Università degli Studi di Bergamo
Facoltà di Ingegneria dell'Informazione

Progetto di Linguaggi e Compilatori

Convertitore GraphML – DOT

Silvio Moioli, Cristian Gatti, Tommaso Bolis

Revisione	Data	Redatto
1,6	14/01/2008	Silvio Moioli

Indice dei contenuti

Storia delle revisioni

Rev. n	Data	Redatto	Descrizione
1.0	16/12/2008	Silvio Moioli	Versione iniziale. Introduzione e specifica di DOT.
1.1	17/12/2008	Silvio Moioli	Inserita parte sulla specifica GraphML, aggiunte note.
1.2	19/12/2008	Silvio Moioli	Modifiche alla parte su GraphML, aggiunti gli esempi e parte della grammatica da implementare.
1.3	02/01/2008	Silvio Moioli	Revisione della descrizione testuale di GraphML, aggiunta la grammatica ristretta completa. Marcate parti da modificare ulteriormente.
1.4	10/01/2008	Silvio Moioli	Corretti alcuni errori, aggiunti link alle specifiche ufficiali del linguaggio. Aggiunti gli alfabeti e modificata la grammatica di GraphML secondo specifiche. Aggiunta la grammatica ad attributi.
1.5	13/01/2008	Silvio Moioli	Modificata la grammatica, corretti alcuni errori, migliorata l'impaginazione.
1.6	14/01/2008	Silvio Moioli	Corretto un errore di copia e uno nell'indice.

Introduzione

Scopo

Scopo del documento è presentare i risultati del progetto di Linguaggi e Compilatori, i linguaggi trattatati e le specifiche semplificative seguite.

Definizioni, abbreviazioni e sigle

- XML: eXtensible Markup Language, linguaggio di marcatura estensibile;
- GraphML, linguaggio basato su XML per la specifica di grafi;

Riferimenti bibliografici e Web

- GraphViz (Wikipedia): http://en.wikipedia.org/wiki/Graphviz
- GraphViz (sito ufficiale): http://graphviz.org/
- Dot: http://en.wikipedia.org/wiki/DOT_language
- GraphML (Wikipedia): http://en.wikipedia.org/wiki/GraphML
- GraphML (sito ufficiale): http://graphml.graphdrawing.org/

Introduzione

L'obiettivo del progetto è di realizzare un convertitore che, preso in ingresso un grafo specificato secondo il linguaggio GraphML, un dialetto dell'XML, produca una rappresentazione in linguaggio DOT. DOT è il linguaggio usato dall'omonimo programma della suite GraphViz, che fra le altre cose è in grado di creare rappresentazioni grafiche dei grafi, ad esempio in forma di immagini o file PDF.

L'idea essenziale è quindi di costruire uno strumento in grado di visualizzare in forma grafica, quindi facile da capire per gli utenti, il dato XML che è tipicamente di facile elaborazione automatica.

Specifica

1 XML

Come menzionato nell'introduzione GraphML è un dialetto di XML, pertanto in questa sezione viene data una descrizione sintetica di quest'ultimo standard.

XML è una specifica che permette di sviluppare linguaggi di marcatura per arbitrari domini applicativi. I linguaggi conformi a XML, o dialetti, hanno in comune una struttura lessicale e sintattica di base. Gli aspetti fondamentali sono presentati qui per agevolare la comprensione di GraphML, che è basato su XML.

Ogni documento XML è un file di testo contenente una dichiarazione della versione in uso e da un albero di elementi. La dichiarazione è normalmente nella forma seguente:

<?xml version="1.0" encoding="UTF-8"?>

dove è specificata la versione del linguaggio e la codifica del testo.

Ogni **elemento** è delimitato da una coppia di marcatori detti **tag** racchiusi tra parentesi angolari come segue:

*<nome_elemento>*contenuto dell'elemento*</nome_elemento>*

Il primo tag è detto **di apertura**, il secondo **di chiusura**. I nomi degli elementi non possono contenere spazi e caratteri speciali e devono iniziare con una lettera, mentre il **contenuto** può essere testo qualunque, altri elementi o testo misto ad elementi. Per evitare ambiguità sono definite delle **entità** che rappresentano nel testo semplice i caratteri particolari come < o > (ad esempio *<* e *>*).

Il contenuto può anche essere vuoto, in tal caso si può scrivere equivalentemente il marcatore nella forma

<nome_elemento></nome_elemento>

oppure nella **forma sintetica**

<nome_elemento />

Inoltre, ad ogni elemento si possono assegnare delle coppie chiave-valore dette attributi. Gli attributi vengono specificati nella forma *chiave=”valore”* all'interno del tag di apertura o nell'unico tag della forma sintetica. Ad esempio:

<nome_elemento attributo1=”valore1” attributo2=”valore2”>contenuto dell'elemento</nome_elemento>

Infine si possono inserire ovunque commenti, che sono esplicitamente trascurati dai programmi che elaborano XML, nella forma:

<!-- Questo è un commento -->

XML è case-sensitive quindi nomi dei tag, degli attributi e così via si considerano diversi anche se differiscono solo per lettere maiuscole o minuscole (ad esempio il tag *<gatto>* è diverso dal tag *<GATTO>*).

Esempio di documento XML completo:

```
<?xml version="1.0" encoding="UTF-8"?>
 <ricetta nome="pane" tempo_preparazione="5 min" tempo_cottura="3 ore">
   <titolo>Pane bianco</titolo>
   <ingrediente quanto="8" misura="dL">Farina</ingrediente>
   <ingrediente quanto="10" misura="grammi">Lievito</ingrediente>
   <ingrediente quanto="4" misura="dL" stato="calda">Acqua</ingrediente>
   <ingrediente quanto="1" misura="cucchiaino">Sale</ingrediente>
   <istruzioni>
     <passo>Mescolare tutti gli ingredienti.</passo>
     <passo>Impastare bene.</passo>
     <passo>Coprire con un panno, lasciare per un'ora in una stanza tiepida.</passo>
     <passo>Impastare nuovamente.</passo>
     <passo>Mettere i panini su una teglia.</passo>
     <passo>Coprire con un panno, lasciare per un'ora in una stanza tiepida.</passo>
     <passo>Cuocere in forno a 180 gradi per 30 minuti.</passo>
   </istruzioni>
 </ricetta>
```

I dialetti di XML possono essere specificati anche tramite linguaggi appositi, come XML-Schema o DTD.

2 GraphML, specifica ufficiale

La specifica ufficiale del linguaggio GraphML è piuttosto lunga e complessa e può essere reperita all'indirizzo http://graphml.graphdrawing.org/specification.html. Viene qui riportata una sintesi qualitativa al fine di agevolare la comprensione del progetto.

Lo scopo di un documento GraphML è di definire uno o più **grafi**. Il documento consiste di un elemento *graphml* e di un insieme di sottoelementi: *graph, node, edge,* di cui in seguito riportiamo la descrizione in dettaglio.

Il sottoelemento *graph* rappresenta un grafo, annidati in esso vi sono le dichiarazioni di **nodi** e **archi**. Un nodo è dichiarato con l'elemento *node* e un arco con l'elemento *edge*; la dichiarazione può essere fatta in qualsiasi ordine. I grafi definiti con GraphML sono misti, in altre parole possono contenere sia **archi orientati** che **archi non orientati**. Un arco può essere esplicitamente dichiarato come orientato o meno nell'elemento *edge*, come spiegato successivamente, altrimenti viene preso per default l'"orientamento" specificato dall'attributo non opzionale *edgedefault* dell'elemento *graph*, che può assumere i valori *directed* o *undirected*. L'elemento *graph* ammette inoltre l'attributo opzionale *id* come identificatore che può essere impiegato per far riferimento al grafo.

Analogamente, ogni elemento *node* ha un attributo *id* che lo identifica, il quale deve essere univoco all'interno dell'intero documento.

Gli elementi *edge* hanno due attributi non opzionali *source* e *target* che identificano i nodi incidenti. Questi attributi devono ovviamente fare riferimento a *id* di nodi dichiarati altrove all'interno del documento; gli **autoanelli** sono definiti assegnando lo stesso valore a sorgente e destinazione. L'attributo opzionale *directed* dichiara se l'arco è orientato o meno, rispettivamente tramite i valori *true* e *false*. Si può definire un identificatore opzionale con l'attributo *id* che può essere impiegato per far riferimento all'arco.

Con l'estensione *GraphML-Attributes* è possibile specificare informazione addizionale (valori numerici e stringhe), di tipo scalare, per gli elementi precedentemente definiti. Per aggiungere contenuto strutturato agli elementi del grafo, è possibile utilizzare un apposito meccanismo di estensione di tipo "chiave-valore" tramite gli elementi *key* e *data.* Gli attributi di GraphML non vanno confusi con quelli di XML, che rappresentano un concetto differente, come dettagliato in seguito.

Un attributo di GraphML è definito da un elemento *key* che specifica un identificatore univoco, un nome, un tipo e un dominio. L'identificatore è definito tramite l'attributo XML *id* di key ed è utilizzato per riferirsi all'attributo di GraphML all'interno del documento. I nomi degli attributi di GraphML, anch'essi univoci, sono invece specificati da *attr.name*. Lo scopo del nome è quello di descrivere il significato dell'attributo: si noti che per far riferimento a quest'ultimo all'interno del documento si utilizza il suo *id* e non il nome. Il tipo degli attributi di GraphML può essere *boolean*, *int*, *long*, *float*, *double*, o *string*, mentre il dominio specifica a quali elementi del documento essi sono associabili (*graph*, *node*, *edge* oppure *all*). E' possibile associare un valore di default agli attributi di GraphML: il contenuto testuale dell'elemento *default* ne definisce il valore.

Il valore di un attributo di GraphML è definito dall'elemento *data* dichiarato all'interno dell'elemento del grafo. L'elemento *data* ha un attributo XML *key* che fa riferimento all'identificatore dell'attributo di GraphML. Il valore assunto dall'attributo è quello definito all'interno dell'elemento *data*, ed il suo tipo deve essere congruo con quanto dichiarato nella definizione dell'elemento *key.* Come anticipato nel paragrafo precedente l'elemento data può essere omesso nel caso in cui la definizione di *key* abbia specificato un valore di default.

GraphML supporta anche i **grafi nidificati**, nei quali i nodi sono ordinati gerarchicamente. La gerarchia è descritta dalla struttura del documento: un nodo può avere un elemento *graph* il quale a sua volta contiene nodi a livello gerarchico inferiore. Gli archi tra due nodi in un sottografo devono essere dichiarati in un grafo che si trovi ad un livello gerarchico superiore ad entrambi i nodi.

Infine GraphML permette di definire **iperarchi** attraverso l'elemento *hyperedge*; un iperarco è una generalizzazione di un arco che esprime una relazione tra un numero arbitrario di nodi. Ad ogni elemento *hyperedge* sono associati uno o più elementi *endpoint* che fanno riferimento ai nodi messi in relazione dall'iperarco. L'elemento *endpoint* deve avere un attributo XML *node*, che contiene l'identificatore del nodo.

3 GraphML, sottoinsieme da implementare

Nello sviluppo del nostro progetto andremo ad utilizzare solo una parte delle funzionalità messe a disposizione da GraphML, andando di fatto a definire una grammatica semplificata.

In particolare ci limiteremo a permettere la definizione degli elementi:

- *graph,* con i relativi attributi *id* e *edgedefault*;
- *node*, con il relativo attributo *id*;
- *edge,* con i relativi attributi *source* e *target*;

La grammatica semplificata di GraphML è esprimibile con la notazione usata nel corso con gli alfabeti e le produzioni seguenti:

$$\Sigma = \{ <, /, >, ", =, ?, graphml, xml, version, encoding, graphml, xmlns, attribute, xmlns:xsi,$$

$xsi : schemaLocation , graph , id , edgedefault , directed , undirected , node , edge , source , target , idvalue\}$

$V = \{$GRAPHML_DOCUMENT , XML , GRAPHML_TAG , GRAPH_LIST , VERSION , ENCODING , OPEN_GRAPHML_TAG , XMLNS , XMLNS_SI , SCHEMA_LOCATION , GRAPH_ELEMENT , OPEN_GRAPH_TAG , GRAPH_ATTR_LIST , GRAPH_ID , EDGE_DEFAULT , ID , EDGE_DEFAULT_VALUE , CONTENT , ELMENT_LIST , ELEMENT , ELEMENT_NODE , ELEMENT_EDGE , EDGE_ATTR_LIST$\}$

S → GRAPHML_DOCUMENT
GRAPHML_DOCUMENT → XML OPEN_GRAPHML_TAG GRAPH_LIST $< / graphml >$

XML → $< ? xml\ version =$ VERSION $encoding =$ ENCODING $? >$
OPEN_GRAPHML_TAG → $< graphml\ xmlns =$ XMLNS $xmlns : xsi =$ XMLNS_XSI
$xsi : schemaLocation =$ SCHEMA_LOCATION $>$
VERSION → " *attribute* "
ENCODING → " *attribute* "
XMLNS → " *attribute* "
XMLNS_SI → " *attribute* "
SCHEMA_LOCATION → " *attribute* "

GRAPH_LIST → GRAPH_ELEMENT GRAPH_LIST
GRAPH_LIST → ϵ

GRAPH_ELEMENT → OPEN_GRAPH_TAG $/ >$
GRAPH_ELEMENT → OPEN_GRAPH_TAG $>$ CONTENT $< /$ *graph* $>$

OPEN_GRAPH_TAG → $<$ *graph* GRAPH_ATTR_LIST

GRAPH_ATTR_LIST → GRAPH_ID EDGE_DEFAULT
GRAPH_ATTR_LIST → EDGE_DEFAULT GRAPH_ID
GRAPH_ID → *id* = ID
GRAPH_ID → ϵ
EDGE_DEFAULT → *edgedefault* = EDGE_DEFAULT_VALUE

EDGE_DEFAULT_VALUE → " *directed* "
EDGE_DEFAULT_VALUE → " *undirected* "

CONTENT → ELEMENT_LIST

ELEMENT_LIST → ELEMENT ELEMENT_LIST
ELEMENT_LIST → ϵ

ELEMENT → ELEMENT_NODE
ELEMENT → ELEMENT_EDGE

ELEMENT_NODE → <*node id* = ID />
ELEMENT_NODE → <*node id* = ID> </*node*>

ELEMENT_EDGE → <*edge* EDGE_ATTR_LIST />
ELEMENT_EDGE → <*edge* EDGE_ATTR_LIST> </*edge*>

EDGE_ATTR_LIST → *source* = ID *target* = ID
EDGE_ATTR_LIST → *target* = ID *source* = ID

ID → "*idvalue*"

A livello lessicale si suppone che il lexer scarti gli spazi vuoti, i ritorni a capo e le tabulazioni non necessarie.

Si lascia inoltre a livello semantico la verifica dei seguenti aspetti:

- il terminale *idvalue* deve essere, per omogeneità con DOT, una stringa alfanumerica senza simboli che non inizia con una cifra;
- il terminale *attribute* può essere a priori una qualunque stringa che XML permette per un attributo, in particolare però poiché il documento deve essere GraphML è necessario che:
  - l'attributo “version” assuma il valore “1.0”;
  - l'attributo “encoding” assuma il valore “UTF-8”;
  - l'attributo “xmlns” assuma il valore “http://graphml.graphdrawing.org/xmlns”;
  - l'attributo “xmlns:xsi” assuma il valore “http://www.w3.org/2001/XMLSchema-instance”;
  - l'attributo “xsi:schemaLocation” assuma il valore “http://graphml.graphdrawing.org/xmlns http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd”;

4 DOT, specifica del linguaggio

La grammatica di DOT è esprimibile con la notazione usata nel corso con le seguenti produzioni:

S → GRAPH

GRAPH → STRICT TYPE ID { STATEMENT_LIST }

STRICT → *strict*

STRICT → ϵ

TYPE → *graph*

TYPE → *digraph*

ID → *id*

ID → ϵ

STATEMENT_LIST → STATEMENT SEMICOLON STATEMENT_LIST

STATEMENT_LIST → ϵ

SEMICOLON → *;*

SEMICOLON → ϵ

STATEMENT → NODE_STATEMENT

STATEMENT → EDGE_STATEMENT

STATEMENT → ATTR_STATEMENT

STATEMENT → *id* = *id*

STATEMENT → SUBGRAPH

ATTR_STATEMENT → GRAPH_NODE_EDGE ATTR_LIST

GRAPH_NODE_EDGE → *graph*

GRAPH_NODE_EDGE → *node*

GRAPH_NODE_EDGE → *edge*

ATTR_LIST → [A_LIST] ATTR_LIST_EMPTY

ATTR_LIST → ϵ

A_LIST → ID = ID COMMA A_LIST

A_LIST → ID COMMA A_LIST

A_LIST → ϵ

COMMA → ,

COMMA → ϵ

EDGE_STATEMENT → NODE_ID EDGE_RHS ATTR_LIST

EDGE_STATEMENT → SUBGRAPH EDGE_RHS ATTR_LIST

EDGE_RHS → EDGE_OP NODE_ID OPTIONAL_EDGE_RHS

EDGE_RHS → EDGE_OP SUBGRAPH OPTIONAL_EDGE_RHS

OPTIONAL_EDGE_RHS → EDGE_RHS

OPTIONAL_EDGE_RHS → ϵ

NODE_STATEMENT → NODE_ID ATTR_LIST

NODE_ID → ID

NODE_ID → ID PORT

PORT → : *id* : COMPASS_PT

PORT → : COMPASS_PT

SUBGRAPH → SUBGRAPH ID { STATEMENT_LIST }

SUBGRAPH → { STATEMENT_LIST }

COMPASS_PT → *n*

COMPASS_PT → *ne*

COMPASS_PT → *e*

COMPASS_PT → *se*

COMPASS_PT → *s*

COMPASS_PT → *sw*

COMPASS_PT → *w*

COMPASS_PT → *nw*

COMPASS_PT → *c*

COMPASS_PT → _

Le stringhe riportate in corsivo rappresentano i terminali della grammatica, mentre quelle in maiuscolo sono i nonterminali. In particolare tutti i terminali sono da intendersi "così come sono" tranne ϵ, che rappresenta la stringa vuota, ed *id*, che può corrispondere a:

- una stringa di caratteri alfabetici, "_" o cifre che non inizia con una cifra,
- una stringa rappresentante un numero decimale,
- una stringa tra doppi apici oppure,
- una stringa HTML.

L'attributo *strict* fa in modo che DOT non visualizzi eventuali autoanelli (anche se in realtà è ignorato dalla implementazione corrente).

Le definizioni ufficiali della grammatica e del lessico sono disponibili all'indirizzo http://www.graphviz.org/doc/info/lang.html.

Esempi

Esempio di traduzione da GraphML a DOT di un grafo non orientato.

Versione GraphML:

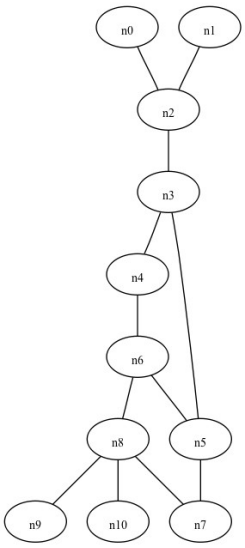
```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <node id="n4"/>
    <node id="n5"/>
    <node id="n6"/>
    <node id="n7"/>
    <node id="n8"/>
    <node id="n9"/>
    <node id="n10"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    <edge source="n2" target="n3"/>
    <edge source="n3" target="n5"/>
    <edge source="n3" target="n4"/>
    <edge source="n4" target="n6"/>
    <edge source="n6" target="n5"/>
    <edge source="n5" target="n7"/>
    <edge source="n6" target="n8"/>
    <edge source="n8" target="n7"/>
    <edge source="n8" target="n9"/>
    <edge source="n8" target="n10"/>
  </graph>
</graphml>
```

Versione DOT:

```
graph G {
  n0 [label="n0"]
  n1 [label="n1"]
  n2 [label="n2"]
  n3 [label="n3"]
  n4 [label="n4"]
```

```
  n5 [label="n5"]
  n6 [label="n6"]
  n7 [label="n7"]
  n8 [label="n8"]
  n9 [label="n9"]
  n10 [label="n10"]
  n0 -- n2
  n1 -- n2
  n2 -- n3
  n3 -- n5
  n3 -- n4
  n4 -- n6
  n6 -- n5
  n5 -- n7
  n6 -- n8
  n8 -- n7
  n8 -- n9
  n8 -- n10
}
```

Rappresentazione grafica prodotta da DOT:



Esempio di traduzione da GraphML a DOT di un grafo orientato.

Versione GraphML:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
      http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
```

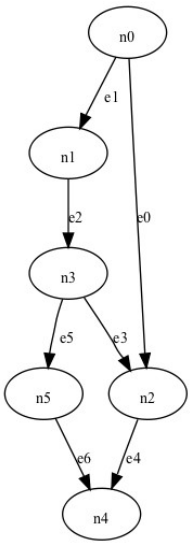
```
  <graph id="G" edgedefault="directed">
    <node id="n0" />
    <node id="n1" />
    <node id="n2" />
    <node id="n3" />
    <node id="n4" />
    <node id="n5" />
    <edge id="e0" source="n0" target="n2" />
    <edge id="e1" source="n0" target="n1" />
    <edge id="e2" source="n1" target="n3" />
    <edge id="e3" source="n3" target="n2" />
    <edge id="e4" source="n2" target="n4" />
    <edge id="e5" source="n3" target="n5" />
    <edge id="e6" source="n5" target="n4" />
  </graph>
</graphml>
```

Versione DOT:

```
digraph G {
  n0 [label="n0"]
  n1 [label="n1"]
  n2 [label="n2"]
  n3 [label="n3"]
  n4 [label="n4"]
  n5 [label="n5"]

  n0 -> n2 [label="e0"]
  n0 -> n1 [label="e1"]
  n1 -> n3 [label="e2"]
  n3 -> n2 [label="e3"]
  n2 -> n4 [label="e4"]
  n3 -> n5 [label="e5"]
  n5 -> n4 [label="e6"]
}
```

Rappresentazione grafica prodotta da DOT:



Grammatica ad attributi

Viene di seguito descritta la grammatica della semplificazione di GraphML considerata con attributi, che è la base per l'implementazione del compilatore.

$$\Sigma = \{ <, /, >, ", =, ?, graphml, xml, version, encoding, graphml, xmlns, attribute, xmlns:xsi, xsi:schemaLocation, graph, id, edgedefault, directed, undirected, node, edge, source, target, idvalue \}$$

$$V = \{ \text{GRAPHML\_DOCUMENT}, \text{XML}, \text{GRAPHML\_TAG}, \text{GRAPH\_LIST}, \text{VERSION}, \text{ENCODING}, \text{OPEN\_GRAPHML\_TAG}, \text{XMLNS}, \text{XMLNS\_SI}, \text{SCHEMA\_LOCATION}, \text{GRAPH\_ELEMENT}, \text{OPEN\_GRAPH\_TAG}, \text{GRAPH\_ATTR\_LIST}, \text{GRAPH\_ID}, \text{EDGE\_DEFAULT}, \text{ID}, \text{EDGE\_DEFAULT\_VALUE}, \text{CONTENT}, \text{ELMENT\_LIST}, \text{ELEMENT}, \text{ELEMENT\_NODE}, \text{ELEMENT\_EDGE}, \text{EDGE\_ATTR\_LIST} \}$$

S → GRAPHML_DOCUMENT

S[0].message ← makeString(GRAPHML_DOCUMENT[1].s_invalid_graph_id_list,
GRAPHML_DOCUMENT[1].s_node_edge_error_list,
GRAPHML_DOCUMENT[1].s_graph_id_list,
GRAPHML_DOCUMENT[1].s_xml_error,
GRAPHML_DOCUMENT[1].s_graphml_error)

S[0].s_code ← GRAPHML_DOCUMENT[1].s_code

GRAPHML_DOCUMENT → XML OPEN_GRAPHML_TAG GRAPH_LIST </*graphml*>

GRAPHML_DOCUMENT[0].s_invalid_graph_id_list ←
checkGraphIdList(GRAPH_LIST[3].s_graph_id_list)

GRAPHML_DOCUMENT[0].s_node_edge_error_list ← GRAPH_LIST[3]. s_node_edge_error_list

GRAPHML_DOCUMENT[0].s_graph_id_list← GRAPH_LIST[3].s_graph_id_list
GRAPHML_DOCUMENT[0].s_xml_error ← XML[1].s_xml_error
GRAPHML_DOCUMENT[0].s_graphml_error← OPEN_GRAPHML_TAG[2].s_graphml_error
GRAPH_LIST[3].node_edge_error_list ← emptyList()
GRAPH_LIST[3].graph_id_list← emptyList()
GRAPHML_DOCUMENT[0].s_code ← GRAPH_LIST[3].s_code
GRAPH_LIST[3].code← emptyString()

XML → < *? xml version* = VERSION *encoding* = ENCODING *?* >
XML[0].s_xml_error← checkXML(VERSION[7].value, ENCODING[12].value)

OPEN_GRAPHML_TAG → < *graphml xmlns* = XMLNS *xmlns* : *xsi* = XMLNS_XSI
** *xsi* : *schemaLocation* = SCHEMA_LOCATION >**
OPEN_GRAPHML_TAG[0].s_graphml_error←
  checkGraphML(XMLNS[6].value, XMLNS_XSI[11].value,
  SCHEMA_LOCATION[16].value)

VERSION → " *attribute* "
VERSION[0].value← attribute.value

ENCODING → " *attribute* "
ENCODING[0].value ← attribute.value

XMLNS → " *attribute* "
XMLNS[0].value← attribute.value

XMLNS_SI → " *attribute* "
XMLNS_XSI[0].s_value← attribute.value

SCHEMA_LOCATION → " *attribute* "
SCHEMA_LOCATION[0].s_value← attribute.value

GRAPH_LIST → GRAPH_ELEMENT GRAPH_LIST
GRAPH_LIST[0].s_node_edge_error_list← GRAPH_LIST[2].s_node_edge_error_list
GRAPH_LIST[0].s_graph_id_list ← GRAPH_LIST[2].s_graph_id_list
GRAPH_LIST[2].node_edge_error_list← GRAPH_ELEMENT[1].s_node_edge_error_list
GRAPH_LIST[2].graph_id_list ← GRAPH_ELEMENT[1].s_graph_id_list
GRAPH_ELEMENT[1].node_edge_error_list← GRAPH_LIST[0].node_edge_error_list

GRAPH_ELEMENT[1].graph_id_list← GRAPH_LIST[0].graph_id_list
GRAPH_LIST[0].s_code ← GRAPH_LIST[2].s_code
GRAPH_ELEMENT[1].code← GRAPH_LIST[0].code
GRAPH_LIST[2].code ← GRAPH_ELEMENT[1].s_code

GRAPH_LIST→ϵ

GRAPH_LIST[0].s_node_edge_error_list ← GRAPH_LIST[0].node_edge_error_list
GRAPH_LIST[0].s_graph_id_list← GRAPH_LIST[0].graph_id_list
GRAPH_LIST[0].s_code ← GRAPH_LIST[0].code

GRAPH_ELEMENT→OPEN_GRAPH_TAG />

GRAPH_ELEMENT[0].s_node_edge_error_list ←
    append(null, GRAPH_ELEMENT[0].node_edge_error_list)
GRAPH_ELEMENT[0].s_graph_id_list ←
    append(OPEN_GRAPH_TAG[1].s_graph_id, GRAPH_ELEMENT[0].graph_id_list)
OPEN_GRAPH_TAG[1].code ← GRAPH_ELEMENT[0].code
OPEN_GRAPH_TAG[1].s_code←
    appendEmptyGraphString(OPEN_GRAPH_TAG[1].code,
    OPEN_GRAPH_TAG[1].s_directed, OPEN_GRAPH_TAG[1].s_graph_id)
GRAPH_ELEMENT[0].s_code ← OPEN_GRAPH_TAG[1].s_code

GRAPH_ELEMENT→OPEN_GRAPH_TAG>CONTENT </ *graph*>

GRAPH_ELEMENT[0].s_node_edge_error_list ←
    append(checkNodesAndEdges(CONTENT[3].s_node_list, CONTENT[3].s_edge_list,
    OPEN_GRAPH_TAG[1].s_directed), GRAPH_ELEMENT[0].node_edge_error_list)
GRAPH_ELEMENT[0].s_graph_id_list←
    append(OPEN_GRAPH_TAG[1].s_graph_id, GRAPH_ELEMENT[0].graph_id_list)
OPEN_GRAPH_TAG[1].code← GRAPH_ELEMENT[0].code
CONTENT[3].code ← OPEN_GRAPH_TAG[1].s_code
GRAPH_ELEMENT[0].s_code← wrap(CONTENT[3].s_code)

OPEN_GRAPH_TAG→< *graph* GRAPH_ATTR_LIST

OPEN_GRAPH_TAG[0].s_graph_id← GRAPH_ATTR_LIST[3].s_graph_id
OPEN_GRAPH_TAG[0].s_directed ← GRAPH_ATTR_LIST[3].s_directed
OPEN_GRAPH_TAG[0].s_code←
    appendGraphString(OPEN_GRAPH_TAG[0].code, OPEN_GRAPH_TAG[0].s_directed,
    OPEN_GRAPH_TAG[0].s_graph_id)

GRAPH_ATTR_LIST → GRAPH_ID EDGE_DEFAULT
GRAPH_ATTR_LIST[0].s_graph_id ← GRAPH_ID[1].s_value
GRAPH_ATTR_LIST[0].s_directed← EDGE_DEFAULT[2].s_directed

GRAPH_ATTR_LIST → EDGE_DEFAULT GRAPH_ID
GRAPH_ATTR_LIST[0].s_graph_id← GRAPH_ID[2].s_value
GRAPH_ATTR_LIST[0].s_directed ← EDGE_DEFAULT[1].s_directed

GRAPH_ID → *id* = ID
GRAPH_ID[0].s_value ← ID[4].s_value

GRAPH_ID → ϵ
GRAPH_ID[0].s_value ← null

EDGE_DEFAULT → *edgedefault* = EDGE_DEFAULT_VALUE
EDGE_DEFAULT[0].s_directed ← EDGE_DEFAULT_VALUE[4].s_value

EDGE_DEFAULT_VALUE → " *directed* "
EDGE_DEFAULT_VALUE[0].s_value ← true

EDGE_DEFAULT_VALUE → " *undirected* "
EDGE_DEFAULT_VALUE[0].s_value ← false

CONTENT → ELEMENT_LIST
CONTENT[0].s_node_list ← ELEMENT_LIST[1].s_node_list
CONTENT[0].s_edge_list← ELEMENT_LIST[1].s_edge_list
ELEMENT_LIST[1].node_list ← emptyList()
ELEMENT_LIST[1].edge_list← emptyList()
CONTENT[0].s_code ← ELEMENT_LIST[1].s_code
ELEMENT_LIST[1].code← CONTENT[0].code

ELEMENT_LIST → ELEMENT ELEMENT_LIST
ELEMENT_LIST[0].s_node_list← ELEMENT_LIST[2].s_node_list
ELEMENT_LIST[0].s_edge_list ← ELEMENT_LIST[2].s_edge_list
ELEMENT[1].node_list← ELEMENT_LIST[0].node_list
ELEMENT[1].edge_list ← ELEMENT_LIST[0].edge_list
ELEMENT_LIST[2].node_list← ELEMENT[1].s_node_list

ELEMENT_LIST[2].edge_list← ELEMENT[1].s_edge_list
ELEMENT_LIST[0].s_code ← ELEMENT_LIST[2].s_code
ELEMENT_LIST[2].code← ELEMENT[1].s_code
ELEMENT[1].code ← ELEMENT_LIST[0].code

ELEMENT_LIST→ϵ
ELEMENT_LIST[0].s_node_list ← ELEMENT_LIST[0].node_list
ELEMENT_LIST[0].s_edge_list← ELEMENT_LIST[0].edge_list
ELEMENT_LIST[0].s_code ← ELEMENT_LIST[0].code

ELEMENT→ELEMENT_NODE
ELEMENT[0].s_node_list ← ELEMENT_NODE[1].s_node_list
ELEMENT_NODE[1].node_list← ELEMENT[0].node_list
ELEMENT[0].s_edge_list ← ELEMENT[0].edge_list
ELEMENT_NODE[1].code← ELEMENT[0].code
ELEMENT[0].s_code ← ELEMENT_NODE[1].s_code

ELEMENT→ELEMENT_EDGE
ELEMENT[0].s_edge_list ← ELEMENT_EDGE[1].s_edge_list
ELEMENT_EDGE[1].edge_list← ELEMENT[0].edge_list
ELEMENT[0].s_node_list ← ELEMENT[0].node_list
ELEMENT_EDGE[1].code← ELEMENT[0].code
ELEMENT[0].s_code ← ELEMENT_EDGE[1].s_code

ELEMENT_NODE→<*node id*=ID/>
ELEMENT_NODE[0].s_node_list ← append(ID[6].s_value, ELEMENT_NODE[0].node_list)
ELEMENT_NODE[0].s_code←
    appendNodeString(ELEMENT_NODE[0].code, ID[6].s_value)

ELEMENT_NODE→<*node id*=ID></*node*>
ELEMENT_NODE[0].s_node_list ← append(ID[6].s_value, ELEMENT_NODE[0].node_list)
ELEMENT_NODE[0].s_code←
    appendNodeString(ELEMENT_NODE[0].code, ID[6].s_value)

ELEMENT_EDGE→<*edge* EDGE_ATTR_LIST/>
ELEMENT_EDGE[0].s_edge_list ← EDGE_ATTR_LIST[3].s_edge_list
EDGE_ATTR_LIST[3].edge_list← ELEMENT_EDGE[0].edge_list

ELEMENT_EDGE[0].s_code← EDGE_ATTR_LIST[3].s_code

EDGE_ATTR_LIST[3].code ← ELEMENT_EDGE[0].code

ELEMENT_EDGE → < *edge* EDGE_ATTR_LIST > < / *edge* >

ELEMENT_EDGE[0].s_edge_list ← EDGE_ATTR_LIST[3].s_edge_list

EDGE_ATTR_LIST[3].edge_list← ELEMENT_EDGE[0].edge_list

ELEMENT_EDGE[0].s_code ← EDGE_ATTR_LIST[3].s_code

EDGE_ATTR_LIST[3].code← ELEMENT_EDGE[0].code

EDGE_ATTR_LIST → *source* = ID *target* = ID

EDGE_ATTR_LIST[0].s_edge_list←

append(couple(ID[4].s_value, ID[9].s_value), EDGE_ATTR_LIST[0].edge_list)

EDGE_ATTR_LIST[0].s_code ←

appendEdgeString(EDGE_ATTR_LIST[0].code, ID[4].s_value, ID[9].s_value)

EDGE_ATTR_LIST → *target* = ID *source* = ID

EDGE_ATTR_LIST[0].s_edge_list ←

append(couple(ID[9].s_value, ID[4].s_value), EDGE_ATTR_LIST[0].edge_list)

EDGE_ATTR_LIST[0].s_code←

appendEdgeString(EDGE_ATTR_LIST[0].code, ID[9].s_value, ID[4].s_value)

ID → " *idvalue* "

ID[0].s_value← idvalue.value

Grammatica ad attributi commentata

Viene di seguito riportata una versione della grammatica con spiegazione degli attributi e delle procedure.

$\Sigma = \{ <, /, >, ", =, ?, graphml, xml, version, encoding, graphml, xmlns, attribute, xmlns:xsi, xsi:schemaLocation, graph, id, edgedefault, directed, undirected, node, edge, source, target, idvalue \}$

$V = \{$ GRAPHML_DOCUMENT, XML, GRAPHML_TAG, GRAPH_LIST, VERSION, ENCODING, OPEN_GRAPHML_TAG, XMLNS, XMLNS_SI, SCHEMA_LOCATION, GRAPH_ELEMENT, OPEN_GRAPH_TAG, GRAPH_ATTR_LIST, GRAPH_ID, EDGE_DEFAULT, ID, EDGE_DEFAULT_VALUE, CONTENT, ELMENT_LIST, ELEMENT, ELEMENT_NODE, ELEMENT_EDGE, EDGE_ATTR_LIST $\}$

S → GRAPHML_DOCUMENT

S[0].message ← makeString(GRAPHML_DOCUMENT[1].s_invalid_graph_id_list,
GRAPHML_DOCUMENT[1].s_node_edge_error_list,
GRAPHML_DOCUMENT[1].s_graph_id_list,
GRAPHML_DOCUMENT[1].s_xml_error,
GRAPHML_DOCUMENT[1].s_graphml_error)

S[0].s_code ← GRAPHML_DOCUMENT[1].s_code

La procedura *makeString* genera una stringa, *message*, che dovrebbe essere visualizzata all'utente al termine della compilazione e descrive gli eventuali errori. In particolare contiene:

- "ID di grafo duplicato: id", dove *id* è il primo elemento duplicato della lista *s_graph_id_list*, che raccoglie gli ID dei grafi per ordine di dichiarazione;
- "ID di grafo non valido: *id*", dove *id* è il primo elemento non nullo di *s_invalid_graph_id_list*. Come definito nel paragrafo successivo, *s_invalid_graph_id_list* contiene tutti gli ID di grafo non validi incontrati nel documento;
- "Dichiarazione non valida nel grafo *id*, *causa*", dove *id* è l'ID o il numero di sequenza di un grafo e *causa* può essere "ID di nodo non valido", "nodo dichiarato due volte", "arco dichiarato due volte" oppure "arco dichiarato su nodo inesistente". La causa può essere tratta dal primo elemento non nullo di *s_node_edge_error_list*, che contiene il codice del primo errore trovato su ogni grafo. L'ID del grafo con errori può essere ricavato da *s_graph_id_list;*
- "Dichiarazione XML non valida" se l'attributo booleano s_xml_error è vero (denota un errore nella stringa di dichiarazione del documento XML);
- "Dichiarazione GraphML non valida" se l'attributo booleano s_graphml_error è vero (denota un errore nella stringa di dichiarazione del documento GraphML);
- "compilazione avvenuta" altrimenti, ossia nel caso in cui non siano effettivamente avvenuti errori di compilazione.

L'attributo sintetizzato *s_code*, invece, contiene il codice generato (indipendentemente dalla correttezza) e viene maggiormente specificato nel seguito.

GRAPHML_DOCUMENT → XML OPEN_GRAPHML_TAG GRAPH_LIST </*graphml*>

GRAPHML_DOCUMENT[0].s_invalid_graph_id_list←
checkGraphIdList(GRAPH_LIST[3].s_graph_id_list)

GRAPHML_DOCUMENT[0].s_node_edge_error_list← GRAPH_LIST[3]. s_node_edge_error_list

GRAPHML_DOCUMENT[0].s_graph_id_list← GRAPH_LIST[3].s_graph_id_list

GRAPHML_DOCUMENT[0].s_xml_error ← XML[1].s_xml_error

GRAPHML_DOCUMENT[0].s_graphml_error← OPEN_GRAPHML_TAG[2].s_graphml_error

GRAPH_LIST[3].node_edge_error_list ← emptyList()

GRAPH_LIST[3].graph_id_list← emptyList()

GRAPHML_DOCUMENT[0].s_code ← GRAPH_LIST[3].s_code

GRAPH_LIST[3].code← emptyString()

La procedura *checkGraphIdList* genera una lista, *s_invalid_graph_id_list*, con tanti elementi quanti sono i grafi dichiarati nel documento. Se un grafo ha un ID che non rispetta le regole sintattiche,

questo ID è salvato nella posizione corrispondente della lista; viceversa in quella posizione è salvato un valore nullo. La lista degli ID dei grafi è fornita in input (*s_graph_id_list*).

s_node_edge_error_list è un attributo sintetizzato che riporta una lista indicizzata per grafo di codifiche degli errori relativi a nodi e archi, se esistono, relativamente alla rilevazione di un ID di nodo non valido, un nodo dichiarato due volte, un arco dichiarato due volte oppure un arco dichiarato su un nodo inesistente. La versione ereditata della lista, *node_edge_error_list*, viene invece inizializzata.

s_graph_id_list è una lista contenente gli ID dei grafi in ordine di dichiarazione o valori nulli nel caso fossero omessi. La versione ereditata della lista, *graph_id_list*, viene invece inizializzata.

s_xml_error e *s_graphml_error* sono attributi sintetizzati booleani che assumono valore vero nel caso di errori rispettivamente nelle dichiarazioni del documento XML e GraphML, e sono descritti nelle prossime produzioni.

emptyList crea e ritorna una lista vuota, *emptyString* una stringa vuota.

L'attributo *code* è la controparte ereditata di *s_code*.

XML → < *? xml version* = VERSION *encoding* = ENCODING *?* >

XML[0].s_xml_error ← checkXML(VERSION[7].value, ENCODING[12].value)

checkXML controlla che la dichiarazione XML sia corretta come definito nella grammatica.

OPEN_GRAPHML_TAG → < *graphml xmlns* = XMLNS *xmlns* : *xsi* = XMLNS_XSI
_____ *xsi* : *schemaLocation* = SCHEMA_LOCATION >

OPEN_GRAPHML_TAG[0].s_graphml_error ←
_____ checkGraphML(XMLNS[6].value, XMLNS_XSI[11].value,
_____ SCHEMA_LOCATION[16].value)

checkGraphML controlla che la dichiarazione GraphML sia corretta come definito nella grammatica.

VERSION → " *attribute* "

VERSION[0].value ← attribute.value

ENCODING → " *attribute* "

ENCODING[0].value ← attribute.value

XMLNS → " *attribute* "

XMLNS[0].value ← attribute.value

XMLNS_SI → " *attribute* "

XMLNS_XSI[0].s_value ← attribute.value

SCHEMA_LOCATION → " *attribute* "

SCHEMA_LOCATION[0].s_value ← attribute.value

GRAPH_LIST → GRAPH_ELEMENT GRAPH_LIST

GRAPH_LIST[0].s_node_edge_error_list ← GRAPH_LIST[2].s_node_edge_error_list

GRAPH_LIST[0].s_graph_id_list ← GRAPH_LIST[2].s_graph_id_list

GRAPH_LIST[2].node_edge_error_list ← GRAPH_ELEMENT[1].s_node_edge_error_list

GRAPH_LIST[2].graph_id_list ← GRAPH_ELEMENT[1].s_graph_id_list

GRAPH_ELEMENT[1].node_edge_error_list ← GRAPH_LIST[0].node_edge_error_list

GRAPH_ELEMENT[1].graph_id_list ← GRAPH_LIST[0].graph_id_list

GRAPH_LIST[0].s_code ← GRAPH_LIST[2].s_code

GRAPH_ELEMENT[1].code ← GRAPH_LIST[0].code

GRAPH_LIST[2].code ← GRAPH_ELEMENT[1].s_code

GRAPH_LIST → ϵ

GRAPH_LIST[0].s_node_edge_error_list ← GRAPH_LIST[0].node_edge_error_list

GRAPH_LIST[0].s_graph_id_list ← GRAPH_LIST[0].graph_id_list

GRAPH_LIST[0].s_code ← GRAPH_LIST[0].code

GRAPH_ELEMENT → OPEN_GRAPH_TAG />

GRAPH_ELEMENT[0].s_node_edge_error_list ←

append(null, GRAPH_ELEMENT[0].node_edge_error_list)

GRAPH_ELEMENT[0].s_graph_id_list ←

append(OPEN_GRAPH_TAG[1].s_graph_id, GRAPH_ELEMENT[0].graph_id_list)

OPEN_GRAPH_TAG[1].code ← GRAPH_ELEMENT[0].code

OPEN_GRAPH_TAG[1].s_code ←

appendEmptyGraphString(OPEN_GRAPH_TAG[1].code,

OPEN_GRAPH_TAG[1].s_directed, OPEN_GRAPH_TAG[1].s_graph_id)

GRAPH_ELEMENT[0].s_code ← OPEN_GRAPH_TAG[1].s_code

Questo gruppo di regole semantiche riguardano, nell'ordine, la gestione degli errori e la generazione del codice. Per quanto concerne la generazione degli errori, *append* aggiunge un elemento (primo argomento) in coda a una lista (secondo argomento). In questo caso, dal momento che il grafo è vuoto, aggiunge un valore nullo ad *s_node_edge_error_list* il che denota l'assenza di errori per questo grafo. Similmente la seconda regola aggiunge alla lista degli ID dei grafi il valore sintetizzato da OPEN_GRAPH_TAG.

Per quanto riguarda la generazione del codice oggetto, invece, la procedura *appendEmptyGraphString* ritorna la concatenazione del codice generato finora, specificato nel primo parametro, con la stringa relativa alla dichiarazione di un grafo vuoto. Per generare la dichiarazione vengono inoltre specificati, nell'ordine, il tipo di grafo (orientato o meno) e il suo ID.

GRAPH_ELEMENT → OPEN_GRAPH_TAG > CONTENT </ *graph* >

GRAPH_ELEMENT[0].s_node_edge_error_list←

append(checkNodesAndEdges(CONTENT[3].s_node_list, CONTENT[3].s_edge_list, OPEN_GRAPH_TAG[1].s_directed), GRAPH_ELEMENT[0].node_edge_error_list)

GRAPH_ELEMENT[0].s_graph_id_list ←

append(OPEN_GRAPH_TAG[1].s_graph_id, GRAPH_ELEMENT[0].graph_id_list)

OPEN_GRAPH_TAG[1].code ← GRAPH_ELEMENT[0].code

CONTENT[3].code← OPEN_GRAPH_TAG[1].s_code

GRAPH_ELEMENT[0].s_code ← wrap(CONTENT[3].s_code)

Per quanto riguarda la gestione degli errori, similmente alla produzione precedente, vengono aggiornate le liste *s_node_edge_error_list* e *s_graph_id_list*. La differenza sostanziale è che in questo caso, essendoci del contenuto, viene chiamata la procedura *checkNodesAndEdges* che controlla le dichiarazioni di nodi e archi e può ritornare un codice di errore, oppure il valore nullo se non ci sono anomalie. A questa procedura vengono passati i parametri:

- *s_node_list*, una lista di stringhe che rappresentano gli ID dei nodi così come sono dichiarati nel documento;
- *s_edge_list*, una lista di coppie di stringhe che rappresentano gli ID dei nodi sorgente e destinazione degli archi così come sono dichiarati nel documento;
- *s_directed*, un booleano che è vero se il grafo è orientato.

I controlli effettuati da *checkNodesAndEdges* sono i seguenti:

1. controllo che gli ID dei nodi siano corretti (stringhe alfanumeriche che non iniziano con una cifra);
2. controllo che uno stesso nodo non sia dichiarato più volte;
3. controllo che gli ID riportati come sorgente e destinazione sugli archi corrispondano a nodi dichiarati;
4. controllo che gli archi non siano dichiarati più volte. Nota che nel caso di grafo non orientato le coppie relative agli archi sono da considerarsi non orientate quindi la dichiarazione di un arco (1, 2) non può seguire la dichiarazione di un arco (2, 1) poiché è duplicato.

La procedura ritorna un codice relativo al primo errore incontrato oppure il valore nullo se non ci sono errori.

Per quanto riguarda la generazione del codice la procedura *wrap* occorre per racchiudere il codice generato relativo al grafo tra parentesi graffe così come richiesto da DOT.

OPEN_GRAPH_TAG→< *graph* GRAPH_ATTR_LIST

OPEN_GRAPH_TAG[0].s_graph_id ← GRAPH_ATTR_LIST[3].s_graph_id

OPEN_GRAPH_TAG[0].s_directed← GRAPH_ATTR_LIST[3].s_directed

OPEN_GRAPH_TAG[0].s_code ←

appendGraphString(OPEN_GRAPH_TAG[0].code, OPEN_GRAPH_TAG[0].s_directed, OPEN_GRAPH_TAG[0].s_graph_id)

La procedura *appendGraphString* ritorna la concatenazione del codice generato finora con la dichiarazione di un grafo in modo analogo al caso del grafo vuoto.

GRAPH_ATTR_LIST → GRAPH_ID EDGE_DEFAULT

GRAPH_ATTR_LIST[0].s_graph_id ← GRAPH_ID[1].s_value

GRAPH_ATTR_LIST[0].s_directed ← EDGE_DEFAULT[2].s_directed

GRAPH_ATTR_LIST → EDGE_DEFAULT GRAPH_ID

GRAPH_ATTR_LIST[0].s_graph_id ← GRAPH_ID[2].s_value

GRAPH_ATTR_LIST[0].s_directed ← EDGE_DEFAULT[1].s_directed

GRAPH_ID → *id* = ID

GRAPH_ID[0].s_value ← ID[4].s_value

GRAPH_ID → ϵ

GRAPH_ID[0].s_value ← null

Essendo l'ID del grafo opzionale, si prevede che l'attributo possa assumere valore nullo. In questo caso le procedure utilizzeranno internamente la posizione nella sequenza di dichiarazione al posto dell'ID per identificare il grafo.

EDGE_DEFAULT → *edgedefault* = EDGE_DEFAULT_VALUE

EDGE_DEFAULT[0].s_directed ← EDGE_DEFAULT_VALUE[4].s_value

EDGE_DEFAULT_VALUE → " *directed* "

EDGE_DEFAULT_VALUE[0].s_value ← true

EDGE_DEFAULT_VALUE → " *undirected* "

EDGE_DEFAULT_VALUE[0].s_value ← false

CONTENT → ELEMENT_LIST

CONTENT[0].s_node_list ← ELEMENT_LIST[1].s_node_list

CONTENT[0].s_edge_list ← ELEMENT_LIST[1].s_edge_list

ELEMENT_LIST[1].node_list ← emptyList()

ELEMENT_LIST[1].edge_list ← emptyList()

CONTENT[0].s_code ← ELEMENT_LIST[1].s_code

ELEMENT_LIST[1].code ← CONTENT[0].code

ELEMENT_LIST → ELEMENT ELEMENT_LIST

ELEMENT_LIST[0].s_node_list ← ELEMENT_LIST[2].s_node_list

ELEMENT_LIST[0].s_edge_list ← ELEMENT_LIST[2].s_edge_list

ELEMENT[1].node_list ← ELEMENT_LIST[0].node_list

ELEMENT[1].edge_list ← ELEMENT_LIST[0].edge_list

ELEMENT_LIST[2].node_list← ELEMENT[1].s_node_list

ELEMENT_LIST[2].edge_list ← ELEMENT[1].s_edge_list

ELEMENT_LIST[0].s_code← ELEMENT_LIST[2].s_code

ELEMENT_LIST[2].code ← ELEMENT[1].s_code

ELEMENT[1].code← ELEMENT_LIST[0].code

ELEMENT_LIST→*ϵ*

ELEMENT_LIST[0].s_node_list← ELEMENT_LIST[0].node_list

ELEMENT_LIST[0].s_edge_list ← ELEMENT_LIST[0].edge_list

ELEMENT_LIST[0].s_code← ELEMENT_LIST[0].code

ELEMENT→ELEMENT_NODE

ELEMENT[0].s_node_list← ELEMENT_NODE[1].s_node_list

ELEMENT_NODE[1].node_list ← ELEMENT[0].node_list

ELEMENT[0].s_edge_list← ELEMENT[0].edge_list

ELEMENT_NODE[1].code ← ELEMENT[0].code

ELEMENT[0].s_code← ELEMENT_NODE[1].s_code

ELEMENT→ELEMENT_EDGE

ELEMENT[0].s_edge_list← ELEMENT_EDGE[1].s_edge_list

ELEMENT_EDGE[1].edge_list ← ELEMENT[0].edge_list

ELEMENT[0].s_node_list← ELEMENT[0].node_list

ELEMENT_EDGE[1].code ← ELEMENT[0].code

ELEMENT[0].s_code← ELEMENT_EDGE[1].s_code

ELEMENT_NODE→<*node id*=ID/>

ELEMENT_NODE[0].s_node_list← append(ID[6].s_value, ELEMENT_NODE[0].node_list)

ELEMENT_NODE[0].s_code ←

appendNodeString(ELEMENT_NODE[0].code, ID[6].s_value)

ELEMENT_NODE→<*node id*=ID></*node*>

ELEMENT_NODE[0].s_node_list← append(ID[6].s_value, ELEMENT_NODE[0].node_list)

ELEMENT_NODE[0].s_code ←

appendNodeString(ELEMENT_NODE[0].code, ID[6].s_value)

appendNodeString ritorna la concatenazione del codice generato finora, specificato nel primo parametro, con la stringa relativa alla dichiarazione di un nodo, nel secondo parametro.

ELEMENT_EDGE→<*edge* EDGE_ATTR_LIST/>

ELEMENT_EDGE[0].s_edge_list← EDGE_ATTR_LIST[3].s_edge_list

EDGE_ATTR_LIST[3].edge_list ← ELEMENT_EDGE[0].edge_list

ELEMENT_EDGE[0].s_code← EDGE_ATTR_LIST[3].s_code

EDGE_ATTR_LIST[3].code ← ELEMENT_EDGE[0].code

ELEMENT_EDGE→<*edge* EDGE_ATTR_LIST></*edge*>

ELEMENT_EDGE[0].s_edge_list ← EDGE_ATTR_LIST[3].s_edge_list

EDGE_ATTR_LIST[3].edge_list← ELEMENT_EDGE[0].edge_list

ELEMENT_EDGE[0].s_code ← EDGE_ATTR_LIST[3].s_code

EDGE_ATTR_LIST[3].code← ELEMENT_EDGE[0].code

EDGE_ATTR_LIST→*source*=ID *target*=ID

EDGE_ATTR_LIST[0].s_edge_list←

append(couple(ID[4].s_value, ID[9].s_value), EDGE_ATTR_LIST[0].edge_list)

EDGE_ATTR_LIST[0].s_code←

appendEdgeString(EDGE_ATTR_LIST[0].code, ID[4].s_value, ID[9].s_value)

couple genera una coppia ordinata (lista di due elementi) dagli argomenti.

appendEdgeString ritorna la concatenazione del codice generato finora, specificato nel primo parametro, con la stringa relativa a un arco dal nodo sorgente (secondo parametro) al nodo destinazione (terzo parametro).

EDGE_ATTR_LIST→*target*=ID *source*=ID

EDGE_ATTR_LIST[0].s_edge_list ←

append(couple(ID[9].s_value, ID[4].s_value), EDGE_ATTR_LIST[0].edge_list)

EDGE_ATTR_LIST[0].s_code ←

appendEdgeString(EDGE_ATTR_LIST[0].code, ID[9].s_value, ID[4].s_value)

ID→"*idvalue*"

ID[0].s_value ← idvalue.value