Http2Java - Documentazione Tecnica

Pellegrinelli Nico, Pellegrinelli Sean, Panzeri Federico  
Versione del 27/01/2024

Sommario

[Contesto 3](#_Toc157262801)

[Requisiti funzionali 4](#_Toc157262802)

[Errori e Warning 5](#_Toc157262803)

[Tecnologie 7](#_Toc157262804)

[Design 8](#_Toc157262805)

[Linguaggio http 11](#_Toc157262806)

[Implementazione analisi semantica 18](#_Toc157262807)

# Indice delle figure

[Figura 1 - Statechart Diagram 4](#_Toc157262819)

[Figura 2 - Package Diagram 8](#_Toc157262820)

[Figura 3 - Class Diagram 9](#_Toc157262821)

[Figura 4 - Sequence Diagram del processo di traduzione 10](#_Toc157262822)

# Contesto

Http2Java nasce dall’esigenza di un team di sviluppatori di rendere più efficiente ed efficacie il proprio lavoro, agendo in particolar modo sulle mansioni più ripetitive e noiose, che inducono spesso i programmatori in errori di distrazione.

In particolar modo è stato identificato il seguente problema: gli sviluppatori del team devono spesso scrivere codice Java per chiamate http, queste chiamate sono spesso molto simili tra loro e di conseguenza lo è anche il codice che le implementa, ciò porta gli sviluppatori a non porre particolare attenzione al lavoro e introdurre svariati errori nelle richieste. Questo problema è accentuato dal fatto che diversi di questi errori non vengono rilevati dalla libreria Java usata per le richieste http e non impediscono l’invio della richiesta stessa (ad esempio, se viene implementata una richiesta con l’header “Usr-Agent” anziché “User-Agent”, non viene segnalato alcun tipo di errore, né in compilazione né a runtime).

Queste problematiche hanno portato allo sviluppo del compilatore Http2Java, che, tramite un’interfaccia grafica, permette di scrivere richieste http in un apposito linguaggio e di ottenere la traduzione in codice Java, implementando una serie di controlli sulla struttura e il contenuto della richiesta così da fornire utili messaggi di errore allo sviluppatore.

# Requisiti funzionali

La funzionalità principale di Http2Java consiste nel tradurre richieste http (con metodo GET o POST) scritte in un linguaggio ad-hoc in codice Java. Il linguaggio per le richieste, la cui specifica è riportata nella sezione [Linguaggio http](#_Linguaggio_http), deve rispettare la struttura tipica delle richieste http:

* request line contenete metodo http, path e versione http;
* una lista di header come coppie nome:valore;
* body della richiesta, opzionale.

Http2Java fornisce una semplice ed intuitiva interfaccia grafica che permette di:

1. scrivere, in un box dedicato, le richieste http nel linguaggio ad-hoc;
2. importare richieste http da file di testo (alla pressione del pulsante “Import File”) e modificare a piacimento tali richieste (funzione abilitata alla pressione del pulsante “Toggle File”), per mezzo del medesimo box di testo;
3. alla pressione del pulsante “Translate”:
   1. se la richiesta http presente nel box dedicato non presenta errori, tradurre la richiesta in codice Java, che viene mostrato in un apposito box di testo;
   2. se la richiesta http presenta uno o più errori, mostrare la lista degli errori e, eventualmente, la posizione dell’elemento che ha generato tale errore (nessun codice Java viene generato);
   3. mostrare la lista degli eventuali warning, che non impediscono la generazione del codice;
4. alla pressione del pulsante “Copy”, copiare nella Clipboard il codice Java generato, se presente.

Il funzionamento di Http2Java è riassunto nel seguente statechart diagram:

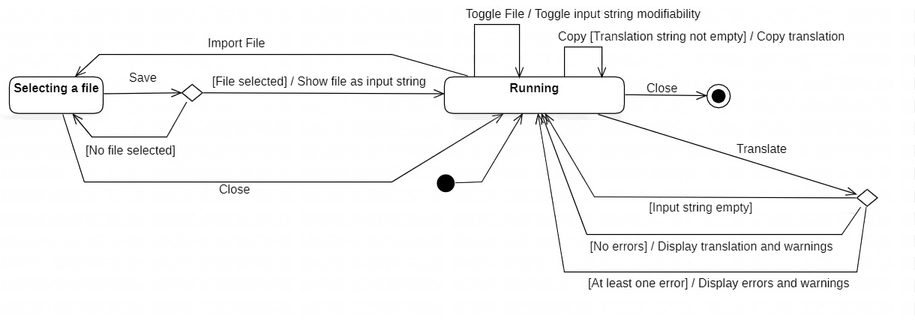


Figura 1 - Statechart Diagram

Http2Java viene distribuito come un eseguibile Jar.

## Errori e Warning

Il focus principale dell’applicazione è la rilevazione di errori e warning per rendere più efficiente la scrittura delle richieste http.

Si fa notare che nell’applicazione è possibile utilizzare tutti gli header possibili (utilizzando una stringa tra singoli apici sia per il nome che per il contenuto dell’header), ma viene implementato un controllo di correttezza lessicale, sintattica e semantica solo su header il cui nome, non tra singoli apici, è uno dei seguenti:

* Host
* User-Agent
* Content-Type
* Accept
* Cookie
* Authorization
* Accept-Language
* Accept-Encoding
* Cache-Control
* Max-Forwards

Sul body non viene fatto nessun controllo se non che sia una stringa delimitata da due caratteri # (controllo sintattico). Sulla request line si implementano solo controlli lessicali e sintattici.

La stringa mostrata in caso di errore ha la seguente struttura:

* **posizione**, nel formato [riga : colonna], del token che ha generato l’errore. Se l’errore non è generato da un token univocamente identificabile la posizione viene omessa;
* **tipo** dell’errore; sono possibili tre tipi:
  1. duplicate information;
  2. missing information;
  3. wrong information;
* **messaggio** specifico dell’errore.

La presenza di errori preclude la generazione della traduzione della richiesta http in codice java. Di seguito vengono indicate le cause di errore per ognuno dei tre tipi di errore:

1. **Duplicate information**: informazione già specificata; si verifica nei seguenti casi:
   1. inserimento di un header il cui nome è già stato utilizzato precedentemente come nome di un altro header (sia tra singoli apici che non);
   2. nel caso di header con nome “Authorization” seguito dall’ID “Digest”, inserimento di un parametro (del tipo ID = STRING) con ID utilizzato precedentemente come ID di un altro parametro all’interno dello stesso header.
2. **Missing information**: mancanza di un’informazione necessaria alla traduzione; si verifica nei seguenti casi:
   1. nessun header con nome “Host” specificato (sia tra singoli apici che non).
3. **Wrong information**: informazione errata che non consente; si verifica nei seguenti casi:
   1. inserimento di sequenze di caratteri non appartenenti alla grammatica (errori lessicali);
   2. inserimento di token che non rispettano la struttura della grammatica (errori sintattici);
   3. nel caso di header con nome “Content-Type”, inserimento, a seguito di un token MIME, di un ID diverso da “charset”;
   4. nel caso di header con nome “Content-Type”, inserimento, a seguito di un token MULTIPART\_MIME, di un ID diverso da “boundary”;
   5. nel caso di header con nome “Authorization”, inserimento di un ID diverso da “Basic” e seguito da una stringa;
   6. nel caso di header con nome “Authorization”, inserimento di un ID diverso da “Digest” e seguito da una lista di parametri (del tipo ID = STRING);
   7. nel caso di header con nome “Authorization” seguito dall’ID “Digest”, inserimento di un parametro (del tipo ID = STRING) con ID diverso da "username", "realm", "uri", "algorithm", "nonce", "nc", "cnonce", "qop", "response" o "opaque";
   8. nel caso di header con nome “Accept-Language”, inserimento di un ID con un solo carattere o più di tre caratteri.
   9. nel caso di header con nome “Accept-Encoding”, inserimento di un ID diverso da "gzip", "compress", "deflate", "br" o "identity".

Per gli errori lessicali e sintattici e per i token si faccia riferimento alla grammatica non decorata riportata nella sezione [Linguaggio http](#_Linguaggio_http).

Vengono inoltre segnalati degli warning. La struttura della stringa mostrata in caso di warning è la seguente:

* la stringa “Warning: ”;
* messaggio specifico dello warning.

Gli warning, che non precludono la generazione della traduzione della richiesta http in codice java, sono mostrati nei seguenti casi:

1. definizione del body in una richiesta GET;
2. definizione dell’header “Content-Type” in una richiesta GET;
3. mancata definizione del body in una richiesta POST;
4. mancata definizione dell’header “Content-Type” in una richiesta POST.

# Tecnologie

Http2Java è stato sviluppato con i seguenti tool e tecnologie:

* **Java**: il linguaggio di programmazione dell’intero progetto (l’IDE utilizzato è **Eclipse**).
* **ANTLR 3**: la grammatica per http è stata definita mediante ANTLR 3.4 (con l’aiuto dell’IDE **ANTLRWorks**). Il codice viene generato in Java.
* **Java Swing**: framework orientato allo sviluppo di interfacce grafiche.
* **Apache Maven**: framework per la gestione del progetto Java e la generazione dell’eseguibile in formato Jar.
* **HTTP client API**: libreria, disponibile a partire da Java 11, per l’implementazione di richieste http. Il codice Java generato da Http2Java sfrutta questa libreria.

Il progetto Maven, la relativa documentazione e l’eseguibile si possono trovare nella seguente repository **Github**: <https://github.com/PellegrinelliNico/ProgettoLFC>.

# Design

Il progetto Java è composto da tre package:

* **compiler**: contiene la specifica della grammatica http (salvata nel file http.g) e i file generati a partire dalla stessa (ovvero HttpLexer.java, HttpParser.java e http.tokens). Contiene anche il file SemanticHandler.java, contenente la classe che implementa l’analisi semantica. In questo package è contenuto quindi il compilatore nella sua interezza (lessico, sintassi e semantica).
* **gui**: contiene i file Gui.java e TextLineNumber.java, contenenti le classi che implementano l’interfaccia grafica. La classe Gui contiene il metodo main e implementa l’interfaccia grafica nella sua interezza, fatta eccezione per i numeri di riga presenti nel box testuale http, che sono implementati nella classe TextLineNumber.
* **variables**: contiene i file Header.java, RequestLine.java e CompilerError.java, ognuno contenente una classe utilizzata come data type. I data type hanno lo scopo di contenere i dati generati dal compilatore e facilitarne lo scambio e l’utilizzo.

Di seguito viene riportato il package diagram, dove vengono indicate le dipendenze e i file contenuti in ogni package:

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Figura 2 - Package Diagram

Di seguito viene riportato il class diagram, per le classi HttpLexer e HttpParser non sono indicati i metodi e i campi ma vengono rappresentate come “black-box”, visto che sono generate automaticamente a partire dalla grammatica decorata:

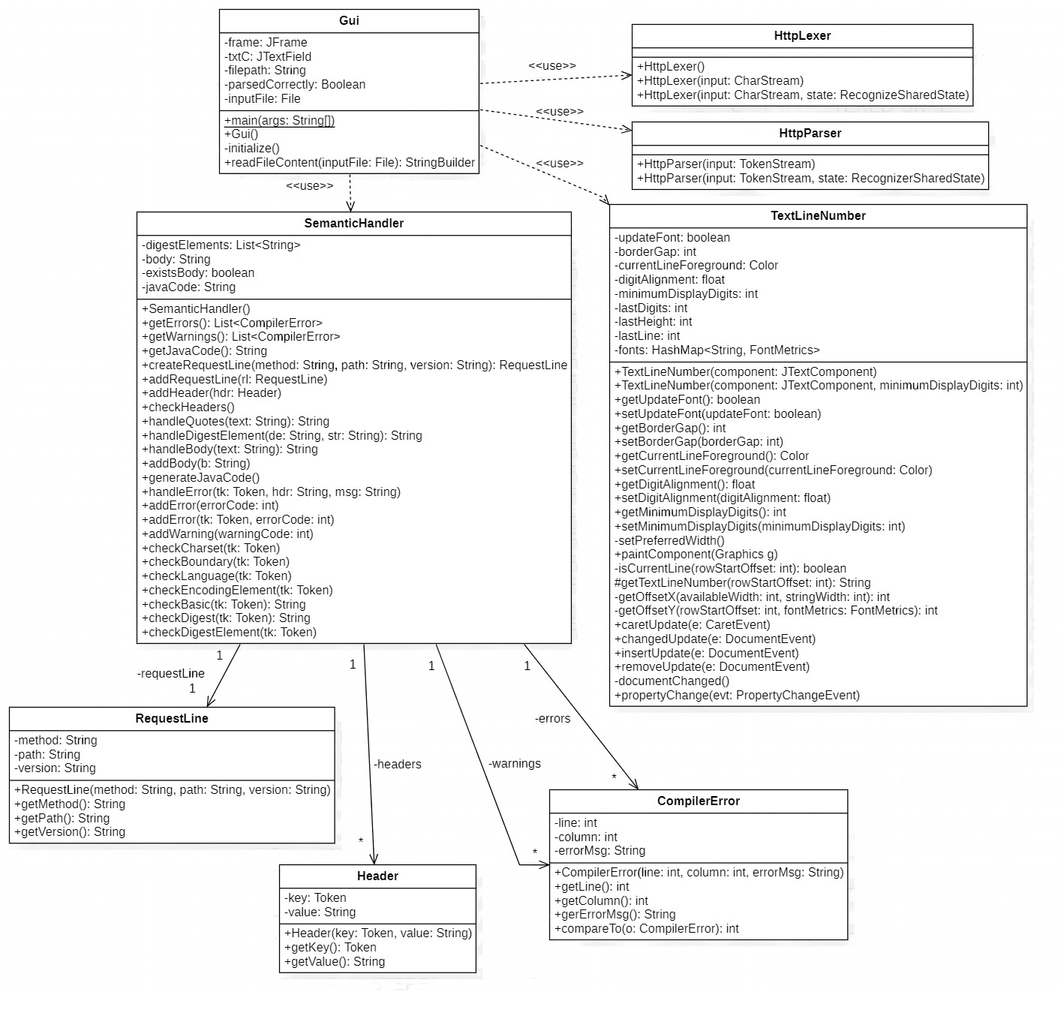


Figura 3 - Class Diagram

Di seguito viene riportato il sequence diagram che illustra la sequenza di operazioni svolte da Http2Java quando viene richiesta la traduzione di una richiesta da http a Java. Come nel caso del class diagram, l’interazione tra HttpParser e HttpLexer è semplificata e di più alto livello.

Immagine che contiene testo, diagramma, Parallelo, numero

Descrizione generata automaticamente

Figura 4 - Sequence Diagram del processo di traduzione

# Linguaggio http

Di seguito viene riportata la specifica ANTLR che definisce lessico e sintassi della grammatica delle richieste http:

grammar Http;

options {  
 language = Java;  
 k = 1;   
}

request  
 : requestLine  
 header\*  
 body?  
 EOF  
 ;

requestLine  
 : method  
 pathRule  
 HTTP\_VERSION  
 TERMINAL  
 ;

pathRule  
 : PATH QUERY?  
 | STRING  
 ;

method  
 : GET  
 | POST  
 ;

header  
 : hostRule  
 | userAgentRule  
 | contentTypeRule  
 | acceptRule  
 | cookieRule  
 | authorizationRule  
 | acceptLanguageRule  
 | acceptEncodingRule  
 | chacheControlRule  
 | maxForwardsRule  
 | genericHeaderRule  
 ;

hostRule  
 : HOST COLUMN  
 (DNS | IPV4)  
 COLUMN INT\_NUM?  
 TERMINAL  
 ;

userAgentRule  
 : USER\_AGENT COLUMN  
 productRule  
 (productRule  
 (extensionRule)\*)?  
 TERMINAL  
 ;

productRule  
 : PRODUCT  
 PRODUCT\_INFO?  
 ;

extensionRule  
 : PRODUCT  
 ;

acceptRule  
 : ACCEPT COLUMN  
 mimeList  
 TERMINAL  
 ;

mimeList  
 : mimeElement  
 (COMMA mimeElement)\*  
 ;

mimeElement  
 : MIME  
 qValueRule?   
 ;

contentTypeRule  
 : CONTENT\_TYPE COLUMN  
 (MIME charsetRule?  
 |MULTIPART\_MIME boundaryRule)  
 TERMINAL  
 ;

charsetRule  
 : SEMI\_COLUMN ID EQUALS STRING  
 ;

boundaryRule  
 : SEMI\_COLUMN ID EQUALS STRING  
 ;

cookieRule  
 : COOKIE COLUMN  
 cookieList  
 TERMINAL  
 ;

cookieList  
 : cookieElement  
 (SEMI\_COLUMN cookieElement)\*  
 ;

cookieElement  
 : STRING EQUALS STRING  
 ;

qValueRule  
 : SEMI\_COLUMN Q EQUALS Q\_VAL  
 ;

authorizationRule  
 : AUTHORIZATION COLUMN  
 ID  
 (basicAuthRule  
 | digestAuthRule)  
 TERMINAL  
 ;

basicAuthRule  
 : STRING  
 ;

digestAuthRule  
 : authRule  
 (COMMA authRule)\*  
 ;

authRule  
 : ID  
 EQUALS STRING  
 ;

acceptLanguageRule  
 : ACCEPT\_LANGUAGE COLUMN  
 languageList  
 TERMINAL  
 ;

languageList  
 : languageElement  
 (COMMA languageElement)\*   
 ;

languageElement  
 : (ID  
 |LANGUAGE\_ELEMENT  
 |STAR)  
 qValueRule?  
 ;

acceptEncodingRule  
 : ACCEPT\_ENCODING COLUMN  
 encodingList  
 TERMINAL  
 ;

encodingList  
 : encodingElement  
 (COMMA encodingElement)\*   
 ;

encodingElement  
 : (ID  
 |STAR)   
 qValueRule?  
 ;

chacheControlRule  
 : CACHE\_CONTROL COLUMN  
 STRING  
 (COMMA STRING)\*  
 TERMINAL  
 ;

maxForwardsRule  
 : MAX\_FORWARDS COLUMN  
 INT\_NUM   
 TERMINAL  
 ;

genericHeaderRule  
 : STRING COLUMN STRING  
 TERMINAL  
 ;

body  
 : BODY\_STRING  
 TERMINAL  
 ;

GET : 'GET';  
POST : 'POST' ;  
HOST : 'Host';  
USER\_AGENT : 'User-Agent';  
CONTENT\_TYPE : 'Content-Type';  
ACCEPT : 'Accept';  
COOKIE : 'Cookie';  
AUTHORIZATION : 'Authorization';  
ACCEPT\_LANGUAGE : 'Accept-Language';  
ACCEPT\_ENCODING : 'Accept-Encoding';  
CACHE\_CONTROL : 'Cache-Control';  
MAX\_FORWARDS : 'Max-Forwards';  
Q : 'q';  
EQUALS : '=';  
COMMA : ',' ;  
COLUMN : ':';  
SEMI\_COLUMN : ';';  
TERMINAL : '|';  
STAR : '\*';

ID  
 : ALPHA\_CHAR ALPHA\_CHAR+  
 ;

INT\_NUM  
 : NUM  
 ;

HTTP\_VERSION  
 : 'HTTP/' ('1.1'|'2')  
 ;

PATH  
 : '/' | (('/' PATH\_ELEMENT)+ '/'?)  
 ;

QUERY  
 : '?' QUERY\_ELEMENT ('&' QUERY\_ELEMENT)\*  
 ;

DNS  
 : DNS\_ELEMENT ('.' DNS\_ELEMENT)\* ('.' ALPHA\_CHAR ALPHA\_CHAR+)  
 ;

IPV4  
 : NUM '.' NUM '.' NUM '.' NUM  
 ;

MIME  
 : MIME\_TYPE '/' MIME\_SUBTYPE  
 | '\*/\*'  
 ;

MULTIPART\_MIME  
 : 'multipart/' MIME\_SUBTYPE  
 ;

Q\_VAL  
 : '0' '.' (('0'..'9')   
 |('0'..'9') ('0'..'9')  
 |('0'..'9') ('0'..'9') ('0'..'9'))  
 | '1.0'  
 ;

PRODUCT  
 : ALPHA\_NUM\_CHAR+ '/' VERS\_NUM  
 ;

PRODUCT\_INFO  
 : '(' (~('('|')'))\* ')'  
 ;

LANGUAGE\_ELEMENT  
 : ALPHA\_CHAR ALPHA\_CHAR ALPHA\_CHAR? ('-' ALPHA\_CHAR+)  
 ;

BODY\_STRING  
 : '#' (ESC\_SEQ | ~('\\' | '#'))\* '#'  
 ;

STRING  
 : '\'' (ESC\_SEQ | ~('\\'|'\''))\* '\''  
 ;

COMMENT  
 : '//' ~('\n'|'\r')\* '\r'? '\n' {$channel=HIDDEN;}  
 | '/\*' ( options {greedy=false;} : . )\* '\*/' {$channel=HIDDEN;}  
 ;

WS  
 : (' '  
 | '\t  
 | '\r'  
 | '\n'  
 ) {$channel=HIDDEN;}  
 ;

fragment  
VERS\_NUM

: ALPHA\_NUM\_CHAR+ ('.' ALPHA\_NUM\_CHAR+)\*

;

fragment  
DNS\_ELEMENT  
 : (ALPHA\_NUM\_CHAR | '-') (ALPHA\_NUM\_CHAR | '-')+  
 ;

fragment  
PATH\_ELEMENT  
 : (ALPHA\_NUM\_CHAR  
 | '.' | '~' | '-' | '\_'  
 | '%' HEX\_DIGIT HEX\_DIGIT)+  
 ;

fragment  
QUERY\_ELEMENT  
 : PATH\_ELEMENT '=' PATH\_ELEMENT  
 ;

fragment  
MIME\_TYPE  
 : 'application'  
 | 'audio'  
 | 'chemical'  
 | 'font'  
 | 'image'  
 | 'message'  
 | 'model'  
 | 'text'  
 | 'video'  
 ;

fragment  
MIME\_SUBTYPE  
 : (ALPHA\_NUM\_CHAR | '-')+ ('.' (ALPHA\_NUM\_CHAR | '-')+)\* ('+' ALPHA\_CHAR+)?  
 | '\*'  
 ;

fragment  
NUM  
 : '0'  
 | ('1'..'9')('0'..'9')\*  
 ;

fragment  
ALPHA\_CHAR  
 : ('a'..'z')  
 | ('A'..'Z')  
 ;

fragment  
ALPHA\_NUM\_CHAR  
 : ('0'..'9')  
 | ALPHA\_CHAR  
 ;

fragment  
ESC\_SEQ  
 : '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')  
 | UNICODE\_ESC  
 | OCTAL\_ESC  
 ;

fragment  
OCTAL\_ESC  
 : '\\' ('0'..'3') ('0'..'7') ('0'..'7')  
 | '\\' ('0'..'7') ('0'..'7')  
 | '\\' ('0'..'7')  
 ;

fragment  
UNICODE\_ESC  
 : '\\' 'u' HEX\_DIGIT HEX\_DIGIT HEX\_DIGIT HEX\_DIGIT  
 ;

fragment  
HEX\_DIGIT  
 : ('0'..'9'|'a'..'f'|'A'..'F')  
 ;

ERROR\_TOKEN  
 : .  
 ;

La grammatica decorata (ovvero contenente anche i controlli semantici) può essere consultata al seguente link: <https://github.com/PellegrinelliNico/ProgettoLFC/blob/main/Documentazione/HttpDecorata.g>.

# Implementazione analisi semantica

Le azioni semantiche sono state implementate, dove possibile, come singole istruzioni nella grammatica decorata. Dove il controllo era più complesso, è stato delegato a metodi ad-hoc della classe SemanticHandler. L’obiettivo del compilatore è quello di fornire una traduzione della richiesta http in codice java e fornire dei messaggi di errore utili nel caso questo non sia possibile. Quindi l’analisi semantica produce come risultato tre elementi, sotto forma di campi dell’istanza di SemanticHandler generata dalla classe HttpParser:

* javaCode: la stringa rappresentate la traduzione Java, vuota se ci sono errori;
* errors: una lista, anche vuota, di oggetti di tipo CompilerError rappresentanti gli errori riscontrati dal compilatore;
* warnings: una lista, anche vuota, di oggetti di tipo CompilerError rappresentati gli warning riscontrati dal compilatore.

La classe CompilerError contiene tre campi: due interi per numero di riga e numero di colonna del token che ha generato errore e la stringa riguardante il messaggio di errore.

Durante il parsing, prevalentemente mediante attributi sintetizzati, oltre alle liste errors e warnings, si popolano i seguenti campi dell’istanza di SemanticHandler:

* requestLine: oggetto di tipo RequestLine, contenente quindi tre stringhe, una per il metodo, una per il path e una per la versione della richiesta http;
* headers: lista di oggetti di tipo Header, uno per ogni dichiarazione di header, contenenti quindi un oggetto di tipo Token, ossia il token relativo al nome dell’header, e una stringa rappresentante il valore dell’header;
* digestElements: una lista di stringhe, una per ogni parameter presente nell’header “Authorization” (solo se di tipo “Digest”);
* body: una stringa rappresentante il body;
* existsBody: booleano, false se il body non è stato definito, true altrimenti.

Tutte le stringhe nominate nella lista precedente vengono ottenute manipolando le stringhe ottenute dai token (tramite il metodo getText()) in modo da rimuovere i caratteri superflui ed al fine di essere compatibili con l’environnement di java.

Al termine del parsing, come ultima azione semantica, viene chiamato il metodo del SemanticHandler generateJavaCode(), che, oltre ad aggiungere gli warnings 1. e 2. se necessario, utilizza i campi sopra descritti per costruire la stringa javaCode.

Per una dettagliata descrizione di errori e warning si veda la sezione [Errori e Warning](#_Errori_e_Warning). A livello puramente implementativo, di seguito vengono fatte delle precisazioni sull’ottenimento di alcuni di questi errori:

* ogni qual volta il parser riconosce correttamente una headerRule, viene controllato se tale header è già presente nella lista; in caso negativo, il nuovo header viene salvato nella lista headers, in caso positivo viene generato un errore. Procedimento analogo per la lista digestElement;
* l’errore 2.1 non è associato ad alcun token; quindi, nella relativa istanza di Header si usa Integer.MAX\_VALUE per numero di riga e di colonna e nel messaggio non vengono specificate le coordinate dell’errore.