

UNIVERSITA' DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica



Realizzazione di un Web Editor per Asmeta

Relatore:

Prof. Angelo Michele GARGANTINI

Tesi di Laurea Triennale

Favio Andree QUINTEROS TERRAZAS

Matricola n.1035982

ANNO ACCADEMICO 2020/2021

A tutte le persone a me care.

Prefazione

La velocissima diffusione di Internet ha portato ad un notevole sviluppo di quelli che sono i servizi forniti da esso. Basti pensare al concetto di *nuvola informatica*, o più propriamente in inglese il *cloud computing*, per l'archiviazione di dati. Con le infrastrutture di due decenni fa, avere tanta comodità nel poter memorizzare qualche gigabyte di dati su un computer remoto era impensabile.

Oggi, l'implementazione di nuove tecnologie, tra i quali fibra ottica, l'abbassamento del rapporto euro/gigabyte, l'integrazione di servizi come Google Drive e iCloud, rendono i prodotti Internet alla portata di tutti. A cascata vi è lo sviluppo della computazione a lato server. Esempi di questo tipo possono essere Word, Matlab e anche servizi di conversione di file.

Questa è la ragione principale per la quale vi è interesse nel realizzare un Web Editor per Asmeta, aver quindi la possibilità di poter scrivere del codice da qualunque computer e poter almeno vedere se quello che si è scritto è sintatticamente corretto, senza la necessità di dover installare nessun software.

Indice

Prefazione	5
Introduzione	9
1.1 Cosa è una Abstract State Machine in generale.....	9
1.2 Informazioni sull'elaborato.....	10
1.2.1 Inquadramento generale.....	11
1.2.2 Struttura dell'elaborato	11
Nozioni su Asmeta ed Xtext	13
2.1 Il framework Asmeta	13
2.2 Il linguaggio AsmetaL	14
2.3 Il framework Xtext.....	16
Fasi dell'implementazione Web	19
3.1 Semantica, sintassi e configurazione	19
3.1.1 Installazione di Xtext su Eclipse e generazione progetto	20
3.1.2 Il file .xtext	23
3.1.3 Il file .mwe2.....	24
3.1.4 Risultato intermedio.....	25
3.2 Progetto Web	26
3.2.1 Scelte progettuali	26
3.2.2 Input del testo tramite JavaScript.....	28
3.2.3 Comunicazione JSP-Servlet.....	29

3.2.4 Implementazione bottone di esecuzione.....	30
3.2.5 Il Controller ed il parsing.....	31
3.2.6 Definizione dello stile in CSS.....	39
Risultati, limiti e conclusioni	43
4.1 Presentazione dei risultati	43
4.2 Limiti del progetto	46
4.3 Difficoltà e conclusioni.....	47
Bibliografia.....	50

Capitolo 1

Introduzione

Al primo approccio, molte persone trovano difficoltà nel comprendere esattamente cosa sia un'Abstract State Machine (ASM), questo può essere principalmente dovuto al metodo di apprendimento tradizionale rivolto in generale alla “computer science”. La sfida principale delle ASM sono quelle di superare le tradizionali assunzioni sui modelli di calcolo. Molti sono infatti i loro sostenitori secondo cui un primo approccio riguardo alle nozioni di algoritmo, sarebbe più facile e naturale con le idee base degli ASM.

1.1 Cosa è una Abstract State Machine in generale

A prima vista, un'Abstract State Machine sembra solo un insieme di istruzioni con assegnamenti e condizioni. In particolare, come vedremo, esistono molte estensioni della versione base di ASM. Cosa rendono le ASM così particolari? Cosa rende le ASM il modello di calcolo più potente ed universale rispetto ai modelli standard tradizionali? Così come scritto da Yuri Gurevich, suo inventore, nel 1985?

L'idea principale degli ASM è il modo sistematico di correlare i simboli nella rappresentazione sintattica di un programma, e gli elementi del mondo reale di uno stato. Uno stato in ASM può quindi includere qualsiasi oggetto o funzione nel mondo reale. In particolare, lo stato non assume nessuna rappresentazione simbolica a livello di bit rispetto ai componenti dello stato. Sta qui la differenza rispetto ai modelli computazionali

tradizionali, quali ad esempio la Macchina di Turing in cui lo stato è strutturato in una collezione di simboli, che hanno il difetto di soffermarsi più sulla trasformazione dei dati e non su ciò che essi rappresentano.

Ora è naturale chiedersi come tutto questo sia attuabile. Sono tanti gli algoritmi che possono essere definiti in ASM, ma che non possono essere implementati, anzi, non sono proprio destinati a tale fine. Piuttosto descrivono delle procedure del mondo reale. Ci possono essere ad esempio algoritmi che descrivono il funzionamento di un ATM con un relativo prelievo di denaro dal conto corrente.

Da questo punto di vista le ASM possono essere viste come la teoria del pseudocodice, in cui vengono descritte le dinamiche per i sistemi discreti.

Per concludere questa breve introduzione, è necessario accennare quello che è la prospettiva delle ASM: quella di apportare nuovi fondamenti di informatica alla scienza, attraverso la capacità di poter dare una particolare nozione di algoritmo in cui quelli “implementabili” sono solo una sottoclasse [1].

1.2 Informazioni sull’elaborato

Nei seguenti paragrafi vi sarà una spiegazione generale dell’elaborato e successivamente una descrizione dettagliata passo a passo. Per il tipo di approccio, a parte per quanto riguardano le conclusioni, si cercherà di evitare quelle che sono state le difficoltà personali, descrivendo invece le ragioni che hanno portato a determinate decisioni.

L’elaborato è rivolto ad un pubblico con conoscenze base in ambito informatico e web. Il codice non verrà mai riportato nella sua totalità ma verranno descritte le porzioni più importanti e funzionali. È comunque possibile andare a visionare il progetto completo all’indirizzo:

https://github.com/Favio-Quinteros/Quinteros_WebEditor

1.2.1 Inquadramento generale

Prima di presentare quelli che sono gli aspetti più tecnici dell'elaborato, si cercherà di descrivere in linea generale quello che è il contesto. Viene quindi descritta la parte teorica su Asmeta ed il linguaggio AsmetaL, per poi affrontare quello che ci è di interesse, cioè il Web Editor. Qui saranno definiti gli aspetti più tecnici, descrivendo prima l'ambiente di sviluppo di contorno, Eclipse ed Xtext, per poi andare più in profondità studiando lo sviluppo Web attraverso, ad esempio, le JSP e Servlet. In particolare, su quest'ultima parte, saranno descritti anche alcuni problemi che un utente può incorrere, presentando quindi anche possibili soluzioni. Infine, le conclusioni cercheranno di descrivere il progetto con un approccio più diretto, attraverso un'analisi auto-critica ed un linguaggio più semplice.

1.2.2 Struttura dell'elaborato

La tesi è strutturata nel seguente modo:

Il secondo capitolo è dedicato ad Asmeta ed Xtext. Si cercherà di rispondere alla domanda, che cos'è Asmeta, con i suoi relativi strumenti, per arrivare ad alcuni esempi sull'Asmeta language. Successivamente verrà spiegato Xtext, quindi cosa è, e come esso viene adoperato nella realizzazione del progetto.

Nel terzo capitolo, quello più corposo, verrà affrontato il cuore del progetto Web Editor. Esso è diviso in due macroaree:

-La prima illustra l'integrazione del framework Xtext nell'ambiente di sviluppo Eclipse, specificando come generare un nuovo progetto. Successivamente si va a definire la semantica del testo sull'editor, attraverso la modifica del file .xtext. Infine si vedrà come modificare le configurazioni attraverso il file .mwe2.

-La seconda parte, invece, riguarda la parte web. Si comincia con l'illustrare quali sono i file prodotti automaticamente alla generazione del progetto e come interagiscono con il file principale dell'editor. Successivamente si discuterà su come modificare i file presenti per poter richiamare una funzione Java, e poter fare il parsing del testo in input. Si spiega quindi cosa sono le JavaServer Pages e le Servlet, necessarie per poter eseguire un metodo Java già esistente. Inoltre, verrà spiegato come prendere il testo da tastiera

attraverso una funzione in JavaScript. Per finire sarà descritta la modifica al file .css che consente di modificare lo “stile” della pagina web e quindi renderla più “gradevole” all’utente.

Il quarto capitolo chiude il disegno del progetto andando a illustrare le conclusioni e naturalmente i risultati finali. Verranno fatte anche delle considerazioni personali su quelle che sono state le difficoltà e definiti quelli che sono i limiti delle scelte effettuate con una conseguente auto-critica.

Capitolo 2

Nozioni su Asmeta ed Xtext

In questo capitolo verrà spiegato cosa sono Asmeta ed Xtext. Quest'ultimo è fondamentale per la generazione del Web Editor.

2.1 Il framework Asmeta

Asmeta è un framework, una struttura utile per la realizzazione di software, per il metodo formale delle Abstract State Machines (ASMs). Questo è basato sul metamodello per ASMs [2]. Per definizione cerca di avvicinare il modo di comprendere umano, la formulazione dei problemi nel mondo reale e lo sviluppo delle soluzioni algoritmiche. Il metamodello ha anche altre sfumature, esso può anche essere inteso come una rappresentazione astratta dei concetti ASMs, in modo da avere uno scambio di formati standard per poter integrare i suoi strumenti. Infatti, Asmeta è composto da diverse utilità e si caratterizzano per le loro attività di validazione e verifica. Questi sono:

- Asmee: un editor per ASMs scritto nel linguaggio AsmetaL.
- AsmetaLc: un compilatore/parser per i modelli AsmetaL.
- AsmetaS: un simulatore.
- AsmetaV: un validatore basato sui scenari.
- AsmetaA: un animatore di esecuzione.
- AsmetaSMV: un controllore di modelli basato su NuSMV.

- AsmetaMA: un advisor di modelli.
- AsmetaVis: un visualizzatore grafico per i modelli AsmetaL.
- AsmetaRefProver: un dimostratore sulla correttezza della raffinatezza.
- Asm2SMT: un traduttore dai modelli AsmetaL a Yices logical contexts.
- Asm2C++: un generatore di codice [3].

2.2 Il linguaggio AsmetaL

AsmetaL è un linguaggio usato per le ASMs. Esso è una derivazione del metamodello ASMs e la sua notazione testuale è usata dai modellisti per scrivere strutture in ASM nel framework Asmeta. Il linguaggio AsmetaL può essere diviso in quattro parti:

- Il linguaggio strutturale, che fornisce i costrutti necessari per descrivere la forma dei modelli ASM.
- Il linguaggio di definizione, che fornisce le dichiarazioni base come funzioni, domini, regole e assiomi caratterizzanti le specificazioni algebriche.
- Il linguaggio dei termini, che fornisce tutti i tipi di espressioni sintattiche che può essere valutato in uno stato di una ASM.
- Il linguaggio dei comportamenti oppure quello delle regole, che fornisce una notazione per specificare le regole di transizione di un ASM [4].

Un esempio di codice sul modello ASM, con una descrizione comunque non troppo dettagliata, in quanto non è scopo principale di questo elaborato, è riportato nella pagina successiva, la quale servirà anche per fare le prove sul Web Editor.

```

asm factorial

import ../STD/StandardLibrary

signature:
  dynamic monitored value: Integer
  dynamic controlled index: Integer
  dynamic controlled factorial: Integer
  dynamic controlled outMess: Any
  derived continue: Integer -> Boolean

definitions:

  function continue($i in Integer) =
    $i>1

  macro rule r_factorial =
    if(continue(index)) then
      seq
        factorial := factorial*index
        index := index-1
      endseq
    endif

  main rule r_Main =
    seq
      if(index=1) then
        if(value>0) then
          par
            index := value
            factorial := 1
            outMess := "Executing the factorial"
          endpar
        else
          outMess := "Insert a value greater than zero"
        endif
      endif
      r_factorial[]
    endseq

default init s0:
  function index = 1

```

Il modello ASM è strutturato in quattro sezioni: un header (intestazione), un body (corpo), una main rule (regola principale) e una initialization (inizializzazione). Il codice, come si può vedere, comincia con *asm factorial*, il quale deve essere uguale al nome del file dove è contenuto, che in questo caso dovrà quindi essere factorial.asm. Per quanto riguarda le quattro sezioni sono così individuate:

Header:

```

import ../STD/StandardLibrary

signature:
  dynamic monitored value: Integer
  dynamic controlled index: Integer
  dynamic controlled factorial: Integer
  dynamic controlled outMess: Any
  derived continue: Integer -> Boolean

```

Body:

```
definitions:

function continue($i in Integer) -
    $i>1

macro rule r_factorial -
    if(continue(index)) then
        seq
            factorial := factorial*index
            index := index-1
        endseq
    endif
```

Main rule:

```
main rule r_Main -
    seq
        if(index=1) then
            if(value>0) then
                par
                    index := value
                    factorial := 1
                    outMess := "Executing the factorial"
                endpar
            else
                outMess := "Insert a value greater than zero"
            endif
        endif
        r_factorial[]
    endseq
```

Initialization:

```
default init s0:
    function index = 1
```

In seguito a questo esempio, bisognerebbe andare ad approfondire il linguaggio, però come detto precedentemente non è scopo di questo elaborato andare avanti nel dettaglio, quindi per ogni approfondimento si rimanda alla bibliografia, dove questi argomenti vengono definiti in modo più esaustivo [5].

2.3 Il framework Xtext

Xtext è un framework per lo sviluppo di linguaggi di programmazione e linguaggi di uno specifico dominio. Esso fornisce un'infrastruttura per quanto riguarda il controllo sintassi, il parsing, il linker e compilatore, fornendo anche supporto per la modifica di Eclipse e Web Editor [6]. Da questa definizione, si capisce che il framework Xtext è adatto agli scopi di questo progetto. Infatti, partendo dai requisiti basilari, è necessario avere una

“casella” di testo in una pagina web su cui poter scrivere il codice. Se si scrivono determinate parole chiave per il linguaggio, nel nostro caso Asmeta, occorre evidenziare quella parola, indicando anche se la sintassi, cioè l’insieme delle parole, ha senso per il linguaggio.

È però necessario fare una precisazione: essendo già presente un plug-in Eclipse per Asmeta, per analizzare il codice si userà una funzione Java già esistente. Non vi sarà quindi l’analisi dettagliata del controllo della sintassi e semantica.

Capitolo 3

Fasi dell'implementazione Web

Questo capitolo è dedicato alla descrizione del progetto in chiave tecnica, presentando passo a passo le operazioni di rilievo effettuate per la realizzazione dello stesso e commentando ed esponendo i problemi maggiori, alcuni dei quali, per loro natura, riescono a incidere in modo rilevante sulle tempistiche. Certe precisazioni sembreranno forse inutili e anche banali, però capita non raramente che i processi in apparenza facili creino delle difficoltà successivamente. Problemi che in alcuni casi sono stati risolti, adottando “soluzioni” drastiche, in quanto non soddisfano i requisiti in tutti i casi ma che comunque coprono in modo soddisfacente gli usi a cui un Web Editor è destinato.

3.1 Semantica, sintassi e configurazione

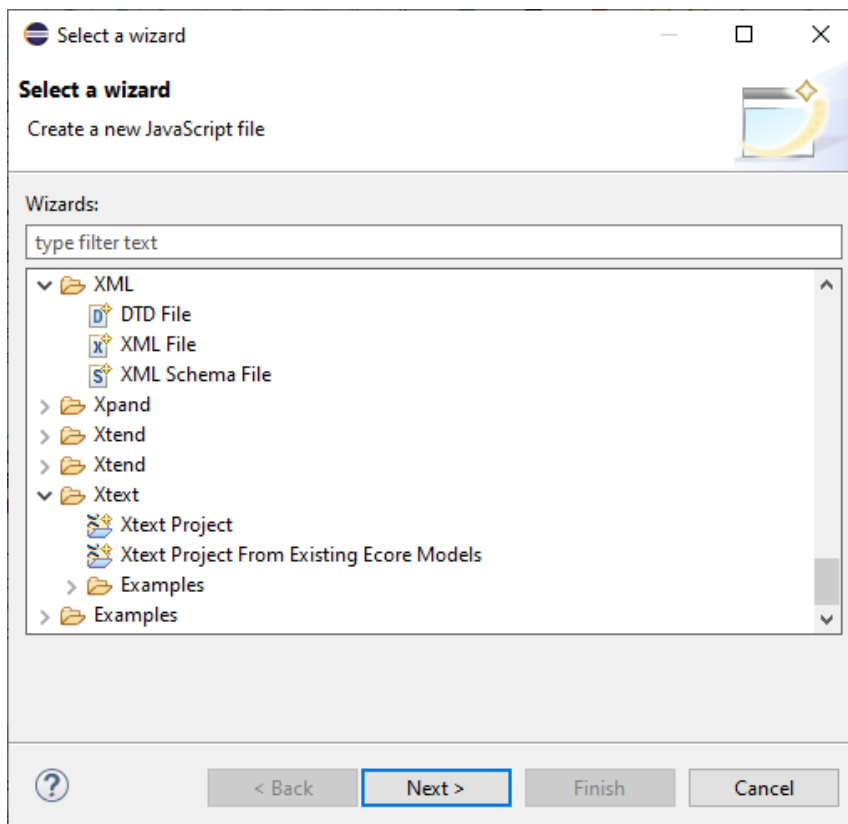
In questo paragrafo verrà trattata l'inizializzazione del progetto e su come adattare i file del plug-in Eclipse, già esistente, per riuscire ad avere un Web Editor che possa evidenziare le parole chiave del linguaggio Asmeta, come ad esempio *asm* oppure *function*, e di segnalare eventuali errori di semantica.

3.1.1 Installazione di Xtext su Eclipse e generazione progetto

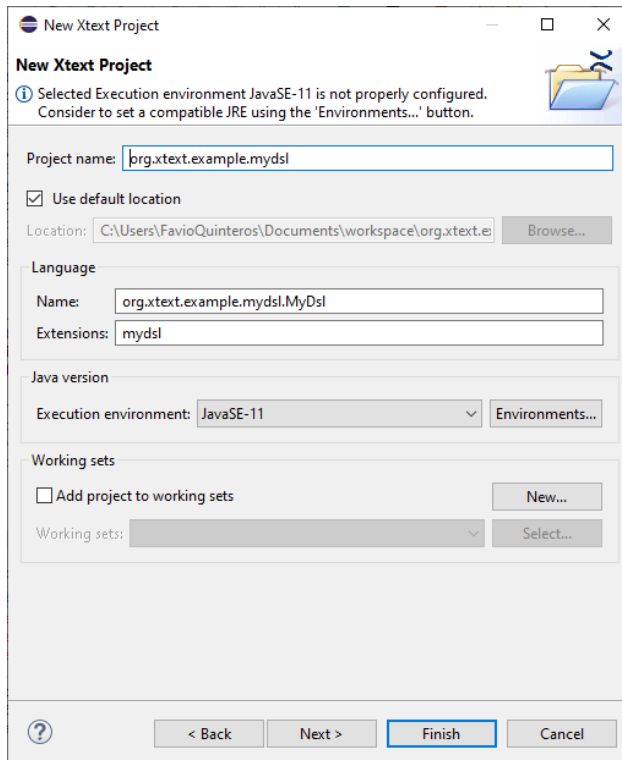
Dopo la breve presentazione su Xtext, viene qui illustrato l'installazione di essa su Eclipse, e la creazione del progetto su cui si andrà ad agire. Prerequisito fondamentale è, naturalmente, disporre di Eclipse for Java Developer. Nel caso di problemi all'avvio di Eclipse, le soluzioni di solito possono essere di due tipi: o manca l'installazione di Java sul proprio computer (infatti Eclipse "usa" la Java Virtual Machine), oppure bisognerà modificare, in base all'errore sorto, un file di configurazione di Eclipse che quindi va a settare le impostazioni di lancio del programma.

Per l'installazione di Xtext, è sufficiente andare nella sua pagina di download, scegliere una versione tra quelle disponibili, copiando l'URL che lo indirizza. Sarà poi quell'URL che permetterà l'avvio dell'installazione. Infatti, su Eclipse bisognerà andare nella sezione *Help>Install New Software>*incollare URL, precedentemente copiato, quindi seguire le indicazioni proposte.

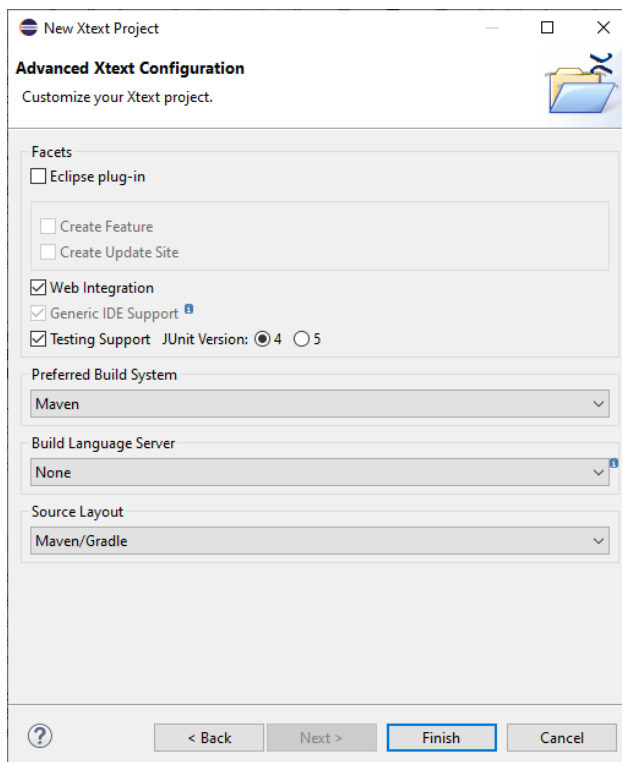
Una volta che si ha a disposizione Xtext su Eclipse, bisogna andare a creare un nuovo progetto, in modo simile a come si crea un progetto Java, quindi *File>New>Other*:



Da cui si seleziona *Xtext Project* > *Next*. Ciò consentirà di passare alla pagina seguente:

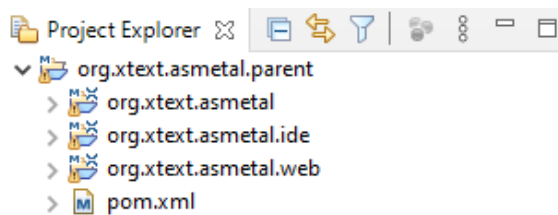


Da qui è possibile modificare il nome del progetto e anche il nome dell'estensione, cioè il nome del linguaggio che vogliamo implementare. Una volta eseguite le opportune modifiche, bisogna andare avanti con *Next* per arrivare al seguente step:



Qui possiamo indicare cosa vogliamo includere nel nostro progetto. Con l'interesse rivolto verso l'ambito web, è necessario selezionare solamente la casella con *Web Integration*. Per quanto riguarda il *Preferred Build System*, è stato scelto Maven il quale permette di semplificare il processo di costruzione. Tramite Maven (che fa parte della Apache Foundation) è possibile compilare il codice sorgente, fare il packaging dei codici compilati in file JAR ed installarli in una repository locale.

L'ultimo passo è *Finish*. Successivamente si vengono a generare le seguenti cartelle:



Prima di continuare con il progetto e su come farlo partire, è doveroso spiegare prima cosa è *localhost:8080*.

Localhost:8080 nei sistemi Windows è fondamentale quando si tratta di dover lanciare un programma lato server e riuscire in questo modo a fare dei test sul funzionamento. Spiegato in maniera più pratica, *localhost*, è più semplicemente un indirizzo locale, per cui ogni qual volta che sul browser si va su tale indirizzo, il computer “chiama” sé stesso, facendo sia da client che da server allo stesso tempo. Questo permette di non dover spendere risorse inutilmente e, soprattutto, quando non si dispone, ad esempio, di un dominio internet.

Fatta la precisazione, è possibile andare avanti sul progetto che, per adesso, è quello generato automaticamente dal Wizard di Xtext. Questo, come si vedrà successivamente, si tratta del classico programma di prova “Hello World”.

Per visualizzare l'editor già generato automaticamente, è necessario seguire i seguenti passaggi che, naturalmente, valgono per ogni tipo di progetto del genere:

- fare click destro sulla cartella .parent>*Run As>Maven install*
- fare click destro sulla cartella .web>*Maven build....* e su *Goals* digitare *jetty:run* ed infine cliccare su *Run*.
- andare su un qualsiasi browser web e digitare *localhost:8080*

Per comodità e brevità definiremo questi passaggi appena illustrati come *eseguire il progetto*.

Dopo l'esecuzione del progetto, il risultato è il seguente:



Si può notare come *Hello* viene evidenziato e non viene rilevato nessun errore di sintassi.

È possibile che durante la fase di *Maven install* si riporti un errore di “test failure”. Per risolvere tale problema è necessario andare ad agire sul file *pom.xml* contenuta nella cartella *.parent* aggiungendo la seguente porzione di codice, che permette di ignorare l'errore prima citato e quindi andare avanti:

```
<configuration>
  <testFailureIgnore>true</testFailureIgnore>
  <useSystemClassLoader>false</useSystemClassLoader>
</configuration>
```

Ora che è stata generata la struttura base del nostro progetto, possiamo andare a modificarlo per definire il linguaggio Asmeta.

3.1.2 Il file *.xtext*

Il file *xtext* è presente nella cartella principale, cioè quella senza l'estensione *.ide* e senza *.web*. All'interno di tale cartella, si trova la directory *main*, contenente il file di interesse. Il file *.xtext* è importante in quanto definisce la sintassi e semantica del linguaggio. Grazie a questo possiamo definire quali sono le parole chiave che devono essere messe in evidenza e quali sono le regole da seguire per avere una “frase” che per il linguaggio Asmeta abbia senso.

Come è già stato accennato, la struttura del codice .xtext per Asmeta era già esistente per via di un progetto plug-in per Eclipse. Quindi, a parte l'intestazione contenente il percorso della cartella su cui è contenuto, è stato sufficiente copiare il resto del file.

3.1.3 Il file .mwe2

Partendo da dove questo è allocato, come in precedenza, il file .mwe2 ha lo stesso percorso del file .xtext. Il campo di azione del file è quello relativo alle configurazioni del progetto. Più esattamente, è un generatore che può essere configurato esternamente in modo dichiarativo. L'utente, a questo punto, può descrivere la composizione di oggetti in modo arbitrario tramite una sintassi semplice e concisa che permette di dichiarare istanze di oggetti, valori di attributi e riferimenti [7].

Di seguito viene qui riportato una porzione di codice che risulta fondamentale per questo progetto:

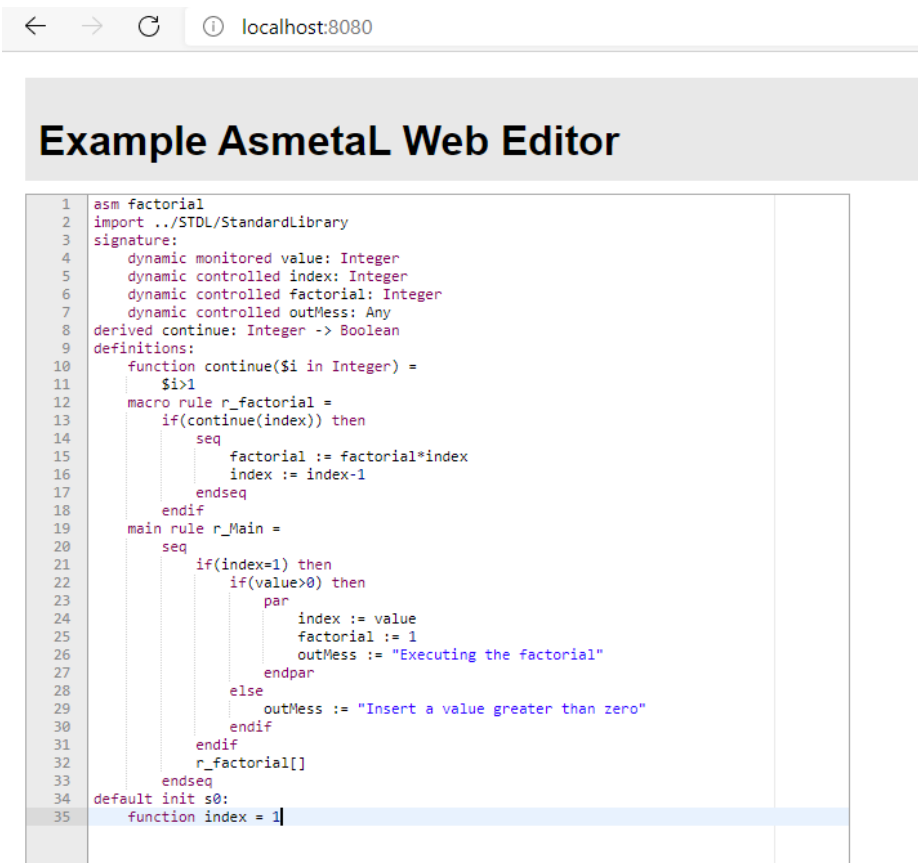
```
component = XtextGenerator {
  configuration = {
    project = StandardProjectConfig {
      baseName = "org.xtext.asmetal"
      rootPath = rootPath
      runtimeTest = {
        enabled = true
      }
      web = {
        enabled = true
      }
      mavenLayout = true
    }
    code = {
      encoding = "windows-1252"
      lineDelimiter = "\r\n"
      fileHeader = "/*\n * generated by Xtext \${version}\n */"
    }
  }
}
```

Dalla parola chiave *component* risulta intuibile che si andrà ad agire su quelli che sono i componenti da generare. Successivamente vi sono le configurazioni di percorso e anche quelle di encoding ma, per questo progetto, è fondamentale abilitare la parte web, quindi

`web={enabled =true}`. Senza questa istruzione, infatti, verrebbe generato un errore nel momento in cui si va a fare un *Maven install*.

3.1.4 Risultato intermedio

Dopo aver fatto le opportune modifiche ai file `.xtext` e `.mwe2`, il progetto contiene ora tutto ciò di cui l'editor ha bisogno per poter definire la semantica e la sintassi del linguaggio Asmeta. Infatti, mandando in esecuzione il progetto, viene visualizzato l'editor. Copiando ed incollando un esempio di codice in linguaggio Asmeta nell'editor, otteniamo il seguente risultato:



The screenshot shows a web browser window with the address bar displaying 'localhost:8080'. The main content area has a title 'Example AsmetaL Web Editor' and a code editor with the following Asmeta code:

```
1 asm factorial
2 import ../STD/StandardLibrary
3 signature:
4   dynamic monitored value: Integer
5   dynamic controlled index: Integer
6   dynamic controlled factorial: Integer
7   dynamic controlled outMess: Any
8   derived continue: Integer -> Boolean
9 definitions:
10  function continue($i in Integer) =
11    $i>1
12  macro rule r_factorial =
13    if(continue(index)) then
14      seq
15        factorial := factorial*index
16        index := index-1
17      endseq
18    endif
19  main rule r_Main =
20    seq
21      if(index=1) then
22        if(value>0) then
23          par
24            index := value
25            factorial := 1
26            outMess := "Executing the factorial"
27          endpar
28        else
29          outMess := "Insert a value greater than zero"
30        endif
31      endif
32      r_factorial[]
33    endseq
34  default init s0:
35    function index = 1
```

Come si può vedere, vengono riconosciute le parole chiave e non viene segnalato nessun errore né per quanto riguarda la semantica né per la sintassi. Se si dovesse scrivere una istruzione che non ha senso per Asmeta, verrebbe segnalata una X vicino ai numeri che indicano il sequenziale delle righe.

3.2 Progetto Web

In questo paragrafo viene presentato quello che è il cuore del progetto. Si comincia con la spiegazione del perché sono state effettuate certe scelte tecnologiche, con le loro relative spiegazioni, illustrando anche quali sono i problemi sorti e le relative soluzioni.

3.2.1 Scelte progettuali

Prima della discussione riguardo al progetto, è necessario spiegare cosa significa fare il parsing di un codice. Questo termine indica il processo di analisi della sintassi per quanto riguarda la struttura dell'intero codice quindi, ad esempio, per quanto riguarda Asmeta, la prima riga deve sempre essere *asm* seguita dal nome del file in cui è contenuto il codice. In parole più semplici, il parser non si limita a fare un controllo sintattico riga per riga ma esegue un controllo per quanto riguarda l'interezza del codice.

Riprendendo l'analisi del progetto, ora che si ha un editor web di base, c'è bisogno di implementare alcune funzionalità, come il *parsing* per l'appunto. Dando un'occhiata nel file system del progetto con estensione *.web* è possibile notare un file di nome *index.html* che si occupa della visualizzazione dell'editor. Essendo un file con nome *index*, si tratta del file che viene automaticamente caricato quando viene composto l'URL *localhost:8080* nel web browser.

Nel plug-in Eclipse di Asmeta si trova già presente una funzione Java che fa da parser al progetto. Il relatore del sottoscritto, assieme ai suoi collaboratori, hanno suggerito di implementare una JavaServer Pages (JSP). Vien naturale chiedersi su quali siano i motivi per cui si è adottato tale scelta.

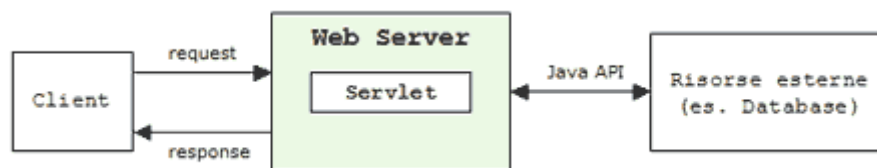
Le JSP sono una tecnologia sviluppata da Oracle e provvede a fornire servizi semplificati per la creazione di pagine web dinamiche, indipendentemente dalla piattaforma usata. In parole povere possiamo anche definire le JavaServer Pages come un html arricchito di funzionalità, essendo di base molto simili. Il vero vantaggio delle JSP, oltre alla sua indipendenza, è però quella di permettere la separazione tra la parte di presentazione e quella di controllo. Tralasciando il fatto che nel progetto non è presente

una logica di accesso ai dati, quindi ad esempio un data-base, si può definire che tramite le JSP si rispetta il pattern Model-View-Controller. Questo, in ambito informatico, soprattutto per quanto riguarda l'ingegneria del software, è fondamentale in quanto, per grandi progetti, è necessario dividere i compiti tra i vari programmatori specializzati in vari ambiti.

Ora, è stata spiegata l'importanza di separare la presentazione dal controllore, ma non è stato spiegato come questo veramente accada nel progetto. Per questo motivo è necessario introdurre il concetto di Servlet.

Spiegato in modo semplice, la Servlet è un codice in Java che risiede sul server in grado di gestire richieste dai client. Questi sono tipicamente collocati all'interno di Application Server, come ad esempio Tomcat.

Le Servlet comunicano con il client attraverso il protocollo http. Risulta d'interesse il seguente schema che illustra la comunicazione tra client e Servlet:



Il server al momento di una richiesta (request) da parte del client di una Servlet, se è la prima volta, istanzia la Servlet ed avvia un nuovo thread per gestire la comunicazione. Mentre nel caso non sia la prima richiesta, allora la non serve ricaricare la Servlet e quindi per ogni nuovo client viene generato un nuovo thread. Successivamente il server invia alla Servlet la richiesta inviata dal client, così che possa elaborare la risposta ed inviarla al server. Infine, il server invia la risposta al client [8].

Ora che dovrebbe essere chiaro in linea generale cosa è una JSP e cosa una Servlet, è possibile definire i ruoli che giocano queste due tecnologie nel progetto. Infatti, come si è visto, le JavaServer Pages sono essenzialmente delle pagine html che permettono una integrazione di codice Java. Esse si occupano della presentazione, di ciò che l'utente vede, mentre, le Servlet, sono il codice Java che si occupano dell'elaborazione quindi la parte di controllo. L'idea progettuale è dunque quella di far comunicare queste due tecnologie, separando la View dal Controller anche se, in realtà per essere precisi, al momento della compilazione le JSP vengono trasformate in Servlet.

3.2.2 Input del testo tramite JavaScript

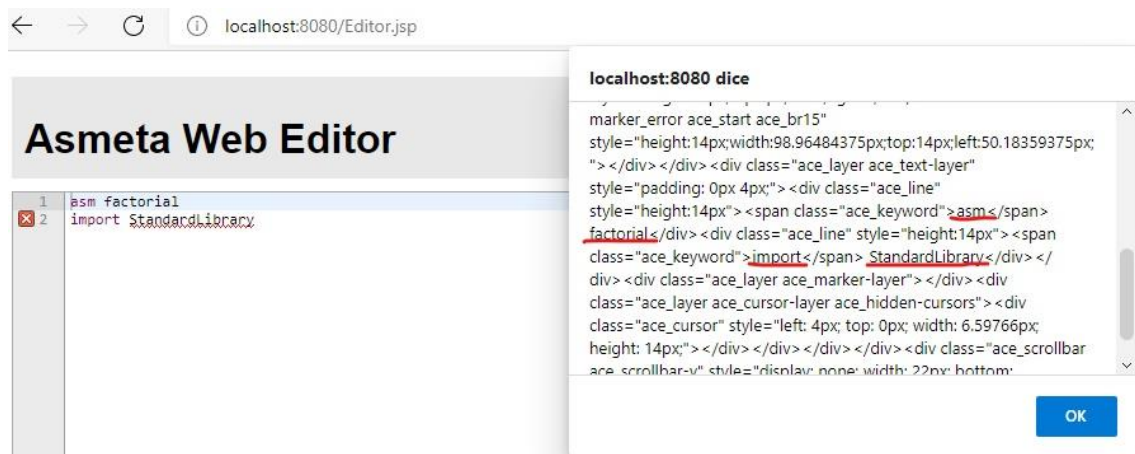
JavaScript è un linguaggio di programmazione per lo sviluppo delle web applications. È un linguaggio di scripting lato client usato per rendere interattive le pagine, tra quelli più usati per la programmazione front-end. Essendo lato client, la correttezza del codice dipende dal browser web usato, siccome alcuni metodi non sono presenti in tutti i tipi di applicativi per la navigazione Internet. Chiarito il concetto di JavaScript, prima di andare avanti con la sua implementazione nel progetto, occorre specificare che, essendo stata scelta la tecnologia delle JSP, il contenuto di *index.html* (che conteneva la presentazione del Web Editor) è stato “trasferito” in un nuovo file di nome *Editor.jsp*. Il contenuto di *index.html* è stato invece cancellato e sostituito con un comando di re-direct alla pagina JSP. In questo modo, digitando *localhost:8080* sul browser web, si viene re-indirizzati sulla pagina *localhost:8080/Editor.jsp*.

Purtroppo, non è stato possibile modificare da *Editor.jsp* ad *index.jsp*, in quanto ogni volta che si andava a fare un *Maven install*, si andava a generare un nuovo *index.html* che non permetteva di aprire direttamente *index.jsp*.

Nell’ambito del progetto, JavaScript che si può definire all’interno di *Editor.jsp*, è stato usato per ricavare l’input testuale che l’utente digita all’interno dell’editor. Il metodo usato a tale fine è il seguente:

```
var cod = document.getElementById("xtext-editor").innerHTML;
```

Che permette di ricavare per la variabile *cod* il seguente testo:



Il valore di *cod* è quello a destra che, come si vede, è pieno di codice html con all’interno però il testo in input (sottolineato in rosso). È stato dunque necessario creare una funzione

in JavaScript capace di “filtrare” la variabile *cod* in modo da ottenere soltanto il testo, necessario per l’elaborazione. Essendo abbastanza corposo il codice non verrà qui riportato. Si ricorda che il codice completo è accessibile pubblicamente su GitHub all’indirizzo: https://github.com/Favio-Quinteros/Quinteros_WebEditor.

Altra funzione importante da citare, importante per il progetto, è quella di lettura dei cookies. L’argomento dei cookies verrà spiegato meglio più avanti. Essendo però la loro lettura codificata in JavaScript, è giusto comunque illustrare il metodo capace per recuperare i cookies, quindi informazioni. Il codice è il seguente:

```
function getCookie(name) {  
    const value = `; ${document.cookie}`;  
    const parts = value.split(`; ${name}=`);  
    if (parts.length === 2){  
        var cookiesString = parts.pop().split(';').shift();  
        var counter = 0;  
        var risultato = "";  
        while (counter < cookiesString.length){  
            if(cookiesString.charAt(counter) === "+") risultato = risultato + " ";  
            else risultato = risultato + cookiesString.charAt(counter);  
            counter ++;  
        }  
        return decodeURIComponent(risultato);  
    }  
    else return "";  
}
```

Tramite la prima riga è possibile ricavare, in modo “grezzo”, il valore dei cookies. Successivamente, in base al nome, si prende solo il cookie di interesse, per poi finire con un ciclo che converte i caratteri “+” in spazi “ ”, indispensabile per il fatto che i cookies non supportano certi caratteri, tra cui gli spazi vuoti, quindi è necessario fare un encoding (codifica) prima di salvarli, per poi “decodificarli” quando li si legge.

3.2.3 Comunicazione JSP-Servlet

Prima è stata illustrata la funzione con cui ottenere il testo di input dall’editor. Si sa anche che quest’ultima è adesso contenuto dentro il file *Editor.jsp*. Ora serve inviare il testo

ottenuto con JavaScript alla Servlet che si occuperà di elaborarlo, facendo il parsing. A questo scopo sono utili le seguenti istruzioni in JavaScript, il quale assegna *tes* (che contiene il testo in input) a *key*:

```
document.formname.key.value = tes;  
document.formname.submit();
```

Insieme al seguente codice da implementare nella JSP, che invia a sua volta *key* alla Servlet:

```
<!-- parte che collega alla servlet -->  
<form name="formname" action="EditorServlet" method="post">  
  <input type="hidden" name="key">  
</form>
```

La Servlet per ottenere il valore del testo, che ora si trova in *key*, dovrà eseguire la seguente istruzione:

```
String text = request.getParameter("key");
```

In questo modo la variabile *text* conterrà il testo dell'editor.

Riassumendo, quando l'utente digita l'indirizzo *localhost:8080*, viene caricato automaticamente il file *index.html*, e questo contiene un reindirizzamento a *localhost:8080/Editor.jsp*. Questa JavaServer Pages "collegata" ad un file JavaScript, avente le utilità necessarie al progetto, contiene la visualizzazione dell'editor. Le utilità prima citate sono quelle di lettura del testo in ingresso e dei cookie. Manca quindi un pulsante da implementare nella JSP che, una volta premuto, mandi in esecuzione la funzione JavaScript che, a sua volta attraverso la *form*, manderà in esecuzione il metodo *doPost()* della Servlet.

3.2.4 Implementazione bottone di esecuzione

Come è stato accennato prima, manca un pulsante che mandi in esecuzione la Servlet una volta che l'utente abbia composto il codice Asmeta nell'editor. A questo scopo è

necessario modificare la JSP ed implementare, oltre al pulsante, anche una console dove poter visualizzare i risultati:

```
<div class="container">
  <div class="header">
    <h1>Asmeta Web Editor</h1> <!-- titolo -->
  </div>
  <!-- parte dell'editor -->
  <div class="content"><div id="xtext-editor" data-editor-xtext-
lang="asm">
  <span id="myText"></span></div></div>
</div>
<div id="titolo_console"><h1>Console:</h1></div>
<!-- box della console-->
<div id="secontainer"><h5><span id="tempo"></span>
  <span id="out" style="font-family:Courier"></span></h5></div>

<!-- bottone -->
<div class="run">
  <button onclick="esegui()"></button>
</div>
```

La prima parte di codice riguarda la presentazione dell'editor. Il punto di maggior interesse è l'aggiunta di un secondo riquadro, necessario per contenere i risultati del parsing, eseguita attraverso l'istruzione `<div id="secontainer">`. Il bottone è dall'istruzione `<button>`. Per mandare in esecuzione, la funzione JavaScript, è necessario aggiungere all'interno di `<button>`: `onclick="esegui()"`. In questo modo, quando l'utente preme il bottone, viene eseguita la funzione `esegui()`, che farà la lettura del testo all'interno dell'editor per poi mandarlo alla Servlet. Ottenuto il testo da parte della Servlet, manca la parte dell'elaborazione.

3.2.5 Il Controller ed il parsing

Come accennato prima, la Servlet legge i parametri passati attraverso `getParameter()`. Il passaggio del controllo avviene però attraverso la `<form>` contenuta nella JSP, tramite il quale è possibile scegliere quale metodo della Servlet chiamare. Le scelte sono due: invocare il `doGet()`, usato principalmente quando la quantità di parametri da inviare non è elevata (passati solitamente tramite URL), oppure invocare il `doPost()` che, invece, è

usato per quantità di dati più sostanziosi. In questo caso tramite *method="post"* è stato scelto di invocare il metodo *doPost()* della Servlet, per non essere troppo vincolati.

Ottenuti i parametri dal controller, è possibile vedere la struttura della Servlet. Affinché sia possibile raggiungerla attraverso la *form*, è necessario inserire all'interno di quest'ultima *action="EditorServlet"*. Per la Servlet si dovrà implementare la seguente istruzione, subito dopo aver definito gli opportuni *import*:

```
@WebServlet(name = "EditorServlet", urlPatterns = {"/EditorServlet"})
```

In questo modo il collegamento avviene tramite il nome. Per poter usare tale istruzione è, però, necessario aggiungere un'importazione all'inizio tramite: *import javax.servlet.annotation.WebServlet;*

Importante accennare che quando si crea una classe Java che faccia da Servlet, è che essa deve estendere *HttpServlet*, essendo il protocollo di comunicazione implementato tramite *http*. È necessario quindi, aggiungere all'inizio gli import corretti, come ad esempio *import javax.servlet.http.HttpServletRequest*, assieme ad altri che non saranno riportati su questo elaborato.

Spiegati gli step fondamentali per il passaggio dei parametri, è possibile ora implementare il parsing al controller attraverso una funzione pre-esistente. Per importare tale funzionalità è necessario prima scaricare il pacchetto *AsmetaS.jar* per poi andare nella cartella *.web* principale del progetto, fare tasto destro su di esso >*Build Path*>*Configure Build Path*>*Libraries*>*Classpath*>*Add External JARs*, e selezionare *AsmetaS.jar*. Successivamente, occorre fare la seguente importazione nel file Java: *import org.asmeta.parser.*;* . Anche il file *pom.xml* della parte *.web* deve essere modificato aggiungendo la dipendenza, attraverso il seguente codice:

```
<dependency>
  <groupId>AsmetaS</groupId>
  <artifactId>AsmetaS</artifactId>
  <scope>system</scope>
  <version>1.0</version>
  <systemPath>${basedir}\src\lib\AsmetaS.jar</systemPath>
</dependency>
```


I metodi importati, interessanti per il progetto, sono:

-*ASMParser.setUpReadAsm(file)* il quale permette di fare il parsing del file in ingresso, restituendo un oggetto *asmeta.AsmCollection*.

-*ASMParser.getResultLogger()* il quale permette di capire se ci sono errori. Restituisce un oggetto di tipo *ParseResultLogger*, che avrà una dimensione nulla nel caso in cui non vi siano errori nel parsing.

-*e.getMessage()* in cui *e* è l'eccezione che viene invocata nel caso di errore nel parsing. In questo modo è possibile prendere gli errori generati dal parser.

Analisi della porzione di codice riguardante il parsing:

```
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    saveStandardLibrary();
    //
    String parseResult = "";
    String text = request.getParameter("key");
    if (text == null) {
        parseResult = "empty asm";
    } else {
        // estrai il nome della asm
        Matcher m = p.matcher(text);
        if (m.find()) {
            String asmName = m.group(0).substring(4).trim();
            //
            File codice = new File(asmName + ".asm");
            if (codice.exists()) {
                codice.delete();
            }
            // save to a new file the asm
            codice.createNewFile();
            PrintWriter scrivi = new PrintWriter(codice);
            scrivi.print(text);
            scrivi.close();
            // call the parser
            try {
                @SuppressWarnings("unused")
                asmata.AsmCollection asms =
                    ASMParser.setUpReadAsm(codice);
                ParserResultLogger resultLogger =
                    ASMParser.getResultLogger();
                if (resultLogger.errors.size() == 0)
                    parseResult = "No errors found. Parsing
                    successful.";
            } catch (Exception e) {
                parseResult = e.getMessage();
            }
        } else {
            parseResult = "Must declare asm <name>.";
        }
    }
}
```

In ordine si può osservare che:

- saveStandardLibrary() è una funzione statica della stessa classe che si occupa di verificare che sia presente nel progetto la StandardLibrary.asm. Se questa è già

presente non succede nulla. Mentre se manca la recupera copiandola da una specifica cartella con denominazione “STDLD”.

-La stringa *text* contiene il valore di testo inserito nell’editor.

-Se *text* è vuota, si segnala subito che il file asm è vuoto, quindi è inutile proseguire con la funzione di parsing.

-Se *text* non è vuoto, attraverso *Matcher m* e la successiva condizione *if*, si vuole capire se è presente la dicitura *asm <nome>*.

-Se lo è, allora seleziona tale nome che andrà a definire il file .asm.

-Se non viene trovata nessuna corrispondenza, allora si scrive su *parseResult* la frase “*must declare asm <name>*”.

-Riprendendo il caso di una corretta dichiarazione dell’asm, prima di creare il file .asm si controlla se non vi sia uno uguale con lo stesso nome. Se presente, lo si cancella e si va avanti creando un nuovo file, scrivendoci la stringa *text*.

-Ottenuto il file .asm, si procede col fare il parsing di tale file, attraverso *ASMParser.setUpReadAsm(file)* e *ASMParser.getResultLogger()* che sono stati spiegati in precedenza. Nel caso di un parsing senza errori, si assegna alla stringa *parseResult* il testo “*No errors found. Parsing successful*”. In caso contrario, durante l’esecuzione del parsing, verrà sollevata un’eccezione che scriverà in *parseResult* la segnalazione degli errori.

A questo punto, *parseResult* contiene il risultato del parsing, segnalando: se non vi sono errori o, in caso contrario, segnalarne il tipo.

Continuando si dovrebbe ritornare alla presentazione. Per tale fine l’idea in origine è stata quella di ritornare il controllo alla JSP, mandandogli il valore di *parseResult*. Questo attraverso l’implementazione della funzione nella Servlet di:

```
request.setAttribute(“message”,parseResult)
```

e di:

```
request.getRequestDispatcher(“/Editor.jsp”).forward(request,response)
```

ed anche di:

```
${message}
```

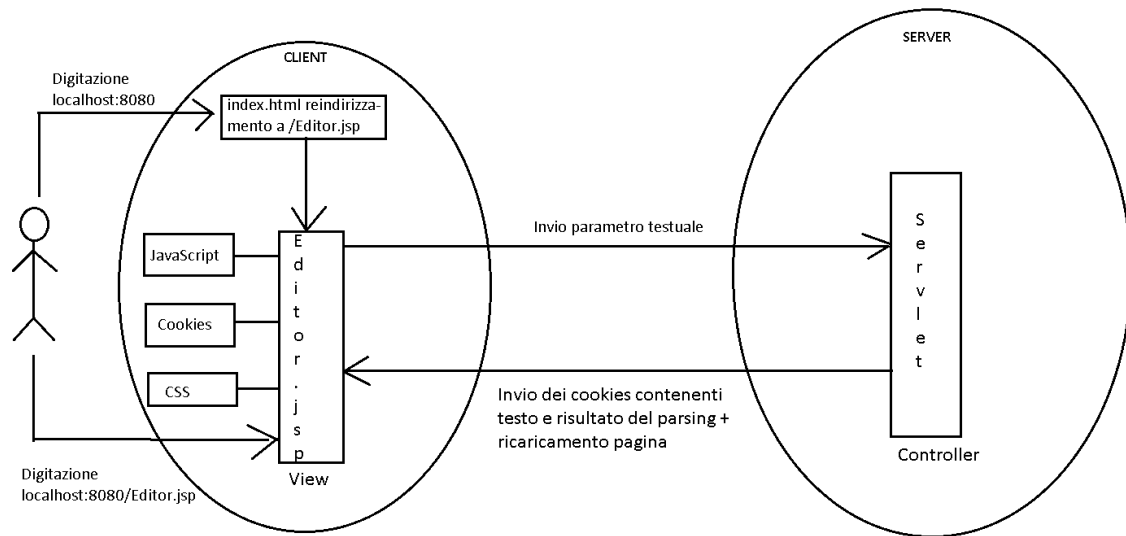
quest’ultima da aggiungere nella JSP.

Con queste istruzioni si dovrebbe passare il valore di *parseResult* alla presentazione, ritornando, attraverso *getServletContext()*, il controllo alla JavaServer Pages. Quello che accade invece è, sì, il passaggio del parametro alla presentazione, viene quindi visualizzato il risultato, però la JSP non riprende la sua esecuzione, dunque l'editor non viene più reso attivo, non più modificabile. Per un Web Editor il fatto che, dopo aver avviato tramite bottone il parsing, esso non sia più modificabile, è naturalmente non accettabile per i requisiti intrinseci. Si può pensare che questo succede perché non viene definito un "mapping" tra JSP e Servlet, vale a dire, in parole semplici, che non vi è una stretta dipendenza tra queste due. Per implementare tale dipendenza, sarebbe necessario introdurre un file *web.xml* che definisca le relazioni dei file presenti nel progetto. In quest'ultimo è già presente un file simile nel nome, si tratta del *pom.xml*, ma internamente è molto diverso, in quanto specifico per l'ambiente Maven. Dato questo problema, è stato preferito andare a ricercare delle alternative che permettessero la modifica del Web Editor, anche dopo il comando di esecuzione del parsing.

Effettuate delle ricerche, le alternative prese in considerazione sono due: uso dei *frameset* oppure dei cookies. I primi sono delle pagine indipendenti, con la possibilità di essere unite nella visualizzazione. Il problema in questo caso riguarderebbe la comunicazione tra le pagine. I secondi, cioè i cookies, sono delle informazioni che fisicamente risiedono nella memoria del client. Ogni qual volta il server ha bisogno dei cookies, esso invia una richiesta al client, che risponde inviandogli i dati richiesti. Il server a sua volta può inviare al client nuovi cookies da memorizzare. Questo è particolarmente utile in quanto il protocollo *http* è privo di stato. Tali caratteristiche hanno spinto all'uso dei cookies, che comporta altri vantaggi che saranno presentati nelle prossime righe.

Implementati nel progetto, i cookies permettono di salvare il testo ed il risultato del parsing sul client. Ogni volta che si carica la pagina, è possibile recuperare questa informazione per visualizzarlo. Se prima, ogni volta che si chiudeva la pagina, il testo sull'editor veniva perso, ora il testo viene prima recuperato dai cookies all'apertura (rispetto all'ultima esecuzione della Servlet) per poi essere visualizzato.

Per chiarire il concetto di funzionamento del progetto, vi è il seguente schema:



Il modello sopra illustrato rende possibile una visualizzazione migliore del funzionamento in generale del progetto, con gli scambi dei parametri e dove sono allocati i componenti. Non è uno schema dettagliato che copre esattamente tutto il funzionamento ma descrive semplicemente la meccanica di passaggio del controllo.

Descritto il funzionamento generale, manca definire esattamente:

- per quanto riguarda il client: un meccanismo che al caricamento della pagina, vada a recuperare i cookies, leggendoli e caricandoli nella vista della pagina, sia per quanto riguarda il testo che il risultato dell'ultimo parsing.
- per quanto riguarda il server: ogni volta che viene eseguito il codice Java, dopo aver fatto le opportune elaborazioni, venga inviato al client il testo mandatogli in precedenza e il risultato del parsing.

Prima di iniziare ad analizzare una porzione di codice del client, è più utile innanzitutto vedere quello della Servlet:

```
@SuppressWarnings("deprecation")
String cod = URLEncoder.encode(text);
Cookie cookie = new Cookie("cookie",cod);
cookie.setMaxAge(Timeout_cookie);
response.addCookie(cookie);
System.out.println(parseResult);
@SuppressWarnings("deprecation")
String cod2 = URLEncoder.encode(parseResult);
Cookie cookie2 = new Cookie("cookie2",cod2);
cookie2.setMaxAge(Timeout_cookie);
response.addCookie(cookie2);
response.sendRedirect (URLjsp);
```

Alla prima istruzione utile vi è un comando *URLEncoder.encode()* che serve per fare un encoding alla stringa che contiene il testo. Questo è utile farlo perché i cookies non supportano certi caratteri, tra cui anche gli spazi vuoti (“ ”), è quindi necessario trasformarli in una serie di digit special. Successivamente si crea un cookie e gli si attribuisce il valore del testo codificato, si setta il suo tempo massimo di vita, con un intero in secondi, per poi aggiungerlo alla risposta, che viene quindi inviata al client. Lo stesso avviene anche per la stringa contenente il risultato del parsing. Da specificare che per usare i cookies occorre fare *import javax.servlet.http.Cookie;* all’inizio. Per ultimo vi è il reindirizzamento alla pagina *Editor.jsp*, che ricarica la pagina.

Per quanto riguarda l’elaborazione lato client:

```
window.onload = function(){
    document.getElementById("myText").innerHTML =
    minmagHtml(getCookie("cookie"));
    document.getElementById("out").innerHTML =
    minmagHtml(getCookie("cookie2"));
    var today = new Date();
    var giorno = today.getDate();
    var mese = today.getMonth()+1;
    var minuti = today.getMinutes();
    var ora = today.getHours();
    if(giorno<10) giorno="0"+giorno;
    if(mese<10) mese="0"+mese;
    if(minuti<10) minuti="0"+minuti;
    if(ora<10) ora="0"+ora;
    var tempo = "["+giorno+'/' +mese+'/' +today.getFullYear()+ " " +ora +
    ":" +minuti+"]";
    document.getElementById("tempo").innerHTML = tempo;
}
```

Il codice presente nella JavaServer Pages ha il compito di leggere i cookies al momento di caricamento della pagina. Tramite *getCookie()* è possibile caricare i cookie, decodificarli, per poi tradurre i caratteri speciali sostituendoli opportunamente, ad esempio con gli spazi. Con la funzione *minmagHtml()* si va a sostituire i caratteri “<” e “>” rispettivamente con “<” e “>”, questo perché, in html e quindi anche per le JSP, sono caratteri speciali che se mandati in visualizzazione non vengono interpretati. Attraverso l’assegnazione *document.getElementById(“myText”).innerHTML = minmagHtml(getCookie(“cookie”))*; è possibile mandare alla presentazione il valore che è stato precedentemente prodotto, quindi il valore dei cookies. Infatti, tramite ** è possibile visualizzare il valore mandato dall’assegnazione precedentemente citata. Per ultimo, per motivi estetici, vi è la porzione di codice che ricava la data e l’ora, che viene visualizzata ad ogni caricamento della pagina.

3.2.6 Definizione dello stile in CSS

CSS, acronimo di Cascading Style Sheets, è un linguaggio che determina lo stile, la formattazione delle pagine html. Si occupa esclusivamente della presentazione delle pagine web. Questo, però, non può lavorare da solo, ha bisogno ad esempio di un html (nel caso del progetto una JSP) che prima definisca quali saranno gli elementi presenti sulla pagina. Elementi le cui caratteristiche vengono meglio definiti dal CSS. Nel progetto, questo file si può trovare in modo predefinito dopo aver eseguito una *Maven install* del progetto web Xtext. In questo modo si va a definire il font dei caratteri, il colore dei bordi, i riempimenti ed il posizionamento. Nel caso la pagina web non dovesse aggiornarsi a seguito di modifiche del file CSS, occorre premere F5 per risolvere il problema.

Come novità progettuali, sono stati introdotti tre nuovi componenti.

Il codice CSS per il titolo della console, che viene visualizzata a destra dell'editor, è il seguente:

```
#titolo_console {  
    display: block;  
    position: fixed;  
    background-color: #e8e8e8;  
    top: 20px;  
    left: 740px;  
    right: 0;  
    height: 60px;  
    padding: 10px;  
}
```

-display: block; definisce il modo in cui verranno rappresentati i paragrafi di testo. In pratica, con questa istruzione ogni inizio di paragrafo in html sarà automaticamente mandato a capo.

-position: fixed; definisce che il titolo della console sarà sempre fissa, indipendentemente se l'utente si muove nella pagina.

-background-color: #e8e8e8; definisce il colore di riempimento dello sfondo che circonda il titolo con la specificazione, tramite codifica, di quale sarà la colorazione.

-top: 20px; definisce la distanza in pixel tra il bordo più in alto dello schermo e l'oggetto in questione, quindi il titolo.

-left: 740px; definisce la distanza in pixel tra il bordo sinistro dello schermo e l'oggetto in questione, cioè il titolo.

-right: 0px; esattamente come per l'istruzione left, ma prendendo in considerazione il bordo destro.

-height: 60px; definisce la grandezza del titolo, quindi quanto questo sarà ampio in termini di altezza.

-padding: 10px; definisce in termini di pixel, la distanza che il testo contenuto all'interno avrà dai bordi che lo circondano.

Per il riquadro contenente il risultato del parsing si ha:

```
#secontainer {  
    display: block;  
    position: fixed;  
    top: 110px;  
    bottom: 20px;  
    left: 740px;  
    right: 20px;  
    padding: 4px;  
    border: 1px solid #aaa;  
}
```

Le prime istruzioni sono uguali a quelle illustrate prima. La parte interessante è l'istruzione *border: 1px solid #aaa*, che definisce in termini di pixel, quanto sarà ampio lo spessore del riquadro che andrà a contenere il risultato del parsing. Con la parola chiave *solid* si va a definire il tipo di bordo, che in questo caso sarà continuo. Ad esempio con *dotted* si andrebbe a definire un bordo fatto di puntini. *#aaa* definisce il colore che il bordo va ad assumere.

Per il bottone che manda in esecuzione il parsing si ha:

```
.run {  
    display: block;  
    position: fixed;  
    top: 110px;  
    bottom: 0;  
    left: 680px;  
}
```

Tutti le istruzioni di questo codice sono già state introdotte. Un'importante osservazione è che la posizione del bottone rimarrà invariata nello schermo, anche se si fa scrolling nell'editor.

Per ultimo, la modifica del codice del contenitore di testo dell'editor:

```
.container {  
    display: block;  
    height: 6000px;  
    position: absolute;  
    top: 0;  
    bottom: 0;  
    left: 0;  
    right: 0;  
    margin: 20px;  
}
```

L'istruzione non ancora vista è la *position: absolute*; la quale al contrario di *fixed*, permette il movimento dell'oggetto insieme allo scrolling della pagina. L'istruzione *margin* riguarda la distanza in termini di pagina, e non di bordo dello schermo come invece è per istruzioni come *bottom* o *left*. La grossa modifica apportata al progetto è *height* il quale è stato notevolmente incrementato rispetto al valore che assumeva prima di default. Questo per supplire al “difetto” della funzione JavaScript *document.getElementById().innerHTML*. Infatti, tale funzione “prende” solamente il testo visibile dall'utente, quindi se nel riquadro dell'editor si andasse oltre un certo numero di righe di codice, le righe non visibili a video non verrebbero considerate, generando inevitabilmente errori nel parsing. Tale argomento sarà comunque approfondito e discusso nel prossimo capitolo.

Con questa considerazione viene chiusa la presentazione dei concetti tecnici più importanti del progetto.

Capitolo 4

Risultati, limiti e conclusioni

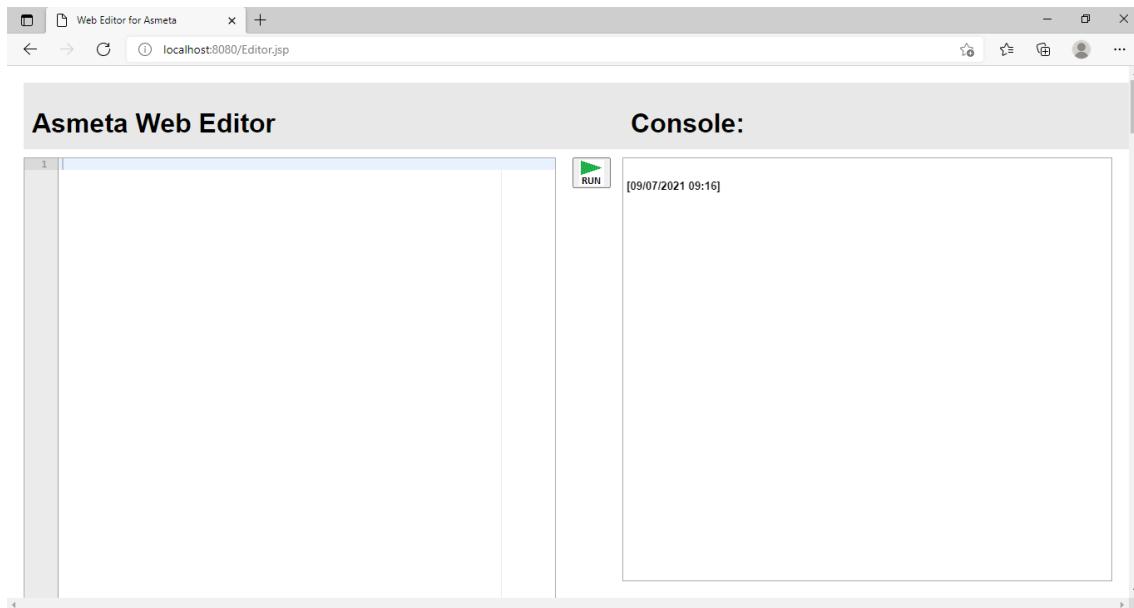
In quest'ultimo capitolo viene esposto il percorso compiuto durante il progetto assieme al risultato finale. Fino a questo punto l'approccio che si è cercato di avere è stato quello di presentare il progetto in maniera impersonale, descrivendo gli aspetti più interessanti dal punto di vista del codice. L'implementazione di diverse tecnologie comporta una difficoltà nel trattare argomenti di comprensione non immediata. Infatti, l'analisi e studio di questo progetto comporta, inevitabilmente, conoscenza, preparazione base in ambito web a livello teorico. Nella pratica è più facile a farsi che a dirsi.

Personalmente, l'approccio adottato fino a qui mi è sembrato adeguato fin quando si trattava di spiegare concetti teorici assieme ad aspetti tecnici, ma ora ritengo adeguato usare un linguaggio più diretto per poter descrivere le difficoltà trovate durante questo percorso.

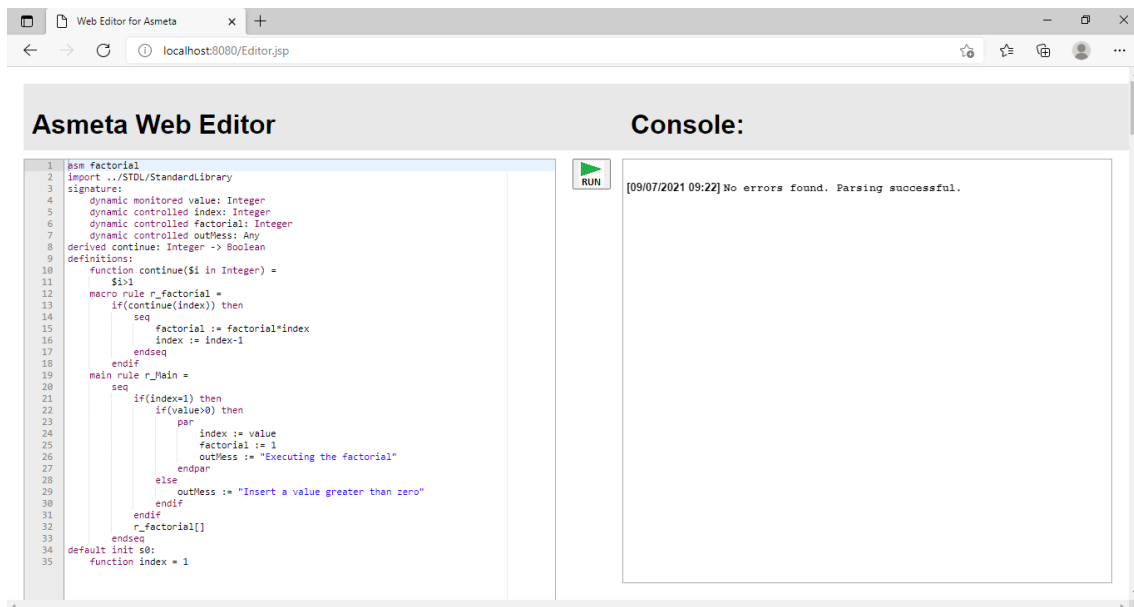
4.1 Presentazione dei risultati

In questa sezione vi saranno le immagini più interessanti che riguardano il risultato dei passaggi descritti nel capitolo precedente.

Digitando *localhost:8080* sul browser veniamo re-indirizzati a *localhost:8080/Editor.jsp* ed abbiamo:

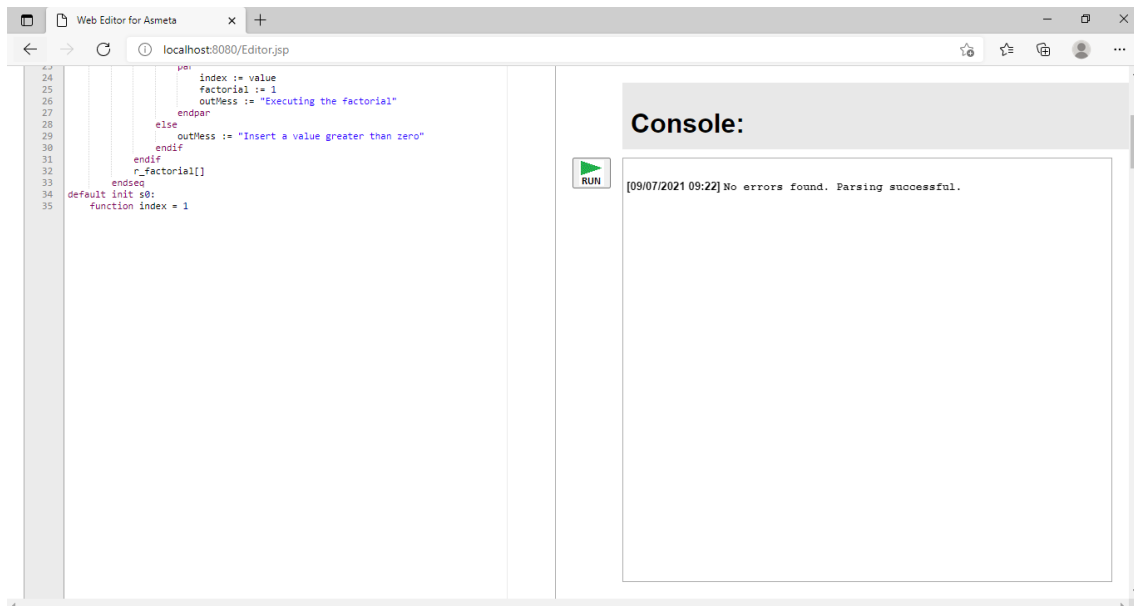


Copiando ed incollando un codice Asmeta e mandando in esecuzione il parsing otteniamo:

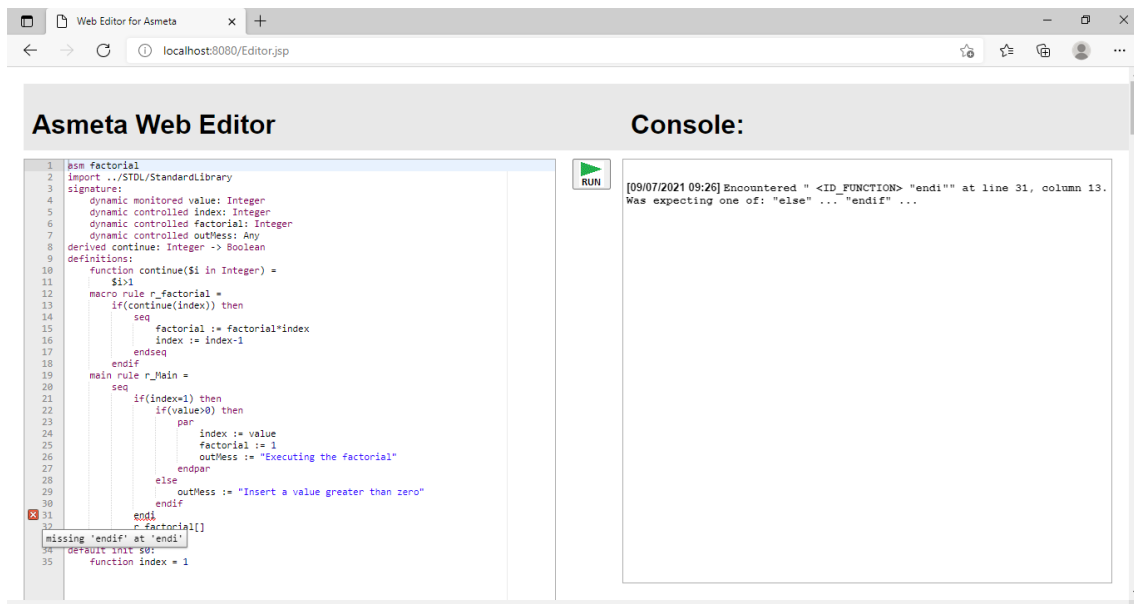


Si può osservare l'assenza di errori, sia per quanto riguarda il controllo di semantica e sintassi dell'editor, sia per il parsing.

Se facciamo scrolling nella pagina osserviamo che la posizione del bottone e l'aspetto della console restano invariate:

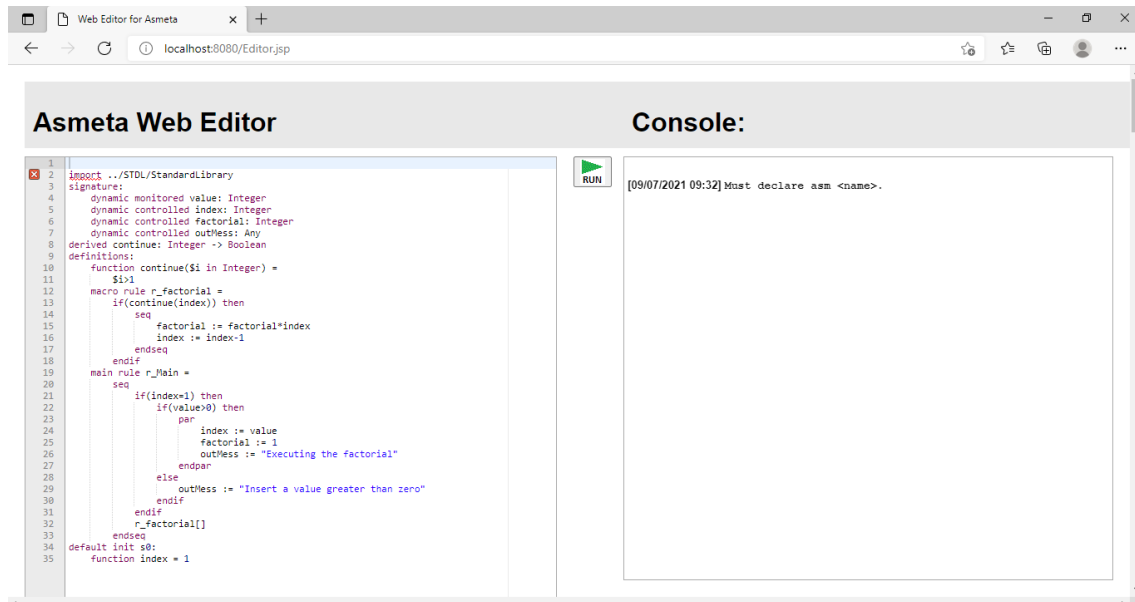


Può risultare utile fare dei test negativi, quindi scrivere un codice non corretto per il linguaggio (modificando ad esempio *endi* al posto di *endif*) :



Vediamo che il parser, non riconoscendo *endi*, ci segnala che si aspetta un *else* o un *endif*.

Se omettiamo la definizione di *asm* ad inizio codice:



Vediamo che il parser ci comunica di dichiarare *asm* *<nome>* in cima.

Questo conclude quelli che sono alcune delle casistiche più rilevanti dei risultati possibili prodotti dal Web Editor.

4.2 Limiti del progetto

Uno dei limiti, per il linguaggio, è quello di non riuscire ad avere una visione del file system del progetto. Infatti, una delle istruzioni comuni è *import ../STDL/StandardLibrary*. A questo scopo è stata aggiunta soltanto questa libreria nel progetto. Perciò, non sarebbe possibile importare altre librerie e nemmeno poter caricarne nuove nel progetto.

Altro limite è sicuramente quello testuale. Infatti, come ho accennato nel capitolo 3, la funzione *document.getElementById().innerHTML* riesce a prendere solamente il testo visibile nell'editor. Prendendo in considerazione l'editor generato da Xtext, senza alcuna modifica, è una casella di dimensione fissa che nel caso in cui si vada a scrivere più righe di quante ne possa contenere (circa quaranta) allora genera automaticamente una barra che permette fare lo scrolling sull'editor. Se dovessimo scrivere, ad esempio, cento righe di codice Asmeta, facendo una istantanea, vedremmo solamente circa quaranta righe di codice. Ecco, tramite *document.getElementById().innerHTML* riusciamo a prendere solo quelle quaranta righe, niente di più, niente di meno. Per questo motivo, nella spiegazione

del file CSS, è stato trattato come ho “allungato” l’editor, in modo che possa contenere circa quattrocento righe di codice. In questo senso è possibile ottenere tutte queste righe, per poter mandarle al parsing. Si può definire una soluzione a metà, in quanto per un sistema informatico tale limite non è corretto. L’uso che però si vuole dare al Web Editor non è quello di generare un progetto complesso, ma semplicemente avere a disposizione, su qualsiasi computer, un servizio che permetta di scrivere del codice non troppo impegnativo. Per i progetti più laboriosi c’è già infatti un plug-in Eclipse. Da questo punto di vista possiamo quindi considerare che il progetto svolto, è sì, non corretto in termini stretti, ma comunque accettabile all’uso per cui è stato pensato.

Restando in ambito testuale, anche i cookies hanno un limite in termini di memoria, che varia in base al browser web che si utilizza. In genere questa oscilla intorno ai 4KB. Se comunque questo non dovesse bastare è sufficiente creare altri cookies, infatti, solitamente i browser web prevedono un massimo di 20 cookies per lo stesso sito, quindi per URL con stesso dominio.

Per l’editor una limitazione importante potrebbe essere quella di non avere la possibilità di tornare indietro attraverso la funzione *Ctrl+Z*. Questo è dovuto al meccanismo implementato nel progetto, cioè quello di fare il refresh della pagina.

In termini progettuali, cioè dello sviluppo del Web Editor, un limite potrebbe essere la mancanza di dipendenza stretta tra JavaServer Pages e Servlet. Questo ha comportato il dover utilizzare la tecnologia dei cookies, rendendo il progetto più complicato dal punto di vista del codice. E si sa; un progetto più complicato comporta una bassa manutenibilità.

4.3 Difficoltà e conclusioni

Inizialmente le difficoltà le ho trovate nel capire bene l’ambito in cui mi stavo cimentando. In particolar modo, dopo che il mio relatore mi aveva spiegato il funzionamento Xtext, non capivo bene il suo meccanismo, non essendomi mai addentrato in ambito web. Per la natura dei miei studi l’argomento mi interessava, però l’intoppo era sempre dietro l’angolo. All’inizio non riuscivo nemmeno a far partire l’esempio di Web Editor predefinito di Xtext, cioè l’“Hello World”. Sono stato parecchio tempo a provare a farlo andare, installando tutti i tipi di Eclipse sul mio computer, senza però ottenere

risultati positivi. Ogni volta che facevo il *Maven install* mi andava sempre in errore. Allora decisi di cambiare computer, ciò mi ha comportato la generazione di un errore in fase di test dell'installazione. Cercando una soluzione, ho trovato che era sufficiente aggiungere al file *pom.xml* un'istruzione capace di ignorare i *test failure*. Il progetto di esempio ora partiva e riuscivo a vedere finalmente il Web Editor di base. Dopo questo primo traguardo, dovevo adattare il progetto di plug-in Eclipse già esistente. Anche in questo caso ho trovato delle difficoltà, ma dopo alcune dritte dei collaboratori del mio relatore, ed anche dopo aver capito meglio l'importanza del file *.mwe2* sono riuscito a far partire il Web Editor per Asmeta.

Il prossimo passaggio è stato quello di implementare il parser attraverso una funzione Java. Il mio relatore mi suggerì di usare le JSP. Come primo step dovevo ricavare il testo; inizialmente non trovavo il modo di prendere l'input testuale dall'editor, successivamente riuscii ad adoperare una funzione che mi dava la soluzione sotto forma di html. Sapevo però che, con qualche algoritmo, sarei riuscito a ricavarli il testo "pulito".

Ottenuto il testo in ingresso, ora mancava solo il modo di implementare la funzione Java del parser. Imparai quindi cosa fossero le Servlet, e come utilizzarle a livello base. A poco a poco implementai la comunicazione tra Servlet e JSP. A quel punto mi interessava vedere che il parsing funzionasse. Qui trovai qualche difficoltà, ma con l'aiuto del mio relatore sono riuscito ad implementare il tutto e quindi mostrare il risultato del parser sulla pagina.

Purtroppo, però, c'era il problema che dopo l'esecuzione della Servlet, la JasvaServer Pages non riprendeva più con il funzionamento dell'editor. Cercando su Internet, trovai alcuni esempi che implementavano i cookies. Di principio, l'idea non mi convinceva ma alla fine decisi di utilizzarli, in quanto mi ritornava utile la memorizzazione del testo e del risultato. È da mettere in luce un errore riscontrato: la non accettazione dei cookies degli spazi vuoti (" "). A questo scopo aggiunsi una codifica al salvataggio seguita dalla decodifica in lettura di questi caratteri, meccanismo simile a quando si passano i parametri via URL.

A questo punto potevo considerare, ragionevolmente, il progetto concluso.

Di seguito vorrei esporre quelle che sono le mie considerazioni prettamente personali. Esteticamente il progetto sembra complicato da un punto di vista concettuale e per certi punti lo è, in quanto, per le funzionalità che il Web Editor compie, si potrebbero semplificare molte cose. Alla fine, però, sono state tutte queste complicazioni, tutte queste difficoltà, a farmi capire meglio certi concetti. Ho capito, fuori dall'ambito teorico, l'importanza di separare la presentazione dalla logica di controllo e come questo influenza l'ingegneria del software. Ho capito la stretta dipendenza tra un codice JavaScript ed il browser web. Ho imparato ad usare i cookies sia lato client che lato server. Ho imparato anche qualche comando in CSS. Se ci mettiamo ad osservare l'Asmeta Web Editor da un punto di vista di efficienza, non è ottimizzato, perché sicuramente ci sono delle operazioni la cui elaborazione potrebbe essere semplificata, comportando un minor carico computazionale.

Nel complesso posso ritenermi soddisfatto, in quanto il progetto mi ha permesso di sviluppare delle conoscenze che mi ritorneranno sicuramente utili, dato che al momento della stesura di questo elaborato la richiesta dei servizi internet, ricercati dal mercato, è in fase crescente, anche perché impulsato dalla pandemia che è ancora, purtroppo, in corso.

Bibliografia

- [1] Wolfgang Reisig. «Abstract State Machines for the Classroom». 2006. URL: https://www.researchgate.net/publication/318521286_Abstract_State_Machines_for_the_Classroom
- [2] URL: <https://asmeta.github.io/index.html>
- [3] URL: <https://asmeta.github.io/download/index.html>
- [4] URL: https://asmeta.github.io/material/AsmetaL_quickguide.html
- [5] URL: https://asmeta.github.io/material/AsmetaL_guide.pdf
- [6] URL: <https://www.eclipse.org/Xtext/>
- [7] URL: https://www.eclipse.org/Xtext/documentation/306_mwe2.html
- [8] URL: <https://www.html.it/articoli/primi-passi-con-le-servlet/>