

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.

Sommario

Introduzione.....	8
1.1 Cosa è una Abstract State Machine in generale	8
1.2 Informazioni sull'elaborato	9
1.2.1 Inquadramento generale.....	9
1.2.2 Struttura dell'elaborato	10
Nozioni su Asmeta ed Xtext.....	11
2.1 Il framework Asmeta	11
2.2 Il linguaggio AsmetaL.....	12
2.3 Il framework Xtext.....	14
Fasi dell'implementazione Web	16
3.1 Semantica, sintassi e configurazione.....	16
3.1.1 Installazione di Xtext su Eclipse e generazione progetto	17
3.1.2 Il file .xtext	20
3.1.3 Il file .mwe2.....	21
3.1.4 Risultato intermedio.....	22
3.2 Progetto Web	23
3.2.1 Scelte progettuali	23
3.2.2 Input del testo tramite JavaScript	25
3.2.3 Comunicazione JSP-Servlet	26
3.2.4 Implementazione bottone di esecuzione	27
3.2.5 Il Controller ed il parsing	28
3.2.6 Definizione dello stile in CSS.....	36
Risultati, limiti e conclusioni	40
4.1 Presentazione dei risultati	40
4.2 Limiti del progetto.....	43
4.3 Conclusioni.....	44

Bibliografia	47
---------------------------	-----------

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.

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 **definizioni 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 avanti nel dettaglio del 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].

Xtext, come si può ben capire, fornisce tutti i servizi necessari per sviluppare il progetto del Web Editor. Infatti, per gli scopi di questo elaborato, partendo dai requisiti più elementari, è necessario avere una “casella” di testo in una pagina web su cui poter scrivere del testo. 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 ci sarà quindi l’analisi dettagliata di 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, commentando ed esponendo i problemi maggiori, alcuni dei quali hanno comunque inciso in modo rilevante le tempistiche. Certe precisazioni sembreranno forse inutili e anche banali, però capita non raramente che le cose in apparenza facili, creino dei problemi successivamente. Problemi che in alcuni casi sono stati “risolti” se così si può dire, 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

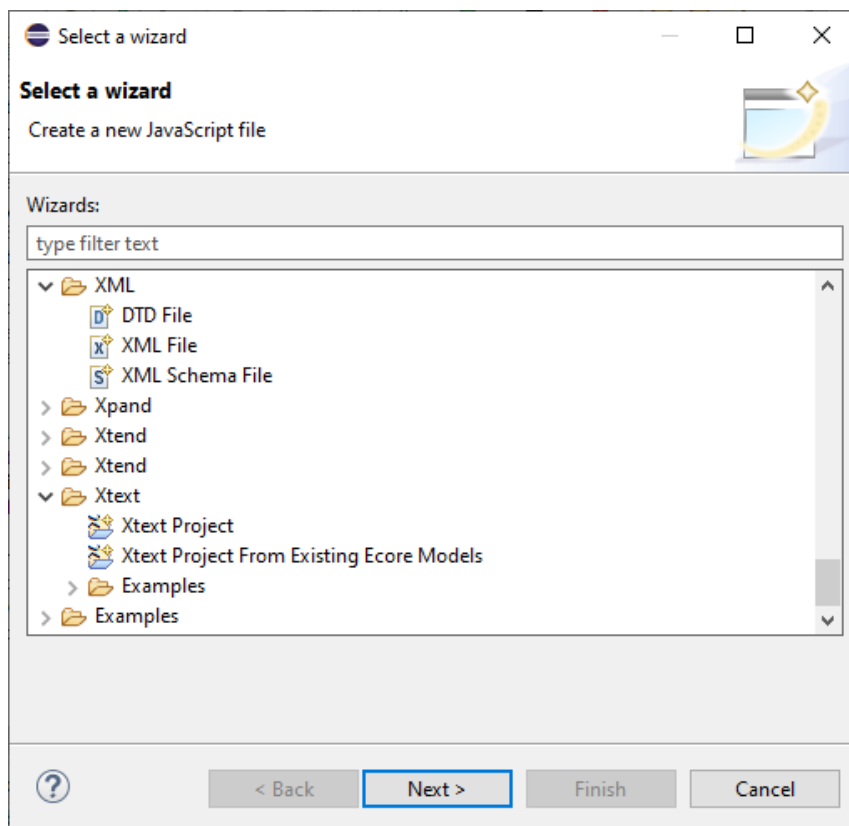
In questo paragrafo è spiegato l'inizio del progetto, per arrivare a come adattare i file del plug-in Eclipse già esistente ed in questo modo riuscire ad avere un Web Editor che possa evidenziare le parole chiave del linguaggio Asmeta, come ad esempio *asm* oppure *function*.

3.1.1 Installazione di Xtext su Eclipse e generazione progetto

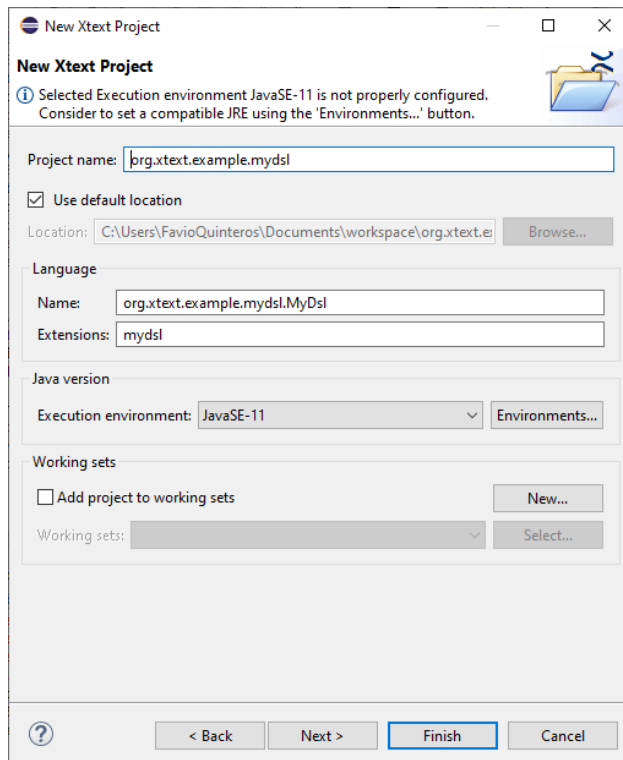
Dopo la breve presentazione su Xtext, viene qui illustrato l'installazione di Xtext su Eclipse, e la creazione del progetto su cui si andrà ad agire. Prerequisito fondamentale è ovviamente avere già Eclipse for Java Developers. Se capita già fin dall'inizio che non si avvia nemmeno Eclipse, le soluzioni di solito possono essere di due tipi: o serve 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 modificare le impostazioni di lancio del programma.

Per l'installazione di Xtext, è sufficiente andare nella pagina di download di Xtext, scegliere una versione tra quelle disponibili copiando l'URL che li indirizza. Sarà poi quell'URL a permetterci fare l'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 qui si seleziona *Xtext Project> Next* e dovrebbe apparire la seguente pagina:



The 'New Xtext Project' dialog box is shown. It has a title bar with a question mark icon, a close button, and a maximize button. The main title is 'New Xtext Project'. Below it is a message: 'Selected Execution environment JavaSE-11 is not properly configured. Consider to set a compatible JRE using the 'Environments...' button.' The dialog contains several sections: 'Project name' with a text field containing 'org.xtext.example.mydsl'; 'Use default location' checked; 'Location' with a text field containing 'C:\Users\FavioQuinteros\Documents\workspace\org.xtext.e' and a 'Browse...' button; 'Language' section with 'Name' 'org.xtext.example.mydsl.MyDsl' and 'Extensions' 'mydsl'; 'Java version' section with 'Execution environment' 'JavaSE-11' and an 'Environments...' button; 'Working sets' section with 'Add project to working sets' unchecked, a 'New...' button, and a 'Working sets' dropdown with a 'Select...' button. At the bottom are buttons for '< Back', 'Next >', 'Finish' (highlighted), and 'Cancel'.

New Xtext Project

Selected Execution environment JavaSE-11 is not properly configured. Consider to set a compatible JRE using the 'Environments...' button.

Project name:

☒ Use default location

Location:

Language

Name:

Extensions:

Java version

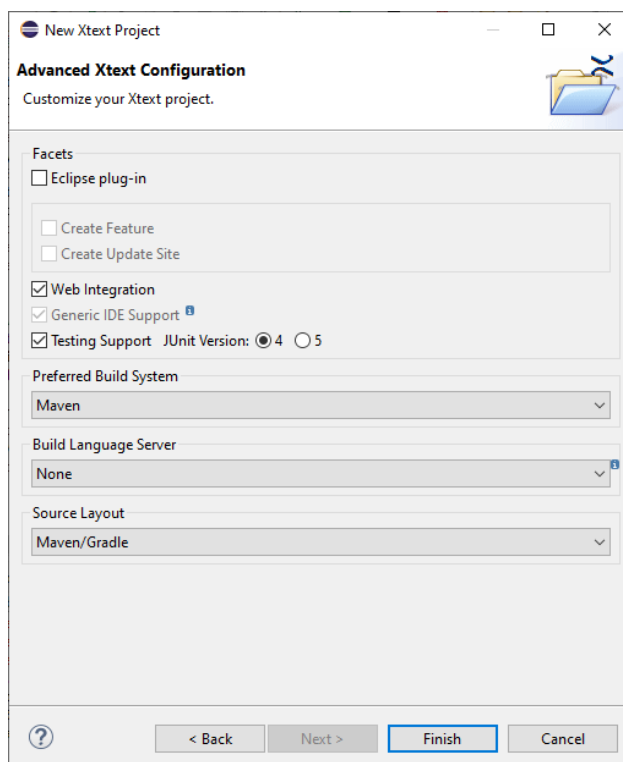
Execution environment:

Working sets

☐ Add project to working sets

Working sets:

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 modificato, bisogna andare avanti con *Next* ed arrivare alla seguente pagina:



The 'Advanced Xtext Configuration' dialog box is shown. It has a title bar with a question mark icon, a close button, and a maximize button. The main title is 'Advanced Xtext Configuration' with the subtitle 'Customize your Xtext project.' The dialog contains several sections: 'Facets' with 'Eclipse plug-in' unchecked, 'Create Feature' and 'Create Update Site' unchecked, 'Web Integration' checked, 'Generic IDE Support' checked, and 'Testing Support' checked with 'JUnit Version' set to '4'; 'Preferred Build System' set to 'Maven'; 'Build Language Server' set to 'None'; and 'Source Layout' set to 'Maven/Gradle'. At the bottom are buttons for '< Back', 'Next >', 'Finish' (highlighted), and 'Cancel'.

Advanced Xtext Configuration

Customize your Xtext project.

Facets

☐ Eclipse plug-in

☐ Create Feature

☐ Create Update Site

☒ Web Integration

☒ Generic IDE Support

☒ Testing Support JUnit Version: ☒ 4 ☐ 5

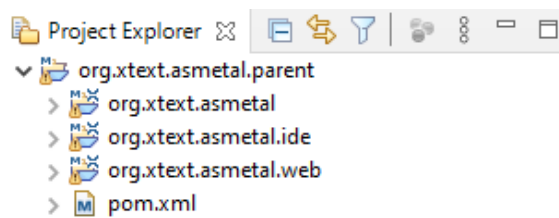
Preferred Build System

Build Language Server

Source Layout

Qui possiamo indicare cosa vogliamo includere nel nostro progetto. Interessandoci solo l'ambito web, è necessario quindi 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 quindi *Finish* e dovrebbero essere generate 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, il quale ogni 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 anche di non dover spendere risorse inutilmente, soprattutto quando non si dispone, ad esempio, di un dominio internet.

Fatta la precisazione, possiamo andare avanti sul progetto, che, per adesso è quello generato automaticamente dal Wizard di Xtext. Infatti, tramite esso è possibile eseguire il classico programma “Hello World”. Per visualizzare quindi l'editor già generato automaticamente, è necessario eseguire i seguenti passaggi (che ovviamente 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*.

Questo dovrebbe essere il risultato:



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 abbiamo generato 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, ne .web. In questa cartella, per essere più precisi, lo si trova all’interno della cartella *main*. Questo file è importante in quanto definisce la sintassi e semantica del linguaggio. Grazie a questo possiamo quindi 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 era già esistente da un progetto di un 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. Quello su cui agisce il file sono le configurazioni per quanto riguarda il progetto in generale. Più esattamente è un generatore che può essere configurato esternamente, in modo dichiarativo. L'utente di questo modo 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:

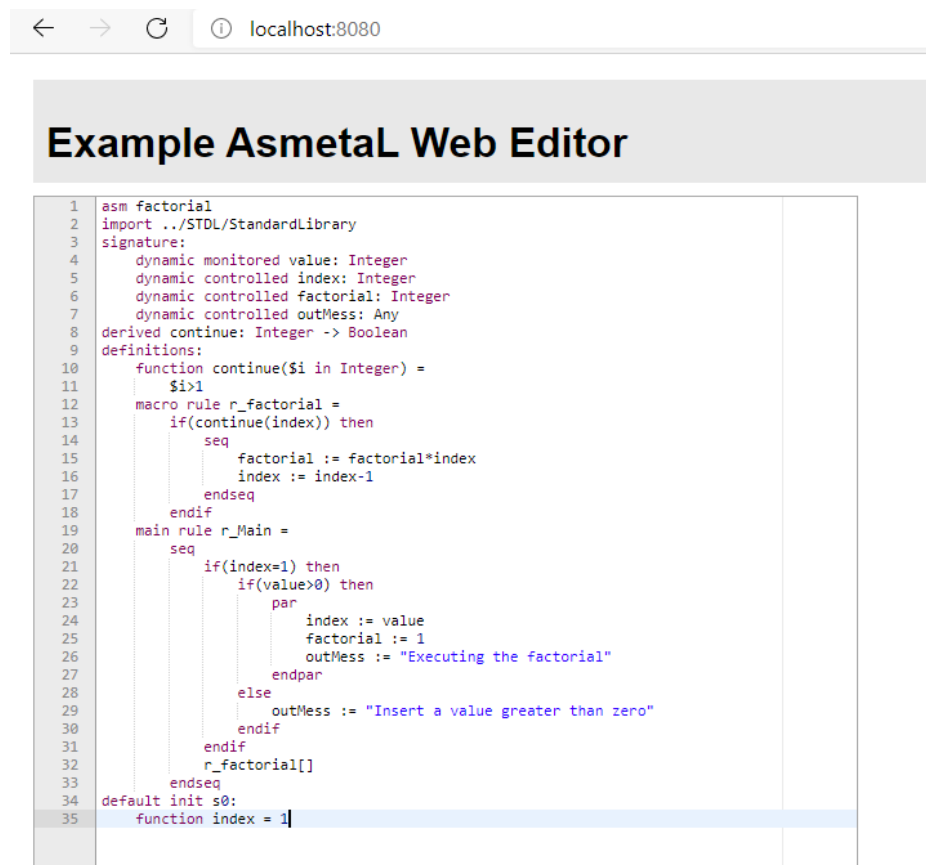
```
component = XtextGenerator {
  configuration = {
    project = StandardProjectConfig {
      baseName = "org.xtext.asmetal"
      rootPath = rootPath
      runtimeTest = {
        enabled = true
      }
      web = {
        enabled = true
      }
      mavenLayout = true
    }
    code = {
    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

Si può ben capire che da qui è possibile impostare quelli che saranno i componenti da generare. È possibile vedere 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 dovrebbe ora contenere 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 at localhost:8080 displaying the 'Example AsmetaL Web Editor'. The editor contains 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 viene riconosciuta la semantica, riconoscendo le parole chiave, inoltre non viene segnalato nessun errore per quanto riguarda la sintassi. Se si scrive qualche istruzione che non ha senso per Asmeta, viene segnalato a fianco vicino ai numeri che segnalano le righe con una X.

3.2 Progetto Web

In questo paragrafo viene presentato quello che è il cuore del progetto della parte web. Si comincia con la spiegazione del perché sono state effettuate certe scelte tecnologiche, per poi andare a spiegare quindi cosa essi sono, spiegando anche quali sono i problemi sorti con le loro relative soluzioni.

3.2.1 Scelte progettuali

Prima della discussione riguardo al progetto, è necessario prima 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. Quindi in parole più semplici, il parser non si limita a fare un controllo sintattico riga per riga, ma fa un controllo per quanto riguarda l'interezza del codice.

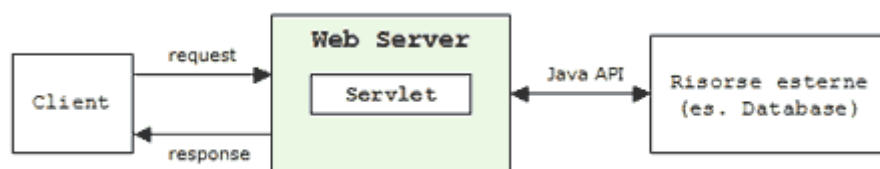
Riprendendo l'analisi del progetto, ora che abbiamo un editor web di base, c'è bisogno di implementare alcune funzionalità, come il *parsing* per l'appunto. Dando un'occhiata nel file system del progetto con estensione .web è stato notato che vi era un file di nome *index.html* che si occupava della visualizzazione dell'editor. Essendo un file con nome *index* si tratta quindi del file che viene automaticamente chiamato dal server quando viene composto l'URL *localhost:8080* nel web browser.

Visto che nel plug-in Eclipse di Asmeta era già presente una funzione Java che faceva da parser al progetto, il mio relatore assieme ai suoi collaboratori avevano suggerito di implementare una JavaServer Pages (JSP). Quali sono i motivi che hanno spinto a 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. Interessante può essere illustrare 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, allora istanzia la Servlet ed avvia un nuovo thread per gestire la comunicazione, mentre nel caso non sia la prima richiesta, allora la non serve ricaricarla 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 abbiamo visto, le JavaServer Pages sono essenzialmente delle pagine html che permettono una integrazione di codice Java, quindi si occupano della presentazione, di ciò che l'utente vede. Mentre per 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.

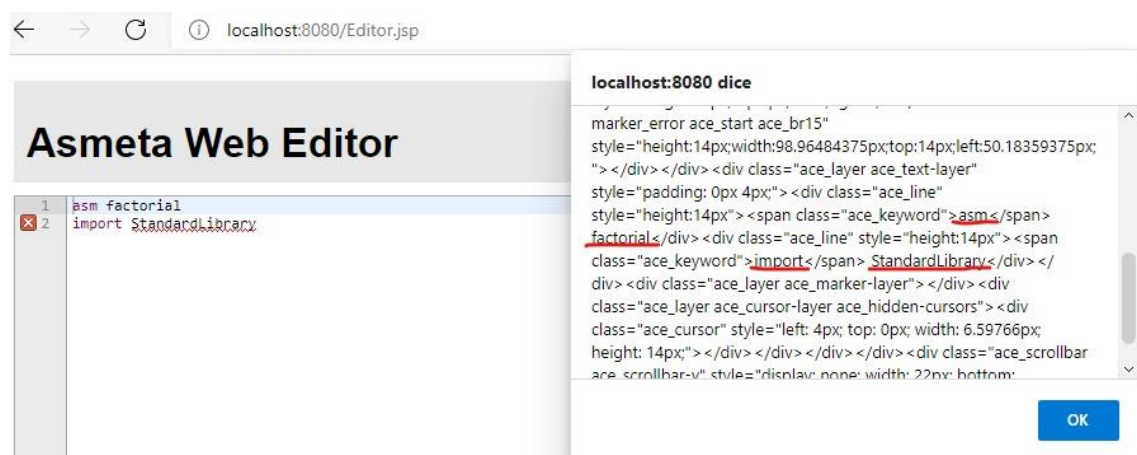
3.2.2 Input del testo tramite JavaScript

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. Essendo lato client la correttezza del codice dipende dal web browser usato, essendo alcuni metodi non presenti in tutti i tipi di applicativi per la navigazione Internet. È uno dei linguaggi più usati per la programmazione front-end. 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 quindi cancellato e sostituito con un comando di re-direct alla pagina JSP. In questo modo, digitando *localhost:8080* sul Web Browser, 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 è stato usato per ricavare l’input testuale che l’utente digita all’interno dell’editor. Il metodo usato 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). È stato dunque necessario creare una funzione in

JavaScript capace di “filtrare” la variabile *cod* in modo da ottenere soltanto il testo, che è quello che serve per l’elaborazione. Essendo abbastanza corposo il codice non verrà qui riportato.

Altra funzione importante da citare, importante per il progetto, è quella di lettura dei cookies. L’argomento dei cookies verrà spiegato meglio più avanti, però, essendo la loro lettura scritta in JavaScript, è giusto comunque illustrare il metodo capace di 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 cookie. Successivamente in base al nome si prende solo il cookie di interesse, per poi finire con un ciclo che converte i caratteri “+” in spazi “ ”, dovuto al fatto che i cookies non supportano certi caratteri, tra cui gli spazi vuoti, quindi è necessario fare un encoding prima di salvarli, per poi “decodificarli” quando li si legge.

3.2.3 Comunicazione JSP-Servlet

Prima è stato illustrato il modo in cui ottenere il testo di input dall’editor. Si sa anche che quest’ultimo 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:

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

Insieme al seguente codice da implementare nel codice JSP:

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

Nel quale il valore di *tes* è il testo da passare alla Servlet. Quest'ultima dovrà prendere il parametro passatogli attraverso l'istruzione:

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

In questo modo è possibile ottenere nella Servlet la stringa di testo in input dall'editor, che potrà quindi essere poi elaborata per il parsing.

Riassumendo, quando l'utente digita l'indirizzo *localhost:8080*, il client carica automaticamente il file *index.html*, ma questo contiene un reindirizzamento a *localhost:8080/Editor.jsp*. Questa JavaServer Pages, cui è "collegato" ad un file JavaScript avente le utilità necessarie al progetto, contiene la visualizzazione dell'editor, quello che era in precedenza *index.html*. 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 i dati alla Servlet che quindi eseguirà il metodo *doPost*.

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 compone il codice Asmeta nell'editor. A questo scopo è necessario modificare la JSP e 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 quello che è l'editor. La cosa che interessa di più riguarda l'aggiunta di un secondo riquadro, necessario per contenere i risultati del parsing, fatto attraverso l'istruzione `<div id="secontainer">`. Il bottone sarà generato 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 premerà il bottone, verrà eseguita la funzione `esegui()`, che quindi 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 il `<form>` contenuto nella JSP, tramite il quale è possibile scegliere quale metodo della Servlet chiamare. Le scelte sono due: o scegliere di invocare il `doGet`, usato principalmente quando la quantità di parametri da inviare non è elevata (passati solitamente tramite URL), oppure `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.

Ora che si sa come passare i parametri, possiamo vedere la struttura della Servlet. Importante affinché sia possibile raggiungerla attraverso la *form*, è inserire all'interno di quest'ultima *action="EditorServlet"*. Per la Servlet si dovrà aggiungere la seguente istruzione, subito dopo aver scritto gli opportuni *import*:

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

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

È possibile ora procedere con i passaggi che riguardano l'elaborazione del testo passato come parametro. 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 i passaggi 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 però 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 importare nel file Java il metodo, attraverso *import org.asmeta.parser.*;*

I metodi 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 si potrà prendere quello che sono 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 se sia presente nel progetto la StandardLibrary.asm. Se questa è già

presente non fa nulla, mentre se manca, allora la recupera, copiandola da una specifica cartella con denominazione “STDLD”.

-La stringa *text* contiene ora 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 invece *text* non è vuoto, attraverso *Matcher m* e la successiva condizione *if*, si vuole capire è presente la dicitura *asm “nome”*, quindi se è presente, allora selezionare tale nome che andrà a definire il file .asm.

-Se non viene trovata nessuna corrispondenza, allora si scrive su *parseResult* che si deve dichiarare un *asm “nome”*.

-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 quindi 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, in cui nel caso di un parsing senza errori, allora 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* dovrebbe contenere il risultato del parsing, segnalando: se non vi sono errori, se manca *asm “nome”*, e nel caso di errore, segnalare quali essi sono. L’idea in origine è stata quella di, una volta dato il controllo alla Servlet, fare l’elaborazione, e successivamente 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}
```

da aggiungere nella JSP.

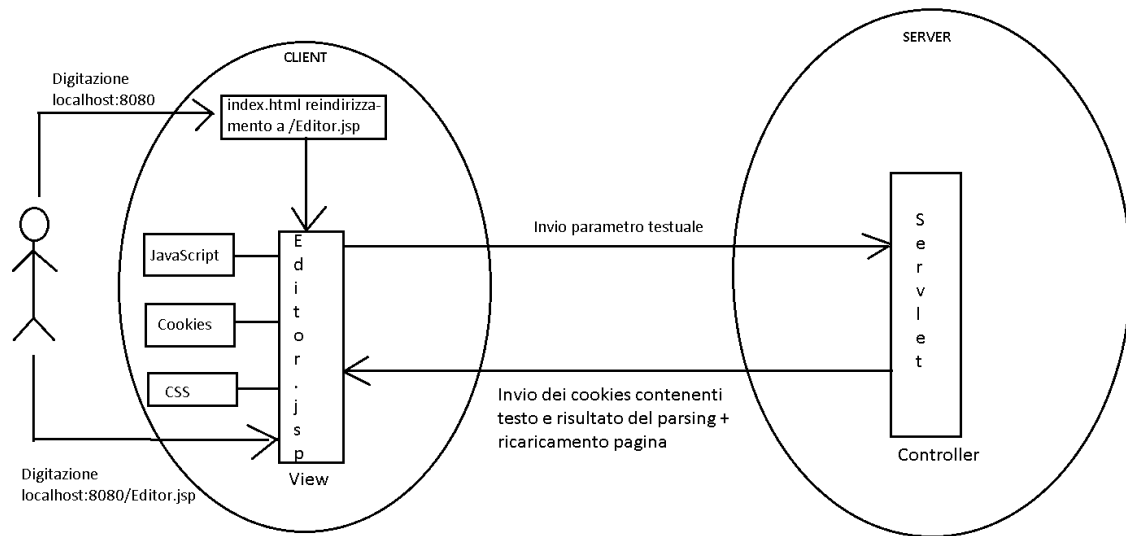
Con queste istruzioni si dovrebbe passare il valore di *parseResult* alla presentazione, ritornando (attraverso *getServletContext()*) il controllo alla JavaServer Pages. Quello che

accade invece è, sì, il passaggio del parametro alla presentazione, quindi viene visualizzato il risultato, però purtroppo, la JSP non riprende la sua esecuzione, quindi l'editor non viene reso più attivo, non più modificabile. Per un Web Editor, il fatto che dopo aver avviato tramite bottone il parsing, l'editor stesso non sia più modificabile, è ovviamente non accettabile per i requisiti intrinseci. Molto probabilmente 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 possibilità prese in considerazione furono due: usare i *frameset* oppure i cookies. I primi sono delle pagine indipendenti, unite per essere visualizzate insieme. Il problema qui riguarderebbe la comunicazione tra le pagine. Le seconde, 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 che dovrà memorizzare. Questo è particolarmente utile in quanto il protocollo *http* è privo di stato. Tali ragioni 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, e quindi ogni volta che si carica la pagina, recuperare questa informazione per visualizzarlo. Se prima ogni volta che si ricaricava la pagina, il testo sull'editor veniva perso, ora il testo viene prima recuperato dai cookies (rispetto all'ultima esecuzione della Servlet), per poi essere visualizzato.

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



Grazie a questo è possibile vedere meglio il funzionamento in generale del progetto, con gli scambi dei parametri e dove sono collocati 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, che ogni volta che viene eseguito il codice Java, dopo aver fatto le opportune elaborazioni, mandi al client il testo inviatogli in precedenza, quindi aggiornandolo, ed anche il testo contenente il risultato del parsing.

Per cominciare è più interessante 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 si vede un comando 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 (" "), quindi è 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, 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*.

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 caricare i cookies al momento di caricamento della pagina. Tramite *getCookie()* è appunto possibile caricare i cookie, decodificarli, quindi di tradurre i caratteri speciali sostituendoli opportunamente, ad esempio, con gli spazi. Con la funzione *minmagHtml()* si va invece 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* è possibile mandare alla presentazione il valore che è stato prima elaborato, quindi il valore dei cookies. Infatti, tramite ** è possibile visualizzare il valore mandato dall’assegnazione precedentemente citata, cioè quello dei cookies. Per ultimo, per motivi estetici, vi è la porzione di codice che ricava la data e l’ora, da far visualizzare 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, quindi la formattazione delle pagine html. Si occupa quindi, 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 verranno, appunto, 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.

Come novità progettuali, sono stati introdotti 3 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 di questo riquadro 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 serve a definire 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 di tipo 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. L'unica cosa che è possibile notare 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*, si ottiene che l'oggetto si muove 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 però *height* il quale è stato notevolmente incrementato rispetto al valore che assumeva prima. Questo per supplire al “difetto”, se così si può definire, della funzione JavaScript *document.getElementById().innerHTML*. Infatti, tale funzione “prende” solamente il testo visibile dall'utente, quindi se nel riquadro dell'editor 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 del progetto nella sua parte più tecnica.

Capitolo 4

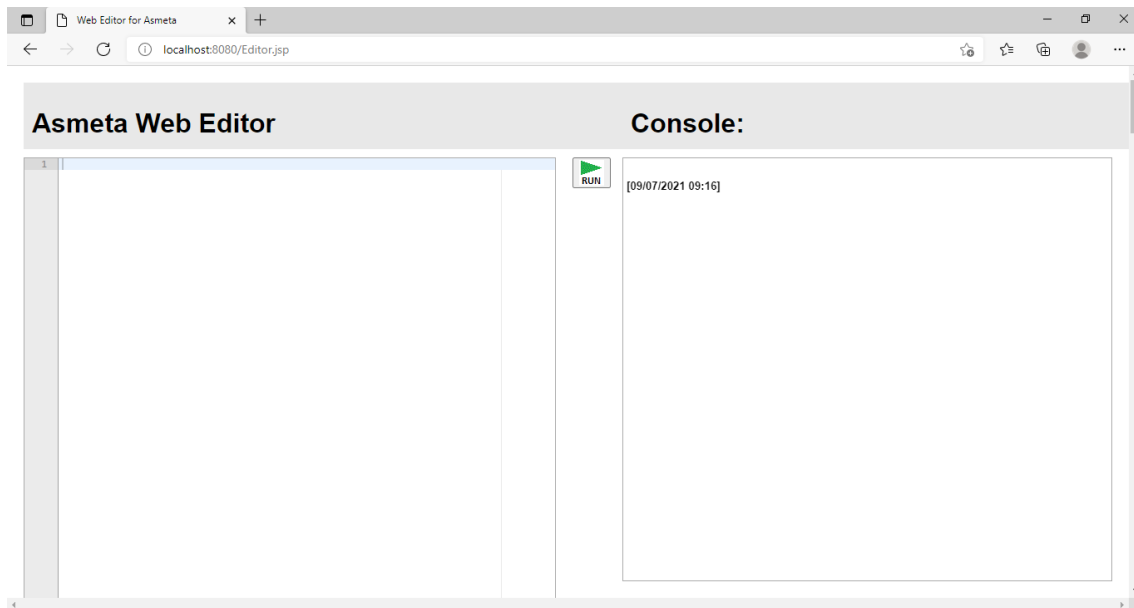
Risultati, limiti e conclusioni

In quest'ultimo capitolo viene illustrato il percorso seguito e ciò che è il risultato finale. Fino a questo punto l'approccio che si è cercato di avere è stato quello di cercare di presentare il progetto in maniera impersonale, descrivendo gli aspetti più interessanti dal punto di vista del codice. La descrizione dei problemi è stata affrontata, descrivendo subito quali fossero le soluzioni, in modo da essere risolti subito. Essendo state implementate diverse tecnologie non è stato facile descrivere il tutto, è un classico esempio, in cui è più facile a farsi che a dirsi, perlomeno lo è per una persona con conoscenze base in ambito web. L'approccio avuto mi è sembrato quindi adeguato fin quando si trattava di spiegare cosa fosse una ASM, oppure spiegare i passaggi del progetto, spiegando in breve le tecnologie usate, ma ora mi ritengo più adeguato usare un linguaggio più diretto per poter descrivere le difficoltà trovate durante questo percorso.

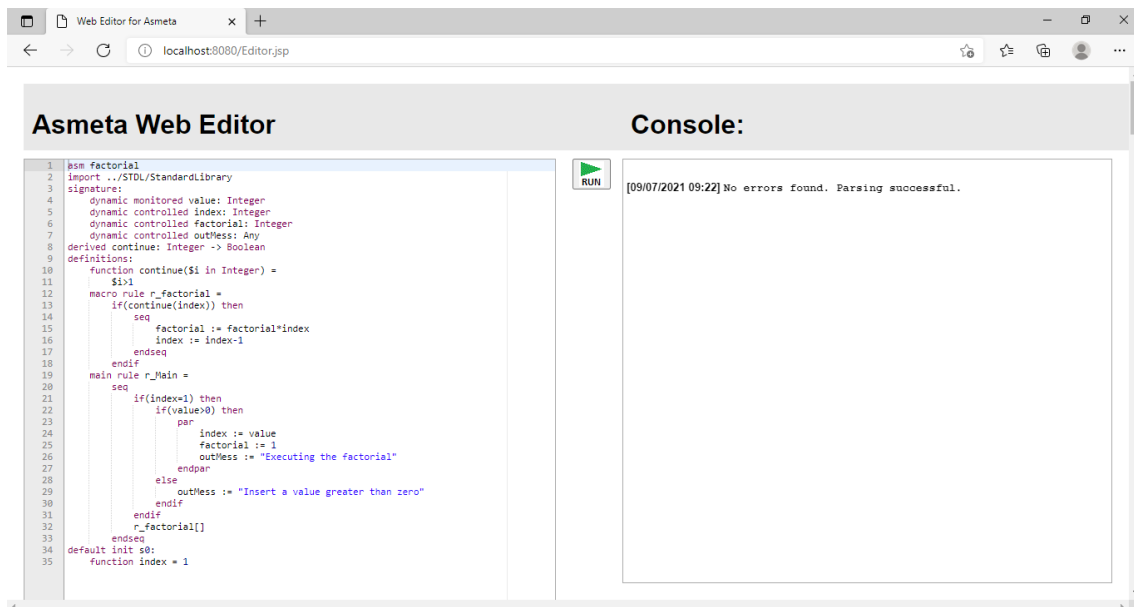
4.1 Presentazione dei risultati

In questa sezione ci 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:

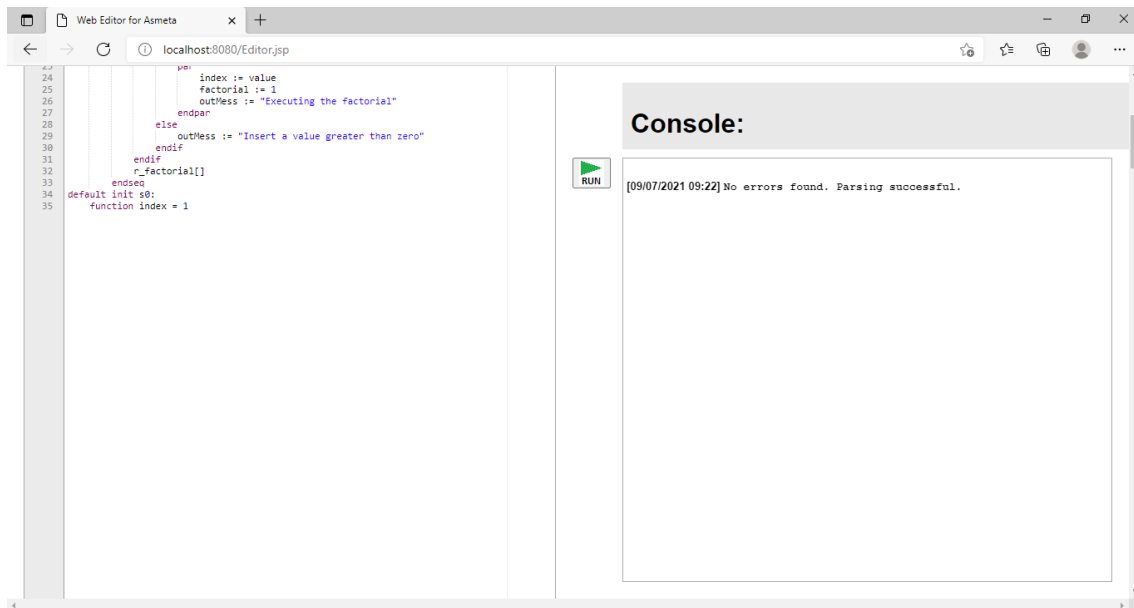


Copiando ed incollando un codice Asmeta di esempio e mandando in esecuzione il parsing otteniamo:

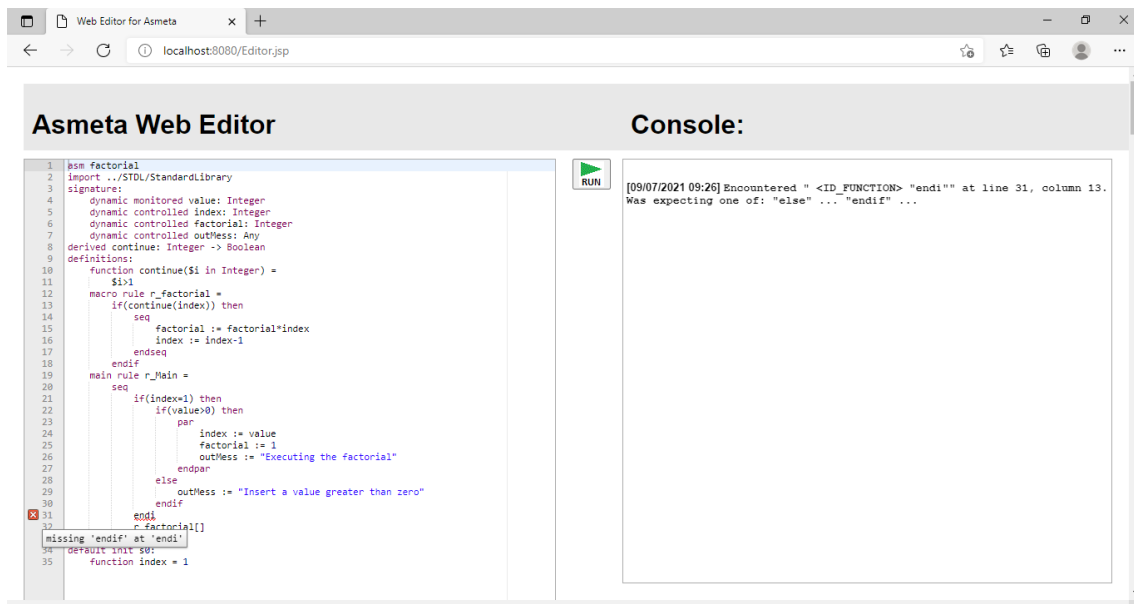


Si può osservare l'assenza di errori, sia per quanto riguarda il controllo di semantica e sintassi dell'editor, sia per il parsing.

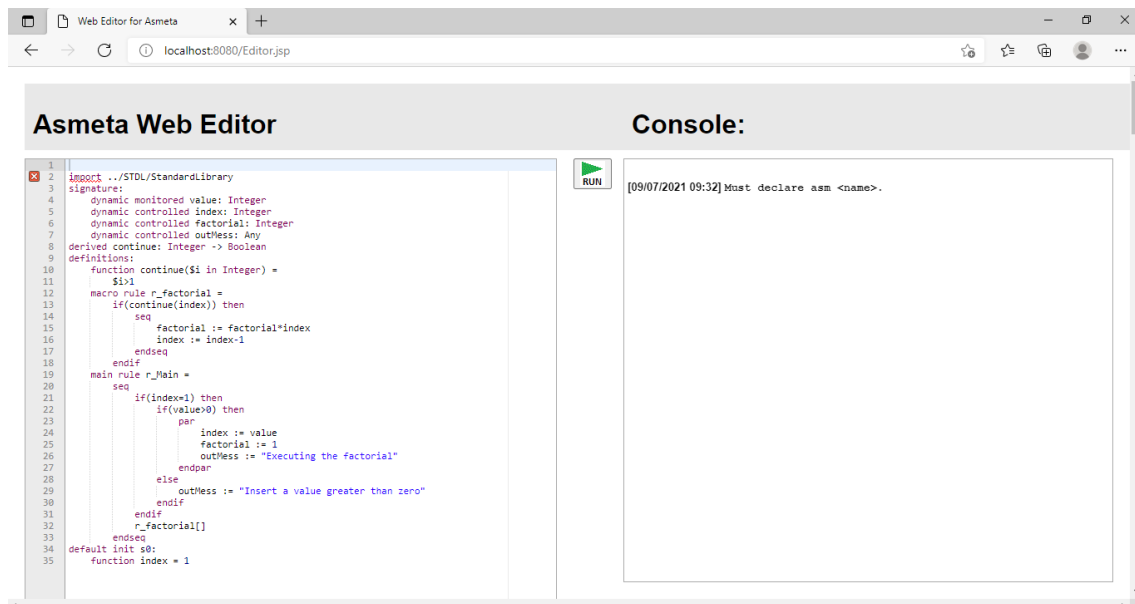
Se facciamo scrolling nella pagina osserviamo che il bottone e la console rimangono fermi:



Può risultare utile fare dei test negativi, quindi un uso scorretto del programma. Modificando il codice, scrivendo ad esempio *endi* invece di *endif*:



Oppure non definendo *asm* ad inizio codice:



Come vediamo, il parser ci comunica che dobbiamo dichiarare *asm* <nome> in cima.

Questo conclude quelli che sono alcuni esempi più rilevanti di risultati possibili prodotti dal Web Editor.

4.2 Limiti del progetto

Uno dei limiti per il linguaggio può essere quello di non riuscire ad avere una visione del file system del progetto. Infatti, una delle istruzioni comuni è *import ../STDL/StandardLibrary*. Questo comporta anche che oltre alla Standardlibrary (che è l'unica implementata nel progetto), non c'è un'altra libreria disponibile, ne vi è la possibilità di aggiungerne

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 solo 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 quindi 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 spiegato anche come ho allungato l'editor, in modo che possa contenere circa quattrocento righe di codice. Si può definire una soluzione a metà, in quanto per un sistema informatico tale limite non è corretto. L'uso che si vuole dare però al Web Editor non è quello di scrivere un progetto complesso, ma semplicemente avere a disposizione su qualsiasi computer, un servizio che permetta di scrivere qualcosa di comunque non troppo complesso. 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, ma comunque accettabile per l'uso a cui è pensato.

Restando in ambito testuale, anche i cookie 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. Infatti, 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 ha una bassa manutenibilità.

4.3 Conclusioni

Inizialmente le difficoltà le ho trovate nel capire bene l'ambito in cui mi stavo cimentando. In particolare modo, dopo che il mio relatore mi aveva spiegato Xtext ed il suo funzionamento, non capivo bene come facesse il tutto a funzionare, non essendomi mai addentrato in ambito web. L'argomento ovviamente mi interessava, però l'intoppo era sempre dietro l'angolo. Infatti, 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 di farlo andare, installando tutti i tipi di Eclipse sul mio computer, ma niente,

ogni volta che facevo il *Maven install* mi andava sempre in errore. Stufo, decisi quindi di cambiare computer, ma stavolta mi appariva un errore in fase di test dell'installazione. Così cercai per un po' una soluzione, trovando che bastava aggiungere al file pom.xml un'istruzione capace di ignorare i *test failure*. Il progetto di esempio ora partiva e riuscivo a vedere finalmente un Web Editor base. Dopo questo primo traguardo, dovevo adattare il progetto di plug-in Eclipse già esistente. Inizialmente ho trovato alcune 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 fu quindi, quello di implementare il parser, attraverso una funzione Java. Il mio relatore mi suggerì di usare le JSP. Sapevo che per far andare il parser, dovevo prima ricavare il testo. Ed ecco che nuovamente non riuscivo a trovare il modo di prendere l'input testuale dell'editor. Dopo un po' di tempo trovai una funzione che mi dava il testo sotto forma di html, ma non mi interessava, sapevo 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. Per quel momento mi interessava vedere che il parsing funzionasse. Inizialmente ho trovato 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, ora vi era il problema che dopo l'esecuzione della Servlet, la JspServer Pages non riprendeva più con l'esecuzione dell'editor. Cercando su internet, trovai alcuni esempi che implementavano i cookies. Inizialmente l'idea non mi convinceva, ma alla fine decisi di implementarlo. Un primo errore apparso fu quello che i cookies non accettavano spazi vuoti (" "), quindi implementai una codifica seguita dalla decodifica di questi caratteri, meccanismo simile a quando si passano i parametri via URL. A questo punto il progetto, potevo considerare, bene o male il progetto concluso.

Posso quindi riflettere su tutto quello che è stato il percorso di tutto questo progetto. 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. Certo, se ci mettiamo ad osservare l'Asmeta Web Editor da un punto di vista di efficienza, non è il massimo, perché sicuramente ci sono delle "strade" più brevi.

Nel complesso posso ritenermi soddisfatto, in quanto il progetto mi ha permesso di sviluppare delle conoscenze che saranno sicuramente utili, dato che al mondo d'oggi i servizi internet, anche per via della pandemia che è ancora in corso, sono sempre più richiesti dal mercato.

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/>