

Progetto di Linguaggi Formali e Compilatori

Giuseppe Mirra, Raul Jose Luizaga Yujra,

Alessio Cicero

Matricole n. 1076257, 1046611, 1071537



Università degli Studi di Bergamo

Prof. Giuseppe Psaila, Prof. Paolo Fosci

Novembre 2022

Indice

1	Introduzione e Toolchain	7
1.1	Introduzione	7
1.2	Toolchain	7
2	Libreria	10
2.1	Linguaggio di programmazione	10
2.2	Principali Classi e Funzioni	10
2.3	Parametri in input	10
2.4	File in output	10
3	Metalinguaggio	13
3.1	Introduzione Metalinguaggio	13
3.2	Struttura Metalinguaggio	13
3.3	Sezioni del metalinguaggio	14
3.3.1	File Input Path	14
3.3.2	File Output Path	14
3.3.3	File Type Conversion	16
3.3.4	Rows Cols Delete	16
3.3.5	modifying Cell	17
4	Grammatica	19
4.1	Approccio iniziale	19
4.2	Token	19
4.2.1	Token di inizio e fine sezione	19
4.2.2	Token operazionali	22
4.3	Regole	24
5	Semantica	35
5.1	Costruzione classe Handler	35

5.2	Campi della classe Handler	35
5.3	Funzioni della classe Handler	36
5.4	Utilizzo delle funzioni in ANTLRWorks	37
5.4.1	Utilizzo della funzione setInputPath()	38
5.4.2	Utilizzo della funzione setOutputPath()	40
5.4.3	Utilizzo della funzione setConversionType()	42
5.4.4	Utilizzo della funzione setSheetName()	44
5.4.5	Utilizzo della funzione fusionDelete()	46
5.4.6	Utilizzo della funzione modifyingCellMeth()	49
6	Gestione degli errori	51
6.1	Errori da evitare	51
6.2	Gestione errori lessicali	51
6.3	Gestione errori sintattici	52
6.4	Gestione errori Semantici	52
6.5	Esempi di errori	52
6.5.1	Errore su Type Conversion - Errore Semantico	53
6.5.2	Errore su Modify Cell - Errore Sintattico/Semantico	54
6.5.3	Errore su Type Conversion - Errore Sintattico	55
6.5.4	Errore su Modify Cell - Errori sintattici	56
6.5.5	Errore su STARTNAMESHEET - Errore Lessicale	57
6.5.6	Warning su STARTOUTPATH - Errore Semantico	58
6.6	Errori formato Data	58
7	Manuale Utente	60
7.1	Introduzione	60
7.2	Import Progetto su Eclipse	60
7.3	Pom.xml	60
7.4	Maven	61
7.5	Input.file	62

7.6	Run del progetto	63
7.7	About Codice	63
7.7.1	Codice Libreria	63
7.8	Complete main Class	64
8	esempi di funzionamento	66
8.1	Primo esempio: NameSheet e TypeConversion	66
8.2	Secondo esempio: eliminazione di una riga e di una colonna	66
8.3	Terzo esempio: file convertito in formato <i>.txt</i>	67
8.3.1	Quarto esempio: modifica cella	67

Elenco delle figure

1	Esempio dei file in output dalla libreria.	12
2	Metalinguaggio.	15
3	InputPath	16
4	OutputPath	16
5	type of conversion	17
6	rows and cols delete	18
7	modifying cell	18
8	language	25
9	body	25
10	inpath	26
11	outpath	27
12	path	27
13	conversion	28
14	type	28
15	namesheet	29
16	sheet	29
17	delete	30
18	row	31
19	col	31
20	todelete	32
21	modifyingcell	32
22	modcell	33
23	scanerror	34
24	body	38
25	setInputPath(i)	39
26	setInputPath()	39
27	path	40

28	setOutputPath(o)	41
29	outpath	41
30	setConversionType(cT)	42
31	conversion	42
32	type	43
33	setSheetName(cS)	44
34	namesheet	44
35	sheet	45
36	printDelete	47
37	delete	47
38	col	48
39	row	48
40	todelete	48
41	modifyingCellMeth(cM)	49
42	modifyingcell	49
43	modcell	50
44	Error Type Conversion	53
45	Error Type Conversion output	53
46	Error Modify Cell	54
47	Error Modify Cell output	54
48	Error Type Conversion	55
49	Error Type Conversion output	55
50	Error Modifying Cell	56
51	Error Modifying Cell output	56
52	Errore lessicale	57
53	Errore lessicale output	58
54	Warning	58
55	Warning output	58
56	Data sbagliata	59

57	Conversione data sbagliata	59
58	Esempio comando Maven.	62
59	Package Libreria	64
60	Package Grammatica	64
61	CompleteMain.java	65
62	Type conversion/Name sheet	66
63	Output run	67
64	Output run	67
65	Sezione modifica cella	68
66	Output run	68

Elenco delle tabelle

1	Toolchain e tecnologie utilizzate	8
2	Parametri in input	11
3	Dipendenze Pom.xml	61

1 Introduzione e Toolchain

1.1 Introduzione

Il progetto è nato esaminando una problematica reale che prende in considerazione la visualizzazione, modifica e la conversione dei file con estensione *.csv*, *.json*, *.txt*. Il problema di questi file risiede nella loro stessa formattazione che rende difficoltosa la modifica e la lettura dei dati. Per evitare queste criticità, le tre estensioni, vengono aperte sfruttando Microsoft Excel, programma prodotto da Microsoft dedicato alla produzione ed alla gestione di fogli elettronici. Microsoft Excel formatta e converte automaticamente il file in formato tabella. Questa formattazione semplifica la lettura e la modifica dei dati. Il secondo problema che emerge da Excel è che, una volta che il file è stato modificato, la nuova versione non può essere salvata con la formattazione originale. Il progetto permette, attraverso un linguaggio di programmazione personalizzato, di:

1. Modificare e leggere il file sfruttando la formattazione e i benefici di excel.
2. Di convertire il file da excel ad una delle tre estensioni citate pocanzi.

1.2 Toolchain

Per la realizzazione della libreria verranno utilizzati i seguenti strumenti che sono descritti in tabella 1:

Tool/Tecnologia	Utilizzo
Java 18.0.1	Linguaggio di programmazione della libreria
IntelliJ IDEA	IDE per sviluppo codice
Antler	Per scrittura e generazione della grammatica
Git e GitHub	Software e piattaforma per versionamento e condivisione di codice e documentazione
Overleaf	Software per la stesura e formattazione della documentazione in linguaggio LaTeX

Tabella 1: Toolchain e tecnologie utilizzate

2 Libreria

2.1 Linguaggio di programmazione

La libreria, che permette di risolvere le problematiche descritte nel paragrafo precedente "Introduzione e Tecnologie", è implementata in *Java*. Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti, a tipizzazione statica e progettato per essere il più possibile indipendente dalla piattaforma hardware di esecuzione.

2.2 Principali Classi e Funzioni

Le 4 classi principali sono: *converterExcelToType*, *excelToCVS*, *excelToJson*, *excelToTXT*. La prima classe prende in ingresso i vari input che poi serviranno alla modifica e alla conversione del file; questa prima classe può essere definita come una "classe d'appoggio". Le tre classi successive, convertono e modificano effettivamente il file originale in una o più estensioni scelte dalla utente.

2.3 Parametri in input

I parametri che la libreria prende in input sono 6, e sono rappresentati e descritti in tabella 2:

2.4 File in output

I file in uscita dalla libreria saranno tutti i singoli fogli elettronici, che compongono il file originale excel, convertiti singolarmente in tutte le estensioni desiderate. L'esempio è mostrato in figura 1. In questo caso specifico l'utente ha voluto convertire i due fogli elettronici "sheet1" e "sheet2", presenti all'interno del file excel "testConversion.xlsx", in tutte e tre le possibili estensioni: .csv, .txt e .json. Inoltre, l'utente tramite una delle funzionalità della libreria, ha rinominato i due fogli elettronici in "Foglio Uno" e "Foglio Due".

Parametri in input	Descrizione	Inserimento
<i>input Path</i>	Percorso del file che l'utente desidera modificare e/o convertire	obbligatorio
<i>output Path</i>	Percorso/percorsi dove l'utente desidera salvare il file modificato e/o convertito	opzionale
<i>typeConversion</i>	L'utente inserisce il tipo/i tipi di estensione in cui vuole convertire il file originale	obbligatorio
<i>nameSheetModifying</i>	L'utente può scegliere se rinominare o meno i singoli fogli elettronici che compongono il file excel	opzionale
<i>deleteRows</i>	Questo parametro contiene al suo interno gli indici delle righe che l'utente vuole eliminare dal foglio elettronico	opzionale
<i>deleteCols</i>	Questo parametro contiene al suo interno gli indici delle colonne che l'utente vuole eliminare dal foglio elettronico	opzionale
<i>modifyingCells</i>	Questo parametro contiene al suo interno la cella che l'utente vuole modificare nel foglio elettronico	opzionale

Tabella 2: Parametri in input

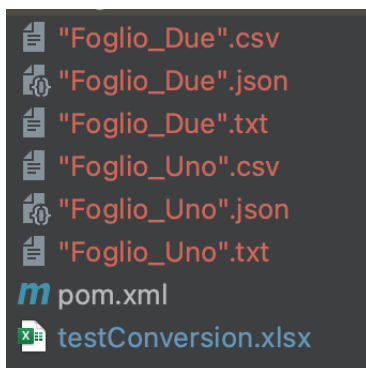


Figura 1: Esempio dei file in output dalla libreria.

3 Metalinguaggio

3.1 Introduzione Metalinguaggio

Nella logica e nella teoria dei linguaggi formali per *metalinguaggio* si intende un linguaggio formalmente definito che ha come scopo la definizione di altri linguaggi artificiali, definiti linguaggi obiettivo o linguaggi oggetto. Tale definizione tende ad essere formalmente rigorosa e completa, tanto da potersi utilizzare per la costruzione o la validazione di strumenti informatici di sostegno per i linguaggi obiettivo. Il metalinguaggio per il progetto è nato basandoci sulle esigenze e sulle azioni che poi avrebbe compiuto l'utente finale.

3.2 Struttura Metalinguaggio

Le caratteristiche, la struttura e le regole generali che compongono il metalinguaggio, mostrato in figura 2, sono:

1. *#STARTactionName* e *#ENDactionName*: questo è un tipo di token che specifica l'inizio e la fine di ogni sezione presente nel metalinguaggio. Le singole sezioni, che verranno spiegate dettagliatamente nei paragrafi successivi, sono le azioni che può compiere la libreria sul file excel in input. Le sezioni sono cinque e ogni singola sezione viene specificata sostituendo *actionName* con il nome specifico dell'azione che verrà eseguita in quella specifica sezione. Ad esempio: *#START-DOCUMENT ... #ENDDOCUMENT*, indicano, rispettivamente, l'inizio e la fine dell'intero documento.
2. *%* Commento *%*: il commento è la parte di testo compresa tra *%* ... *%*, tutto ciò che è compreso fra questi due simboli non viene considerato come dati da prendere in considerazione per generare il file di output.

3. < "stringa" o valore numerico >: le parentesi angolari sono coloro che contengono i dati, che l'utente inserisce, sia di testo che numerici, che verranno presi in considerazione quando si genera il file di output.

3.3 Sezioni del metalinguaggio

In questo paragrafo verranno descritte le 5 sezioni che compongono il metalinguaggio. Le sezioni sono:

1. Percorso del file in input. *File Input Path*
2. Percorso o percorsi del file in output. *File Output Path*
3. Tipo di conversione che si vuole fare. *File Type Conversion*
4. Righe e/o colonne che si vogliono cancellare dal file originale. *Rows Cols Delete*
5. Celle da modificare nel file originale. *Modifying Cell*

3.3.1 File Input Path

Questa prima sezione è obbligatoria. Il percorso del file excel dev'essere obbligatoriamente inserito dall'utente per poter far lavorare la libreria. Come mostrato in figura 3, il percorso del file viene inserito sottoforma di stringa all'interno delle parentesi angolari.

3.3.2 File Output Path

Questa seconda sezione non è obbligatoria. Il percorso del file excel può non essere inserito dall'utente. La libreria, anche senza percorso d'uscita può compiere le varie azioni. In questo caso la libreria ha due comportamenti differenti:

1. Se l'utente inserisce il percorso d'uscita, come mostrato in figura 4, allora la libreria salverà il file generato in quella specifica posizione.

```
#STARTDOCUMENT

%*COMMENTO*%

#STARTINPATH

    <"input">

#ENDINPATH

#STARTOUTPATH

    <"OutputPath1">
    <"OutputPath2">

#ENDOUTPATH

#STARTTYPECONVERSION

    <"CSV">
    <"JSON">
    <"TXT">

#ENDTYPECONVERSION

#STARTNAMESHEET

    <sheetIndex:1,nameSheet:"Foglio_Uno">
    <sheetIndex:2,nameSheet:"Foglio_Due">

#ENDNAMESHEET

#STARTDELETE

    #STARTROW

        <sheetIndex:1,indexDelete:10>
        <sheetIndex:2,indexDelete:9>
        <sheetIndex:3,indexDelete:8>
        <sheetIndex:4,indexDelete:7>

    #ENDROW

    #STARTCOL

        <sheetIndex:1,indexDelete:2>

    #ENDCOL

#ENDELETE

#STARTMODIFYINGCELL

    <sheetIndex:1,row:1,col:1,value:1>
    <sheetIndex:1,row:1,col:1,value:"ProvaStringa">

#ENDMODIFYINGCELL

#ENDDOCUMENT
```

Figura 2: Metalinguaggio.


```
4 #STARTINPATH
5
6 <"/Users/beppeemirra/Desktop/LFC/testConversion.xlsx">
7
8 #ENDINPATH
```

Figura 3: InputPath

2. Se l'utente non inserisce il percorso d'uscita, allora la libreria genererà il file modificato e/o convertito nello stesso percorso del file in input.

Anche in questo caso il file di output è inserito tra le parentesi angolari.

```
10 #STARTOUTPATH
11
12 <"/Users/beppeemirra/Desktop/LFC/">
13
14 #ENDOUTPATH
```

Figura 4: OutputPath

3.3.3 File Type Conversion

Questa terza sezione è obbligatoria. L'utente deve inserire almeno una delle tre estensioni possibili (*.csv*, *.json*, *.txt*), anche in questo caso l'estensione inserita dall'utente è immessa all'interno delle parentesi angolari. Inserito il/i tipi di conversione, come mostrato in figura 5, la libreria genererà i singoli file con tutte le estensioni desiderate dall'utente. Esempio in figura 1.

3.3.4 Rows Cols Delete

Questa quarta sezione non è obbligatoria. In questo caso l'utente può inserire o meno le righe o le colonne che desidera eliminare. In questo caso, come mostrato in figura 6, si ha una struttura ad albero. La parte superiore si occuperà di contenere gli indici delle righe che l'utente vuole eliminare, quella inferiore gli indici delle colonne. Come si può

```
16 #STARTTYPECONVERSION
17
18 <"CSV">
19 <"JSON">
20 <"TXT">
21
22 #ENDTYPECONVERSION
```

Figura 5: type of conversion

notare l'intera sezione *#STARTDELETE ... #ENDDELETE* è suddivisa in due sottosezioni *#STARTROW ... #ENDROW* e *#STARTCOL ... #ENDCOL*. La struttura per inserire i dati è la stessa; anche in questo caso i dati vengono inseriti all'interno delle parentesi angolari. Ma a differenza delle precedenti sezioni, le informazioni da inserire sono due:

1. *sheetIndex*: cioè l'indice del foglio dal quale si vuole eliminare una certa riga o colonna.
2. *indexDelete*: cioè l'indice effettivo della riga o della colonna che si vuole eliminare dal foglio elettronico.

3.3.5 modifying Cell

Questa quinta sezione non è obbligatoria. In questo caso l'utente può inserire o meno le celle che desidera modificare nel file originale. In questo caso, come mostrato in figura 7, la struttura per inserire i dati è differente rispetto alle sezioni precedenti. Anche questo caso i dati vengono inseriti all'interno delle parentesi angolari. Ma a differenza delle precedenti sezioni, le informazioni da inserire sono quattro:

1. *sheetIndex*: indice del foglio nel quale si vuole modificare una cella.
2. *row*: indice della riga della cella che si vuole modificare.
3. *col*: indice della colonna della cella che si vuole modificare.

```
30
31
32 #STARTDELETE
33
34 #STARTROW
35 |
36 |   <sheetIndex:2,indexDelete:20>
37 |
38 #ENDROW
39
40 #STARTCOL
41 |
42 |   <sheetIndex:2,indexDelete:10>
43 |
44 #ENDCOL
45
46 #ENDDELETE
```

Figura 6: rows and cols delete

4. value: valore che si vuole sostituire al posto del valore della cella nel file originale.
Il valore può essere di qualsiasi tipo.

```
#STARTMODIFYINGCELL

<sheetIndex:1,row:2,col:1,value:100>
<sheetIndex:1,row:2,col:2,value:"TestModCell">

#ENDMODIFYINGCELL
```

Figura 7: modifying cell

4 Grammatica

4.1 Approccio iniziale

Come primo approccio per la definizione della grammatica si è scritto un file di input che contenesse tutti i costrutti definiti nella fase di stesura del metalinguaggio. Dal caso più difficile e completo si è passati ad esempi più semplici contenenti solo i costrutti essenziali per far avvenire la conversione del file in input. Quest'ultimo è presente nella sezione *Metalinguaggio* in figura 2. I principali problemi che si sono riscontrati durante la stesura del metalinguaggio sono stati :

- Distinzione tra Token e Regole
- Rispetto della prospezione 1
- Inserimento di vincoli per rendere il linguaggio meno flessibile

4.2 Token

4.2.1 Token di inizio e fine sezione

Per Token strutturali s'intendono quei token che definiscono inizio e fine di una sezione appartenente al metalinguaggio. Ogni sezione definita nel metalinguaggio permette delle operazioni specifiche. Ad esempio non è possibile eseguire operazioni di tipo *DELETE* all'interno di una struttura *STARTINPATH*. Il lavoro di riconoscimento dei Token in questa fase sarà gestita dallo Scanner.

1. STARTDOC

Al token è assegnata la stringa *#STARTDOCUMENT* , che definisce l'inizio delle operazioni di conversione da parte dell'utente.

2. ENDDOC

Al token è assegnata la stringa *#ENDDOCUMENT*, che definisce la fine delle operazioni di conversione da parte dell'utente

3. STARTINPATH

Al token è assegnata la stringa *#STARTINPATH*, che definisce l'inizio della operazione di inserimento del percorso file di input da parte dell'utente.

4. ENDINPATH

Al token è assegnata la stringa *#ENDINPATH*, che definisce la fine della operazione di inserimento del percorso file di input.

5. STARTOUTPATH

Al token è assegnata la stringa *#STARTOUTPUTPATH*, che definisce l'inizio delle operazioni di inserimento dei percorsi file di output.

6. ENDOUTPATH

Al token è assegnata la stringa *#ENDOUTPATH*, che definisce la fine delle operazioni di inserimento percorso file di output.

7. STARTCONVERSION

Al token è assegnata la stringa *#STARTCONVERSION*, che definisce l'inizio delle operazioni di inserimento dei formati dei file di output.

8. ENDCONVERSION

Al token è assegnata la stringa *#ENDTYPECONVERSION*, che definisce la fine delle operazioni di inserimento dei formati dei file di output.

9. STARTNAMESHEET

Al token è assegnata la stringa *#STARTNAMESHEET*, che definisce l'inizio delle operazioni di assegnamento degli indici del file di input al nome del file di output.

10. ENDNAMESHEET

Al token è assegnata la stringa *#ENDNAMESHEET*, che definisce la fine delle operazioni di assegnamento degli indici del file di input al nome del file di output.

11. STARTDELETE

Al token è assegnata la stringa *#STARTDELETE*, che definisce l'inizio delle operazioni di cancellazione di righe e colonne. *#STARTDELETE* al suo interno contiene due sotto sezioni:

- *#STARTROW ... #ENDROW*
- *#STARTCOL ... #ENDCOL*

sotto sezioni dotate rispettivamente delle operazioni di cancellazione righe e cancellazione colonne.

12. ENDDELETE

Al token è assegnata la stringa *#ENDDELETE*, che definisce la fine delle operazioni di cancellazione di righe e colonne.

13. STARTROW

Al token è assegnata la stringa *#STARTROW*, che definisce l'inizio delle operazioni di cancellazione delle righe e deve stare all'interno della sezione *STARTDELETE ... ENDDELETE*.

14. ENDROW

Al token è assegnata la stringa *#ENDROW*, che definisce la fine delle operazioni di cancellazione delle righe.

15. STARTCOL

Al token è assegnata la stringa *#STARTCOL*, che definisce l'inizio delle operazioni di cancellazione delle colonne e deve stare all'interno della sezione *STARTDELETE ... ENDDELETE*.

16. ENDCOL

Al token è assegnata la stringa *#ENDCOL*, che definisce la fine delle operazioni di cancellazione delle colonne.

17. STARTMODIFYINGCELL

Al token è assegnata la stringa *#STARTMODIFYINGCELL*, che definisce l'inizio delle operazioni di modifica del contenuto delle celle.

18. ENDMODIFYINGCELL

Al token è assegnata la stringa *#ENDMODIFYINGCELL*, che definisce la fine delle operazioni di modifica del contenuto delle celle.

4.2.2 Token operazionali

Per token operazionali s'intendono quei token, che definiscono delle keywords, per le operazioni che l'utente può eseguire tramite il metalinguaggio. Le operazioni che possono essere eseguite all'interno di una sezione hanno il seguente standard di costruzione *< ... >*, ed in base alla sezione in cui siamo prenderà una determinata forma. I principali token che accomunano quasi tutte le operazioni sono :

- **BAOPEN**

Al token è assegnato il carattere "<" che identifica l'inizio dell'inserimento dei dati da parte dell'utente all'interno della sezione.

- **BACLOSE**

Al token è assegnato il carattere ">" che identifica la fine dell'inserimento dei dati da parte dell'utente all'interno della sezione.

- **COLON**

Al token è assegnato il carattere ":" che rappresenta la separazione tra i token operazionali e i dati inseriti dall'utente. es (sheetIndex (token strutturale) : 1 (indice del foglio inserito dall'utente))

- **COMMA**

Al token è assegnato il carattere "," che rappresenta la separazione tra i vari campi all'interno delle parentesi triangolari.

- **INT**

token di tipo *valore*.

- **STRINGA**

Token di tipo *valore* che rappresenta le stringe. Quest'ultimo è racchiuso all'interno dei doppi apici ("Stringa")

Token operazionali:

1. **SHEETINDEX**

Al token è assegnata la stringa *'sheetIndex'*, quest'ultima definisce lo spazio in cui l'utente può inserire l'indice del foglio elettronico che la libreria deve prendere in considerazione.

2. **NEWNAMESHEET**

Al token è assegnata la stringa *'nameSheet'*, quest'ultima definisce lo spazio in cui l'utente può inserire l'indice del foglio elettronico a cui desidera modificare il nome.

3. **INDEXDELETE**

Al token è assegnata la stringa *'indexDelete'*, quest'ultima definisce lo spazio in cui l'utente può inserire gl'indici delle righe e/o colonne che desidera eliminare dal foglio elettronico.

4. **COL**

Al token è assegnata la stringa *'col'*, quest'ultima definisce lo spazio in cui l'utente può inserire l'indice della colonna che identifica la cella da modificare all'interno del foglio elettronico.

5. **ROW**

Al token è assegnata la stringa *'row'*, quest'ultima definisce lo spazio in cui l'utente può inserire l'indice della riga che identifica la cella da modificare all'interno del foglio elettronico.

6. NEWVALUE

Al token è assegnata la stringa *'value'*, quest'ultima definisce lo spazio in cui l'utente può inserire il nuovo valore da sostituire ad una specifica cella all'interno del foglio elettronico.

Altri Token :

- **COMMENT**

Al token è assegnato il simbolo *'%*'* quest'ultimo definisce i commenti su una sola riga. Inoltre tutto ciò che è inserito all'interno dei delimitatori *'%*'*... *'*%'* verrà riconosciuto come commento.

- **WS**

Token utilizzato per gestire gli spazi bianchi.

- **SCAN_ERROR**

Al token sono assegnati tutti quelle stringhe e/o caratteri che non sono stati riconosciuti come token, è utile per la gestione degli errori.

4.3 Regole

I token identificati nella fase di *scanning* vengono raggruppati gerarchicamente tramite delle strutture ad albero che specificano la struttura sintattica delle frasi del programma. Si cerca, quindi, di mettere insieme le parole identificate in fase di analisi lessicale per formare parole sintatticamente corrette. Questo è ciò che s'intende quando si parla di regole. Piccola parentesi sulla notazione di ANTLRWorks, i token sono scritti in maiuscolo, mentre le regole sono scritte in minuscolo. Nella teoria studiata durante il corso la notazione era esattamente l'opposta.

1. language

```

language :
    STARTDOC wsc body wsc ENDDOC
    ;

```

Figura 8: language

Rappresenta l'assioma del linguaggio, ovvero la regola da cui il parser inizierà a lavorare. Rappresenta la sezione principale che conterrà tutte le altre sezioni, quest'ultima è composta da :

- *STARTDOC*
- **wsc**
Regola che contiene (*WS* || *COMMENT* || **scanerror**)^{*}, permette all'utente di scrivere le sezioni successive dopo un numero arbitrario di commenti e/o di spazi.
- **body**
Regola che contiene tutte le altre sotto sezioni
- *ENDDOC*

2. **body**

```

body :
    inpath
    wsc
    outpath?
    wsc
    conversion
    wsc
    namesheet?
    wsc
    delete?
    wsc
    modifyingcell?
    ;

```

Figura 9: body

Composto da :

- **inpath**
Regola per l'inserimento del path di input
- **outpath**
Regola per l'inserimento del path di output
- **conversion**
Regola per l'inserimento dei tipi di conversione desiderati
- **namesheet**
Regola per l'inserimento del nome da modificare dei singoli fogli elettronici presenti all'interno del file di Microsoft excel
- **delete**
Regola per la cancellazione sia di righe che di colonne all'interno del singolo foglio elettronico
- **modifyingcell**
Regola per la modifica del contenuto delle celle all'interno del singolo foglio elettronico

tra ognuna di queste regole sarà presente un **wsc**.

3. **inpath**

```
inpath :  
      STARTINPATH path ENDINPATH wsc  
      ;
```

Figura 10: inpath

Composto da :

- *STARTINPATH*

- **path**
- *ENDINPATH*
- **wsc**

4. **outpath**

```
outpath :  
    STARTOUTPATH path+ ENDOUTPATH  
    ;
```

Figura 11: outpath

Composto da :

- *STARTOUTPATH*
- **path**⁺
- *ENDOUTPATH*

5. **path**

```
path :  
    wsc BAOPEN STRINGA BACLOSE wsc  
    ;
```

Figura 12: path

Regola in comune ad *inpath* ed *outpath*, composto da :

- *BAOPEN*
- *STRINGA*
- *BACLOSE*

Non sono ammessi spazi o commenti dove viene inserito il percorso del file in input, ma si possono inserire prima di *BAOPEN* e dopo *BACLOSE*.

6. conversion

```
conversion :
    STARTCONVERSION wsc type wsc ENDCONVERSION
    ;
```

Figura 13: conversion

Composto da :

- *STARTCONVERION*
- **type**
regola che contiene le effettive estensioni in cui l'utente vuole convertire i singoli fogli elettronici presenti all'interno del file Microsoft excel.
- *ENDCONVERSION*

Prima e dopo *type* è possibile inserire spazi e commenti

7. type

```
type :
    BAOPEN STRINGA BACLOSE wsc
    (BAOPEN STRINGA BACLOSE) ? wsc
    (BAOPEN STRINGA BACLOSE) ?
    ;
```

Figura 14: type

Composto da :

- *BAOPEN*

- *STRINGA*
- *BACLOSE*

Sequenza di token che nel caso più complesso viene eseguita per 3 volte. La prima è obbligatoria, mentre le altre 2 sono opzionali. Le tre sequenze di token sono separate da *wsc*.

8. **namesheet**

```
namesheet :
    STARTNAMESHEET sheet wsc ENDNAMESHEET
    ;
```

Figura 15: namesheet

Composto da :

- *STARTNAMESHEET*
- **sheet**
Regola che permette di inserire all'utente il nuovo nome da modificare nei singoli fogli elettronici
- **wsc**
- *ENDNAMESHEET*

9. **sheet**

```
sheet :
    (wsc BAOPEN SHEETINDEX COLON (INT|STRINGA)
    COMMA NEWNAMESHEET COLON STRINGA BACLOSE wsc)+
    ;
```

Figura 16: sheet

Composto da :

- *BAOPEN*
- *SHEETINDEX*
- *COLON*
- *INT*
- *COMMA*
- *NAMESHEET*
- *STRINGA*
- *BACLOSE*

Questa sequenza di token è all'interno di $(...)^+$, inoltre tra le varie sequenze è presente la regola *wsc*

10. delete

```
delete :  
    STARTDELETE wsc row? wsc col? wsc ENDDELETE  
    ;
```

Figura 17: delete

Composto da :

- *STARTDELETE*
- **wsc**
- **row?**
- **col?**
- *ENDDELETE*

Delete contiene le regole per l'eliminazione delle righe *row* e/o colonne *col*. Questi ultimi sono opzionali in quanto sono a discrezione dell'utente. L'utente può usare anche solo una tra e due sottosezioni

11. row

```
row      :  
          STARTROW todelete ENDROW  
          ;
```

Figura 18: row

Composto da :

- *STARTROW*
- **todelete**
- *ENDROW*

12. col

```
col      :  
          STARTCOL todelete ENDCOL  
          ;
```

Figura 19: col

Composto da :

- *STARTCOL*
- **todelete**
- *ENDCOL*

13. todelete


```
todelete :
    (wsc BAOPEN SHEETINDEX COLON (INT | STRINGA) COMMA
    INDEXDELETE COLON INT BACLOSE wsc )+
    ;
```

Figura 20: todelete

Composto da :

- *BAOPEN*
- *SHEETINDEX*
- *COLON*
- *INT*
- *COMMA*
- *INDEXSHEET*
- *BACLOSE*

Questa sequenza di token sta all'interno di $(...)^+$, inoltre tra le varie sequenze è presente la regola *wsc*. Regola dove avviene l'effettiva cancellazione.

14. **modifyingcell**

```
modifyingcell :
    STARTMODIFYINGCELL modcell ENDMODIFYINGCELL
    ;
```

Figura 21: modifyingcell

Composto da :

- *STARTMODIFYINGCELL*
- **modcell**

- *ENDMODIFYINGCELL*

15. **modcell**

```
modcell :
    (wsc BAOPEN SHEETINDEX COLON (INT | STRINGA) COMMA
    ROW COLON (INT | STRINGA) COMMA COL COLON (INT | STRINGA) ) COMMA
    NEWVALUE COLON (INT | STRINGA) BACLOSE wsc )+
    ;
```

Figura 22: modcell

Composto da :

- *BAOPEN*
- *SHEETINDEX*
- *COLON*
- *INT*
- *COMMA*
- *ROW*
- *COL*
- *NEWVALUE*
- *STRINGA*
- *BACLOSE*

Questa sequenza di token sta all'interno di $(...)^+$, inoltre tra le varie sequenze è presente la regola *wsc*. Regola dove avviene l'effettiva modifica della cella.

16. **scanerror**

```
scanerror :  
    SCAN_ERROR  
    ;
```

Figura 23: scanerror

Composto da :

- *SCAN_ERROR*

Utilizzato per la gestione degli errori, verrà mostrato meglio il suo utilizzo successivamente.

5 Semantica

5.1 Costruzione classe Handler

Per estrapolare i dati, inseriti dall'utente nel file di input, si è implementata una classe chiamata *Handler* la quale contiene al proprio interno dei campi che saranno inizialmente istanziati vuoti dall'analisi semantica. *L'Handler* al suo interno ha anche delle funzioni destinate al riempimento dei campi, citati pocanzi, durante le operazioni di *Parsing*.

5.2 Campi della classe Handler

I campi in discussione sono i seguenti :

1. **input_path**

Campo di tipo *String* destinato a contenere la stringa che corrisponde al percorso del file che si intende modificare e/o convertire.

2. **output_path**

Campo di tipo *ArrayList* di *String* che conterrà la stringa e/o le stringhe corrispondenti al percorso dove l'utente desidera salvare i vari file convertiti e/o modificati.

3. **set_conversion**

Campo di tipo *ArrayList* di *String* che potrà contenere al massimo le tre stringhe equivalenti alle tre estensioni in cui modificare il/i fogli elettronici in output. Le tre estensioni sono: *CSV*, *TXT* e *JSON*.

4. **delete_row**

Campo di tipo *ArrayList* di classe di *Delete*. Quest'ultima appositamente implementata per gestire le cancellazioni di righe e/o colonne che l'utente desidera effettuare. La classe *Delete* è implementata in modo generico ma è destinata a contenere 2 campi di tipo token. Questi due oggetti verranno gestiti per rappresentare i valori dei campi *sheetIndex* ed *indexDelete*.

5. **delete_col**

Campo di tipo *ArrayList* di *Delete* ha la stessa implementazione, struttura e funzionamento del campo al punto precedente, *delete_row*.

6. **cellModifyingArrayList**

Campo di tipo *ArrayList* di classe *cellModifyingClass*. Quest'ultima appositamente implementata per gestire le modifiche, che vuole apportare l'utente, di una o più celle all'interno dei singoli fogli elettronici. La classe *cellModifyingClass* è implementata in modo generico ma è destinata a contenere 4 campi di tipo *token*. Questi quattro oggetti verranno gestiti per rappresentare i valori dei campi *sheetIndex*, *row*, *col* e *value*. I quattro campi all'interno della classe istanziati a *Token* verranno poi manipolati per ottenere, rispettivamente, tre campi *Integer* e il quarto che si adatterà al contenuto della cella presente all'interno del foglio elettronico.

5.3 **Funzioni della classe Handler**

Tutte le funzioni che sono state definite all'interno della classe *Handler* contengono campi di tipo *Token* poichè non si è interessati solo al valore dei campi ma anche ad altre informazioni che serviranno in seguito per gestire gli errori tra cui :

- Posizione nella linea
- Valore in stringa
- Numero di *token* associato
- Tipo
- etc...

Le funzioni contenute all'interno della classe *Handler* sono le seguenti :

1. **setInputPath()**

Funzione di tipo *void* avente come campo un oggetto di tipo *Token*.

2. setOutputPath()

Funzione di tipo *void* avente come campo un *ArrayList* di *Token*.

3. setConversionType()

Funzione di tipo *void* avente come campo un *ArrayList* di *Token*.

4. setSheetName()

Funzione di tipo *void* avente come campo una *Map* che utilizza come chiave e valore due oggetti di tipo *Token*.

5. fusionDelete()

Funzione di tipo *void* avente come campo due *ArrayList* di oggetti di tipo *Delete* il quale a sua volta prende due oggetti di tipo *Token*. Questi due *ArrayList* rappresentano rispettivamente le informazioni delle righe e delle colonne da cancellare.

6. printDelete()

Funzione, di controllo, per stampare i valori di righe e colonne.

7. modifyingCellMeth()

Funzione di tipo *void* avente come campo un *ArrayList* di oggetti di tipo *cellModifyingClass* il quale a sua volta prende quattro oggetti di tipo *Token*.

5.4 Utilizzo delle funzioni in ANTLRWorks

Le regole definite per il metalinguaggio vengono interpretate dal *Parser* come delle funzioni. Si può osservare attentamente che il codice generato da *ANTLRWorks*, in base a come viene manipolata la grammatica, utilizza delle variabili di appoggio. Queste ultime hanno lo scopo di contenere il valore ritornato delle funzioni che rappresentano le regole. Infine i valore saranno utilizzati per riempire i campi delle funzioni appartenenti alla classe *Handler*.

```

body      :
    { i = inpath{h.setInputPath(i);}}
    WSC
    { o = outpath{h.setOutputPath(o);}}?
    WSC
    { cT = conversion { h.setConversionType(cT);} }
    WSC

    { nS = namesheet { h.setSheetName(nS);}}?

    WSC
    { d = delete{h.printDelete(d);}}?

    WSC
    { cM = modifyingcell{h.modifyingCellMeth(cM);}}?

    ;

```

Figura 24: body

Come si può osservare dall'immagine 24 all'interno del *body* sono state effettuate alcune modifiche. Dal punto di vista del codice generato questo equivale a creare una funzione per ogni regola presente nel body :

1. **public final void body()**
2. **public final void inpath()**
3. **public final void conversion()**
4. **public final void namesheet()**
5. **public final void delete()**
6. **public final void modifyingcell()**

con la stessa logica verranno create altre funzioni nei livelli più bassi.

5.4.1 Utilizzo della funzione setInputPath()

Alla regola **inpath** viene assegnata una variabile definita "i", che durante la generazione del codice, da parte del *Parser*, equivale a *i=inpath()*. In seguito si chiama, allo stesso livello della regola, la funzione *h.setInputPath(i)*. Questa funzione si aspetta un valore

ritornato dalla funzione *inpath()*. Il tipo del valore ritornato sarà definito all'interno della regola a cui abbiamo assegnato la variabile "i".

```
(i= inpath{h.setInputPath(i);})
```

Figura 25: setInputPath(i)

All'interno della regola **inpath** si devono eseguire le seguenti modifiche :

```
inpath returns [Token p]
:
    STARTINPATH (x=path {p=x;}) ENDINPATH wsc
;
```

Figura 26: setInputPath()

Si osserva nello specifico :

- valore restituito dalla regola **inpath**
- variabile di appoggio che conterrà il valore restituito da **path**
- assegnamento del valore restituito da **path** al valore restituito da **inpath**

Nella generazione del codice da parte del *Parser* in **inpath** sono presenti :

- Definizione di due nuove variabili di tipo *Token*
Token p = null e Token x = null
- variabile di appoggio che conterrà il valore restituito dalla funzione *path()*
x = path()
- Assegnamento del valore restituito da *path()* al valore restituito da *inpath()*
p = x
- valore di ritorno
return p

Infine si passa alla regola **path** dove è presente il *Token* che rappresenta i percorsi inseriti dall'utente.

```

path      returns [Token p]
:
    wsc BAOPEN (x = STRINGA {p = $x; }) BACLOSE wsc
;

```

Figura 27: path

Si osserva nello specifico :

- valore restituito dalla regola **path**
- variabile di appoggio che conterrà il valore restituito dal token **STRINGA**
- assegnamento del contenuto della variabile di appoggio *x* al valore restituito da **path**

p ed *x* del livello **path** sono diversi da quelli definiti nel livello **inpath**. Inoltre si può notare l'utilizzo del simbolo "\$" durante l'assegnamento, quest'ultimo è utilizzato quando si assegna ad una variabile un'altra variabile che contiene un token.

Nella generazione del codice da parte del *Parser* in **path** sono presenti :

- Definizione delle nuove variabili locali di tipo *Token*
Token *p* = null e Token *x* = null
- Assegnamento del valore contenuto in *x* al valore restituito da **path**
p = *x*
- valore di ritorno
return *p*

5.4.2 Utilizzo della funzione setOutputPath()

Alla regola **outpath** viene assegnata una variabile definita "*o*", che durante la generazione del codice, da parte del *Parser*, equivale all'assegnamento *o=outpath()*. In seguito si è

chiamata, allo stesso livello della regola, la funzione *h.setOutputPath(o)*. Questa funzione si aspetta un valore ritornato dalla funzione *outpath()*. Il tipo del valore ritornato sarà definito all'interno della regola a cui abbiamo assegnato la variabile "o".

```
(o = outpath{h.setOutputPath(o);})?
```

Figura 28: setOutputPath(o)

All'interno della regola **outpath** si devono eseguire le seguenti modifiche :

```
outpath returns[ArrayList<Token> p]
@init { p = new ArrayList<Token>(); }
:
STARTOUTPATH (x=path {p.add(x);})+ ENDOUTPATH
;
```

Figura 29: outpath

Si osserva nello specifico :

- valore restituito dalla regola **path**
- inizializzazione dell'*ArrayList* con la parola chiave **@init**, questa serve ad istanziare "p" una ed una sola volta
- assegnamento del contenuto della variabile di appoggio "x" al valore restituito da **path**
- inserimento in lista del valore restituito da **path**

Nella generazione del codice da parte del *Parser* in **outpath** sono presenti :

- Definizione di due nuove variabili di tipo *Token*
Token p = null e *Token x = null*
- creazione istanza della lista "p"

- variabile di appoggio che conterrà il valore restituito dalla funzione *path()*

x = path()

- inserimento del valore restituito da **path** all'interno della lista

p.add(x)

- valore di ritorno

return "p"

Infine si entra nella regola **path** dov'è presente il token che rappresenta il path inserito dall'utente. La descrizione è identica a quella vista per la regola *setInputPath()*.

5.4.3 Utilizzo della funzione *setConversionType()*

Alla regola **conversion** viene assegnata una variabile definita "*cT*", che durante la generazione del codice, da parte del *Parser*, equivale a *cT=conversion()*. In seguito si è chiamata, allo stesso livello della regola, la funzione *h.setConversionType(cT)*. Questa funzione si aspetta un valore ritornato dalla funzione *conversion()*. Il tipo del ritornato sarà definito all'interno della regola a cui abbiamo assegnato la variabile "*cT*".

```
{cT = conversion { h.setConversionType(cT); } }?
```

Figura 30: *setConversionType(cT)*

All'interno della regola **conversion** si devono eseguire le seguenti modifiche :

```
conversion returns [ArrayList<Token> cT]
:
STARTCONVERSION wsc { cType = type{ cT = cType; }} wsc ENDCONVERSION
;
```

Figura 31: *conversion*

Si osserva nello specifico :

- valore restituito dalla regola **conversion**
- assegnamento del contenuto della variabile di appoggio *cType* al valore restituito da **type**
- assegnamento del valore restituito da **type** al valore restituito da **conversion**

Nella generazione del codice da parte del *Parser* in **conversion** sono presenti :

- Definizione di *cT* e *cType* di tipo *ArrayList<Token>*
- variabile di appoggio che conterrà il valore restituito dalla funzione *type()*
cType = type()
- valore di ritorno
return cT

Infine si entra nella regola **type** dove sono presenti i possibili *Token* che rappresentano il formato del file che l'utente desidera avere in output.

```
type returns [ArrayList<Token> cType]
@init { cType = new ArrayList<Token>(); }
:
    BAOPEN {s1 = STRINGA { cType.add(s1); }} BACLOSE wsc
  (BAOPEN {s2 = STRINGA {cType.add(s2); }} BACLOSE)? wsc
  (BAOPEN {s3= STRINGA {cType.add(s3); }} BACLOSE)?
;
```

Figura 32: type

Si osserva nello specifico :

- valore restituito dalla regola **type**
- inizializzazione dell'*ArrayList* con la parola chiave **@init**, questa serve ad istanziare *cType* una sola volta

- variabili di appoggio che conterranno il valore restituito dei token **STRINGA**
- inserimento in lista delle variabile che contengono i token

Nella generazione del codice da parte del *Parser* in **type** sono presenti :

– Definizione di due nuove variabili di tipo *Token* e *ArrayList<Token>*
ArrayList<Token> cType , Token s1 , Token s2, Token s3

– creazione istanza della lista *cType*

- inserimento del valore contenuto nelle variabili *s* all'interno della lista
cType.add(s1),cType.add(s2),cType.add(s3)

– valore di ritorno

return cType

5.4.4 Utilizzo della funzione `setSheetName()`

Alla regola **namesheet** viene assegnata una variabile definita "*cS*", che durante la generazione del codice, da parte del *Parser*, equivale a *nS=conversion()*. In seguito si è chiamata, allo stesso livello della regola, la funzione *h.setSheetName(nS)*. Questa funzione si aspetta un valore ritornato dalla funzione *namesheet()*. Il tipo del valore ritornato sarà definito all'interno della regola a cui abbiamo assegnato la variabile "*nS*".

```
{nS = namesheet { h.setSheetName {nS} ; } } ?
```

Figura 33: `setSheetName(cS)`

All'interno della regola **namesheet** si devono eseguire le seguenti modifiche :

```
namesheet returns [Map<Token, Token> nS]
:
STARTNAMESHEET { nameS = sheet { nS = nameS ; } } wsc ENDNAMESHEET
;
```

Figura 34: `namesheet`

Si osserva nello specifico :

- valore restituito dalla regola **namesheet**
- assegnamento del contenuto della variabile di appoggio *nameS* al valore restituito da **sheet**
- assegnamento del valore restituito da **sheet** al valore restituito da **namesheet**

Nella generazione del codice da parte del *Parser* in **namesheet** sono presenti :

- Definizione di due nuove variabili di tipo *Map<Token,Token>*
`Map<Token,Token> nS, Map<Token,Token> nameS`
- variabile di appoggio che conterrà il valore restituito dalla funzione *sheet()*
`nameS = sheet()`
- valore di ritorno
`return nS`

All'interno della regola **sheet** si devono eseguire le seguenti modifiche :

```
sheet    returns  [Map<Token,Token> nS]

@init { nS = new HashMap<Token, Token>(); }
      :

      (wsc BAOPEN SHEETINDEX COLON ( index = (INT|STRINGA) )
      { System.out.println("tipo : "+ index.getType()); }
      COMMA NEWNAMESHEET COLON
      ( nameSheet = STRINGA {nS.put(index, nameSheet);} ) BACLOSE wsc)+

      ;
```

Figura 35: sheet

Si osserva nello specifico :

- valore restituito dalla regola **sheet**

- inizializzazione di *Map* con la parola chiave **@init**, questa serve ad istanziare *nS* una ed una sola volta
- variabili di appoggio che conterranno il valore restituito dei token **STRINGA** e **INT**
- inserimento nella mappa della coppia delle variabili *chiave* e *valore* che contengono i token

Nella generazione del codice da parte del *Parser* in **sheet** sono presenti :

- Definizione di nuove variabili di tipo *Map<Token,Token>* e *Token*
 $Map<Token,Token> \textit{nS}, \textit{Token nameSheet}, \textit{Token index}$
- creazione istanza della mappa *nS*
- inserimento del valore contenuto nelle variabili *index* e *nameSheet* all'interno della mappa
 $\textit{nS.put(index,nameSheet)}$
- valore di ritorno
 $\textit{return nS}$

5.4.5 Utilizzo della funzione *fusionDelete()*

Questa funzione a differenza delle altre non è costruita a livello del **body**, ma in un sotto livello.

All'interno della regola **delete** vengono assegnate due variabili, per le regole **row** e **col**, nel seguente modo: $r = \textbf{row}$ e $c = \textbf{col}$. In seguito si è chiamata, allo stesso livello della regola, la funzione $h.fusionDelete(r,c)$, quindi ciò che viene restituito da queste due ultime regole viene utilizzato all'interno della funzione dell'*Handler*. Il tipo del valore ritornato sarà definito all'interno delle regole a cui abbiamo assegnato le corrispondenti variabili r e c .

```
(d = delete{h.printDelete(d);})?
```

Figura 36: printDelete

Nella immagine 36 si può osservare la funzione di controllo che abbiamo utilizzato per stampare i valori contenuti nella lista di *Delete*. Alla regola **delete** si devono eseguire le seguenti modifiche :

```
delete returns [ArrayList<Delete<Token,Token>> con]

@init{con = new ArrayList<>();}
:

STARTDELETE wsc (r = row)? wsc (c = col)? wsc ENDELETE
{con.addAll(r);con.addAll(c);h.fusionDelete(r,c);}

;
```

Figura 37: delete

- valore restituito dalla regola **delete**
- inizializzazione di *ArrayList* di *Delete* con la parola chiave **@init**, questa serve ad istanziare *con* una ed una sola volta
- variabili di appoggio che conterranno il valore restituito dalle regole **row** e **col**
- inserimento nella lista dei valori restituiti dalle funzioni **row** e **col**

Nella generazione del codice da parte del *Parser* in **delete** sono presenti :

- Definizione di nuove variabili di tipo *ArrayList<Delete<Token,Token>>*
ArrayList<Delete<Token,Token>> con , r , c
- creazione istanza della lista *con*
- chiama della funzione della classe *Handler*
- valore di ritorno
return con

All'interno della regola **col** si devono eseguire le seguenti modifiche :

```
col      returns  [ArrayList<Delete<Token,Token>> c]
:
      STARTCOL {dc = todelete{c=dc;}} ENDCOL
;

```

Figura 38: col

All'interno della regola **row** si devono eseguire le seguenti modifiche :

```
row      returns  [ArrayList<Delete<Token,Token>> r]
:
      STARTROW {dr= todelete{r=dr;}} ENDROW
;

```

Figura 39: row

Si può osservare che entrambe le regole richiamano **todelete** che rappresenta la foglia dove si andrà, effettivamente a prendere i token utili per l'operazione di cancellazione.

All'interno della regola **todelete** si devono eseguire le seguenti modifiche :

```
todelete returns  [ArrayList<Delete<Token,Token>> d]

@init{d = new ArrayList<>();}
:

    (wsc BAOPEN SHEETINDEX COLON {index = (INT | STRINGA)} COMMA INDEXDELETE COLON
    {del = INT{Delete p= new Delete(index,del); d.add(p);}}
    BACLOSE wsc )+
;

```

Figura 40: todelete

Si osserva nello specifico :

- valore restituito dalla regola **todelete**

- inizializzazione di *ArrayList* con la parola chiave **@init**, questa serve ad istanziare *d* una sola volta
- variabili di appoggio che conterranno il valore restituito dei token **INT**
- inserimento nella lista dell'oggetto *Delete*

Nella generazione del codice da parte del *Parser* in **col**, **row** e **todelete** viene utilizzata la stessa logica vista per le regole precedenti.

5.4.6 Utilizzo della funzione **modifyingCellMeth()**

Alla regola **modifyingcell** viene assegnata una variabile definita "*cM*", che durante la generazione del codice da parte del *Parser* equivale a *cM=modifyingcell()*. In seguito si è chiamata, allo stesso livello della regola, la funzione *h.modifyingCellMeth(cM)*. Questa funzione si aspetta un valore ritornato dalla funzione *modifyingcell()*. Il tipo del valore ritornato sarà definito all'interno della regola a cui si è assegnata la variabile "*cM*"

```
{cM = modifyingcell{h.modifyingCellMeth(cM);}} ?
```

Figura 41: **modifyingCellMeth(cM)**

All'interno di **modifyingcell** si devono eseguire le seguenti modifiche :

```
modifyingcell returns [ArrayList<cellModifyingClass<Token,Token,Token,Token>> aCM]
:
STARTMODIFYINGCELL {aCMM = modcell{aCM = aCMM;}} ENDMODIFYINGCELL
;
```

Figura 42: **modifyingcell**

Si osserva nello specifico :

- valore restituito da **modifyingcell**
- variabili di appoggio che conterranno il valore restituito dei token **STRINGA** e **INT**

- assegnamento del contenuto della variabile di appoggio x al valore restituito da **path**

All'interno di **modcell** si devono eseguire le seguenti modifiche :

```
modcell returns [ArrayList<cellModifyingClass<Token,Token,Token,Token>> aCM]

@init {aCM = new ArrayList<>();}
:
    (wsc BAOPEN SHEETINDEX COLON (ind = (INT | STRINGA)) COMMA
    ROW COLON (rw = (INT | STRINGA)) COMMA COL COLON ( cl = (INT | STRINGA)) COMMA
    NEWVALUE COLON ( vle = (INT | STRINGA)
    {cellModifyingClass ce = new cellModifyingClass(ind,rw,cl,vle); aCM.add(ce);})
    BACLOSE wsc )+
```

Figura 43: modcell

Si osserva nello specifico :

- valore restituito da **modcell**
- inizializzazione dell'*ArrayList* con la parola chiave **@init**, questa serve ad istanziare *aCM* una sola volta
- variabili di appoggio che conterranno il valore restituito dei token **STRINGA** e **INT**
- inserimento dei valore dei token all'interno della lista

Nella generazione del codice da parte del *Parser* in **modifyingcell** e **modcell** viene utilizzata la stessa logica vista per le regole precedenti.

6 Gestione degli errori

6.1 Errori da evitare

L'utente durante la stesura del metalinguaggio deve stare attento ad un insieme di errori che possono essere classificati come :

1. Errori Lessicali :

Sono errori che riguardano come vengono utilizzate le parole all'interno del metalinguaggio. Questo tipo di errore può essere scaturito da una parola inesistente o da una scritta in modo scorretto. Quindi le parole inserite dall'utente, che creano un errore lessicale, non sono riconosciute come *token* appartenenti al metalinguaggio.

2. Errori Sintattici :

Sono errori che riguardano le parole chiave del metalinguaggio, ovvero i *token*. Quest'ultimi nella scrittura del metalinguaggio sono utilizzati in maniera errata . Quindi la sequenza di *token* per definire sezioni e/o operazioni all'interno del metalinguaggio non viene riconosciuta.

3. Errori semantici :

Sono errori che si verificano durante l'esecuzione del codice anche se il metalinguaggio è stato scritto correttamente. Tali errori sono riferiti ai campi che l'utente inserisce per effettuare le modifiche del file in input.

6.2 Gestione errori lessicali

Per la gestione degli errori lessicali si è utilizzato un *token* definito **SCAN_ERROR**, il quale ha la funzione di assegnare a tutti quei caratteri non riconosciuti dal *Lexer* un *token* che poi verrà gestito per avvisare l'utente . In questa fase di segnalazione è risultato estremamente importante l'utilizzo di oggetti di tipo *Token* per riuscire a stampare in output l'esatta posizione dell'errore commesso dall'utente.

6.3 Gestione errori sintattici

Per la gestione degli errori sintattici metà del lavoro è svolto dal parser stesso il quale manda un messaggio di "mismatch" nel caso in cui una sequenza di token riconosciuti non coincide con le regole definite all'interno della grammatica. Per quanto riguarda le sequenze di token con all'interno dei token non riconosciuti si è utilizzato una regola definita **scanerror** il quale contiene il *token* `SCAN_ERROR`. La gestione è avvenuta grazie all'utilizzo della funzione `h.printerror()` che appartiene all'*Handler*. Questa funzione si occupa di intercettare i *token* non riconosciuti all'interno di una sequenza di *token* che definisce una sezione e/o un'operazione.

6.4 Gestione errori Semantici

Per la gestione degli errori semantici si sono utilizzati i campi della classe *Handler* che sono riempiti durante la fase di *Parsing*. I controlli sono stati possibili perché i campi di questa classe sono di tipo *Token* e di conseguenza è possibile eseguire un controllo approfondito sul tipo e sul contenuto inserito dall'utente grazie ai metodi `getText()` e `getType()`. Nel caso in cui l'inserimento da parte dell'utente sia effettivamente sbagliato dal punto di vista semantico, la classe *Token* permette di segnalare con precisione la posizione in termini di righe e colonne grazie ai metodi `getLine()` e `getCharPositionInLine()`. Sono state create anche delle classi per la gestione delle eccezioni, nello specifico abbiamo la classe *InputErrorException* e *OutputErrorException* che estendono entrambi la super classe *FileNotFoundException*.

6.5 Esempi di errori

In questo sottocapitolo verranno mostrati degli esempi di errori sia lessicali che sintattici. Questi errori sono stati volutamente commessi al fine di mostrare all'utente finale il comportamento dell'applicazione a seguito di un errore. Gli errori nello specifico sono stati inseriti nella sezione *Type Conversion* (responsabile della conversione dei file da excel ad

altra estensione) e nella sezione *Modify Cell* (sezione dedicata alla modifica di una o più celle specifiche del file excel dato in input alla libreria).

6.5.1 Errore su Type Conversion - Errore Semantico

Il test effettuato consiste nell'inserire all'interno della sezione *TYPECONVERSION* una stringa che non rappresenta un formato file.

```
#STARTTYPECONVERSION

    <"CSV"> <"TT">
    <"JSON">

#ENDTYPECONVERSION
```

Figura 44: Error Type Conversion

Questo tipo di errore è stato gestito, infatti in output avviene la conversione del file definito nella sezione *INPATH* solo nei formati scritti correttamente. Quindi non essendo disponibile un formato *TT* non è possibile eseguire la conversione.

```
Non sono presenti errori all'interno della grammatica
numero di fogli: 2
C:\Users\user\Desktop\Universit \Linguaggi_Progetto_17_10\output\Foglio_1.csv has been created.
C:\Users\user\Desktop\Universit \Linguaggi_Progetto_17_10\output\Sheet2.csv has been created.
ERROR: type conversion  "'TT'" is not supported or it wrong
numero di fogli: 2
C:\Users\user\Desktop\Universit \Linguaggi_Progetto_17_10\output\Foglio_1.json has been created.
C:\Users\user\Desktop\Universit \Linguaggi_Progetto_17_10\output\Sheet2.json has been created.
```

Figura 45: Error Type Conversion output

Il messaggio *Non sono presenti errori all'interno della grammatica* sta ad indicare che non sono presenti caratteri sconosciuti all'interno del metalinguaggio scritto dall'utente.

Controllo dal punto di vista del codice :

1. I token all'interno delle operazioni di conversione vengono riconosciuti ed inseriti in una *ArrayList*.
2. Gli elementi della lista vengono letti tramite il metodo *getText()*.

3. Le stringhe vengono confrontate con le stringhe *CSV,JSON,TXT*.
4. Se non c'è il match con le stringhe di riferimento non avviene la conversione e viene generato un messaggio in output.

6.5.2 Errore su Modify Cell - Errore Sintattico/Semantico

Il test effettuato consiste nell'inserire all'interno della sezione *MODIFYINGCELL* nel campo *sheetIndex* una stringa al posto di un valore intero.

```
#STARTMODIFYINGCELL

<sheetIndex:"1",row:2,col:4,value:100>
<sheetIndex:1,row:2,col:4,value:100>

#ENDMODIFYINGCELL
```

Figura 46: Error Modify Cell

Questo tipo di errore è stato gestito, infatti in output la conversione avviene correttamente perché l'operazione contenente questo errore non viene considerata.

```
ERROR LINE: 66, Parametri errati[ index, row, col, value ] = [ "1",2,4, 100], l'input non verra considerato
FINE DEL PARSER
lista degli errori :
Non sono presenti errori all'interno della grammatica
```

Figura 47: Error Modify Cell output

Dal punto di vista sintattico ciò che c'è scritto dentro la sezione viene considerato corretto dal compilatore perché *sheetIndex* accetta una stringa o un intero. Questo però crea ambiguità in quanto la stringa rappresenta un numero, di conseguenza è stato utilizzato il tipo del token per gestirlo. Quindi anche se è un errore sintattico questo viene gestito in modo semantico.

Controllo dal punto di vista del codice :

1. I token all'interno delle operazioni di modifica vengono riconosciuti ed inseriti in una *Map*.
2. Gli elementi della Map devono essere tutte di tipo *INT*.
3. Nel caso in cui fossero di un'altro tipo queste non vengono inserite nella mappa, di conseguenza viene stampato un messaggio di errore.
4. Il codice continua ad andare avanti perché dato che il tipo non ha fatto match con il tipo *INT* non viene considerato.

6.5.3 Errore su Type Conversion - Errore Sintattico

Il test effettuato all'interno della sezione *TYPECONVERSION* consiste nella dimenticanza di una parentesi triangolare in una delle operazioni di conversione.

```
<"CSV"> <"TXT"
<"JSON">
```

Figura 48: Error Type Conversion

Questo tipo di errore è stato gestito, durante l'operazione di *Parsing* il compilatore si accorge che la sequenza di token di una regola non viene eseguita correttamente. Quindi lancia un errore in output con le rispettive righe e colonne. Dato che c'è un errore nella sintassi la conversione non va a buon fine.

```
FINE DEL PARSER
lista degli errori :
*****
1 - Errore sintattico 1 a [25, 16]: Trovato WS ( '
') - missing BACLOSE at '\r' - missing BACLOSE at '\r'
```

Figura 49: Error Type Conversion output

Controllo dal punto di vista del codice :

1. I token all'interno delle operazioni di conversione vengono riconosciuti ed inseriti in una *ArrayList*.
2. La chiusura non viene messa e di conseguenza il parser legge uno spazio.
3. Lo spazio è un token appartenente al metalinguaggio, quindi il parser va avanti ma comunque segnala che c'è un errore in quella determinata prosizione e che si dovrebbe aspettare > che corrisponde al token *BACLOSE*.

6.5.4 Errore su Modify Cell - Errori sintattici

Le figure seguenti mostrano l'errore sintattico (ed il relativo errore di log) a seguito della non chiusura del tag `#ENDMODIFYINGCELL`. Come precedentemente accennato, infatti, l'errore sintattico è la mancata sequenza di token che viene interrotta. Qui la catena viene interrotta perché dopo il tag di apertura ci si aspetta il tag di chiusura con in mezzo una sequenza di stringa che caratterizzano le cella (o cella) da modificare.

```
#STARTMODIFYINGCELL

%*<sheetIndex:"1",row:2,col:4,value:100>*&
<sheetIndex:1,row:2,col:4,value:100>

#ENDDOCUMENT
```

Figura 50: Error Modifying Cell

Questo tipo di errore è stato gestito, durante l'operazione di *Parsing* il compilatore si accorge che la sequenza di token di una regola non viene eseguita correttamente. Quindi lancia un errore in output con le rispettive righe e colonne. Dato che c'è un errore nella sintassi la conversione non va a buon fine.

```
lista degli errori :
*****
1 - Errore sintattico 1 a [80, 1]: Trovato ENDDOC ('#ENDDOCUMENT') - missing ENDMODIFYINGCELL at '#ENDDOCUMENT'
```

Figura 51: Error Modifying Cell output

Controllo dal punto di vista del codice :

1. I token all'interno delle operazioni di modifica vengono riconosciuti ed inseriti in una *ArrayList*.
2. La chiusura della sezione non viene messa e di conseguenza il parser segnala la mancanza del token di chiusura della sezione.
3. Lo spazio è un token appartenente al metalinguaggio, quindi il parser va avanti ma comunque segnala che c'è un errore in quella determinata posizione e che si dovrebbe aspettare *#ENDMODIFYINGCELL* che corrisponde al token *ENDMODIFYINGCELL*.

6.5.5 Errore su STARTNAMESHEET - Errore Lessicale

Il test effettuato all'interno della sezione *STARTNAMESHEET* consiste nell'inserimento di caratteri che non appartengono alla grammatica scritta per il metalinguaggio.

```
<sheetIndex:1,nameSheet:"Foglio_1">  
!£$%&/()  
#ENDNAMESHEET  
%*
```

Figura 52: Errore lessicale

Questo tipo di errore è stato gestito, durante l'operazione di *Parsing* il compilatore si accorge che la sequenza di token di una regola non viene eseguita correttamente. Inoltre sono presenti che token non appartenenti alla grammatica, per questo motivo verranno considerati come token di tipo *SCAN_ERROR*. Dato che c'è un errore nel lessico la conversione non va a buon fine.

```

FINE DEL PARSER
lista degli errori :
ERROR Token Type num: [0] , [row, col] : [38,0], Contenuto del Token: !
ERROR Token Type num: [1] , [row, col] : [38,1], Contenuto del Token: #
ERROR Token Type num: [2] , [row, col] : [38,2], Contenuto del Token: $
ERROR Token Type num: [3] , [row, col] : [38,3], Contenuto del Token: %
ERROR Token Type num: [4] , [row, col] : [38,4], Contenuto del Token: &
ERROR Token Type num: [5] , [row, col] : [38,5], Contenuto del Token: /
ERROR Token Type num: [6] , [row, col] : [38,6], Contenuto del Token: (
ERROR Token Type num: [7] , [row, col] : [38,7], Contenuto del Token: )

```

Figura 53: Errore lessicale output

6.5.6 Warning su STARTOUTPATH - Errore Semantico

Il test effettuato all'interno della sezione *STARTOUTPATH* consiste nell'inserimento di una stringa che però non rappresenta un vero e proprio path.

```

#STARTOUTPATH

    <"C:\Users\user\Desktop\Università\Linguaggi_Progetto_17_10\output">
    <"asd">

#ENDOUTPATH

```

Figura 54: Warning

Questo tipo di errore è stato gestito, durante l'operazione di *Parsing*, nello specifico quando viene letta la stringa viene fatto un controllo tramite l'utilizzo della classe *File*. Dato che è un errore semantico poco rilevante la conversione avviene secondo alcune condizioni descritti nel messaggio di output.

```

WARNING!!!  il percorso : asd e' risultato scorretto...
Se e' l'unico output verr@ messo nel path del file di conversione
Se sono presente altri destinazioni di output, allora non verra' considerato

```

Figura 55: Warning output

6.6 Errori formato Data

All'interno del file excell che si desidera convertire è necessario prestare attenzione all'inserimento delle date. Nello specifico, se vengono inserite, date nel formato **dd/mm/yy**

verranno convertire in una stringa di interi che però non rappresenta la data di partenza. Per questo motivo è consigliato utilizzare il formato **dd-mm-aaaa**.

Item Name	Qty	Item Price	Sold Date
Book	2	10	01-01-2020
Table	1	50	02-01-2021
Lamp	5	100	01-01-2022
Pen	100	20	02-01-2023
Book	2	10	01/01/2024
Table	1	50	02-01-2025
Lamp	5	100	01-01-2026
Pen	100	20	02-01-2027
Book	2	10	01-01-2028
Table	1	50	02-01-2029
Lamp	5	100	01-01-2030
Pen	100	20	02-01-2031
Book	2	10	01-01-2032
Table	1	50	02-01-2033
Lamp	5	100	01-01-2034

Figura 56: Data sbagliata

```
{
  "RowId": 1,
  "Qty": "1.0",
  "Item Price": "50.0",
  "Sold Date": "02-01-2021",
  "RowId": 2,
  "Qty": "5.0",
  "Item Price": "100.0",
  "Sold Date": "01-01-2022",
  "RowId": 3,
  "Qty": "100.0",
  "Item Price": "20.0",
  "Sold Date": "02-01-2023",
  "RowId": 4,
  "Qty": "2.0",
  "Item Price": "10.0",
  "Sold Date": "45292.0",
  "RowId": 5,
  "Qty": "1.0",
  "Item Price": "50.0",
  "Sold Date": "02-01-2025",
  "RowId": 6,
  "Qty": "5.0",
  "Item Price": "100.0",
  "Sold Date": "01-01-2026",
  "RowId": 7,
  "Qty": "100.0",
  "Item Price": "20.0",
  "Sold Date": "02-01-2027",
  "RowId": 8,
  "Qty": "2.0",
  "Item Price": "10.0",
  "Sold Date": "01-01-2028",
  "RowId": 9,
  "Qty": "1.0",
  "Item Price": "0.0",
  "Sold Date": "02-01-2029",
  "RowId": 10,
  "Qty": "5.0",
  "Item Price": "0.0",
  "Sold Date": "01-01-2030",
  "RowId": 11,
  "Qty": "100.0",
  "Item Price": "20.0",
  "Sold Date": "02-01-2031",
  "RowId": 12,
  "Qty": "2.0",
  "Item Price": "10.0",
  "Sold Date": "01-01-2032",
  "RowId": 13,
  "Qty": "1.0",
  "Item Price": "50.0",
  "Sold Date": "02-01-2033",
  "RowId": 14,
  "Qty": "5.0",
  "Item Price": "100.0",
  "Sold Date": "01-01-2034"
}
```

Figura 57: Conversione data sbagliata

7 Manuale Utente

7.1 Introduzione

Questo capitolo intende offrire una linea guida utile all'utente finale al fine di mostrare l'utilizzo del convertitore di file da excel a *csv*, *txt* e *json*. Verranno dunque illustrate le azioni da svolgere per la corretta installazione e il corretto utilizzo del convertitore. L'intero applicazione è costituita da due parti:

- Grammatica (codice java);
- Libreria per la conversione dei file (codice java);

Le due parti sono state combinate in un'unica classe principale che verrà fornita all'utente. Quest'ultima fa sì che il lancio del convertitore avvenga in una sola fase piuttosto che in due separate. L'ambiente di sviluppo preso come riferimento è *Eclipse*.

7.2 Import Progetto su Eclipse

Questo paragrafo illustra i vari passaggi per importare correttamente l'intero progetto sull'IDE di sviluppo. Dopo aver scaricato Eclipse ed avviato il programma, il primo passo da fare è quello di aprire il progetto contenente il codice. Il codice è presente sotto la repository: [GitHub Link](#) . Da GitHub per scaricare il codice ci sono diversi modi:

1. Download zip;
2. Clone da GitHub Desktop;
3. Comando Git Clone da prompt dei comandi.

7.3 Pom.xml

Dopo l'apertura del progetto all'interno di Eclipse, si gestiscono le dipendenze attraverso il file "*pom.xml*". Il POM (*Project Object Model*). Quest'ultimo contiene le informazioni di configurazione per la corretta importazione del progetto (dipendenze, directory di

compilazione, directory di origine, directory di origine di test, plugin, obiettivi...). Se qualche dipendenza manca occorre inserirla manualmente nella sezione `<< dependency >> .. << /dependency >>`. Le dipendenze vengono aggiunte copiando direttamente la sezione interessata dal sito: Maven repository.

Le dipendenze presenti nel progetto sono inserite in tabella 3:

Dipendeza	Link
org.apache.poi	org.apache.poi
commons-io	org.apache.poi
org.apache.logging.log4j	org.apache.logging.log4j
org.slf4j	org.slf4j
net.sf.json-lib	net.sf.json-lib
org.antlr	org.antlr

Tabella 3: Dipendenze Pom.xml

7.4 Maven

E' importante, dopo aver impostato le dipendenze, fare un *"upload project"* attraverso il comando *"Maven"*. Il Maven effettua automaticamente il download delle librerie Java e plug-in dai vari repository definiti. Questo permette di recuperare in modo uniforme i vari file *"JAR"* e di spostare il progetto da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie. Per poter importare tutto correttamente si utilizza il tasto Maven, come rappresentato in figura numero 58, seguendo i seguenti passaggi:

1. *"cliccare"* tasto destro su *"Pom.xml"*;
2. Update Project;
3. OK.

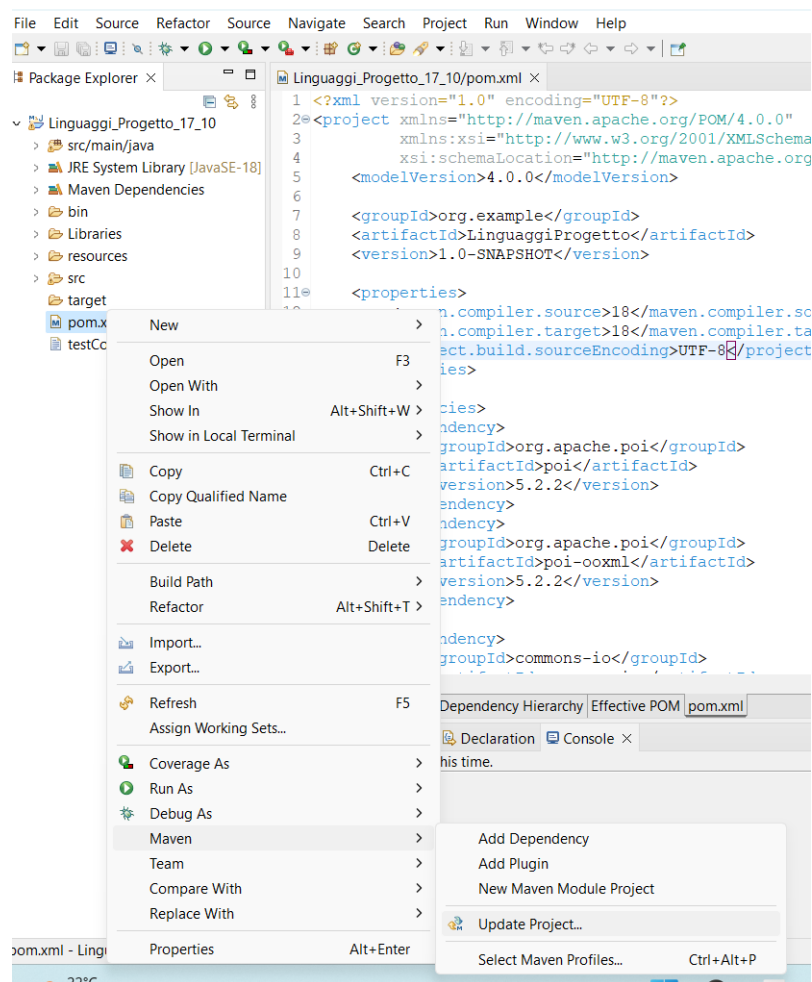


Figura 58: Esempio comando Maven.

7.5 Input.file

Dopo aver importato e settato il progetto con le librerie necessarie al funzionamento, si può iniziare ad utilizzare l'applicazione. Quest'ultima permette di convertire file excel in uno o in tutti i seguenti formati *.json*, *.csv* e *.txt* oppure di modificare parti specifiche del file excel stesso. Il file che permette di mettere a punto le modifiche è l' *"input.file"*. Quest'ultimo, infatti, prende i parametri di input inseriti dall'utente per fornirli alla libreria che si occuperà delle varie modifiche. L'*input.file* si trova nella directory *"resources"* all'interno del progetto.

In seguito, dopo aver compilato i campi desiderati all'interno del file di input è possibile salvare il documento e lanciare il progetto.

7.6 Run del progetto

Il progetto viene lanciato attraverso la classe *"completeMain"*, quest'ultima si trova al di sotto di queste tre sub-directory:

1. *"src"*;
2. *"main"*;
3. *"java"*.

Lanciando la classe, quindi, si otterranno le modifiche indicate nell'input.file. Di default, qualora non venga esplicitamente impostato, l'output path prenderà in automatico il path in input. Pertanto, è consigliabile creare una folder in output per avere un riferimento dei file creati/modificati.

7.7 About Codice

In questa sottosezione vengono mostrati, a scopo illustrativo, i package ed il relativo codice di LFCLibrary (Figura 59) e Grammatica (Figura 60). Le classi contenute nei vari package contengono il codice java che permette il funzionamento della libreria. Il codice, inoltre, è stato suddiviso in base alle funzionalità dello stesso.

7.7.1 Codice Libreria

Il codice della libreria viene suddiviso nei package mostrati in figura:

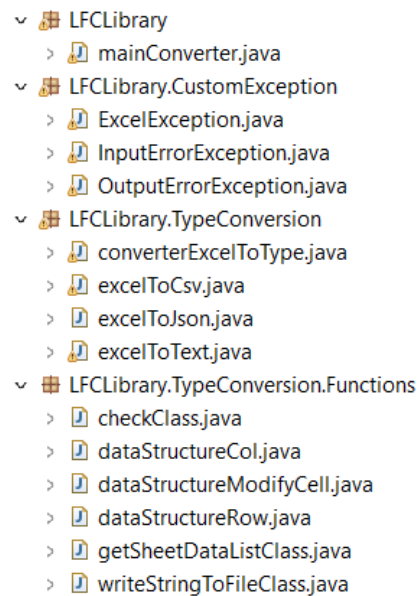


Figura 59: Package Libreria

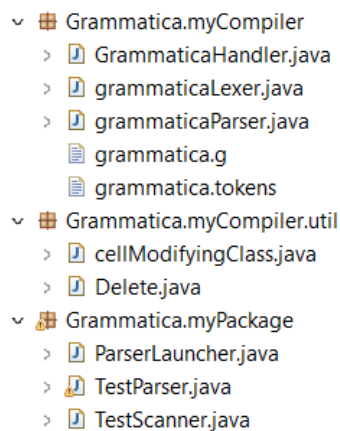


Figura 60: Package Grammatica

7.8 Complete main Class

La classe *"completeMain.java"*, mostrata in Figura 61, permette l'integrazione della libreria e della grammatica. Inoltre, questa classe permette un'usabilità maggiore per l'utente. L'utente, infatti, si concentra solamente sul file *"input.file"*, modifica i parametri interessanti e, dopo aver lanciato la classe, vede l'output nella folder interessata. Questa classe, infat-

ti, evita che l'utente debba preoccuparsi di lanciare prima il parsing e poi successivamente la libreria per la conversione.

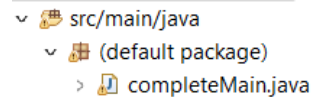


Figura 61: CompleteMain.java

8 esempi di funzionamento

In questo capitolo vengono mostrati all'utente alcuni esempi di running. Lo scopo è quello di dare consapevolezza sull'output atteso in base alle modifiche fatte nell'input.file.

8.1 Primo esempio: NameSheet e TypeConversion

Tra i tag `#STARTTYPECONVERSIONE` ed `#ENDTYPECONVERSION` vengono impostati i tipi di conversione che si desidera apportare al file excel. Mentre tra i tag `#STARTNAMESHEET` ed `#ENDNAMESHEET` è possibile modificare il nome dei singoli fogli elettronici del file excel stesso. Esempio mostrato in figura 62.

```
#STARTTYPECONVERSION
  <"CSV">
  <"JSON">
  <"TXT">
#ENDTYPECONVERSION

%*COMMENTO*%

#STARTNAMESHEET
  %*COMMENTO*%
  <sheetIndex:1,nameSheet:"Test_1">
  <sheetIndex:2,nameSheet:"Test_2">
  %*COMMENTO*%
#ENDNAMESHEET
```

Figura 62: Type conversion/Name sheet

8.2 Secondo esempio: eliminazione di una riga e di una colonna

Come mostrato in figura 63 è possibile definire una riga e/o una colonna da eliminare. La figura mostra il punto esatto da modificare nel *"input.file"* per poter eliminare una riga e una colonna.

```

#STARTDELETE

#STARTROW

<sheetIndex:1,indexDelete:3>

#ENDROW

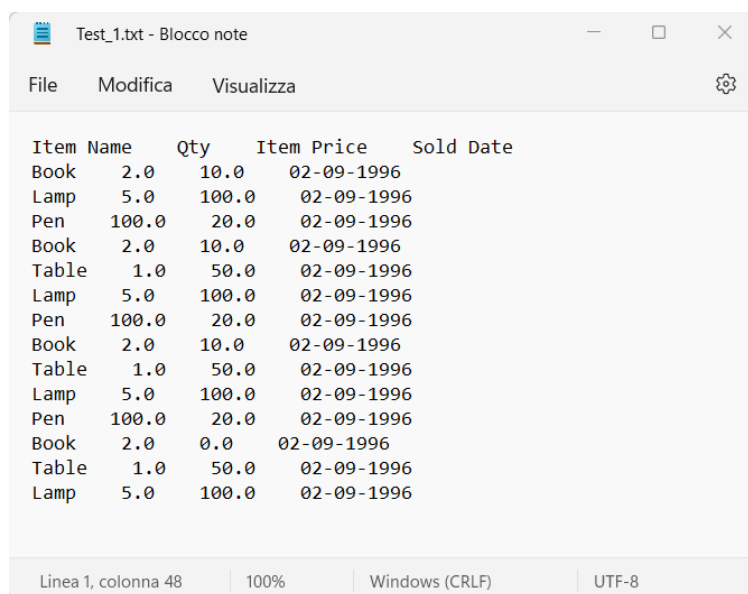
#STARTCOL
<sheetIndex:1,indexDelete:1>
#ENDCOL
#ENDDELETE

```

Figura 63: Output run

8.3 Terzo esempio: file convertito in formato .txt

Nella figura 64 si mostra l'output di un file excel convertito in estensione .txt.



Item Name	Qty	Item Price	Sold Date
Book	2.0	10.0	02-09-1996
Lamp	5.0	100.0	02-09-1996
Pen	100.0	20.0	02-09-1996
Book	2.0	10.0	02-09-1996
Table	1.0	50.0	02-09-1996
Lamp	5.0	100.0	02-09-1996
Pen	100.0	20.0	02-09-1996
Book	2.0	10.0	02-09-1996
Table	1.0	50.0	02-09-1996
Lamp	5.0	100.0	02-09-1996
Pen	100.0	20.0	02-09-1996
Book	2.0	0.0	02-09-1996
Table	1.0	50.0	02-09-1996
Lamp	5.0	100.0	02-09-1996

Figura 64: Output run

8.3.1 Quarto esempio: modifica cella

Le due figure seguent, 65 e 66 mostrano la funzionalità della modifica di una o più celle del file excel in input con il relativo output a seguito del lancio della libreria.

```
#STARTMODIFYINGCELL

<sheetIndex:1,row:14,col:4,value:00>

#ENDMODIFYINGCELL
```

Figura 65: Sezione modifica cella

A	B	C
Item Name,"Qty","Item Price","Sold Date"		
Book,"2.0","10.0","02-09-1996"		
Lamp,"5.0","100.0","02-09-1996"		
Pen,"100.0","20.0","02-09-1996"		
Book,"2.0","10.0","02-09-1996"		
Table,"1.0","50.0","02-09-1996"		
Lamp,"5.0","100.0","02-09-1996"		
Pen,"100.0","20.0","02-09-1996"		
Book,"2.0","10.0","02-09-1996"		
Table,"1.0","50.0","02-09-1996"		
Lamp,"5.0","100.0","02-09-1996"		
Pen,"100.0","20.0","02-09-1996"		
Book,"2.0","0.0","02-09-1996"		
Table,"1.0","50.0","02-09-1996"		
Lamp,"5.0","100.0","02-09-1996"		

Figura 66: Output run