

Indice

1. Prefazione	3
1.1 Version History	3
1.2 Sommario modifiche	3
2. Introduction	4
2.1 Definizioni e glossario	4
2.2 Scopo	4
2.3 Toolchain	5
3. Iterazione 0	6
3.1 Requisiti funzionali	6
3.1.1 Struttura corretta della grammatica	6
3.1.2 Errori lessicali	6
3.1.3 Errori sintattici	7
3.1.4 Errori semantici	7
3.2 Casi d'uso	8
3.2.1 UC1 - Analisi di una grammatica	8
3.3 Struttura del programma	9
3.3.1 Scelta architetturale	9
3.4 Programma di testing	10
4. Iterazione 1	11
4.1 Design Architetturale	11
4.1.1 Class diagram	11
4.2 Testing	13
4.2.1 Descrizione del prodotto software	13
4.2.2 Funzionalità oggetto di test	14
4.2.2.1 Elenco delle funzionalità oggetto di test	14
4.3 Esiti dei test	16
4.3.1 Esiti e copertura dei test di unità	16
4.3.1.1 Esiti dei test per la classe Stato	16
4.3.1.2 Esiti dei test per la classe RegolaDiProduzione	17
4.3.1.3 Esiti dei test per la classe NonTerminale	17
4.3.1.4 Esiti dei test per la classe Solver	18
4.3.1.5 Esiti dei test per la classe Terminale	18
4.3.2 Esiti e copertura dei test di sistema	19
5. Iterazione 2	20
5.1 Design Architetturale	20
5.1.1 Librerie utilizzate	21
5.1.2 Class Diagram	21
5.2 Casi d'uso	23
5.2.1 UC1 - Analisi di una grammatica	23
5.2.2 UC2 - Visualizzazione della grammatica analizzata	23
5.3 Testing	25
5.3.1 Descrizione del prodotto software	25

5.3.2	Funzionalità oggetto di test	26
5.3.2.1	Elenco delle funzionalità oggetto di test	26
5.4	Esiti dei test	28
5.4.1	Esiti e copertura dei test di unità	28
5.4.1.1	Esiti dei test per la classe Stato	28
5.4.1.2	Esiti dei test per la classe RegolaDiProduzione	29
5.4.1.3	Esiti dei test per la classe NonTerminale	29
5.4.1.4	Esiti dei test per la classe Solver	30
5.4.1.5	Esiti dei test per la classe Terminale	30
5.4.2	Esiti e copertura dei test di sistema	31
6.	Strutture dati e algoritmi	32
6.1	Terminale	32
6.1.1	Dati	32
6.1.2	Operazioni	32
6.2	NonTerminale	32
6.2.1	Dati	32
6.2.2	Operazioni	33
6.2.3	Algoritmi	33
6.3	RegolaDiProduzione	33
6.3.1	Dati	33
6.3.2	Operazioni	34
6.4	Stato	34
6.4.1	Dati	34
6.4.2	Operazioni	34
6.4.3	Algoritmi	35
7.	Sviluppi futuri	36

1. Prefazione

1.1 Version History

Versione 0.0 - 16/04/2019

1.2 Sommario modifiche

Definizione dei requisiti funzionali, dei casi d'uso, della struttura generale del software, della struttura dei test e degli strumenti utilizzati per lo sviluppo.

2. Introduction

2.1 Definizioni e glossario

Grammatica libera da contesto

In informatica e in linguistica, una grammatica libera dal contesto è una grammatica formale in cui ogni regola sintattica è espressa sotto forma di derivazione di un simbolo a sinistra a partire da uno o più simboli a destra.

Ciò può essere espresso con due simbolismi equivalenti:

1) $S := \alpha$

2) $S \rightarrow \alpha$

dove S è un simbolo detto *non terminale*, sostituibile con altri simboli non terminali e/o con simboli terminali, e α è una sequenza di simboli non terminali e/o terminali, ossia simboli che non possono essere sostituiti con null'altro.

L'espressione "libera dal contesto" si riferisce al fatto che il simbolo non terminale S può sempre essere sostituito da α , indipendentemente dai simboli che lo precedono o lo seguono; un linguaggio formale si dice libero dal contesto se esiste una grammatica libera dal contesto che lo genera.

Parser

Un parser LR è un parser di tipo bottom-up per grammatiche libere da contesto che legge il proprio input partendo da sinistra verso destra, producendo una derivazione a destra. Laddove indicato come parser $LR(k)$, il k si riferisce al numero di simboli letti (ma non "consumati") per prendere le decisioni di parsing.

2.2 Scopo

Scopo del programma che si andrà a realizzare è il riconoscimento di una grammatica $LR(1)$ contenuta in un file di input selezionato dall'utente.

2.3 Toolchain

Per lo sviluppo di questo progetto è stata scelta la seguente toolchain:

- Git, utilizzato per il controllo di versione, così da consentire il lavoro a più mani sul progetto anche qualora non ci si trovi tutti nello stesso ambiente. L'utilizzo di Git prevederà la creazione di branch secondari per gli sviluppi incrementali del programma, così da conservare sempre una versione funzionante del programma stesso all'interno del branch "master" della repository ospitante il progetto;
- GitKraken, client che fornisce un'interfaccia grafica intuitiva per l'utilizzo di github, implementa funzioni che velocizzano l'utilizzo di git implementando un grafo rappresentante l'avanzamento della repository.
- Modelio, utilizzato per la creazione di diagrammi UML, si è scelto questo tool visto che implementa dei tool che partendo da codice java genera il modello UML del codice fornito. Essendo questo progetto un'espansione di un progetto già in essere realizzato per il corso Linguaggi Formali e Compilatori questo tool agevolerà la creazione del modello UML del codice già esistente;
- TexMaker, utilizzato per la redazione della documentazione;
- Eclipse, utilizzato per lo sviluppo dell'applicazione, inoltre integra dei tool utili al fine di analizzare la copertura del codice;
- ANTLR, tool per la generazione di parser, una volta definita una grammatica questo genera un parser e tutto il codice di contorno per l'individuazione di errori lessicali e semantici;
- Junit, tool utilizzato per la definizione dei casi di test e l'esecuzione dei casi stessi;
- SpotBugs, PMD, plugin di Eclipse utilizzati per l'analisi statica del codice.

3. Iterazione 0

3.1 Requisiti funzionali

Il programma deve consentire all'utente le seguenti azioni:

- selezione del file di input da sottoporre al parsing e all'identificazione.

Il programma deve fornire le seguenti funzionalità:

- effettuare il parsing del file ricevuto in input, individuando eventuali errori sintattici, lessicali o semantici;
- qualora vengano individuati errori di qualsiasi genere nella fase di parsing, il programma deve comunicare i dettagli relativi agli errori individuati all'utente;
- qualora non vengano individuati errori nella fase di parsing, il programma deve procedere nell'identificare la grammatica come grammatica $LR(1)$ o non $LR(1)$.

3.1.1 Struttura corretta della grammatica

Il programma deve riconoscere come formalmente corrette (quindi prive di errori sintattici, lessicali e/o grammaticali) soltanto grammatiche che presentino la seguente struttura:

- una prima regola pr che abbia come elemento di sinistra il non terminale S_0 , definita come segue

$$SZ \ EQ \ NT \ TER \ SC$$

- altre $n \geq 1$ regole di produzione ar , che formano il resto della grammatica, definite come segue

$$NT \ EQ \ (NT|CT)^* \ SC$$

I blocchi componenti le regole appena definite sono così traducibili:

Simbolo	Caratteri
SZ	S_0
EQ	$- \ > \ \ :=$
NT	$A \ \dots \ Z$
CT	$a \ \dots \ z \ \ 0 \ \dots \ 9 \ \ + \ \ - \ \ * \ \ /$
TER	$/swa \ \ /cjswa$
SC	$;$

Tabella 1: Corrispondenza tra caratteri della grammatica e blocchi di definizione delle regole

Nota: Per la definizione della struttura delle regole è stata utilizzata la notazione formale di Backus-Naur estesa (EBNF).

3.1.2 Errori lessicali

L'utilizzo di qualsiasi carattere non riconducibile alla colonna "Caratteri" della Tabella 1 corrisponde a un errore lessicale.

3.1.3 Errori sintattici

Gli errori sintattici sono dati dal mancato rispetto della struttura delle regole *pr* e *ar* come definite nel paragrafo "Struttura corretta della grammatica" a pagina 6.

3.1.4 Errori semantici

Gli errori semantici si verificano nei seguenti casi:

- nella grammatica è presente un carattere non terminale che non presenta regole di produzioni associate;
- nella grammatica è presente una regola duplicata (nota bene questo non è un errore bloccante).

3.2 Casi d'uso

In questa fase preliminare si è identificato un solo caso d'uso riportato di seguito

3.2.1 UC1 - Analisi di una grammatica

Descrizione: identificazione di una grammatica $LR(1)$

Attori coinvolti: utente

Precondizioni: esistenza del file d'input contenente la definizione della grammatica

Trigger: necessità di identificare una grammatica

Post condizioni: la classificazione della grammatica viene mostrata a schermo

Procedimento standard:

1. l'utente avvia il programma;
2. l'utente seleziona il file di input desiderato;
3. il programma esegue il parsing e l'analisi del file di input;
4. il programma mostra a schermo l'esito dell'analisi della grammatica;
5. l'utente chiude il programma.

Procedimenti alternativi o eccezioni:

- allo step 3, il programma rileva errori nella grammatica
viene mostrato un messaggio d'errore contenente informazioni relative all'errore individuato, l'utente chiude il programma e, corretta la grammatica, la procedura riparte dallo step 1

3.3 Struttura del programma

Il programma, definito in linguaggio Java, sarà composto di tre moduli:

- un primo modulo che verrà implementato all'interno del modulo generato da ANTLR, che si occuperà della lettura di un file .txt contenente la grammatica da identificare.
- un modulo generato da ANTLR, che si occuperà del parsing e dell'individuazione di errori lessicali e sintattici;
- un modulo scritto manualmente che si occuperà dell'analisi della grammatica e della sua classificazione.

L'interfacciamento con l'utente nella prima versione avverrà attraverso una CLI (Command Line Interface).

3.3.1 Scelta architetturale

Vista la struttura del programma si è scelto la struttura ideale è di tipo pipe and filter dove si possono identificare specifici moduli e ogni modulo genera dei risultati intermedi che verranno poi utilizzati dai moduli successivi.

In figura è riportata una bozza della struttura implementata nella fase 1.

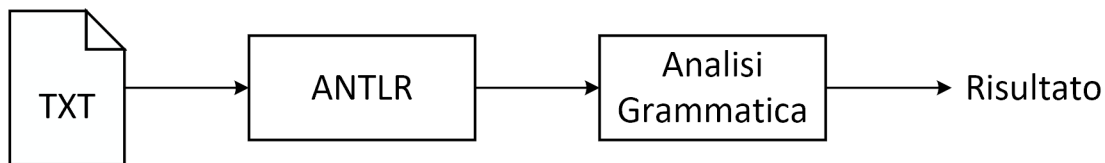


Figura 1: Bozza dell'architettura

3.4 Programma di testing

Il testing del programma sarà incentrato sul modulo *non* generato da ANTLR, assumendo che i controlli su quello generato da ANTLR siano già stati svolti dal produttore del tool, e sarà composto di:

- casi di test per ogni classe generata scritti in JUnit, per i quali sarà richiesta una coverage del codice $\geq 95\%$;
- verifiche di coverage del codice scritto tramite l'esecuzione del programma con diversi input, al fine di garantire l'assenza di dead code;
- analisi statica del codice tramite strumenti quali SpotBugs e PMD.

A ogni revisione del codice, nonché in precedenza al rilascio di una nuova versione, saranno svolti test di non regressione per evitare l'introduzione di nuovi difetti nel codice.

4. Iterazione 1

In questa iterazione è stata creata la versione a CLI (Command Line Interface), è stata inserita un'unica interfaccia grafica utilizzata per permettere all'utente la selezione del file che contiene la grammatica da analizzare, l'esito dell'analisi è riportato all'interno della CLI indicando se la grammatica è o meno $LR(1)$, l'elenco di tutti gli stati ed infine un elenco di tutte le transizioni.

4.1 Design Architetture

In questa prima versione il design architetture è rimasto quello previsto nella iterazione 0.

4.1.1 Class diagram

Per la generazione del class diagram della fase 1 è stato utilizzato il tool "java designer" di Modelio che permette di estrapolare dal codice già presente il diagramma UML delle classi. In particolare si può notare come la struttura sequenziale del programma è molto simile a quella ipotizzata nella fase zero a pagina 9.

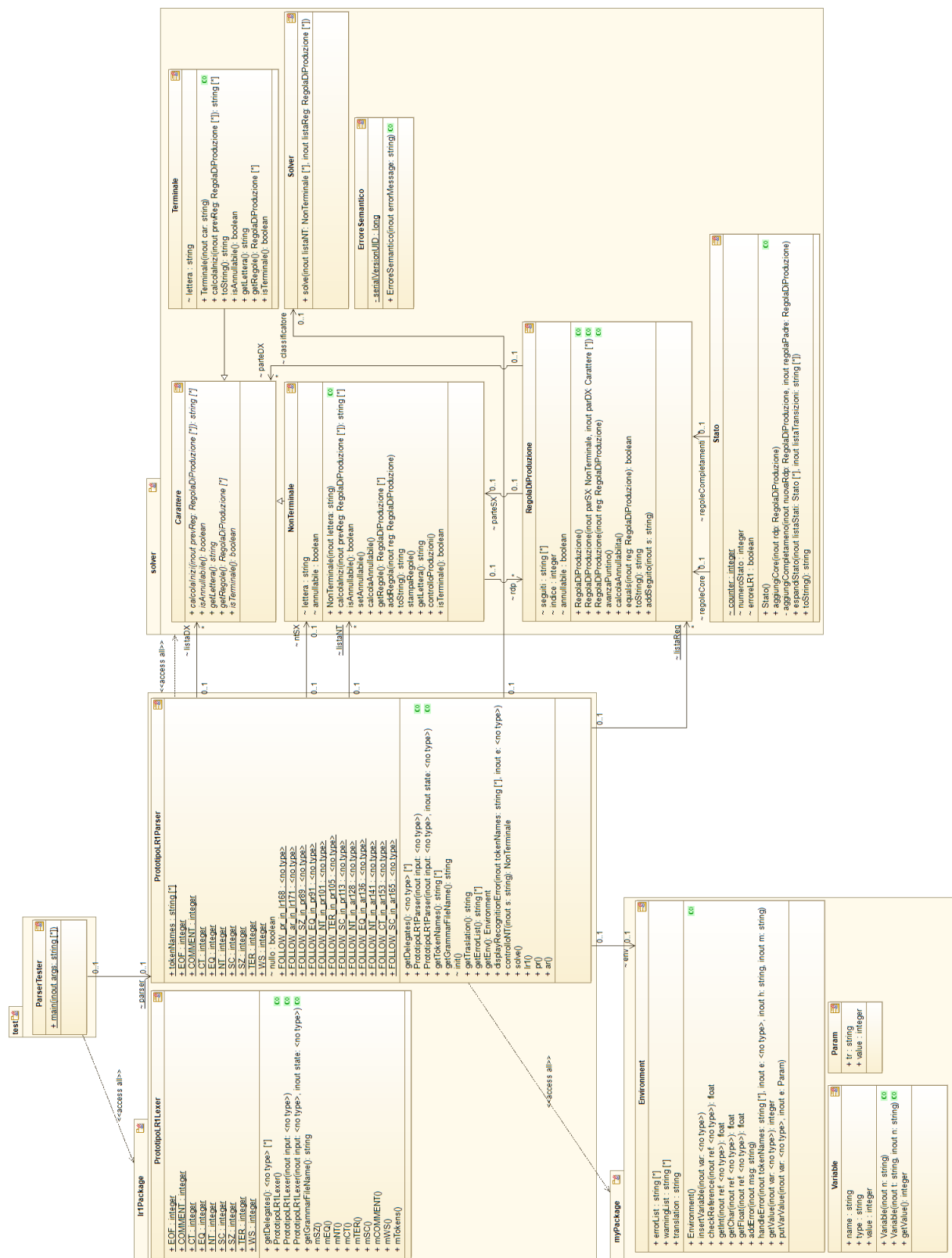


Figura 2: UML Class Diagram delle funzionalità dell'applicazione nell'iterazione 1

4.2 Testing

4.2.1 Descrizione del prodotto software

Il software, sviluppato in linguaggio Java in ambiente di sviluppo Eclipse, è suddiviso nei 4 packages descritti di seguito:

- **lr1package** - questo package contiene due degli output di ANTLR, strumento utilizzato per generare il compilatore, ossia *PrototipoLR1Lexer.java* e *PrototipoLR1Parser.java*; il terzo e ultimo file di output (*PrototipoLR1.tokens* è invece situato all'esterno di questo package;
- **main_package** - questo package contiene il file *Riconoscitore.java*, all'interno del quale è definito il metodo *main* che permette l'effettiva esecuzione del programma;
- **myPackage** - questo package contiene il file *Environment.java*, all'interno del quale è definita la classe *Environment*, necessaria per il corretto funzionamento dei file output di ANTLR ed ereditata da un progetto di esempio;
- **solver** - questo package contiene tutti i file che contengono definizioni di classi custom utilizzate per il processo di riconoscimento delle grammatiche e per rappresentare gli elementi costituenti delle grammatiche.

La versione oggetto di test è la 1.0, pubblicata sul branch *master* della repository di Github del progetto (<https://github.com/d-presciani/progettoLFC>) in data 26/03/2019.

4.2.2 Funzionalità oggetto di test

Le funzionalità sottoposte a test di unità saranno quelle definite nelle classi contenute all'interno del package *solver*, essendo queste le classi scritte manualmente e quelle che consentono l'effettivo funzionamento del programma; si trascurano di effettuare test di unità sui restanti package in quanto generati automaticamente da ANTLR e difficilmente sottoponibili a test di tale granularità.

È previsto anche un test di sistema, fornendo al sistema diversi input, per verificare la corretta integrazione tra i vari componenti e, al contempo, il corretto funzionamento del programma nel suo complesso.

Oltre ai suddetti test, è prevista anche l'esecuzione di un'analisi statica del codice tramite i due plugin di Eclipse *SpotBugs* e *PMD*.

4.2.2.1 Elenco delle funzionalità oggetto di test

Di seguito vengono riportate le classi, con i relativi metodi, per cui saranno effettuati i test di unità.

- Stato
 - private void aggiungiCompletamento(RegolaDiProduzione nuovaRdp, RegolaDiProduzione regolaPadre)
 - public void aggiungiCore(RegolaDiProduzione rdp)
 - public Stato()
 - public void espandiStato(LinkedList<Stato> listaStati, LinkedList<String> listaTransizioni)
 - public String toString()
- RegolaDiProduzione
 - public RegolaDiProduzione()
 - public RegolaDiProduzione(NonTerminale parSX, List<Carattere> parDX)
 - public RegolaDiProduzione(RegolaDiProduzione reg)
 - public void addSeguito(String s)
 - public void avanzaPuntino()
 - public void calcolaAnnullabilita()
 - public boolean equals(Object o)
 - public String toString()
- NonTerminale
 - public NonTerminale (String lettera)
 - public void addRegola (RegolaDiProduzione reg)
 - public void calcolaAnnullabile()
 - public List<String> calcolaInizi(LinkedList<RegolaDiProduzione> prevReg)
 - public void controlloProduzioni()

- public String getLettera()
 - public List<RegolaDiProduzione> getRegole()
 - public boolean isAnnullabile()
 - public boolean isTerminale()
 - public void setAnnullabile()
 - public void stampaRegole()
 - public String toString()
- Solver
 - public boolean solve(LinkedList<NonTerminale> listaNT,
LinkedList<RegolaDiProduzione> listaReg)
- Terminale
 - public Terminale(String car)
 - public List<String> calcolaInizi(LinkedList<RegolaDiProduzione> prevReg)
 - public String getLettera()
 - public List<RegolaDiProduzione> getRegole()
 - public boolean isAnnullabile()
 - public boolean isTerminale()
 - public String toString()

4.3 Esiti dei test

4.3.1 Esiti e copertura dei test di unità

Di seguito sono riportati gli esiti dei test e la copertura degli stessi per ognuna delle classe precedentemente evidenziate come oggetto di test di unità; nel complesso tali test hanno consentito una copertura del package *solver* pari al 96,4%.














▼  solver	 96,4 %	2.101	79	2.180
>  Stato.java	 95,6 %	1.485	68	1.553
>  RegolaDiProduzione.java	 95,6 %	238	11	249
>  Carattere.java	100,0 %	3	0	3
>  ErroreSemantico.java	100,0 %	4	0	4
>  NonTerminale.java	 100,0 %	177	0	177
>  Solver.java	 100,0 %	165	0	165
>  Terminale.java	100,0 %	29	0	29

Figura 3: Copertura dei test per il package solver

4.3.1.1 Esiti dei test per la classe Stato

Per la classe Stato sono stati scritti, all'interno della classe StatoTest, 22 test, superati correttamente dal programma, che garantiscono una copertura della classe Stato pari al 95,6%.

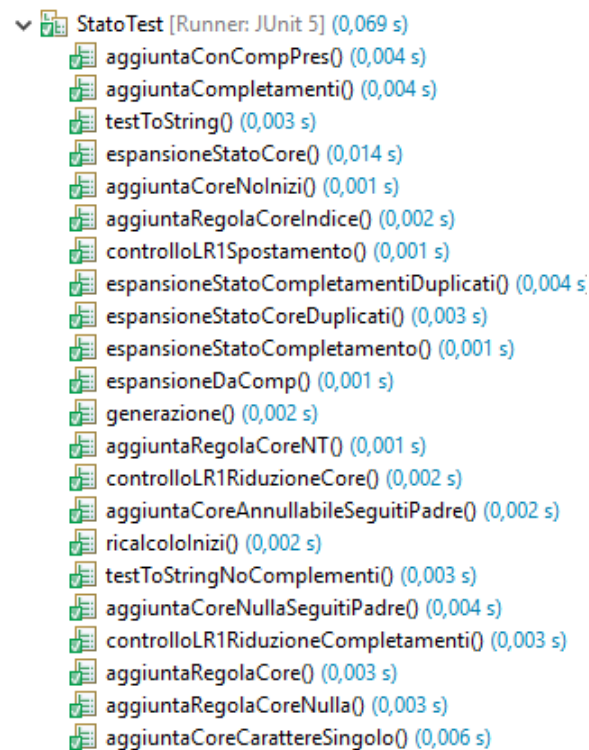


Figura 4: Esiti dei test della classe Stato

4.3.1.2 Esiti dei test per la classe RegolaDiProduzione

Per la classe RegolaDiProduzione sono stati scritti, all'interno della classe RegolaDiProduzioneTest, 6 test, superati correttamente dal programma, che garantiscono una copertura della classe RegolaDiProduzione pari al 95,6%.

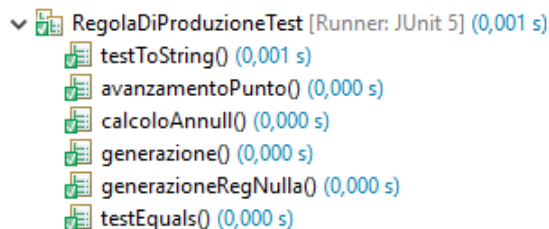


Figura 5: Esiti dei test della classe RegolaDiProduzione

4.3.1.3 Esiti dei test per la classe NonTerminale

Per la classe NonTerminale sono stati scritti, all'interno della classe NonTerminaleTest, 13 test, superati correttamente dal programma, che garantiscono una copertura della classe NonTerminale pari al 100,0%.

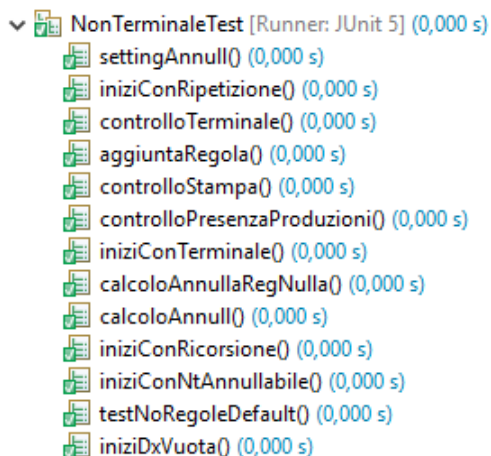


Figura 6: Esiti dei test della classe NonTerminale

4.3.1.4 Esiti dei test per la classe Solver

Per la classe Solver sono stati scritti, all'interno della classe SolverTest, 2 test, superati correttamente dal programma, che garantiscono una copertura della classe Solver pari al 100,0%.

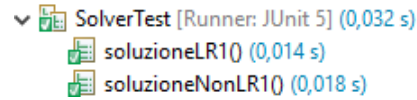


Figura 7: Esiti dei test della classe Solver

4.3.1.5 Esiti dei test per la classe Terminale

Per la classe Terminale sono stati scritti, all'interno della classe TerminaleTest, 7 test, superati correttamente dal programma, che garantiscono una copertura della classe Terminale pari al 100,0%.

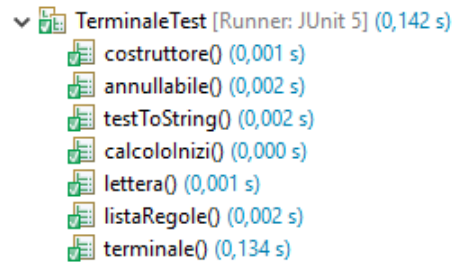


Figura 8: Esiti dei test della classe Terminale

4.3.2 Esiti e copertura dei test di sistema

I test di sistema sono stati effettuati eseguendo il metodo *main* della classe *Riconoscitore* più volte, passandogli come input i file contenuti all'interno della cartella "lfc\resources" presente nella repository del progetto (<https://github.com/d-presciani/progettoLFC>); come oracoli per valutare la correttezza dei test sono stati utilizzati i temi svolti in classe durante il corso di Linguaggi formali e compilatori.

Durante i test di sistema è stata rilevata anche la copertura del codice nelle varie esecuzioni; nella seguente immagine è possibile vedere la copertura complessiva (ottenuta combinando la copertura delle singole esecuzioni).












Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ Ir1package	 67,1 %	937	459	1.396
> ▢ PrototipoLR1Lexer.java	 48,9 %	343	358	701
> ▢ PrototipoLR1Parser.java	 85,5 %	594	101	695
▼ solver	 96,4 %	2.128	80	2.208
> ▢ Solver.java	 86,5 %	167	26	193
> ▢ Stato.java	 98,5 %	1.530	23	1.553
> ▢ NonTerminale.java	 89,3 %	158	19	177
> ▢ RegolaDiProduzione.java	 96,0 %	239	10	249
> ▢ Terminale.java	93,1 %	27	2	29
> ▢ Carattere.java	100,0 %	3	0	3
> ▢ ErroreSemantico.java	100,0 %	4	0	4
▼ myPackage	22,2 %	16	56	72
> ▢ Environment.java	 22,2 %	16	56	72
▼ main_package	 78,6 %	99	27	126
> ▢ Riconoscitore.java	 78,6 %	99	27	126

Figura 9: Copertura del codice relativa ai test di sistema

5. Iterazione 2

In questa iterazione è stata implementata un interfaccia grafica che permette di interagire con il programma in un ambiente completamente grafico.

È stata aggiunta inoltre la possibilità di visualizzare graficamente il risultato dell'analisi generando un grafo della grammatica analizzata.

5.1 Design Architeturale

In questa seconda versione la struttura è rimasta di tipo pipe and filter ma è stata ampliata per includere la generazione e la visualizzazione del grafo, come si può vedere in figura.

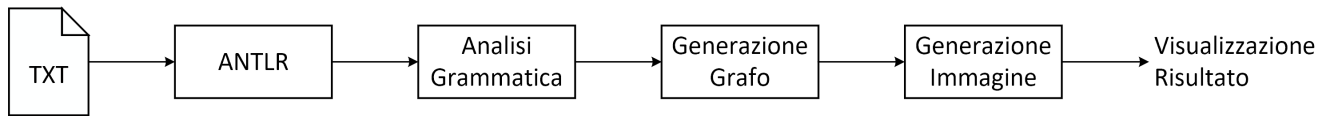


Figura 10: Bozza dell'architettura nella seconda iterazione

5.1.1 Librerie utilizzate

Per l'implementazione dell'interfaccia grafica e la creazione del grafo sono state utilizzate le seguenti librerie:

- JavaFX, framework grafico utilizzato per la realizzazione dell'interfaccia utente;
- JgraphT, libreria realizzata in swing che permette la creazione di grafi di vario tipo, implementando inoltre iteratori può essere utilizzata per la creazione di algoritmi di vario tipo;
- mxGraph, libreria che permette la creazione di diagrammi e utilizza SVG e HTML per il rendering delle immagini, in questo progetto questa libreria è utilizzata per la visualizzazione del grafico creato con JgraphT, in particolare questa libreria è stata scelta perché permette se integrata in un JFrame di modificare l'aspetto del grafico in realtime oltre che offrire molte opzioni per la personalizzazione dei grafici. Nella pratica questa libreria è stata utilizzata per modificare l'aspetto delle box cambiando colore agli stati che contengono delle regole che rendono il linguaggio non $LR(1)$ e per modificare lo stile grafico di rappresentazione.

Durante lo sviluppo di questa iterazione sono però sorte delle problematiche nell'integrazione delle librerie per la generazione del grafo e JavaFx.

Provando ad integrare all'interno dell'interfaccia utente un modulo sviluppato per testare le funzionalità di mxGraph che implementava la modifica realtime del grafo, sono stati riscontrati alcuni problemi di integrabilità essendo questa libreria realizzata in swing.

Per implementare questo modulo si è provato ad utilizzare la classe SwingNode per "incapsulare" questo modulo che integra componenti realizzate in swing e riuscire quindi a riutilizzare correttamente il modulo all'interno dell'interfaccia utente creata con JavaFX.

Dopo diversi tentativi si è riusciti ad integrare il modulo contenente il grafo con l'interfaccia utente realizzata con JavaFX, perdendo però la possibilità di editare in tempo reale il grafico, si è dunque scelto per motivi di performance di salvare l'immagine generata tramite mxGraph (la scelta del formato è lasciata all'utente) e poi caricare l'immagine salvata all'interno dell'interfaccia grafica per la visualizzazione del grafo.

5.1.2 Class Diagram

Rispetto al class diagram presente a pagina 12 in questa nuova iterazione si può vedere come è stato necessario modificare la struttura delle classi per permettere il trasferimento di informazioni tra le classi in modo da poter mostrare all'interno dell'interfaccia grafica i vari messaggi da mostrare all'utente.

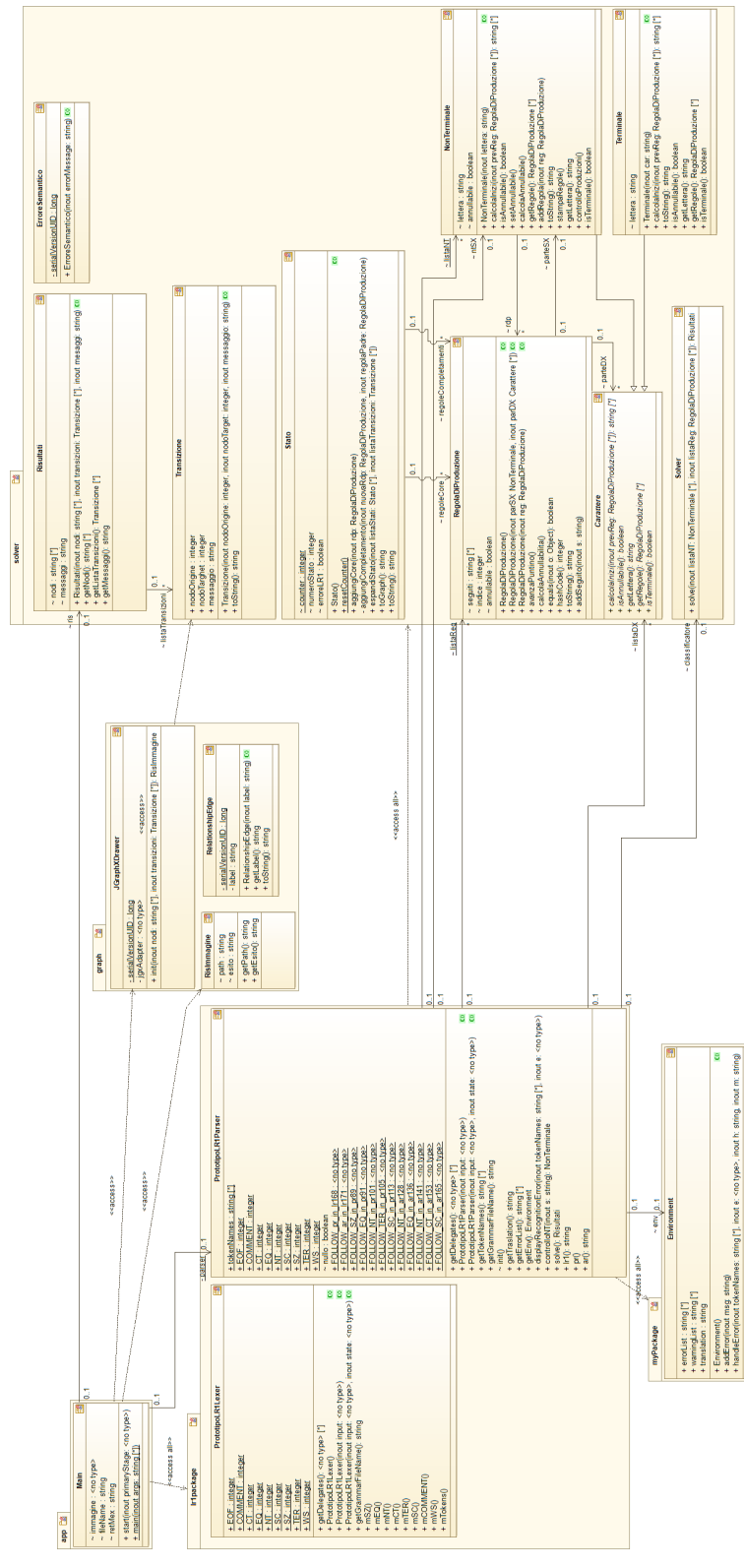


Figura 11: UML Class Diagram delle funzionalità dell'applicazione nell'iterazione 2

5.2 Casi d'uso

Vista l'introduzione di un'interfaccia grafica per l'interazione con il programma sono stati ridefiniti i casi d'uso come segue:

5.2.1 UC1 - Analisi di una grammatica

Descrizione: identificazione di una grammatica $LR(1)$

Attori coinvolti: utente

Precondizioni: esistenza del file d'input contenente la definizione della grammatica

Trigger: necessità di identificare una grammatica

Post condizioni: la classificazione della grammatica viene mostrata a schermo

Procedimento standard:

1. l'utente avvia il programma;
2. l'utente clicca il pulsante "carica file";
3. l'utente seleziona il file di input desiderato;
4. il programma esegue il parsing e l'analisi del file di input;
5. il programma mostra a schermo l'esito dell'analisi della grammatica.

Procedimenti alternativi o eccezioni:

- allo step 3, il programma rileva errori nella grammatica
viene mostrato un messaggio d'errore contenente informazioni relative all'errore individuato, l'utente, corretta la grammatica riprende la procedura dallo step 2

5.2.2 UC2 - Visualizzazione della grammatica analizzata

Descrizione: visualizzazione della grammatica $LR(1)$ analizzata precedentemente

Attori coinvolti: utente

Precondizioni: analisi di una grammatica avvenuta correttamente

Trigger: necessità di visualizzare la grammatica identificata

Post condizioni: creazione di un file contenente il grafo relativo alla grammatica e visualizzazione dello stesso

Procedimento standard:

1. l'utente clicca il pulsante "genera e mostra grafo";
2. l'utente seleziona la posizione dove salvare il grafo e il formato del file creato;
3. il programma genera e mostra a schermo il grafo relativo alla grammatica.

Procedimenti alternativi o eccezioni:

- allo step 3, il programma rileva un errore in fase di generazione dell'immagine
viene mostrato un messaggio d'errore contenente informazioni relative all'errore individuato, l'utente, riprende la procedura dal punto 1 seguendo le indicazioni fornite dal programma

5.3 Testing

5.3.1 Descrizione del prodotto software

Il software, sviluppato in linguaggio Java in ambiente di sviluppo Eclipse, è suddiviso nei 5 packages descritti di seguito:

- **app** - questo package contiene il file *Main.java*, oltre ad altri file di supporto, utilizzati per la creazione dell'interfaccia grafica;
- **graph** - questo package contiene tutti i file necessari alla generazione del grafo;
- **lr1package** - questo package contiene due degli output di ANTLR, strumento utilizzato per generare il compilatore, ossia *PrototipoLR1Lexer.java* e *PrototipoLR1Parser.java*; il terzo e ultimo file di output (*PrototipoLR1.tokens* è invece situato all'esterno di questo package;
- **myPackage** - questo package contiene il file *Environment.java*, all'interno del quale è definita la classe Environment, necessaria per il corretto funzionamento dei file output di ANTLR ed ereditata da un progetto di esempio;
- **solver** - questo package contiene tutti i file che contengono definizioni di classi custom utilizzate per il processo di riconoscimento delle grammatiche e per rappresentare gli elementi costituenti delle grammatiche.

La versione oggetto di test è la 2.0, pubblicata sul branch *master* della repository di Github del progetto (<https://github.com/d-presciani/progettoLFC>) in data 09/10/2019.

5.3.2 Funzionalità oggetto di test

Le funzionalità sottoposte a test di unità saranno quelle definite nelle classi contenute all'interno del package *solver*, essendo queste le classi scritte manualmente e quelle che consentono l'effettivo funzionamento del programma; si trascurano di effettuare test di unità sui restanti package in quanto generati automaticamente da ANTLR e difficilmente sottoponibili a test di tale granularità.

È previsto anche un test di sistema, fornendo al sistema diversi input, per verificare la corretta integrazione tra i vari componenti e, al contempo, il corretto funzionamento del programma nel suo complesso.

Oltre ai suddetti test, è prevista anche l'esecuzione di un'analisi statica del codice tramite i due plugin di Eclipse *SpotBugs* e *PMD*.

5.3.2.1 Elenco delle funzionalità oggetto di test

Di seguito vengono riportate le classi, con i relativi metodi, per cui saranno effettuati i test di unità.

- Stato
 - private void aggiungiCompletamento(RegolaDiProduzione nuovaRdp, RegolaDiProduzione regolaPadre)
 - public void aggiungiCore(RegolaDiProduzione rdp)
 - public Stato()
 - public void espandiStato(LinkedList<Stato> listaStati, LinkedList<Transizione> listaTransizioni)
 - public String toString()
- RegolaDiProduzione
 - public RegolaDiProduzione()
 - public RegolaDiProduzione(NonTerminale parSX, List<Carattere> parDX)
 - public RegolaDiProduzione(RegolaDiProduzione reg)
 - public void addSeguito(String s)
 - public void avanzaPuntino()
 - public void calcolaAnnullabilita()
 - public boolean equals(Object o)
 - public String toString()
- NonTerminale
 - public NonTerminale (String lettera)
 - public void addRegola (RegolaDiProduzione reg)
 - public void calcolaAnnullabile()
 - public List<String> calcolaInizi(LinkedList<RegolaDiProduzione> prevReg)
 - public void controlloProduzioni() throws ErroreSemantico

- public String getLettera()
- public List<RegolaDiProduzione> getRegole()
- public boolean isAnnullabile()
- public boolean isTerminale()
- public void setAnnullabile()
- public void stampaRegole()
- public String toString()
- Solver
 - public Risultati solve(LinkedList<NonTerminale> listaNT,
LinkedList<RegolaDiProduzione> listaReg)
- Terminale
 - public Terminale(String car)
 - public List<String> calcolaInizi(LinkedList<RegolaDiProduzione> prevReg)
 - public String getLettera()
 - public List<RegolaDiProduzione> getRegole()
 - public boolean isAnnullabile()
 - public boolean isTerminale()
 - public String toString()

5.4 Esiti dei test

5.4.1 Esiti e copertura dei test di unità

Di seguito sono riportati gli esiti dei test e la copertura degli stessi per ognuna delle classi precedentemente evidenziate come oggetto di test di unità; nel complesso tali test hanno consentito una copertura del package *solver* pari al 95,6%.

solver		95,6 %	2.314	107	2.421
> Carattere.java		100,0 %	3	0	3
> ErroreSemantico.java		100,0 %	4	0	4
> NonTerminale.java		100,0 %	177	0	177
> RegolaDiProduzione.java		95,6 %	238	11	249
> Risultati.java		62,5 %	15	9	24
> Solver.java		100,0 %	162	0	162
> Stato.java		96,1 %	1.674	68	1.742
> Terminale.java		100,0 %	29	0	29
> Transizione.java		38,7 %	12	19	31

Figura 12: Copertura dei test per il package solver

5.4.1.1 Esiti dei test per la classe Stato

Per la classe Stato sono stati scritti, all'interno della classe StatoTest, 22 test, superati correttamente dal programma, che garantiscono una copertura della classe Stato pari al 96,1%.

StatoTest [Runner: JUnit 5] (0,001 s)
aggiuntaConCompPres() (0,000 s)
aggiuntaCompletamenti() (0,000 s)
testToString() (0,000 s)
espansioneStatoCore() (0,000 s)
aggiuntaCoreNoInizi() (0,000 s)
aggiuntaRegolaCoreIndice() (0,000 s)
controlloLR1Spostamento() (0,000 s)
espansioneStatoCompletamentiDuplicati() (0,000 s)
espansioneStatoCoreDuplicati() (0,000 s)
espansioneStatoCompletamento() (0,000 s)
espansioneDaComp() (0,000 s)
generazione() (0,000 s)
aggiuntaRegolaCoreNT() (0,000 s)
controlloLR1RiduzioneCore() (0,000 s)
aggiuntaCoreAnnullabileSeguitiPadre() (0,000 s)
ricalcoloInizi() (0,000 s)
testToStringNoComplementi() (0,000 s)
aggiuntaCoreNullaSeguitiPadre() (0,000 s)
controlloLR1RiduzioneCompletamenti() (0,000 s)
aggiuntaRegolaCore() (0,000 s)
aggiuntaRegolaCoreNulla() (0,000 s)
aggiuntaCoreCarattereSingolo() (0,000 s)

Figura 13: Esiti dei test della classe Stato

5.4.1.2 Esiti dei test per la classe RegolaDiProduzione

Per la classe RegolaDiProduzione sono stati scritti, all'interno della classe RegolaDiProduzioneTest, 6 test, superati correttamente dal programma, che garantiscono una copertura della classe RegolaDiProduzione pari al 95,6%.

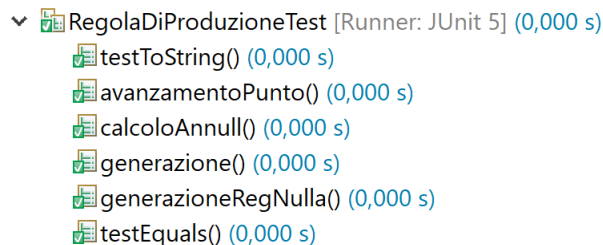


Figura 14: Esiti dei test della classe RegolaDiProduzione

5.4.1.3 Esiti dei test per la classe NonTerminale

Per la classe NonTerminale sono stati scritti, all'interno della classe NonTerminaleTest, 13 test, superati correttamente dal programma, che garantiscono una copertura della classe NonTerminale pari al 100,0%.

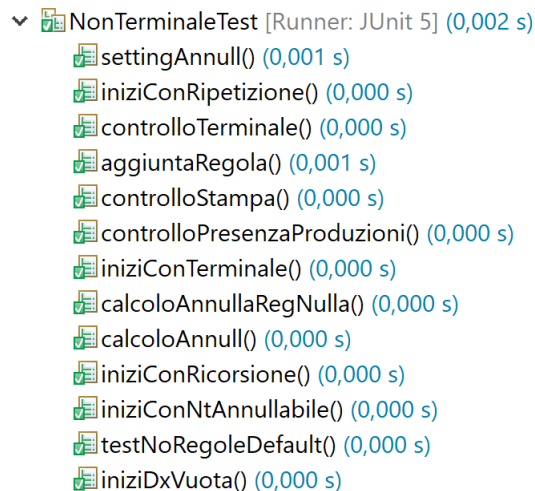


Figura 15: Esiti dei test della classe NonTerminale

5.4.1.4 Esiti dei test per la classe Solver

Per la classe Solver sono stati scritti, all'interno della classe SolverTest, 2 test, superati correttamente dal programma, che garantiscono una copertura della classe Solver pari al 100,0%.

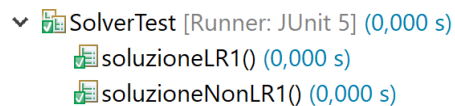


Figura 16: Esiti dei test della classe Solver

5.4.1.5 Esiti dei test per la classe Terminale

Per la classe Terminale sono stati scritti, all'interno della classe TerminaleTest, 7 test, superati correttamente dal programma, che garantiscono una copertura della classe Terminale pari al 100,0%.

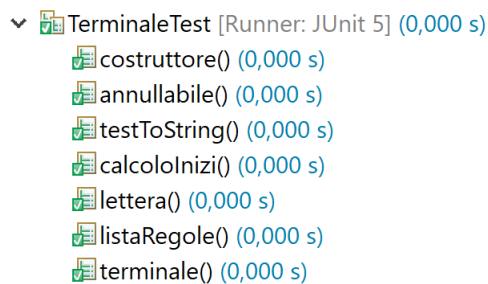


Figura 17: Esiti dei test della classe Terminale

5.4.2 Esiti e copertura dei test di sistema

I test di sistema è stato effettuato eseguendo il metodo *main* della classe *app*, passandogli come input i file contenuti all'interno della cartella "lfc/resources" presente nella repository del progetto (<https://github.com/d-presciani/progettoLFC>); come oracoli per valutare la correttezza dei test sono stati utilizzati i temi svolti in classe durante il corso di Linguaggi formali e compilatori.

Durante il test di sistema è stata rilevata anche la copertura del codice durante l'esecuzione; nella seguente immagine è possibile vedere la copertura complessiva.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
lfc	87,8 %	4.287	594	4.881
src	87,8 %	4.287	594	4.881
app	95,7 %	563	25	588
Main.java	95,7 %	563	25	588
graph	97,1 %	363	11	374
JGraphXDrawer.java	97,7 %	345	8	353
RelationshipEdge.java	75,0 %	9	3	12
RisImmagine.java	100,0 %	9	0	9
lr1package	72,6 %	1.035	391	1.426
PrototipoLR1Lexer.java	48,9 %	343	358	701
PrototipoLR1Parser.java	95,4 %	692	33	725
myPackage	100,0 %	72	0	72
Environment.java	100,0 %	72	0	72
solver	93,1 %	2.254	167	2.421
Carattere.java	100,0 %	3	0	3
ErroreSemantico.java	100,0 %	4	0	4
NonTerminale.java	86,4 %	153	24	177
RegolaDiProduzione.java	96,0 %	239	10	249
Risultati.java	100,0 %	24	0	24
Solver.java	100,0 %	162	0	162
Stato.java	93,6 %	1.630	112	1.742
Terminale.java	93,1 %	27	2	29
Transizione.java	38,7 %	12	19	31

Figura 18: Copertura del codice relativa al test di sistema

6. Strutture dati e algoritmi

In questa sezione si illustrano le strutture dati utilizzate e successivamente gli algoritmi più rilevanti implementati. Le strutture fondamentali che costituiscono le fondamenta delle strutture più complesse sono le classi `Terminale` e `NonTerminale`. Queste classi sono rappresentate di seguito.

6.1 Terminale

Questa classe viene utilizzata per rappresentare i caratteri terminali, questi caratteri sono i più semplici da gestire dato che la maggior parte delle proprietà che condividono con i caratteri `NonTerminale` sono ben definite.

6.1.1 Dati

- *lettera* una variabile utilizzata per memorizzare il carattere terminale.

6.1.2 Operazioni

- Costruttore(*car*): costruisce il `Terminale` memorizzando la variabile *car* ricevuta;
- CalcolaInizi: restituisce il carattere `Terminale`.
- toString: restituisce la rappresentazione testuale del carattere terminale;
- isAnnullabile: dato che un carattere terminale non è mai annullabile questa funzione restituisce sempre *false*;
- getLettera: restituisce la rappresentazione testuale del carattere terminale;
- getRegole: dato che un carattere terminale non ha regole associate questa funzione ritorna sempre *null*;
- isTerminale: questa funzione restituisce *true* dato che il carattere è un carattere terminale.

6.2 NonTerminale

Questa classe viene utilizzata per rappresentare i caratteri non terminali, questi caratteri hanno proprietà in comune con i caratteri terminali, ma come si può vedere dai dati memorizzati e dall'implementazione delle funzioni questi ultimi richiedono l'implementazione di particolari algoritmi per restituire alcune informazioni rilevanti.

6.2.1 Dati

- *lettera*: come per la classe `Terminale` questa variabile viene utilizzata per memorizzare la rappresentazione testuale del carattere;
- *rdp*: una `LinkedList` contenente tutte le regole di produzione che hanno questo non terminale come parte sinistra;
- *annullabile*: una flag utilizzata per memorizzare se il carattere è o meno annullabile senza dovere ricalcolare questa proprietà ogni volta.

6.2.2 Operazioni

- Costruttore(*lettera*): costruisce il Terminale memorizzando la variabile *lettera* ricevuta, inoltre inizializza la lista *rdp* come LinkedList vuota, viene infine inizializzata la variabile *annullabile* a false;
- CalcolaInizi: questa funzione calcola gli inizi del non terminale e li restituisce;
- calcolaAnnullabile: questa funzione deve essere chiamata al termine della creazione del non terminale e calcola se il terminale è o meno annullabile andando ad analizzare tutte le regole di produzione che ha associate;
- toString: restituisce la rappresentazione testuale del carattere terminale;
- isAnnullabile: getter del campo annullabile;
- getLettera: restituisce la rappresentazione testuale del carattere terminale;
- getRegole: questa funzione restituisce l'elenco delle regole di produzione associate al non terminale;
- isTerminale: questa funzione restituisce *false* dato che il carattere è un carattere non terminale.

6.2.3 Algoritmi

L'unico algoritmo di interesse in questa struttura dati è quello che consente di calcolare gli inizi del non terminale visualizzabile di seguito:

TODO: inserire algoritmo

6.3 RegolaDiProduzione

Questa classe viene utilizzata per rappresentare le varie regole di produzione, questa classe è stata creata come contenitore per semplificare il codice delle altre strutture dati

6.3.1 Dati

- *parteSX*: il non terminale presente a sinistra del simbolo di equivalenza;
- *parteDX*: una LinkedList di caratteri terminali e non;
- *seguiti*: una LinkedList che rappresentante i seguiti della regola di produzione (NOTA: i seguiti sono da intendersi in relazione alla posizione del puntino (*indice* in questa implementazione));
- *indice*: una variabile utilizzata per memorizzare la posizione dell'indice all'interno della parte destra della regola di produzione;
- *annullabile*: flag utilizzata per memorizzare se la regola è o meno annullabile senza dover ricalcolare questa proprietà tutte le volte.

6.3.2 Operazioni

- Costruttore(*parSX*, *parDX*): costruisce la regola di produzione partendo dai campi ricevuti, viene posto *indice* = 0;
- Costruttore(*reg*): costruisce la regola partendo da una regola esistente, questo costruttore viene utilizzato quando la regola è già esistente ma bisogna crearne una nuova uguale per poi incrementare il valore di *indice*;
- avanzaPuntino: incrementa di 1 il valore di *indice*;
- calcolaAnnullabilita: questa funzione deve essere chiamata al termine della creazione della regola di produzione e calcola se la regola è o meno annullabile andando ad analizzare i caratteri presenti in *parteDX*;
- equals(*o*): ridefinizione del metodo equals;
- toString: ritorna una stringa rappresentante la regola di produzione;
- addSeguito(*s*): aggiunge all'elenco dei seguiti il carattere ricevuto.

6.4 Stato

Questa è la struttura dati principale in questo progetto, qui sono presente la maggior parte degli algoritmi che permettono di calcolare la se una grammatica fornita è o meno di tipo $LR(1)$.

6.4.1 Dati

- *counter*: variabile statica utilizzata per il conteggio progressivo degli stati che vengono creati;
- *regoleCore*: LinkedList contenente le regole core dello stato;
- *regoleCompletamenti*: LinkedList contenente le regole di completamento dello stato;
- *numeroStato*: variabile utilizzata per la memorizzazione del numero dello stato;
- *erroreLR1*: flag utilizzata per memorizzare se uno stato contiene o meno delle regole che sono in conflitto e quindi rendono la grammatica non $LR(1)$.

6.4.2 Operazioni

- Costruttore: costruisce uno stato vuoto;
- resetCounter: funzione per resettare la variabile *counter* a 1;
- aggiungiCore(*rdp*): funzione che inserisce la regola ricevuta all'elenco delle regole core, successivamente analizza la regola inserita per verificare se questa aggiunge delle regole di completamento allo stato oppure modifica i seguiti delle regole già presenti;
- aggiungiCompletamento(*nuovaRdp*, *regolaPadre*): funzione ausiliaria chiamata in aggiungiCore, si occupa di aggiungere *nuovaRdp* alle regole di completamento e aggiunge i seguiti di *regolaPadre* in caso questa sia annullabile;

- espandiStato(*listaStati*, *listaTransizioni*): quando chiamata questa funzione genera nuovi stati esplorando ogni regola che lo compone;
- toGraph: questa funzione crea e ritorna una String che verrà utilizzata per la rappresentazione dello stato all'interno del grafo;
- toString: restituisce una stringa rappresentante lo stato.

6.4.3 Algoritmi

TODO: inserire lo pseudo-codice degli algoritmi

7. Sviluppi futuri

Il programma, allo stato attuale, potrebbe essere migliorato sia sotto l'aspetto dell'interfaccia grafica che delle funzionalità, ad esempio:

- espandere le capacità di analisi implementando dei moduli in grado di riconoscere altri tipi di grammatiche come $LALR(1)$ e $LL(1)$;
- aggiungere la possibilità di scrittura della grammatica direttamente nel programma, senza l'utilizzo di un file txt, in modo da evidenziare anche nella grammatica le regole problematiche;
- migrazione della generazione del grafo a librerie compatibili con JavaFX al fine di avere un grafo editabile;
- porting dell'applicazione anche per dispositivi mobili.