Indice

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. Introduzione

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) $S \to \alpha$

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.

3. 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 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 $S0$, 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	$S0$
EQ	$-> \mid :=$
NT	$A\ \ldots\ Z$
CT	$a\ \ldots z \mid 0\ \ldots\ 9 \mid +\mid -\mid *\mid /$
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).

3.2 Errori lessicali

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

3.3 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 4.

3.4 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).

4. Casi d'uso

4.1 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

5. Struttura del programma

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

- 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 avverrà attraverso una CLI (Command Line Interface).

6. 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.

7. Strumenti utilizzati

7.1 Controllo di versione

Per il controllo di versione verrà utilizzato Git, 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.

7.2 Strumenti di sviluppo

ANTLR sarà utilizzato per definire e generare il modulo che effettuerà il parsing del file in input contenente la grammatica da analizzare.

Le classi Java generate come output da ANTLR saranno poi incluse nel progetto Eclipse, IDE utilizzato per la realizzazione del programma, per essere integrate con il modulo di analisi della grammatica che verrà invece scritto a mano all'interno di Eclipse stesso.

7.3 Strumenti di testing

Per il testing verrà utilizzato JUnit per la definizione dei casi di test e l'esecuzione dei casi stessi, mentre per quanto riguarda l'analisi statica del codice verranno usati due plugin di Eclipse: SpotBugs e PMD.

Per verificare la copertura del codice durante le esecuzioni del programma verrà utilizzato lo strumento di coverage messo a disposizione da Eclipse.