Relazione Progetto Esame di Linguaggi e Compilatori

Autore: Luca Bianchi, Matricola: 114150
(IV appello del 04/07/2023)

Grammatica

Per realizzare il progetto richiesto dalla traccia, ho deciso di adottare come grammatica una grammatica molto simile alla grammatica LL(1) per poter scrivere le espressioni aritmetiche.

Tale grammatica è la seguente:

init → expr1 , stringList
expr1→ expr2 expr1'
expr1'→ + expr2 expr1'
expr1'→ ε
expr2 → expr3 expr2'
expr2'→ expr3 expr2'
expr2'→ ε
expr3 → fact expr3'
expr3'→ *
expr3'→ ε
fact → CHAR
fact → EPSILON
fact → (expr1)
stringList → string stringList'
stringList'→ , stringList
stringList'→ ε
string → CHAR string
string → ε

La grammatica in questione è infatti a sua volta una grammatica LL(1) come dimostrano i seguenti FIRST e FOLLOW:

FIRST(init) = {CHAR, EPSILON, '('}
FIRST(expr1) = {CHAR, EPSILON, '('}
FIRST(expr1') = {+, ε}
FIRST(exp2) = {CHAR, EPSILON, '('}
FIRST(expr2') = {ε, CHAR, EPSILON, '('}
FIRST(exp3) = {CHAR, EPSILON, '('}
FIRST(expr3') = {*, ε}
FIRST(fact) = {CHAR, EPSILON, '('}
FIRST(stringList) = {CHAR, ' , ' , ε}

FIRST(stringList') = {' , ' , ε}
FIRST(string) = {CHAR, ε}

FOLLOW(init) = {$}
FOLLOW (expr1) = {' , ' , ')'}
FOLLOW (expr1') = {' , ' , ')'}
FOLLOW (exp2) = {+, ' , ' , ')'}
FOLLOW (expr2') = {+, ' , ' , ')'}
FOLLOW (exp3) = {CHAR, EPSILON, '(', +, ' , ' , ')'}
FOLLOW (expr3') = {CHAR, EPSILON, '(', +, ' , ' , ')'}
FOLLOW (fact) = {*, CHAR, EPSILON, '(', +, ' , ' , ')'}
FOLLOW (stringList) = {$}
FOLLOW (stringList') = {$}
FOLLOW (string) = {' , ' , $}

Da questi si può notare infatti che neanche nel caso delle produzioni che cancellano i simboli non terminali con 'epsilon' si hanno conflitti nella tabella di Parsing. Perciò la grammatica è LL(1).

Per questo motivo nel codice ho praticamente scritto uno Schema di Traduzione per quanto riguarda la semantica (che implementa il Pattern Visitor ed effettua una visita in profondità).

Per quanto riguarda invece la parte relativa all'analisi lessicale ho semplicemente utilizzato tre token:

- Un token CHAR che rappresenta un qualsiasi singolo carattere alfanumerico appartenente all'alfabeto;
- Un token EPSILON che invece sarebbe la stringa 'epsilon' (che è diversa dalla epsilon sintattica presente nella sintassi, infatti questo token è quello relativo al linguaggio delle espressioni regolari);
- Un token WS che invece viene semplicemente scartato perché rappresenta spazi, TAB, e a capo.

Tale grammatica permette quindi di accettare tutte le stringhe di esempio presenti nella specifica del progetto. Inoltre, permette anche di accettare nella sequenza di stringhe da eseguire nell'NFA anche le stringhe vuote (perché anche se non specificatamente richiesto nella traccia, se viene passata come sola espressione regolare "epsilon, " allora il risultato dovrà essere OK ovviamente perché la stringa passata dopo la virgola è appunto vuota).

NOTA:
Nel codice e nel file g4 tuttavia non ho chiamato ad esempio expr1' in questo modo, ma ho chiamato simboli non terminali di questo tipo con expr1_sub .

Lexer e Parser

Una volta scritto il file g4, attraverso il tool ANTLR4 (che ho utilizzato all'interno di IntelliJ), ho fatto generare tutti i file Java dal tool, e per quanto riguarda la fase di analisi lessicale e sintattica non ho toccato nulla, concentrandomi piuttosto sull'analisi semantica.

Classi custom realizzate

Calculator

Prima di tutto ho realizzato una classe Calculator.java che contiene il 'main' da eseguire.

Al suo interno utilizza come Lexer la classe RegularExprLexer, come Parser RegularExprParser (a cui passa la sequenza di token riconosciuti dal Lexer e di cui si richiede la visita del simbolo non terminale iniziale). Ed infine per l'analisi semantica utilizza invece la classe RegularExprEvalVisitor, che è invece una classe che estende la classe generata RegularExprBaseVisitor (perciò vuol dire che nel mio progetto, avendo una grammatica LL(1), ho deciso di optare per utilizzare come visita il pattern Visitor).

NOTA:
Un piccolo problema che ho avuto con il tool è che in IntelliJ (oppure forse per la versione del tool stesso) non si riesce a prendere in input da console il testo che si vuole passare al programma utilizzando la classe CharStream, però sono riuscito ad aggirare il problema solamente passando in input tra gli argomenti ("args") il percorso al file contenente l'input da passare al programma. Consiglio quindi per l'esecuzione di passare in input il file passandone il Path tra gli argomenti dell'esecuzione.

NFAState

Questa classe rappresenta uno stato dell'NFA, ed infatti è composto da un nome (che è semplicemente un numero intero incrementale automatico), da due boolean per specificare se lo stato è iniziale e/o di accettazione. Ed infine ha una lista di transizioni uscenti.

NFATransition

Questa classe rappresenta invece una transizione nell'automa. Infatti ha uno stato iniziale, uno finale ed il simbolo con cui può "scattare" (che chiaramente sarà un simbolo dell'alfabeto o 'epsilon').

ThompsonNFA

Per quanto riguarda questa classe essa ovviamente è l'automa non deterministico risultante dall'algoritmo di Thompson e per questo è infatti composta da un solo stato iniziale e da un solo stato finale. Ma la cosa più importante è il metodo 'accept' presente all'interno della classe, che è in grado di riconoscere se una stringa è generata dall'espressione regolare (e quindi è accettata dal rispettivo NFA) oppure no.

ExprInheritedAttribute

Questa classe rappresenta invece gli attributi ereditati dei nodi dell'albero di parsing.
In questo caso l'unico attributo che ci interessa è infatti solo uno che si chiama 'i_automata' di tipo ThompsonNFA e che viene tecnicamente associato ai soli simboli non terminali expr1', expr2' ed expr3'. Tuttavia andando a realizzare un applicativo del genere in realtà non viene fatta questa distinzione, infatti anche nel caso degli attributi sintetizzati in cui sono più di uno, in realtà viene fatto in modo che ogni singola istanza di attributi sintetizzati o ereditati li contiene tutti (sarà poi l'algoritmo a decidere quali utilizzare in base alle necessità).

ExprSynthesisedAttribute

Questa classe rappresenta invece l'insieme di tutti gli attributi sintetizzati come già accennato. Qui troviamo un attributo chiamato 'automata' di tipo ThompsonNFA (che utilizzeremo per costruire ricorsivamente l'automa come da algoritmo), un attributo chiamato 'stringToAccept' di tipo String che invece viene utilizzato per costruire "salendo verso l'alto" le stringhe passate in input da far eseguire all'automa (quindi man mano che sale vengono concatenati altri caratteri all'inizio della stringa fino a

formare le intere stringhe), ed infine un attributo ‘accepted’ di tipo List<String> che invece contiene al suo interno l’insieme di tutte le stringhe “OK” e “KO” che sono state trasformate a partire dal boolean restituito dall’automa NFA una volta eseguite su di esso (ed anche in questo caso è sintetizzato perché costruiamo questa lista a partire dal basso, aggiungendo ogni volta un nuovo elemento come primo elemento della lista).

RegularExprBaseVisitor

In questa classe viene implementato come già detto il Pattern Visitor fornito da ANTLR4 che ha appunto generato la classe base che implementa a sua volta l’interfaccia Visitor.
All’interno vengono poi sovrascritti tutti i metodi relativi alle produzioni della grammatica.

All’interno troviamo prima di tutto un attributo privato ‘automata’ di tipo ThompsonNFA, questo perché una volta generato l’automa completo, questo viene salvato all’interno di questo attributo ed utilizzato in modo quasi “globale” all’interno della classe per eseguire le stringhe sullo stesso.

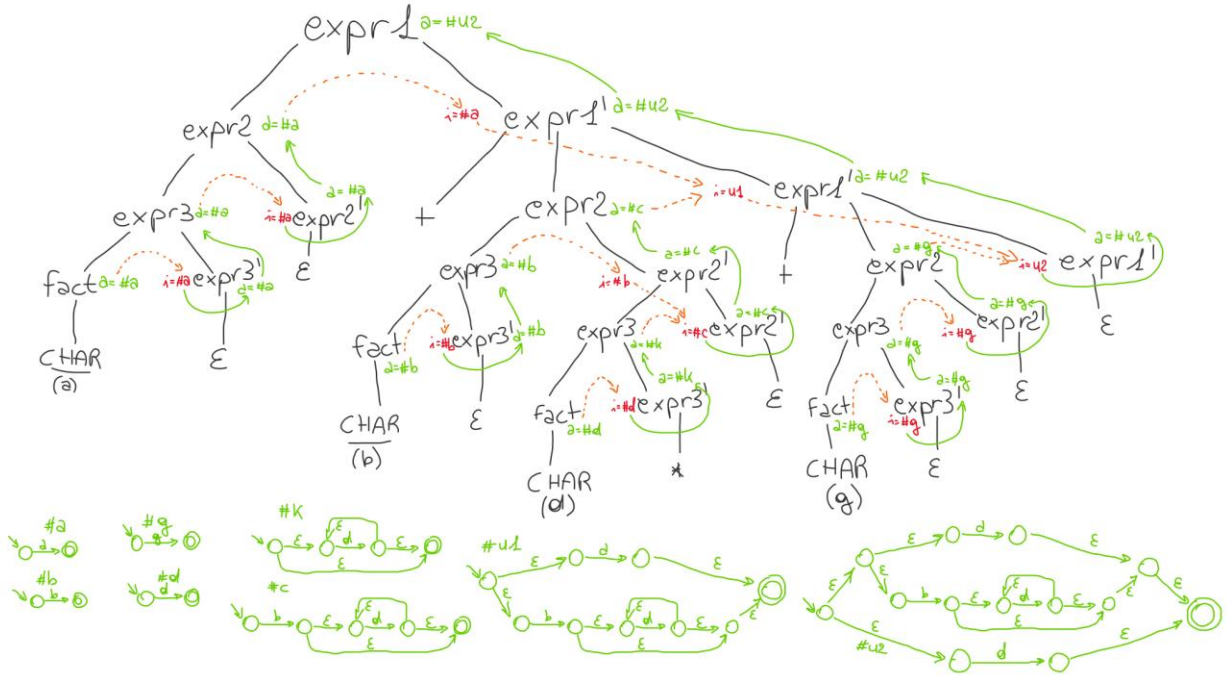
In generale possiamo distinguere due fasi all’interno di questa classe, infatti le prime produzioni della grammatica (fino al ‘fact’) sono relative alla generazione dell’automa non deterministico, mentre la parte successiva si occupa di ricostruire le stringhe e di eseguirle nell’automa di Thompson ricostruendone il risultato.

Costruzione NFA con algoritmo di Thompson

Normalmente sarebbe estremamente facile costruire questo automa in modo “bottom-up” utilizzando appunto una grammatica che ammette un Parser bottom-up. Avendo però voluto realizzare una grammatica specificatamente LL(1), ho utilizzato una coda di attributi ereditati, così facendo possiamo fare quel classico “passaggio” che si può trovare negli Schemi di Traduzione (ed utilizzando la coda ci sgraviamo da molte complicazioni, infatti venendo utilizzata come una coda appunto si è certi che l’attributo tirato fuori è quello atteso) con cui una volta che si cancella un simbolo non terminale, il rispettivo attributo ereditato viene fatto salire nell’albero facendolo diventare un attributo sintetizzato.

Per quanto riguarda “come fare il passaggio di valori da un attributo di un nodo all’altro” e quindi come sono arrivato a determinare la soluzione al problema ho realizzato prima per una espressione regolare di esempio “a + bd* + g” l’albero di derivazione (solo per la parte della derivazione dell’automa) e poi ho fatto muovere gli automi tra gli attributi dei simboli non terminali componendoli tra loro e facendoli salire verso l’alto in modo che poi alla radice dell’albero sia presente l’automa non deterministico.

L’albero realizzato è il seguente:



In questo modo ho saputo appunto come procedere nella costruzione dell’automa, tanto che nei metodi che rappresentano le produzioni non faccio altro che implementare per filo e per segno lo schema di traduzione che mi permette di realizzare in modo ricorsivo questo automa.

Per quanto riguarda come costruire gli automi più piccoli e come poi “combinarli”, mi baso semplicemente sull’algoritmo stesso. La cosa principale però è che ad ogni passo (quando viene creato un nuovo automa), l’automa ha sempre solo ed esclusivamente un solo stato iniziale ed uno di accettazione, così facendo si possono definire i metodi ricorsivamente.

Partendo da quello base infatti vengono prima creati due stati (uno iniziale ed uno finale) e viene creata una transizione con il simbolo letto nel token CHAR.
La stessa cosa vale quando la transizione tra i due stati deve essere con ‘epsilon’.

Unione

Per quanto riguarda l’unione essa viene fatta quando sappiamo che nell’attributo ereditato di expr1’ (e cioè nella coda) abbiamo il risultato dell’automa del figlio sinistro del padre, perciò a quel punto una volta fatta la visita del figlio centrale di expr1’, vengono utilizzati i due atomi contenuti nei due attributi (quello ereditato del nodo stesso e quello sintetizzato del figlio) per poter creare un nuovo automa non deterministico che rappresenti l’unione. Vengono infatti poi creati due nuovi stati (uno iniziale ed uno finale), ed aggiunte un totale di 4 transizioni con ‘epsilon’ come simbolo, in modo da connettere il nuovo stato iniziale ai due precedenti stati iniziali (che sono stati resi non-iniziali), e per connettere i precedenti stati finali (non più finali) con il nuovo stato finale.
Infine il risultato viene messo nuovamente nella coda, in quanto è come se così facendo lo inserissimo nell’attributo ereditato del figlio destro di expr1’.

Concatenazione

Nel caso della concatenazione è un po' più articolato, infatti non vengono creati nuovi stati, ma una volta che è stato valorizzato l'attributo ereditato di expr2' (ed è quindi nella coda perché ce l'ha messo il padre una volta visitato il figlio sinistro), viene visitato il figlio sinistro expr3, in modo da prenderne l'attributo sintetizzato che contiene l'automa risultante e lo concatena "dopo" (in quanto l'ordine conta) quello presente nell'attributo ereditato. Per fare questo prima rende non finale lo stato finale dell'automa di sinistra e non iniziale quello dell'automa di destra. Poi fa in modo che tutte le transizioni dello stato iniziale dell'automa di destra siano modificate, andando a settare come "inizio" della transizione lo stato finale dell'automa di sinistra, e vengono infatti poi aggiunte a questo. Così vengono "trasferite" le transizioni allo stato indietro.
Al termine, anche in questo caso inserisce l'automa risultante nella coda, in quanto così sarà accedibile dal figlio destro.

Kleene

Per quanto riguarda infine Kleene, questo viene fatto semplicemente una volta che viene fatto expr3' → * (e quindi nell'attributo ereditato del nodo c'è l'automa su cui applicare Kleene). In questo caso infatti, vengono prima creati due nuovi stati (uno iniziale ed uno finale), e poi vengono aggiunte 4 transizioni con 'epsilon', una che collega direttamente il nuovo stato iniziale al nuovo stato finale, una che collega il nuovo stato iniziale al precedente stato iniziale, una che fa lo stesso per gli stati finali (ma al contrario) ed infine una transizione che permette di tornare indietro andando dal precedente stato finale al precedente stato iniziale.
Alla fine, l'automa risultante viene ritornato verso l'alto come un attributo sintetizzato.

Transizioni con Epsilon

Tutte le produzioni che vanno in Epsilon, come già detto fanno quel giochetto per cui prima viene prelevato un attributo ereditato dalla coda, e poi viene fatto "salire" come attributo sintetizzato, perché questo "passaggio" indica che la sequenza (in quanto ad esempio può esservi una situazione di "associatività" come nel caso di ' a + b + c + d ') è finita e deve risalire il valore.

Coda

Come si può notare nel codice molto spesso non viene fatto direttamente il pop() degli attributi ereditati, in quanto potrebbero essere necessari se si "scende ancora nell'albero", perciò prima viene fatto il peek() (che non rimuove l'elemento) e solo alla fine della visita del nodo figlio questo viene eliminato dalla coda.

Conclusione

Una volta specificate in modo generico le 3 operazioni, lavorando sempre e solo con stati iniziali e di accettazione, è indifferente cosa si trova all'interno degli automi.

(NOTA: per quanto riguarda il caso in cui si utilizzano le parentesi, il valore viene semplicemente fatto salire verso l'alto).

L'automa risultante alla fine inoltre, data la grammatica che segue lo schema visto a lezione, è in grado di seguire correttamente le precedenze e l'associatività degli operatori.

Al termine di questa prima fase, l'obiettivo è avere nell'attributo 'automata' della classe l'automa calcolato dall'algoritmo.

Accettazione delle stringhe

In questa seconda fase invece, una volta disponibile l'automa, vengono prima ricostruite le stringhe (a partire dai singoli caratteri) e salendo verso l'alto, una volta identificata una stringa essa viene passata all'automa che restituirà un boolean, che indica con 'true' se è stata accettata e con 'false' il contrario.

In questo caso, sia per quanto riguarda la costruzione delle singole stringhe (come sequenza di caratteri), sia per quanto riguarda la costruzione della sequenza di stringhe, non ho preventivamente realizzato un albero di derivazione di prova in quanto in realtà il funzionamento è molto banale.
Infatti per quanto riguarda le stringhe non deve far altro che scendere verso il figlio destro (utilizzando la ricorsione a destra) fino a raggiungere il caso in cui la stringa si cancella con epsilon. Arrivato fin qui fa risalire la stringa concatenando sempre come primo carattere (perché utilizza la ricorsione a destra) il lessema del token CHAR presente come figlio sinistro alla stringa ritornata verso l'alto dalla sottostringa presente nel figlio destro.
Per quanto riguarda invece la sequenza di stringhe in realtà fa praticamente la stessa cosa, infatti scende fino a quando la sequenza di stringhe non si cancella, ma con le stringhe. Però prima di concatenare un nuovo elemento alla sequenza di stringhe ritornata dal figlio destro, esegue prima la stringa presa dal figlio sinistro all'interno dell'automa e prende il risultato (true = "OK" e false = "KO") concatenandolo all'inizio della sequenza di stringhe risultato.

Costruzione delle stringhe

Per costruire le stringhe viene utilizzato l'attributo sintetizzato 'stringToAccept', che viene appunto composto in modo ricorsivo salendo verso l'alto facendo semplicemente una concatenazione di stringhe, infatti salendo aggiunge il nuovo carattere come primo carattere della stringa da accettare.

Prima però scende a destra fino a cancellare il simbolo non terminale della stringa della stringa, per poi ricostruirla salendo verso l'alto (utilizzando appunto per questo solo un attributo sintetizzato).

NOTA: come accennato prima, è ammessa ovviamente anche la stringa vuota, in quanto potendo definire tutte le espressioni regolari utilizzando anche epsilon, allora è ovvio che vi debba essere questa possibilità.

Costruzione della lista di stringhe

Come già detto, anche in questo caso fa la stessa cosa ma con le stringhe al posto dei caratteri, utilizzando un solo attributo sintetizzato chiamato 'accept'.

Infatti, prima scende verso il basso, poi concatena alla lista di stringhe di risultati il nuovo risultato restituito dall'esecuzione dell'automa sulla stringa restituita dal figlio sinistro.

Infine la lista di risultati salirà fino al nodo radice 'init', e verrà preso dal rispettivo attributo sintetizzato all'interno della classe Calculator.java.

Esecuzione dell'automa

Come accennato prima, nella classe ThompsonNFA.java che rappresenta l'automa, è presente inoltre il metodo 'accept', che presa una stringa (quella da accettare) è in grado di dire se essa è accettata dall'automa o meno.

Per fare questo viene prima di tutto definita ovviamente la funzione 'epsilonClosure()', che restituisce per lo stato passato, tutti gli stati raggiungibili con 'epsilon transizioni' a partire dallo stesso (andando poi ad

applicare in modo incrementale questo metodo per poter raggiungere anche gli stati che si trovano dopo un solo passo).

Successivamente viene anche prima definita una funzione ausiliaria 'move()', che a partire da uno stato passato in input ed un carattere (quello letto dalla stringa), restituisce l'insieme di stati raggiungibili con transizioni che hanno come simbolo lo stesso passato (dove però ad ogni stato viene prima applicata la 'epsilon closure').

A questo punto, nel metodo stesso, ad ogni passo legge un simbolo in sequenza della stringa passata, e controlla quali sono gli stati raggiungibili con quel simbolo a partire dagli stati correnti dell'automa (che sono appunto un insieme di stati dato che si tratta di un automa non deterministico).
Se poi mentre vengono computati i nuovi stati raggiungibili, arriviamo ad un passo in cui non c'è nessun nuovo stato raggiungibile, vuol dire che l'automa si è bloccato e termina non accettando la stringa.
Altrimenti termina la stringa, e alla fine controlla se tra gli stati raggiunti ve ne è almeno uno di accettazione. Se è presente allora la stringa è accettata, altrimenti no.

Lista di stringhe

Trattandosi come già detto di una serie di stringhe da accettare, anche in questo caso viene costruita salendo verso l'alto una lista di stringhe (che sono "OK" o "KO"), dove dato che stiamo tornando indietro, ogni risultato viene sempre inserito come primo elemento (in modo che siano ordinati correttamente).

Conclusione

Al termine di tutto viene stampata nella classe Calculator la lista di stringhe, andando a stampare una stringa dopo l'altra.

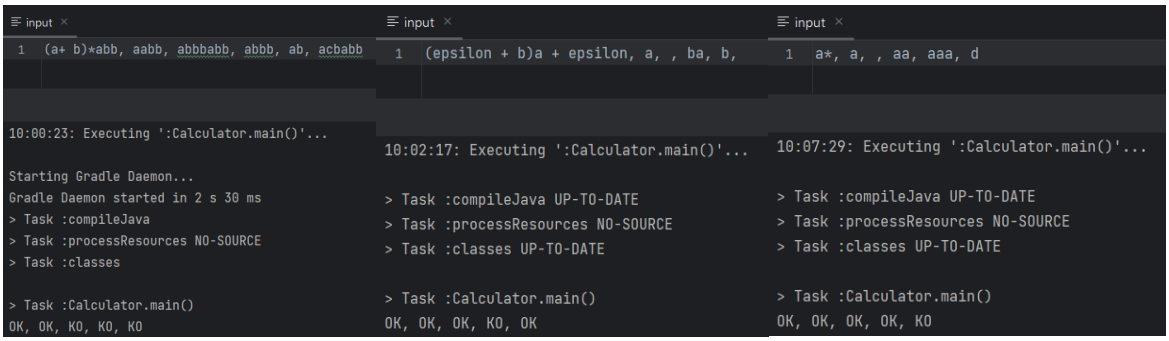
Codice

Per maggiori chiarimenti confrontare direttamente i commenti nel codice.

Esecuzione del programma

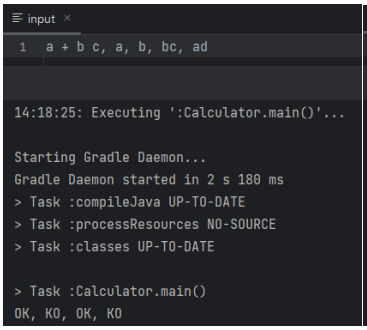
Come già detto, sono riuscito ad utilizzare il tool solo passando in input il percorso del file contenente la stringa da passare alla grammatica. Perciò ho semplicemente creato un file 'input.txt' e passato come argomento il cammino assoluto a questo.

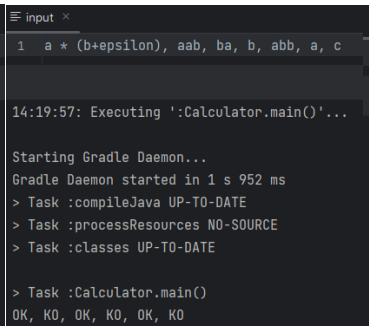
Alcuni input ed output del programma sono i seguenti:

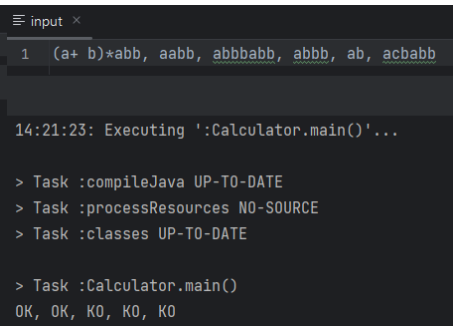


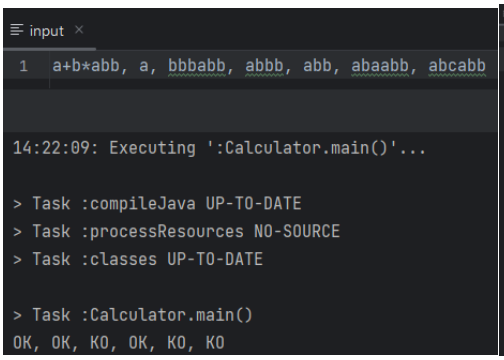
Interessante è ad esempio il secondo esempio, in cui vi sono parecchie epsilon.

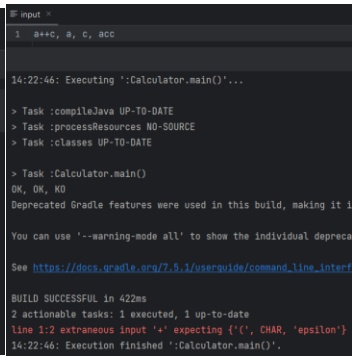
ESEMPI nella specifica del progetto:



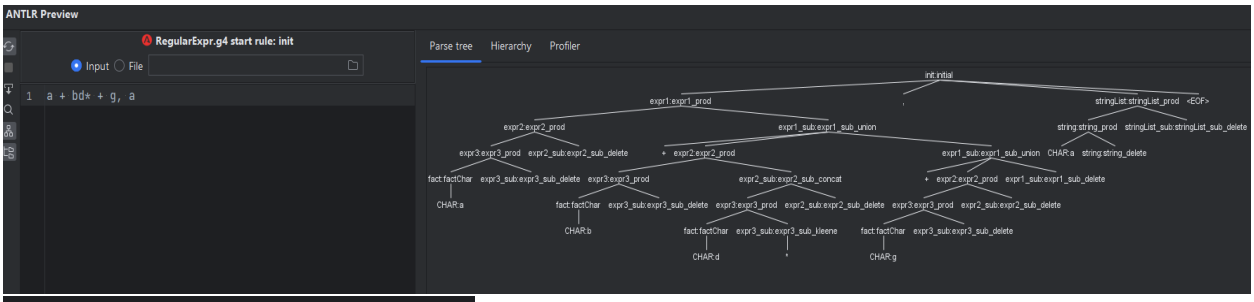


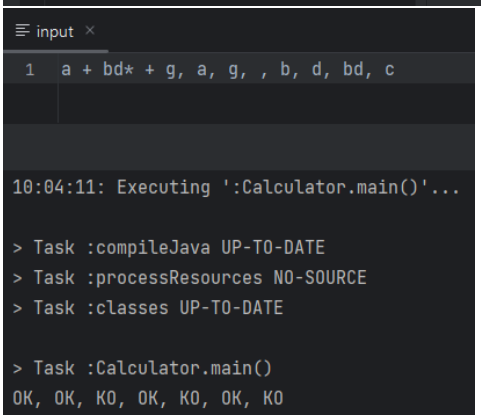






(Questo è l'esempio dell'albero di prima)





Come si può notare quindi i risultati sono sempre quelli attesi.