ELABORATO: Parsing in ANTLR4

Carlo Rossanigo (Camillo Fiorentini)

Indice

1 INTRODUZIONE

Scopo di questo elaborato è esplorare l'argomento dell'analisi sintattica (parsing) nel contesto del software ANTLR4. A tal fine, per comprendere l'attività di parsing e la sua implementazione in ANTLR4, è necessario introdurre le nozioni fondamentali e i concetti chiave relativi ai linguaggi e alle grammatiche. Nel corso dell'elaborato, verranno citati e discussi due libri di riferimento fondamentali per approfondire questi temi. Il primo è "The Definitive ANTLR 4 Reference" di Terence Parr [2], che offre una guida completa all'uso di ANTLR4. Il secondo è "Linguaggi modelli complessità" di Giorgio Ausiello, Fabrizio d'Amore, Giorgio Gambosi e Luigi Laura [1], che fornisce una panoramica dettagliata sui linguaggi formali e sulla complessità computazionale.

2 LINGUAGGI

In questo capitolo concentriamo la nostra attenzione sulla definizione di linguaggio; assumeremo come noti i concetti elementari di insieme, insieme delle parti, di cardinalità di insiemi, di relazione e di operazione. Assumeremo noti anche i significati dei quantificatori e dei connettivi logici, mentre, pur di risultare pedanti, recupereremo i concetti di analisi lessicale, sintattica, semantica e le definizioni base delle strutture algebriche di nostro interesse.

2.1 Nozioni e concetti matematici di base

Ricordiamo in primis la differenza fra sintassi e semantica:

Def 2.1 (**Sintassi**)**.** La sintassi rappresenta l'insieme delle regole grammaticali che governano le relazioni tra le parole in una frase.

Def 2.2 (**Semantica**)**.** La semantica è l'insieme dei significati da attribuire alle frasi sintatticamente corrette costruite nel linguaggio.

È importante sottolineare che i primi passaggi nella creazione di un'applicazione includono l'analisi lessicale, sintattica e semantica:

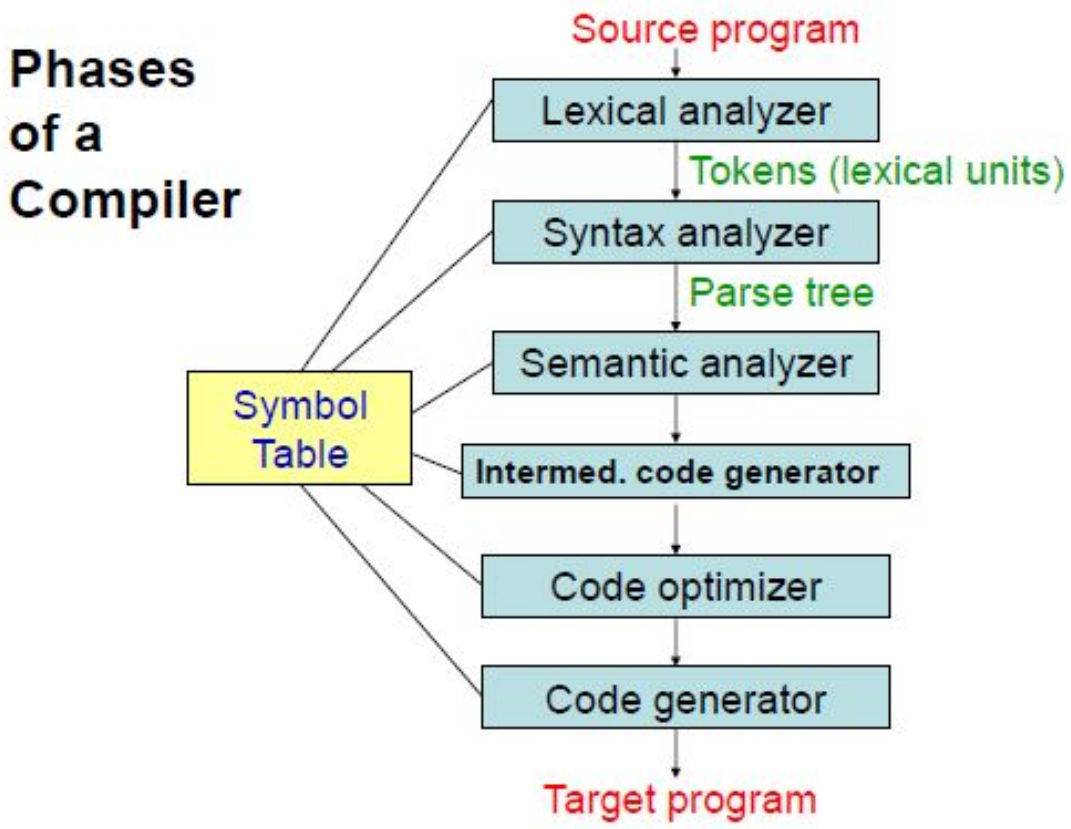


Figura 1: Creazione di un'applicazione

riprendiamo dunque le definizioni operative di Lexer, Parser e analizzatore semantico.

Def 2.3 (**Analizzatore lessicale**)**.** Il lexer - analizzatore lessicale - legge il flusso di caratteri in ingresso e lo suddivide in unità significative dette token. Ogni token rappresenta una categoria lessicale (ad esempio, una parola chiave, un numero, un identificatore, un simbolo).

Def 2.4 (**Analizzatore sintattico**)**.** Il parser, detto anche analizzatore sintattico, una volta trasferitigli i token prodotti dal lexer, organizza tali token secondo la grammatica del linguaggio, creando un Abstract Syntax Tree (AST): l'AST rappresenta la struttura gerarchica del programma e le relazioni tra le diverse parti.

Def 2.5 (**Analizzatore semantico**)**.** L'analizzatore semantico verifica che l'AST costruito dal parser rispetti le regole semantiche del linguaggio, come la coerenza dei tipi e l'uso corretto delle variabili e delle funzioni, determinandone il significato. Durante questa fase vengono eseguiti controlli di tipo, risoluzione dei nomi e verifiche logiche.

A questo punto, prima di poter parlare di linguaggi è doveroso recuperare le definizioni di alcune strutture algebriche di nostro interesse:

Def 2.6 (**Semigruppo**)**.** Sia A un insieme. La coppia $\langle A, \circ \rangle$ viene detta semigruppo se l'operazione binaria $\circ$ soddisfa la proprietà associativa:

$$\forall x, y, z \in A \quad (x \circ y) \circ z = x \circ (y \circ z)$$

Qualora valga anche la proprietà commutativa:

$$\forall x, y \in A \quad x \circ y = y \circ x$$

il semigruppo è detto commutativo.

Esempio 2.1. Sono semigruppi le seguenti coppie:

1. $(\mathbb{N}, +)$ semigruppo dei numeri naturali con l'operazione di somma
2. $(\mathbb{Z}, \cdot)$ semigruppo degli interi con l'operazione di moltiplicazione
3. $(\{1, 2, 3, 4\}, \max)$ semigruppo dell'insieme $\{1, 2, 3, 4\}$ con l'operazione di massimo

Passiamo alla definizione di monoide, un semigruppo dotato di un elemento neutro:

Def 2.7 (**Monoide**)**.** La terna $\langle A, \circ, e \rangle$ viene detta monoide se $\langle A, \circ \rangle$ è un semigruppo, ed esiste $e \in A$ tale che:

$$\forall x \in A \quad e \circ x = x \circ e = x$$

L'elemento e è detto elemento neutro o unità del monoide; se l'operazione $\circ$ è commutativa, il monoide è detto commutativo.

Esempio 2.2. Sono monoidi le seguenti terne:

1. $(\mathbb{N}, +, 0)$ monoide dei numeri naturali con l'operazione di somma e elemento neutro 0
2. $(\mathcal{M}_n(\mathbb{R}), \cdot, I_n)$ monoide delle matrici $n \times n$ con il prodotto righe per colonne e unità è I_n
3. $(\mathbb{Z}, \cdot, 1)$ monoide degli interi con l'operazione di moltiplicazione e elemento neutro 1

Un gruppo è un monoide in cui ogni elemento ammette un inverso:

Def 2.8 (**Gruppo**)**.** La terna $\langle A, \circ, e\rangle$ viene detta gruppo se $\langle A, \circ, e\rangle$ è un monoide e $\forall x \in A\ \exists y \in A$ tale che:

$$x \circ y = y \circ x = e$$

L'elemento y è detto inverso di x, e si denota x^{-1}. Ovviamente, un gruppo è commutativo se lo è l'operazione $\circ$.

Esempio 2.3. Sono gruppi le seguenti terne:

1. $(\mathbb{Z}, +)$ gruppo degli interi con l'operazione di addizione
2. $(D_3, \cdot)$ gruppo diedrale D_3, simmetrie di un triangolo equilatero
3. $(S_3, \circ)$ gruppo di permutazioni su un insieme di 3 elementi

Risulta chiaro che un gruppo è anche un monoide, così come un monoide è anche un semigruppo.
Terminiamo con la definizione di semigruppo libero:

Def 2.9 (**Semigruppo Libero**)**.** Un semigruppo libero S su un insieme X è un semigruppo con una corrispondenza biunivoca tra X e un sottoinsieme di S, tale che ogni funzione da X in un semigruppo T si estende in modo unico a un omomorfismo di semigruppi da S a T.
Ricordiamo che un omomorfismo è una funzione tra due semigruppi che mantiene la struttura algebrica, cioè preserva l'operazione del semigruppo tra due suoi elementi.

Esempio 2.4. Il semigruppo libero generato da un singolo elemento x è $\langle x\rangle = \{x^n \mid n \in \mathbb{N}\}$, dove l'operazione è la moltiplicazione.

2.2 Alfabeti, stringhe e linguaggi

Definiamo ora i concetti elementari di alfabeto e stringa:

Def 2.10 (**Alfabeto**)**.** Un alfabeto è un insieme Σ finito e non vuoto di simboli, detti caratteri.

Def 2.11 (**Stringa**)**.** Dato un alfabeto Σ , una stringa è una sequenza di simboli dell'alfabeto.

Per convenzione, la stringa vuota è indicata con ε.
Indicheremo spesso l'alfabeto con la lettera Σ, mentre useremo i termini parola e stringa in modo equivalente; si noti che la richiesta di finitezza non implica che non si possano generare infinite parole, mentre la richiesta che l'alfabeto sia non vuoto è del tutto naturale: se fosse $\Sigma = \emptyset$, non si potrebbe generare alcuna stringa.
Un esempio di alfabeto è l'alfabeto binario $\Sigma = \{0, 1\}$, da cui si possono generare infinite stringhe di 0 e 1.
Considerando i caratteri di un alfabeto Σ come generatori di un semigruppo, possiamo fare riferimento al monoide libero corrispondente:

Def 2.12 (**Monoide libero**)**.** Dato un alfabeto Σ, denotiamo con $\langle \Sigma^*, \circ, \varepsilon \rangle$ il monoide libero su Σ, detto monoide sintattico. Gli elementi di Σ^* sono detti parole e ε è la parola vuota. L'operazione di concatenazione $\circ$ è definita come segue:

$$x \circ y = xy \quad \text{per } x, y \in \Sigma^*$$

dove xy è la giustapposizione di x e y.

È chiaro che vale la proprietà:

$$x \circ \varepsilon = x = \varepsilon \circ x \quad \forall x \in \Sigma^*$$

Si può concatenare una stringa con se stessa più volte usando l'esponente:

$$x^h = \underbrace{x \circ x \circ \ldots \circ x}_{h \text{ volte}}$$

Si indica con $|x|$ la lunghezza della stringa x, ovvero il numero di caratteri che la compongono.
Possiamo ora definire cos'è un linguaggio:

Def 2.13 (**Linguaggio**)**.** Dato un alfabeto Σ, un linguaggio è un sottoinsieme di Σ^*.
Definiamo il linguaggio vuoto, indicato con Λ, il linguaggio che non contiene alcuna parola.
Il linguaggio che contiene solo la parola vuota è $L = \{\epsilon\}$.

È interessante considerare anche la stringa inversa di una parola x, ottenuta invertendo l'ordine dei caratteri di x:

Def 2.14 (**Stringa inversa**)**.** Data una stringa x, chiamiamo inversa-o riflessa-la stringa seguente:

$$\tilde{x} = \begin{cases} x & \text{se } x = a \text{ o } x = \epsilon, \\ a\tilde{y} & \text{se } x = ya, \end{cases}$$

2.3 Operazioni sui linguaggi

Dati due linguaggi L_1 e L_2 si possono definire su di essi diverse operazioni, tra cui quelle binarie di unione, intersezione e concatenazione - prodotto - e quelle unarie di complementazione e iterazione.
In particolare unione, intersezione e complemento sono definite come normali operazioni su insiemi, mentre concatenazione e iterazione sono definite per mezzo del monoide sintattico $\langle \Sigma^*, \circ, \varepsilon \rangle$.

Def 2.15 (**Unione**)**.** L'unione di due linguaggi L_1 e L_2 è definita come:

$$L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$$

Def 2.16 (**Intersezione**)**.** L'intersezione di due linguaggi L_1 e L_2 è definita come:

$$L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$$

Si noti che $L_1 \cup \Lambda = L_1$ e $L_1 \cap \Lambda = \Lambda$.

Def 2.17 (**Concatenazione**)**.** Il prodotto concatenato di due linguaggi L_1 e L_2 è definito come:

$$L_1 \circ L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Vale $L_1 \circ \Lambda = \Lambda \circ L_1 = \Lambda$ e $L_1 \circ \varepsilon = \varepsilon \circ L_1 = L_1$.
Inoltre, sebbene si utilizzi lo stesso simbolo $\circ$ per indicare la concatenazione di stringhe e di linguaggi, le due operazioni sono diverse e dunque il contesto consente di evitare ambiguità.

Esempio 2.5. Dati i linguaggi $L_1 = \{a^n \mid n \in \mathbb{N}\}$ e $L_2 = \{b^m \mid m \in \mathbb{N}\}$, la concatenazione dei due linguaggi è data da:

$$L_1 \circ L_2 = \{a^n b^m \mid n, m \in \mathbb{N}\}$$

Dati i linguaggi $L_1 = \{\text{BIO}, \text{ENO}\}$ e $L_2 = \{\text{LOGIA}\}$ La concatenazione dei due linguaggi è data da:

$$L_1 \circ L_2 = \{\text{BIOLOGIA}, \text{ENOLOGIA}\}$$

Def 2.18 (**Complemento**)**.** Fissato un alfabeto Σ, il complemento di un linguaggio L è definito come:

$$\overline{L} = \Sigma^* \setminus L$$

cioè il linguaggio $\overline{L}$ è costituito dalle parole appartenenti a Σ^* ma non a L, ovvero:

$$\overline{L} = \{x \in \Sigma^* \mid x \notin L\}.$$

Il concetto di potenza inteso come iterazione della concatenazione viene esteso ai linguaggi mediante la seguente definizione:

Def 2.19 (**Potenza**)**.** La potenza di un linguaggio L^h per $h \geq 0$ è definita ricorsivamente come:

$$L^0 = \{\varepsilon\}, \quad L^h = L \circ L^{h-1} \text{ per } h \geq 1$$

Ricordiamo che ε è la parola vuota.

Con questa definizione possiamo indicare con Σ^h l'insieme delle stringhe di lunghezza h sull'alfabeto Σ.

Def 2.20 (**Chiusura riflessiva**)**.** La chiusura riflessiva di un linguaggio L, denotata L^*, è definita come:

$$L^* = \bigcup_{h=0}^{\infty} L^h$$

L'operatore $*$ prende il nome di iterazione o Stella di Kleene.

Def 2.21 (**Chiusura non riflessiva**)**.** La chiusura non riflessiva di un linguaggio L, denotata $L^\dagger$, è definita come:

$$L^\dagger = \bigcup_{h=1}^{\infty} L^h$$

Si noti che $L^* = L^\dagger \cup \{\varepsilon\}$: in particolare dato un qualunque linguaggio L, vale $\varepsilon \in L^*$, e si ha $\Lambda^* = \{\varepsilon\}$.
Dunque per un linguaggio L, la stella di Kleene L^* include la parola vuota ε e tutte le concatenazioni di parole in L, mentre la chiusura positiva L^+ include tutte le concatenazioni di parole in L a partire da L^1, escludendo la parola vuota.

Esempio 2.6. Consideriamo i seguenti esempi:

1. Dati $L_1 = \{a, ab\}$ e $L_2 = \{b, ba\}$, calcolare $L_1 \circ L_2$, $L_1 \cup L_2$, $L_1 \cap L_2$:

$$L_1 \circ L_2 = \{ab, aba, abb, abba\}$$
$$L_1 \cup L_2 = \{a, ab, b, ba\}$$
$$L_1 \cap L_2 = \emptyset$$

2. Se $L = \{bb\}$, allora $L^+ = \{b^{2n} \mid n \geq 1\}$.
3. Se $L = \{aa\}$, allora $L^* = \{a^{2n} \mid n \geq 0\}$.

Abbiamo quindi fornito una panoramica delle operazioni fondamentale nell'analisi formale dei linguaggi; a questo punto possiamo occuparci della questione della cardinalità di un linguaggio.

2.4 Cardinalità dei linguaggi

Trattiamo in questo paragrafo la questione della cardinalità di un linguaggio.

Def 2.22 (**Cardinalità di un linguaggio**)**.** Per cardinalità di un linguaggio si intende il numero di parole del linguaggio stesso. Essa può essere finita o infinita: nel primo caso, per rappresentare il linguaggio basterà elencare le sue parole; nel secondo, si utilizzerà la notazione propria della teoria degli insiemi - l'assioma di specificazione - e si esplicherà il linguaggio mediante le sue proprietà.

Esempio 2.7. Un linguaggio con cardinalità finita è il linguaggio delle stringhe binarie di lunghezza massima 3:
$\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111\}$.
Questo linguaggio ha un totale di 15 parole. Il linguaggio vuoto Λ è chiaramente un linguaggio di cardinalità finita; anche il linguaggio costituito dalla sola parola vuota $L = \{\varepsilon\}$ è un linguaggio di cardinalità finita.
Si noti che c'è una differenza tra Λ, il linguaggio vuoto, e $L = \{\varepsilon\}$, il linguaggio costituito dalla sola stringa nulla.
Un esempio di linguaggio con cardinalità infinita è invece il linguaggio $L_1 = \{b^{2n} \mid n \geq 0\}$.

2.5 Espressioni regolari

Introduciamo, prima di terminare il capitolo, uno strumento di natura algebrica che ci consente di descrivere tutti i linguaggi appartenenti a un'importante classe.

Def 2.23 (**Espressioni regolari**). Dato un alfabeto Σ e dato l'insieme dei simboli $\{+, *, (,), \circ, \emptyset\}$, si definisce espressione regolare sull'alfabeto Σ una stringa $r \in (\Sigma \cup \{+, *, (,), \circ, \emptyset\})^+$ tale che valga una delle seguenti condizioni:

1. $r = \emptyset$;
2. $r \in \Sigma$;
3. $r = (s + t)$ oppure $r = (s \circ t)$ oppure $r = s^*$, dove s e t sono espressioni regolari sull'alfabeto Σ.

Per convenzione, date due espressioni regolari s e t, si scrive st per $(s \circ t)$ e si assume che il simbolo $*$ abbia la precedenza sul simbolo $\circ$, che a sua volta ha precedenza sul simbolo $+$; tenendo conto dell'associatività di tali operazioni, si possono eliminare le parentesi.
La figura seguente mostra la corrispondenza tra un'espressione regolare e il linguaggio che essa rappresenta.

RE r	**Language L(r)**
a	**{a}**
ϵ	$\{\epsilon\}$
r | s	L(r) ∪ L(s)
rs	$\{rs \mid r \in \text{L(r)}, s \in \text{L(s)}\}$
r^+	L(r) ∪ L(rr) ∪ L(rrr) ∪ . . . (any number of r's concatenated)
r^* $(\text{r}^* = \text{r}^+ \mid \epsilon)$	$\{\epsilon\} \cup \text{L(r)} \cup \text{L(rr)} \cup \text{L(rrr)} \cup \ldots$
(s)	L(s)

Figura 2: Corrispondenza tra Espressione regolare e Linguaggio generato

Le espressioni regolari consentono di rappresentare linguaggi mediante un'opportuna interpretazione dei simboli che le compongono.

Esempio 2.8. L'espressione regolare che su $\Sigma = \{a, b\}$ definisce l'insieme delle stringhe il cui terzultimo carattere è b è $(a|b)^*b(a|b)(a|b)$.

3 GRAMMATICHE

In questo capitolo concentriamo la nostra attenzione sul cosiddetto approccio generativo, nel quale si utilizzano opportuni strumenti quali le grammatiche formali per costruire stringhe di un linguaggio tramite un insieme prefissato di regole, dette regole di produzione. Introdurremo dunque la definizione di grammatica formale, la classificazione secondo Chomsky dei tipi di grammatiche e le caratteristiche principali delle grammatiche di tipo diverso.

3.1 Grammatiche di Chomsky

In generale, una grammatica è una notazione formale con cui esprimere in modo rigoroso la sintassi di un linguaggio. Le grammatiche formali presentate in questo capitolo sono denominate Grammatiche di Chomsky poiché introdotte dal linguista Noam Chomsky al fine di rappresentare i procedimenti sintattici elementari alla base della costruzione di frasi della lingua inglese.
Pur presentando dei limiti, tali grammatiche formali hanno un ruolo fondamentale nello studio delle proprietà sintattiche dei programmi e dei linguaggi di programmazione; vediamo ora in modo più preciso la definizione di grammatica formale:

Def 3.1 (**Grammatica formale**)**.** Una grammatica formale è una quadrupla $G = (V_T, V_N, P, S)$ in cui:

- V_T è un insieme finito di simboli terminali,
- V_N è un insieme finito di simboli non terminali,
- P è un insieme finito di produzioni, ossia regole di riscrittura $\alpha \to \beta$, dove α e β sono stringhe: $\alpha \in (V_T \cup V_N)^+$, $\beta \in (V_T \cup V_N)^*$,
- S è un particolare simbolo non terminale detto simbolo iniziale della grammatica.

Per convenzione, l'unione $V = V_T \cup V_N$ è detta vocabolario della grammatica. I simboli terminali sono caratteri o stringhe su un alfabeto Σ, mentre i simboli non terminali sono meta-simboli che rappresentano categorie sintattiche. Le produzioni riscrivono una stringa non nulla α in una nuova stringa β, che può essere eventualmente nulla. I simboli terminali si indicano con lettere minuscole, i meta-simboli con lettere maiuscole, e le lettere greche rappresentano stringhe

miste di simboli terminali e meta-simboli.
Ciò porta dunque a una distinzione fra forme di frase e frasi:

Def 3.2 (**Forma di frase**)**.** Una forma di frase (sentential form) è una qualsiasi stringa comprendente sia simboli terminali sia meta-simboli, ottenibili dal simbolo iniziale applicando una o più regole di produzione.

Def 3.3 (**Parola**)**.** Una parola (sentence) è una forma di frase che comprende solo simboli terminali.

Esempio 3.1. Data la grammatica $G = \{V_T, V_N, P, S\}$ con $V_T = \{0, 1\}$, $V_N = \{S, Z, U\}$, e $P = \{S \to ZU, Z \to 0, Z \to 0Z, U \to 1, U \to U1\}$, scriviamo 3 forme di parola e 3 parole.

3 forme di frase sono:

- $ZU1$ (derivazione: $S \Rightarrow ZU \Rightarrow ZU1$)
- $Z1$ (derivazione: $S \Rightarrow ZU \Rightarrow Z1$)
- $0U1$ (derivazione: $S \Rightarrow ZU \Rightarrow 0U \Rightarrow 0U1$)

3 parole sono:

- 01 (derivazione: $S \Rightarrow ZU \Rightarrow 01$)
- 011 (derivazione: $S \Rightarrow ZU \Rightarrow ZU1 \Rightarrow Z11 \Rightarrow 011$)
- 001 (derivazione: $S \Rightarrow ZU \Rightarrow 0ZU \Rightarrow 001$)

Introduciamo formalmente il significato di derivazione e di sequenza di derivazione, benché si sia già notato nell'esempio:

Def 3.4 (**Derivazione**)**.** Date due stringhe α e $\beta \in V^*$, $\alpha \neq \epsilon$, si dice che β deriva direttamente da α (e si scrive $\alpha \Rightarrow \beta$) se α e β possono essere decomposte in

$$\alpha = \eta A \delta, \beta = \eta \gamma \delta$$

ed esiste la produzione $A \to \gamma$.

Si dice che β deriva da α (anche non direttamente) se esiste una sequenza di N derivazioni dirette che da α possono infine produrre β:

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_N = \beta$$

Def 3.5 (**Sequenza di derivazione**)**.** Una sequenza di derivazione è la sequenza di passi che produce una forma di frase σ dal simbolo iniziale S:

- $S \Rightarrow \sigma$ se σ deriva da S in un solo passo, cioè con una sola applicazione di produzioni.
- $S \Rightarrow^+ \sigma$ se σ deriva da S con una o più applicazioni di produzioni.
- $S \Rightarrow^* \sigma$ se σ deriva da S con zero o più applicazioni di produzioni.

Esempio 3.2. Data la grammatica dell'Esempio 3.1, una derivazione per la parola 0011 è:

$S \Rightarrow ZU$ (con $S \to ZU$) $\Rightarrow 0ZU1$ (con $Z \to 0Z$ e $U \to U1$) $\Rightarrow 0011$ (con $Z \to 0$ e $U \to 1$)

Definiamo ora il linguaggio generato da una grammatica, prima di introdurre la classificazione delle grammatiche secondo Chomsky:

Def 3.6 (**Linguaggio generato da una grammatica**)**.** Dato una grammatica G, si dice linguaggio $L(G)$ generato da G l'insieme delle frasi derivabili dal simbolo iniziale S applicando le produzioni P:

$$L(G) = \{s \in V_T^* \mid S \Rightarrow^* s\}$$

Un linguaggio può essere generato da grammatiche distinte; in tal caso, se esistono due grammatiche G_1 e G_2 che generano lo stesso linguaggio, G_1 e G_2 sono dette equivalenti.
È chiaro che stabilire se due grammatiche siano equivalenti è in generale un problema indecidibile; la questione cambia se ci si restringe a tipi particolari di grammatiche, che abbiano regole di produzione alquanto semplici.

Esempio 3.3. Data la grammatica $G = \{V_T, V_N, P, S\}$ definita come segue:

$$G = \{V_T, V_N, P, S\} \quad \text{con} \quad V_T = \{a, b\}, \quad V_N = \{C, N\}, \quad S = C,$$

con le seguenti produzioni:

$$P = \{C \to b \mid bN, \quad N \to b \mid a \mid bN \mid aN\}$$

Il linguaggio generato da G è

$$L(G) = \{s \in \{a, b\}^* \mid s = b \ \vee \ s = b(a \mid b)^*\}.$$

In altre parole, le stringhe in $L(G)$ sono del tipo b, ba, bab, $babb$, $baab$.

Esempio 3.4. Consideriamo la grammatica $G = \{V_T, V_N, P, S\}$ con $V_T = \{a, b\}$, $V_N = \{S, X, Y, K\}$, $S = S$, e le seguenti produzioni:

$$\begin{aligned} &S \to KX, \\ &K \to a \mid b, \\ &aX \to aYb, \\ &bX \to bYa, \\ &Y \to aYb \mid a. \end{aligned}$$

Il linguaggio generato da G è $L(G) = \{s \in V_T^* \mid s = a^{n+1}b^n, \ b^n a^{n+1} b^n a, \ n \geq 0, \ n \in \mathbb{N}\}$. Esempi di stringhe in $L(G)$ sono ab, aab, $aababb$, ba, $baab$.

Esempio 3.5. Data la grammatica G_1 con le produzioni $S \to aS \mid b$ e la grammatica G_2 con le produzioni $S \to b \mid Ab$, $A \to Aa \mid a$. Verifichiamo se G_1 e G_2 sono equivalenti in termini di linguaggio generato:

- G_1 genera il linguaggio $L(G_1) = \{a^j b \mid j \geq 0,\ j \in \mathbb{N}\}$, con parole come b, ab, aab, ecc.
- G_2 genera lo stesso linguaggio $L(G_2) = \{a^j b \mid j \geq 0,\ j \in \mathbb{N}\}$.

Quindi, le due grammatiche G_1 e G_2 sono equivalenti perché generano lo stesso linguaggio.

3.2 Classificazione di Chomsky

Introduciamo ora la classificazione di Chomsky delle grammatiche in base alla loro complessità. Chomsky ha definito quattro tipi di grammatiche, ognuno con caratteristiche specifiche nelle produzioni. Questi tipi formano una gerarchia:

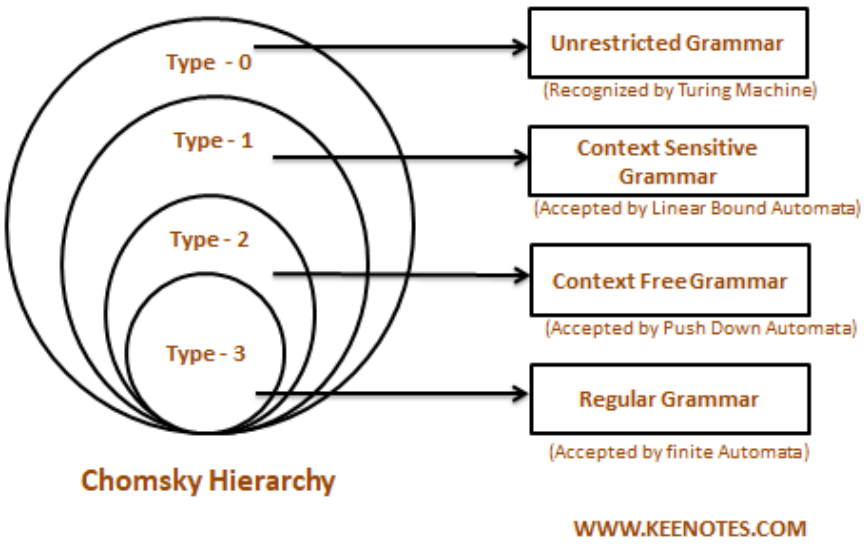


Figura 3: Classificazione delle grammatiche

Le grammatiche di tipo 3, note anche come grammatiche regolari, ammettono produzioni vincolate a forme lineari, sia a destra che a sinistra:

Def 3.7 (Grammatiche di tipo 3 - Regolari). Le grammatiche di tipo 3, note anche come grammatiche regolari, ammettono produzioni vincolate a forme lineari, sia a destra che a sinistra:

$$A \to \sigma \quad \text{o} \quad A \to \sigma B$$

dove $A, B \in V_N$ e $\sigma \in V_T^*$.

È importante notare che tutte le produzioni di una grammatica di tipo 3 devono essere o lineari a destra o lineari a sinistra e non possono essere miste,cioè

con alcune produzioni lineari a destra e altre lineari a sinistra. Inoltre, σ può essere la stringa vuota ϵ. È spesso conveniente trasformare queste grammatiche in forma strettamente lineare, sostituendo $\sigma \in V_T^*$ con un singolo carattere $a \in V_T$:

$$X \to a \quad \text{o} \quad X \to aY$$

per lineare a destra, e

$$X \to a \quad \text{o} \quad X \to Ya$$

per lineare a sinistra.

Esempio 3.6. Riportiamo di seguito 3 esempi di grammatiche regolari:

- **Grammatica Lineare a Destra:**

  $$G = \{V_T, V_N, P, S\} \quad \text{con} \quad V_T = \{a, b\}, \quad V_N = \{S\}$$

  le cui produzioni sono:

  $$P = \{S \to aS \mid b\}$$

  Questa grammatica genera il linguaggio $L(G) = \{a^m b \mid m \geq 0\}$.

- **Grammatica $G2$ lineare a sinistra, resa strettamente lineare a sinistra:**

  $$G2 = \{V_T, V_N, P, S\} \quad \text{con} \quad VT = \{a, b\}, \quad V_N = \{S\}$$

  le cui produzioni sono:

  $$P = \{S \to Sa \mid b\}$$

  $G2$ genera stringhe della forma $a^n b$, per $n \geq 0$.

- **Grammatica $G3$ lineare a destra:**

  $$S \to cC$$
  $$C \to iI$$
  $$I \to aA$$
  $$A \to o$$

  Questa grammatica genera la stringa "ciao".

Def 3.8 (**Grammatiche di tipo 2 - Acontestuali**)**.** Le grammatiche di tipo 2, dette anche grammatiche acontestuali o context-free, ammettono produzioni della forma:

$$A \to \sigma$$

dove $\sigma \in (V_T \cup V_N)^*$ e $A \in V_N$.

Questo significa che A può sempre essere sostituito da σ, indipendentemente dal contesto. Se σ ha la forma u oppure uBv con $u, v \in V_T^*$, $B \in V_N$, la grammatica è detta lineare.

Def 3.9 (**ϵ-rules**)**.** Le ϵ-rules, o regole ϵ, sono produzioni di una grammatica in cui il lato destro della produzione è la stringa vuota ε. In altre parole, una ϵ-rule ha la forma:

$$A \to \varepsilon$$

dove A è un simbolo non terminale e ε rappresenta la stringa vuota.

Si osservi che, data una grammatica di tipo 2 con ϵ-rules, è sempre possibile trovare una grammatica equivalente che ha al più una ϵ-rule, corrispondente al simbolo iniziale S. In altre parole, una grammatica di tipo 2 con ϵ-rules può essere trasformata in una grammatica equivalente che contiene al massimo una sola ϵ-rule, e questa ϵ-rule è utilizzata esclusivamente per generare la stringa vuota (se necessario).

Esempio 3.7. La grammatica

$$G = \{V_T, V_N, P, S\} \quad \text{con} \quad V_T = \{a, b\}, \quad V_N = \{S, A, B\}$$

Con produzioni:

$$P = \{S \to aSb \mid \epsilon\}$$

genera il linguaggio $L(G) = \{a^n b^n \mid n \geq 0\}$.

Def 3.10 (**Grammatiche di tipo 1- Dipendenti dal contesto**)**.** : Le grammatiche di tipo 1, dette anche grammatiche dipendenti dal contesto, ammettono produzioni della forma:

$$\beta A \delta \to \beta \alpha \delta$$

dove $\beta, \delta \in V^*$, $\alpha \in V^+$, e $A \in VN$.
Qui $\alpha \neq \epsilon$, quindi A può essere sostituito da α solo nel contesto $\beta A \delta$. Le riscritture non accorciano mai la forma di frase corrente.

Esempio 3.8. La grammatica

$$G = \{V_T, V_N, P, S\} \quad \text{con} \quad V_T = \{a, b\}, \quad V_N = \{S, A, B\}$$

le cui produzioni sono:

$$S \to aSa \mid aAb \mid aAa$$
$$aA \to aa$$
$$Ab \to aab$$

genera il linguaggio

$$L(G) = \{a^n \mid n \geq 3, \text{ n dispari}\} \cup \{a^n b a^j \mid n \geq 2, j \geq 0\}$$

Questo linguaggio consiste di stringhe che possono essere della forma a^n con $n > 1$ e dispari, oppure $a^n b a^j$ con $n \geq 2$ e $j \geq 0$.

Def 3.11 (**Grammatiche di tipo 0**)**.** Le grammatiche di tipo 0 non impongono alcuna restrizione sulle produzioni. Possono includere riscritture che accorciano la forma di frase corrente.

Esempio 3.9. La grammatica

$$G = \{V_T, V_N, P, S\} \quad \text{con} \quad V_T = \{a, b, c\}, \quad V_N = \{S, B, C, F, G\}$$

le cui produzioni sono:

$$\begin{aligned} P = \{ \quad & S \to aSBC, \\ & CB \to BC, \\ & FC \to cG, \\ & GC \to cG, \\ & SB \to bF, \\ & G \to \epsilon, \\ & FB \to bF\} \end{aligned}$$

appartiene al tipo 0.

Con riferimento alla fig. 3 (Classificazione delle grammatiche), le grammatiche di tipo 2 e di tipo 3 possono generare la stringa vuota ε. La relazione di inclusione vale quindi solo se si accetta che nelle grammatiche dipendenti dal contesto (tipo 1) possa esistere la produzione $S \to \epsilon$, con S come simbolo iniziale che non appare a destra in nessuna produzione.
Ciò sembra portare a una apparente contraddizione, risolta dal seguente teorema:

Teorema 3.1. Se G è una grammatica acontestuale, con produzioni nella forma $A \to \alpha$ dove $\alpha \in V^*$, allora esiste una grammatica acontestuale G' che genera lo stesso linguaggio $L(G) = L(G')$ con produzioni nella forma $A \to \alpha$ dove $\alpha \neq \epsilon$, o $S \to \epsilon$ con S simbolo iniziale.

Questo teorema stabilisce che la sola differenza tra una grammatica di tipo 2 con regole ϵ e una senza è che la prima può includere la stringa vuota ϵ.
Esiste un algoritmo per l'eliminazione delle ϵ-rules: data una regola $A \to \alpha \mid \beta \mid \epsilon$, è possibile rimuovere $A \to \epsilon$ sostituendo A con $\alpha \mid \beta$ in tutti i punti in cui appare A; tale algoritmo deve essere applicato con attenzione per evitare loop e mantenere l'integrità della grammatica.

Esempio 3.10. Data la grammatica $G = \{V_T, V_N, P, S\}$ con $V_T = \{a, b, c, d\}$, $V_N = \{S, A, B, C\}$, e le produzioni:

$$P = \{S \to ABC; \quad A \to aAd \mid \epsilon; \quad B \to bBd \mid \epsilon; \quad C \to c \mid dA\}$$

scrivere se possibile una grammatica equivalente senza ϵ-rules.
Soluzione: Le produzioni nulle sono $\{A \to \epsilon, B \to \epsilon\}$ e otteniamo:

$$G' = (\{a, b, c, d\}, \{S, A, B, C\}, P', S) \quad \text{con}$$

$$P' = \{S \to ABC \mid AC \mid BC \mid C; \quad A \to aAd \mid ad; \quad B \to bBd \mid bd; \quad C \to c \mid dA\}$$

In sintesi, le grammatiche di tipo 1 non ammettono la stringa vuota nelle produzioni, mentre quelle di tipo 2 e 3 sì. Inoltre, i linguaggi generati dalle grammatiche di tipo J sono un sottoinsieme dei linguaggi generati dalle grammatiche di tipo $J-1$, escludendo la presenza della stringa vuota nel linguaggio.

3.3 Caratteristiche delle Grammatiche

Prima di descrivere le caratteristiche peculiari che distinguono le grammatiche di tipo diverso, facciamo la seguente osservazione riguardante la classificazione dei linguaggi:

Def 3.12. Un linguaggio è di un certo tipo se quel tipo è il tipo della grammatica minimale in grado di generarlo.

Ad esempio, i linguaggi contestuali sono quelli che richiedono almeno una grammatica di tipo 1 per essere generati.

Esempio 3.11. Il linguaggio $L = \{a^n b^n c^n \mid n \geq 1\}$ è (almeno) di Tipo 1 in quanto esiste una grammatica di Tipo 1 che lo genera:

$$G_1 = \{(V_T, V_N, P, S)\}$$

dove:

$$V_t = \{a, b, c\}, \quad V_n = \{S, A, B, C, D\}$$

Le produzioni sono:

$$\begin{aligned}
&S \to aBC \mid aSBC,\\
&CB \to DB,\\
&DB \to DC,\\
&DC \to BC,\\
&aB \to ab,\\
&bB \to bb,\\
&bC \to bc,\\
&cC \to cc.
\end{aligned}$$

La grammatica diventa più compatta se espressa con la definizione alternativa di grammatica di Tipo 1, che ammette lo scambio:

$$\begin{aligned}
&S \to aBC \mid aSBC,\\
&CB \to BC,\\
&aB \to ab,\\
&bB \to bb,\\
&bC \to bc,\\
&cC \to cc.
\end{aligned}$$

Il linguaggio sarebbe però generabile anche da una grammatica di Tipo 0:

$$S \to aSBC,$$
$$CB \to BC,$$
$$SB \to bF,$$
$$FB \to bF,$$
$$FC \to cG,$$
$$GC \to cG,$$
$$G \to \epsilon.$$

Osservando le regole di produzioni delle grammatiche di tipo 1, emerge una caratteristica cruciale di tali grammatiche: non garantiscono che qualunque sequenza di derivazione porti a una frase.
Questa caratteristica non si applica alle grammatiche di tipo 2 e 3. In particolare, considerando i vincoli sulle produzioni delle grammatiche:
Grammatiche di tipo 1: $\sigma A\delta \to \alpha$,
Grammatiche di tipo 2: $A \to \alpha$,
notiamo che le grammatiche di tipo 1 ammettono produzioni come $BC \to CB$, che scambiano due simboli. Tale caratteristica è impossibile da esprimere nelle grammatiche di tipo 2, in quanto è ammesso solo un metasimbolo a sinistra.
Analogamente, c'è una caratteristica distintiva delle grammatiche contestuali che le distingue dalle grammatiche regolari: nel tipo 2, i metasimboli possono essere nel mezzo della forma della frase poiché le produzioni sono del tipo $A \to \alpha$, mentre nel tipo 3 questo non è possibile.
Esplichiamo questa caratteristica con la seguente definizione e il teorema ad essa associato:

Def 3.13 (**Self-embedding**)**.** Una grammatica è self-embedding se esiste un non terminale A tale che $A \to^* \alpha_1 A \alpha_2$, con $\alpha_i \in V^+$.
Il self-embedding è la caratteristica cruciale delle grammatiche di tipo 2, che le differenzia da quelle regolari.

Teorema 3.2. Una grammatica di tipo 2 che non è self-embedding genera un linguaggio regolare. Pertanto, in assenza di self-embedding, esiste una grammatica equivalente di tipo 3 e il linguaggio generato è regolare.
È importante notare che non vale il viceversa: esistono grammatiche con self-embedding che generano comunque un linguaggio regolare, in quanto il self-embedding può essere disattivato da altre regole; in questi casi si parla di self-embedding finto.

Esempio 3.12. Self-Embedding vs finto Self-Embedding
Consideriamo la grammatica G con le seguenti produzioni:

$$S \to aSc \mid A,$$
$$A \to bAc \mid \epsilon.$$

Questa grammatica contiene un esempio di self-embedding e genera il linguaggio $L(G)$:

$$L(G) = \{a^n b^m c^{n+m} \mid n, m \geq 0\}.$$

In questo caso, le produzioni permettono di generare stringhe dove le sezioni centrali sono "incastrate" l'una nell'altra.
Consideriamo ora la grammatica:

$$S \to aSa \mid \epsilon.$$

In questo caso, anche se appare un self-embedding, esso è di fatto inutile. Con un alfabeto di un solo carattere, si possono generare solo stringhe molto semplici. Il linguaggio risultante è:

$$L(G) = \{(aa)^n \mid n \geq 0\}.$$

Poiché tutte le stringhe generate sono composte dallo stesso simbolo ripetuto, è impossibile distinguere tra una parte "destra" e una "sinistra".
Una grammatica di Tipo 3 equivalente potrebbe essere espressa con le seguenti produzioni:

$$S \to aaS \mid \epsilon.$$

In questa forma, la grammatica produce lo stesso linguaggio ma senza utilizzare il self-embedding.

Da questi esempi, possiamo dedurre il seguente teorema:

Teorema 3.3. Ogni linguaggio context-free con alfabeto unitario è un linguaggio regolare.

3.4 Grammatiche di Tipo 2

Prima di passare alla discussione del software ANTLR4, è utile approfondire le grammatiche di tipo 2 e 3, dato che ci occuperemo della loro implementazione. In questa sezione, discuteremo alcune peculiarità delle grammatiche contestuali, come l'ambiguità, gli alberi di derivazione e il Pumping lemma.
Per facilitare la discussione sulle grammatiche di tipo 2 e 3, adotteremo una notazione diversa rispetto a quella utilizzata finora nel capitolo.
Useremo la Backus-Naur Form (BNF), dove le regole di produzione hanno la forma $\alpha ::= \delta$, con $\delta \in V^*$ e $\alpha \in V_N$.
I metasimboli X in V_N avranno la forma $< nome >$, e il simbolo "|" indicherà l'alternativa.
Questa estensione ci permette di esprimere in modo compatto un insieme di regole con la stessa parte sinistra: se $X ::= A_1 \dots X ::= A_N$, possiamo scrivere $X ::= A_1|A_2| \dots |A_N$. Menzioniamo anche l'Extended BNF (EBNF), una forma estesa della BNF, che introduce alcune notazioni compatte per semplificare la scrittura delle regole di produzione.

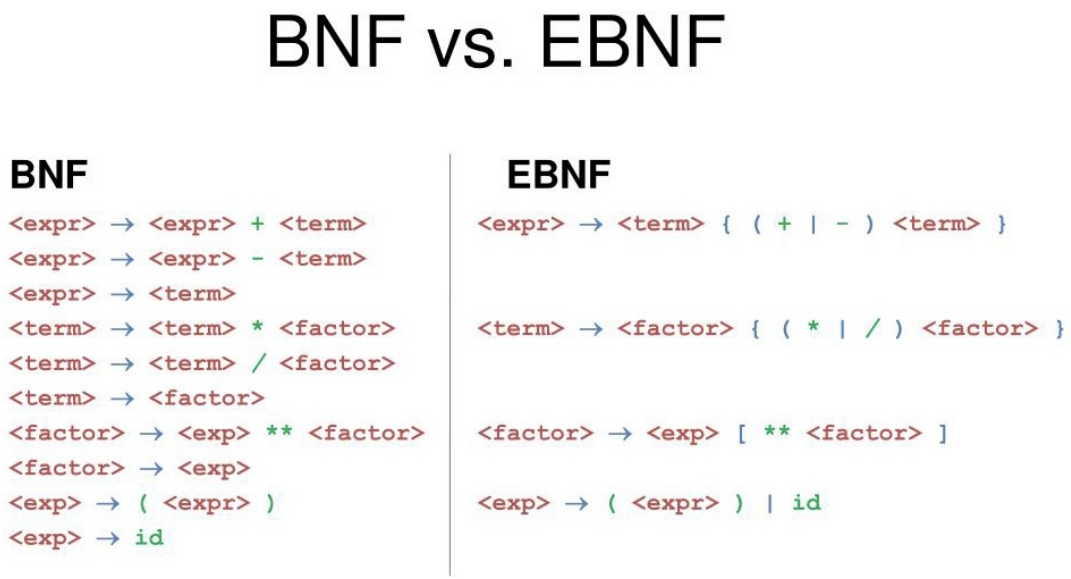


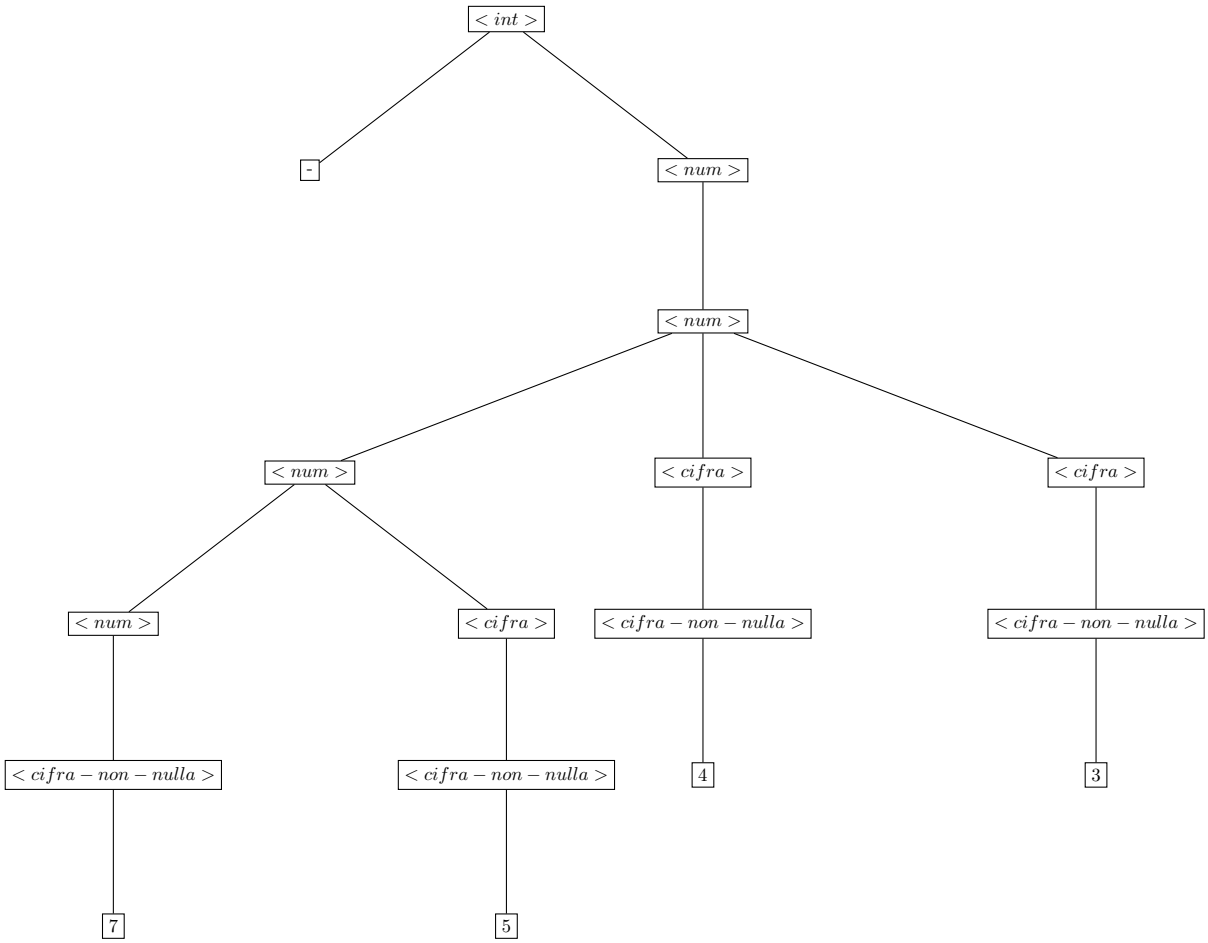
Figura 4: Differenza tra BNF e EBNF nell'aritmetica

Esempio 3.13. Consideriamo la grammatica $G = \langle VT, VN, P, S\rangle$ per i numeri interi, dove:

$$VT = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\},$$
$$VN = \{\langle \text{int}\rangle, \langle \text{num}\rangle, \langle \text{cifra}\rangle, \langle \text{cifra-non-nulla}\rangle\},$$

$$P = \begin{cases} \langle \text{int}\rangle \to +\langle \text{num}\rangle \mid -\langle \text{num}\rangle, \\ \langle \text{num}\rangle \to \langle \text{cifra}\rangle \mid \langle \text{cifra-non-nulla}\rangle\{<\text{cifra}>\}, \\ \langle \text{cifra}\rangle \to 0 \mid \langle \text{cifra-non-nulla}\rangle, \\ \langle \text{cifra-non-nulla}\rangle \to 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{cases}$$

L'albero di derivazione per la stringa -3457 è il seguente:



Possiamo notare che:
1. Ogni nodo dell'albero è associato a un simbolo del vocabolario V.
2. La radice dell'albero coincide con lo scopo S.
3. Se $a_1, a_2, \ldots, a_k$ sono i k figli ordinati di un dato nodo X associato al simbolo X in V_N, allora la grammatica contiene la produzione $X ::= A_1 A_2 \ldots A_k$, dove A_i è il simbolo associato al nodo a_i.

L'albero di derivazione non può esistere per grammatiche di tipo 1 e 0, poiché il lato sinistro delle produzioni ha più di un simbolo e quindi i nodi figli avrebbero più di un genitore: non avremmo più un albero, ma un grafo.
Affrontiamo ora la questione dell'ambiguità; consideriamo un esempio generico di grammatica di tipo 2.

Esempio 3.14. Date le produzioni

$$S \to aSb \mid T \mid \epsilon$$

$$T \to ab$$

La stringa "aabb" ammette più derivazioni:

1. Derivazione 1:
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$
2. Derivazione 2:
$$S \Rightarrow aSb \Rightarrow aTb \Rightarrow aabb$$

Come mostrato in questo esempio, esistono casi in cui una parola può avere derivazioni distinte. Questo fenomeno è noto come ambiguità.

Def 3.14 (**Ambiguità**)**.** Una grammatica è ambigua se esiste almeno una parola che ha due o più derivazioni canoniche distinte (sinistre) - Left Most Derivation -. Il grado di ambiguità è il numero di alberi sintattici distinti che possono derivare la stessa parola.
È importante notare che l'ambiguità può essere indesiderabile e che stabilire se una grammatica di tipo 2 sia ambigua è un problema indecidibile; tuttavia, spesso è possibile trovare una grammatica contestuale ambigua e trasformarla in una che non lo è, ma non sempre.

Esempio 3.15. Per rendere la grammatica dell'esempio 3.14 non ambigua, possiamo riscriverla come segue:

$$S \to aSb \mid ab \mid \epsilon$$

Esistono anche linguaggi L per cui tutte le grammatiche che li generano sono intrinsecamente ambigue.
Concludiamo il paragrafo e il capitolo con alcune osservazioni sulla stringa vuota e un lemma importante: il pumping lemma. Ricordiamo, come già detto, che la stringa vuota può far parte delle parole generate da una grammatica di tipo 0, poiché la regola generica di produzione $\alpha \to \beta$ prevede $\alpha \in V^+$ e $\beta \in V^*$.
La stringa vuota, tuttavia, non può far parte delle parole generate da una grammatica di tipo 1, 2 o 3, poiché vale la condizione $\alpha \neq \epsilon$, e quindi le forme frasali non possono accorciarsi; questo non è in contraddizione col fatto che le grammatiche contestuali e regolari possano apparentemente ammettere ϵ sul lato destro delle produzioni, in quanto esiste sempre una grammatica equivalente senza ϵ-rules, escludendo al più la produzione top-level $S \to \epsilon$. Questa proprietà è espressa nel seguente teorema:

Teorema 3.4. Dato un linguaggio L di tipo 0, 1, 2 o 3, i linguaggi $L \cup \{\epsilon\}$ e $L - \{\epsilon\}$ sono dello stesso tipo.

Concentriamo ora la nostra attenzione sul Pumping Lemma: il Pumping Lemma è un importante strumento per stabilire proprietà di non regolarità per i linguaggi generati da grammatiche di tipo 2 e 3.

Lemma 3.1 (Pumping Lemma per linguaggi Regolari)**.** Se L è un linguaggio di Tipo 3 allora esiste un intero N tale che, per ogni stringa z di lunghezza almeno N:

- z può essere scomposta come $z = xyw$
- La parte centrale xy ha lunghezza limitata: $|xy| \leq N$
- $|y| \geq 1$
- $xy^i w \in L \ \forall i \geq 0$

Lemma 3.2 (Pumping Lemma per linguaggi di Tipo 2)**.** Se L è un linguaggio di Tipo 2 allora esiste un intero N tale che, per ogni stringa z di lunghezza almeno N:

- z è decomponibile in 5 parti: $z = uvwxy$
- La parte centrale vwx ha lunghezza limitata: $|vwx| \leq N$
- $|vy| \geq 1$
- $uv^i wx^i y \in L \ \forall i \geq 0$

Questi lemmi forniscono una condizione necessaria ma non sufficiente per affermare che un linguaggio L sia di tipo 2 o 3: sono utilizzati dunque in contronominale per affermare che un linguaggio L non è regolare o di tipo 2.

Esempio 3.16. Per mostrare che il linguaggio $L = \{a^n b^n c^n \mid n \geq 1\}$ non è di tipo 2, verifichiamo che esso non soddisfa il pumping lemma. Mostriamo quindi che per ogni valore di n esiste una stringa z, con $|z| \geq n$, che non è decomponibile nelle 5 stringhe u, v, w, x, y aventi le proprietà richieste dal lemma.

Sia $z = a^n b^n c^n = uvwxy$. Allora:

- Se v e x contengono almeno due simboli diversi, la stringa uv^2wx^2y conterrà simboli mescolati.
- Se invece v e x contengono un solo simbolo, la stringa uv^2wx^2y non avrà più la proprietà che le stringhe del linguaggio contengono un numero uguale di a, di b e di c.

In entrambi i casi, $uv^2wx^2y \notin L$, il che contraddice il pumping lemma. Quindi, L non è un linguaggio di tipo 2.

Con questo abbiamo concluso l'introduzione sui linguaggi e sulle grammatiche.

4 ANTLR4

ANTLR4 (ANother Tool for Language Recognition) è un generatore di parser che consente di leggere, processare, eseguire e tradurre testi strutturati. A partire da una grammatica specificata, ANTLR4 genera un parser per il linguaggio definito da quella grammatica; questo parser non solo costruisce automaticamente gli alberi di parsing - parse tree - ma genera anche i tree walker. I tree walker sono strumenti che attraversano i nodi di questi alberi per eseguire operazioni specifiche di elaborazione del codice. In questo capitolo, dopo aver presentato il software e descritto le sue funzionalità, ci occuperemo dello sviluppo di due progetti: una teoria per le operazioni aritmetiche elementari e un esempio che descrivi la teoria degli insiemi.

4.1 Installazione

ANTLR4 è scritto in Java, quindi è necessario installare Java; in particolare ANTLR4 richiede almeno la versione 1.6 di Java.
L'installazione di ANTLR4 è semplice. Si può scaricare l'ultimo file JAR dal sito ufficiale di ANTLR4 oppure utilizzare il comando `curl` dal terminale per scaricare direttamente il file nella cartella desiderata:

```
$ cd /directoryLib
$ curl -O http://www.antlr.org/download/antlr-4.13.1-complete.jar
```

Per comodità, consigliamo di utilizzare la linea di comando per eseguire e verificare i risultati delle operazioni. Pertanto, è utile configurare la variabile di ambiente CLASSPATH come segue:

```
$ export CLASSPATH=".:/directoryLib/antlr-4.13.1-complete.jar:$CLASSPATH"
```

Chiaramente, per directoryLib si intende la directory nella quale si vuole installare ANLTR4.
In questo modo, si utilizza ANTLR4 direttamente dal terminale, facilitandone l'esecuzione e la gestione delle operazioni necessarie per i progetti; è utile anche utilizzare degli alias per lanciare ANTLR4 e gli strumenti di test, come per esempio TestRig, presente nella libreria runtime:

```
$ alias antlr4= ‘java -jar /directoryLib/antlr-4.13.1-complete.jar’
$ alias grun=’java org.antlr.v4.gui.TestRig’
```

Naturalmente, rimane al lettore la facoltà di seguire questi suggerimenti.
Menzioniamo l'utilità della libreria TestRig, che permette di eseguire vari tipi di test sulle grammatiche create, rendendo il processo di sviluppo più efficiente.

4.2 ANTLR4: Esempi d'uso

Prima di procedere con la descrizione dettagliata di ANTLR4, è doveroso effettuare alcune osservazioni.

In primis, i file contenenti le grammatiche sono specificati nel formato ".g4"; questi file sono composti dal nome della grammatica, da sezioni dedicate - opzioni, import, token, canali, azioni - e da un insieme di regole.
In particolare, ANTLR4 permette di specificare il linguaggio tramite una grammatica context-free e utilizzando notazioni EBNF.
Il generatore di parser produce parser adattivi di tipo LL(k), ossia un tipo di parser che legge l'input da sinistra a destra (Left-to-right) e costruisce la derivazione più a sinistra (Left-most derivation) utilizzando k simboli di lookahead[1] per decidere quale regola di produzione applicare.
Questo significa che il parser analizza la grammatica e cerca di ottimizzare le prestazioni, adattandosi alla struttura del linguaggio: per essere più precisi, ANTLR4 supporta l'uso di regole ricorsive per la definizione del parser, semplificandone la scrittura grazie a espressioni che si avvicinano di più al linguaggio naturale. Un esempio di grammatica potrebbe essere il seguente:

```
grammar Esempio;

options {
    language = Java;
}

r : 'hello' ID
     | 'goodbye' ID
     ;

ID : [a-zA-Z]+ ;
WS : [ \t\r\n]+ -> skip;
```

Questa grammatica definisce il parser denominato `Esempio` e descrive un linguaggio che riconosce comandi di saluto composti dalle parole chiave "hello" o "goodbye" seguite da un identificatore, permettendo di distinguere tra i due diversi tipi di saluti. L'opzione indicata specifica che il linguaggio di destinazione è Java.
Le regole di parsing sono definite come segue:

- La regola `r` può corrispondere a due alternative:
  - La stringa "hello" seguita da un identificatore (`ID`).
  - La stringa "goodbye" seguita da un identificatore (`ID`).
- Il token `ID` è definito come una sequenza di almeno uno o più caratteri alfabetici (minuscoli o maiuscoli).

[1] Il lookahead rappresenta la capacità del parser di esaminare anticipatamente il flusso di simboli in input per migliorare l'efficienza e l'accuratezza nell'analisi sintattica del linguaggio; consente al parser di predire quale regola di produzione applicare sulla base di un numero definito di simboli successivi nel flusso di input, senza dover retrocedere nella lettura dell'input stesso.

- Il token `WS` gestisce lo "spazio bianco" - whitespace - composto da spazi, tabulazioni e caratteri di fine linea che sono ignorati durante l'analisi.

Effettuata l'analisi della grammatica presente nel file in formato ".g4", ANTLR4 genera le classi nel linguaggio target - Java nel nostro caso - e i seguenti file:

- `Esempio.interp`: Contiene le informazioni interpretative necessarie per il parser generato da ANTLR, come le tabelle di analisi per l'interpretazione dei token e delle regole grammaticali.
- `Esempio.tokens`: Contiene l'elenco dei token riconosciuti dalla grammatica definita in `Esempio.g4`.
  Ogni token è associato a un identificatore univoco, utilizzato internamente dal parser.
- `EsempioLexer.java`: Contiene il codice sorgente per il lexer generato da ANTLR; gestisce la suddivisione del testo in input in token riconoscibili dal parser.
- `EsempioLexer.tokens`: Contiene l'elenco dei token riconosciuti dal lexer generato da ANTLR, con i rispettivi identificatori univoci.
- `EsempioLexer.interp`: Contiene le informazioni interpretative per il lexer generato, come tabelle di analisi per ottimizzare l'analisi lessicale.
- `EsempioListener.java`: Contiene l'interfaccia Java generata per gestire eventi durante l'attraversamento dell'albero di parsing. Include metodi chiamati dal parser durante l'analisi.
- `EsempioParser.java`: Contiene l'implementazione del parser generato da ANTLR, responsabile dell'analisi sintattica secondo le regole definite nella grammatica.
- `EsempioBaseListener.java`: Classe astratta che fornisce implementazioni vuote dei metodi dell'interfaccia `EsempioListener.java`.
  Può essere estesa per gestire specificamente gli eventi durante l'analisi.

Lavoriamo nel terminale con la libreria TestRig - alias grun - :

```
$ antlr4 Esempio.g4
$ javac *.java // compilo tutti i file con estensione .java
$ grun Esempio r -tokens
```

A questo punto è necessario fornire un input, seguito da `Ctrl-d` su Unix o `Ctrl+Z` su Windows per indicare l'EOF (End of File) e terminare l'inserimento. Con l'opzione `-tokens`, si richiede al parser di stampare la lista di token generati dall'input:

```
$hello carlo
$goodbye carlo
$EoF
```

ottenendo:

```
[@0,0:4='hello',<'hello'>,1:0]
[@1,6:10='carlo',<ID>,1:6]
[@2,12:18='goodbye',<'goodbye'>,2:0]
[@3,20:24='carlo',<ID>,2:8]
[@4,26:25='<EOF>',<EOF>,3:0]
```

Si può anche stampare il parse-tree in formato LISP con l'opzione -tree:

```
$grun Esempio r -tree
$ hello Parrot
$EoF
```

ottenendo come output:

```
(r hello Parrot)
```

o anche visualizzare il parse-tree con l'opzione -gui:

```
$grun Esempio r -gui
$ hello Parrot
$EoF
```

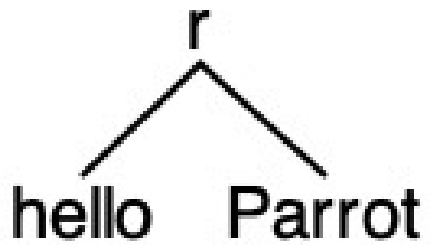


Figura 5: Parse-tree

Di seguito elenchiamo alcune opzioni - le più utilizzate - di `grun`:

- **tokens**: Visualizza i token riconosciuti durante l'analisi del testo di input.
- **tree**: Visualizza l'albero di parsing generato dall'analisi del testo di input in formato LISP.

- **gui**: Mostra un'interfaccia grafica per visualizzare l'albero di parsing.
- **ps file.ps**: Salva l'albero di parsing come file PostScript con il nome specificato.
- **encoding encodingname**: Specifica l'encoding dei file di input.
- **trace**: Mostra i dettagli di debug durante l'analisi.
- **SLL**: Forza l'utilizzo del parsing LL per migliorare le prestazioni, ma con una strategia di parsing più debole.
- **diagnostics**: Mostra informazioni diagnostiche durante l'analisi.

Notiamo che in questo semplice esempio stiamo considerando una grammatica combinata: sono presenti sia le regole per il parser, che sono scritte in minuscolo, come r, che per il lexer, scritte invece in maiuscolo, come ID e WS.
ANTLR4 mette a disposizione anche la possibilità di importare grammatiche esistenti, sia per il lexer che per il parser, facilitando il riutilizzo e la modularizzazione delle grammatiche: questo è particolarmente utile in progetti complessi dove è necessario mantenere una chiara separazione delle responsabilità, permettendo di costruire componenti riutilizzabili e facilmente gestibili.
Quando si importa una grammatica si ereditano le regole, i token e le azioni, ma la sezione delle opzioni viene ignorata: le regole della nuova grammatica eseguono l'override di quella importata, implementando il concetto di ereditarietà similmente al modello object-oriented; l'import non è una semplice inclusione del codice, ma un merge di regole, token e azioni fra la grammatica importata e quella corrente.
Vi sono delle limitazioni sull'import riguardanti le grammatiche specifiche, in cui sono scritte solo le regole per il parser o per il lexer : tali grammatiche possono importare solo altre grammatiche dello stesso tipo, mentre le grammatiche combinate non hanno questo tipo di limitazione.
Elenchiamo le keyword di ANTLR4:

- **import**: Viene utilizzata per importare altre grammatiche o definizioni all'interno di una grammatica principale.
- **fragment**: Definisce regole di token che non vengono esposte al lexer principale e possono essere utilizzate solo all'interno di altre regole di token.
- **lexer**: Indica che la grammatica contiene regole per il lexer.
- **parser**: Indica che la grammatica contiene regole per il parser.
- **grammar**: Definisce l'inizio di una grammatica in ANTLR4; ricordiamo che il nome di una grammatica deve essere lo stesso del file in cui viene scritta.
- **returns**: Specifica il tipo di ritorno di una regola di parser.

- **locals**: Indica la dichiarazione di variabili locali all'interno di una regola di parser.
- **throws**: Specifica le eccezioni che possono essere lanciate da una regola di parser o lexer.
- **catch**, **finally**: Sono usate nelle sezioni di gestione delle eccezioni per catturare e gestire le eccezioni.
- **mode**: Viene utilizzata per definire modalità alternative per il lexer; consente la definizione di diverse regole di token per diversi contesti.
- **options**: Definisce opzioni specifiche per la grammatica, come l'encoding dei file di input o altre configurazioni.
- **tokens**: Definisce token aggiuntivi o personalizzati che possono essere utilizzati nella grammatica.

Esistono anche comandi specifici per il lexer in ANTLR4, da inserire dopo l'ultima alternativa con la sintassi `->`:

- **skip**: Impedisce al lexer di trasmettere il token corrente al parser, ignorandolo completamente.
- **more**: Forza il lexer a considerare il token successivo immediatamente dopo aver ricavato l'attuale. Il tipo del token sarà quello dell'ultimo token considerato.
- **mode(x)**: Modifica lo stack mode del lexer, cambiando il mode corrente a `x`. È possibile navigare nello stack mode utilizzando i comandi `popMode` e `pushMode(x)`.
- **type(x)**: Assegna il tipo associato al token corrente a `x`.
- **channel(x)**: Assegna il numero di canale del token corrente a `x`.

4.3 Proprietà di ANTLR4

Una delle caratteristiche fondamentali di ANTLR4 è la capacità di generare automaticamente sia parser che tree walker. Questo permette di visitare i nodi degli alberi di parsing per eseguire operazioni specifiche, una funzione che si realizza attraverso due principali modalità: i *listener* e i *visitor*.
I *listener* generano automaticamente una classe che implementa un'interfaccia con metodi "enter" e "exit" per ogni regola della grammatica: questo approccio è utile per eseguire azioni specifiche quando il parser entra o esce da una regola.
I *visitor*, d'altra parte, offrono un controllo manuale dell'attraversamento dell'albero di parsing, applicando la logica specifica a ogni nodo tramite un metodo "visit" per ciascuna regola.
Dopo che il lexer ha prodotto la sequenza di token derivata dalla stringa di input, il parser procede alla creazione del *parse tree*: le foglie rappresentano

direttamente i token estratti dall'input, mentre i nodi interni sono etichettati per raggruppare e identificare logicamente i figli corrispondenti.
Se si applica una label (`#label`) a ciascuna alternativa di una regola che presenta punti di scelta, il *parse tree* includerà un nodo distintivo per ogni scelta effettuata, contrassegnato dalla label corrispondente.
Un'altra caratteristica interessante di ANTLR4 sono le azioni incorporate, note come *embedded action*. Identificate dal simbolo @, queste azioni permettono di inserire codice direttamente all'interno delle regole della grammatica: ciò è utile per modificare lo stato interno del parser, costruire strutture dati o eseguire altre operazioni necessarie durante il parsing. Ad esempio, è possibile incrementare un contatore ogni volta che si entra in una regola specifica, oppure costruire una lista di simboli man mano che vengono riconosciuti.
In particolare, `@header{}` viene utilizzata per iniettare codice prima della definizione della classe, ad esempio per specificare il package, mentre `@members{}` permette di inserire codice tra le definizioni, i campi e i metodi della classe generata (ad esempio per la dichiarazione di variabili). Infine, `@after{}` consente di eseguire azioni dopo che il parser ha completato il suo compito.
Ecco un esempio di grammatica con azioni incorporate:

```
grammar Simboli;

@members {
    private int count = 0;
    private List<String> simboli = new ArrayList<>();
}

rule: 'start' ID {
    System.out.println("inizio regola");
    count++;
    simboli.add($ID.text);
}
    'end' {
    System.out.println("fine regola");
}
;

ID : [a-zA-Z]+ ;
WS : [ \t\r\n]+ -> skip ;

@after {
    System.out.println("Simboli inseriti: " + count);
    System.out.println("Simboli: " + simboli);
}
```

Terminiamo questa sezione parlando di predicati semantici, che rappresentano un potente strumento fornito da ANTLR4.
Essi permettono di aggiungere condizioni logiche alle regole della grammatica che

devono essere soddisfatte affinché la regola sia applicata. I predicati semantici sono particolarmente utili per risolvere ambiguità e per applicare logica complessa nel processo di parsing, garantendo che solo le produzioni sintatticamente valide vengano considerate:

```
grammar PredSem;
@members {
    private boolean IdValido(String id) {
//condizione su id: ha almeno 3 caratteri di cui il primo maiuscolo
    return id.length() >= 3 && Character.isUpperCase(id.charAt(0));
         }
    }

r: {IdValido($ID.text)}? ID 'end'
    {System.out.println("Identificatore valido: " + $ID.text);}
    | ID 'end'
    {System.out.println("Identificatore non valido: " + $ID.text);}
    ;

ID : [a-zA-Z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

In questo esempio, la grammatica utilizza un predicato semantico per verificare la validità di un identificatore. La funzione `IdValido` definisce un criterio di validazione: un identificatore è valido se la sua lunghezza è almeno di 3 caratteri e la prima lettera è maiuscola. La regola `r` contiene un predicato semantico `{IdValido($ID.text)}?` che verifica se l'identificatore soddisfa la condizione specificata. Se la condizione è soddisfatta, viene applicata la prima alternativa e viene visualizzato un messaggio che indica che l'identificatore è valido. Altrimenti, si applica la seconda alternativa, stampando un messaggio che indica che l'identificatore non è valido.
Inoltre, nella regola `r`, ci si aspetta che l'identificatore sia seguito dalla stringa `'end'`. Pertanto, l'input al parser deve essere una sequenza di caratteri che corrisponde a un identificatore seguita dalla parola `'end'`. Ad esempio, l'input `'ABCend'` verrà riconosciuto come un identificatore valido, mentre `'ABend'` verrà riconosciuto come un identificatore non valido.

4.4 Progetti

In questa sezione presentiamo i progetti realizzati, commentando le grammatiche utilizzate e le loro implementazioni in Java; i progetti sono disponibili su Github.[2]
Iniziamo con le grammatiche per il calcolo aritmetico: una è implementata con *visitors* e senza *embedded action* mentre l'altra è implementata senza *visitor* e con *embedded action*. Successivamente, trattiamo la grammatica relativa alla teoria degli insiemi, implementata mediante *visitor*.
La struttura dei progetti è uniforme per ciascuno di essi:

[2] `https://github.com/CarloRossanigo`

```
Folder/
|
+-- lib/
|   \-- antlr-4.13.1-complete.jar
|
\-- src/
    +-- ANTLR4/
    |   +-- file.g4
    |   \-- (file generati)
    |
    +-- App/
    |   \-- EspressioneApp.java
    |
    +-- Espr/
    |   \-- (classi Java per la modellizzazione dei progetti)
    |
    \-- Test/
        \-- (file di testo per il testing)
```

Su Github si trovano i file JAR per ogni progetto e la cartella `src`. Si consiglia di seguire la struttura del progetto, in particolare, di inserire una cartella dedicata alla libreria di `ANTLR4` se si intende lavorare su IDE come Eclipse o Maven. In tal caso, è bene assicurarsi che nelle proprietà del progetto la libreria di `ANTLR4` sia inserita nel `Modulepath`, nell'`Order and Export` e che nella configurazione per la compilazione sia scelta come `Main class` la classe `App.EspressioneApp`, che accetti come argomento file di testo (ad esempio, `"file-prompt"`) e che le `Dependencies` della libreria di `ANTLR4` siano inserite sia nel `Modulepath Entries` che nel `ClassPath Entries`.
Non ripeteremo le regole e le implementazioni comuni a tutti i progetti dopo averle esposte nel primo, né descriveremo le regole la cui funzione è intuitiva, salvo nel primo progetto; ciò vale anche per la classe `EspressioneApp`, in cui è presente il `main`, in quanto è la stessa per ogni progetto, con la differenza che i lexer e i parser assumono il nome della rispettiva grammatica: verrà presentata solo per il primo progetto.

4.4.1 Grammatica Calc

La grammatica `Calc` è progettata per eseguire operazioni aritmetiche di base, gestire dichiarazioni di variabili, eseguire funzioni matematiche e includere espressioni condizionali. Di seguito, descriviamo in maniera più o meno discorsiva le regole del parser e le regole lessicali utilizzate nella grammatica.
La regola principale della grammatica è `prog`, che definisce la struttura di un programma:un programma è costituito da una serie di dichiarazioni, espressioni o istruzioni condizionali, seguite dalla fine del file (EOF).

Le dichiarazioni di variabili sono gestite dalla regola `decl`. Ogni dichiarazione deve contenere un identificatore (`ID`), seguito da un tipo (`TYPE`) e da un valore numerico (`NUM`).
Le espressioni aritmetiche e logiche sono definite dalla regola `expr`. Le espressioni possono essere racchiuse tra parentesi per gestire la precedenza delle operazioni; possono includere operazioni di moltiplicazione, divisione e potenza, oltre a somme e sottrazioni. Inoltre, le espressioni possono valutare confronti logici come maggiore, minore, uguale e diverso. Gli operandi delle espressioni possono essere variabili, numeri , costanti matematiche come `PI` o `EULER`, o funzioni.
Le regole lessicali,che definiscono i token che il lexer riconosce nel codice sorgente, sono le seguenti:

- `ID`: Rappresenta un identificatore che inizia con una lettera minuscola e può contenere lettere maiuscole, minuscole, numeri e underscore.
- `NUM`: Definisce un numero che può essere un intero o un numero decimale, eventualmente preceduto da un segno negativo.
- `TYPE`: Rappresenta i tipi di dati supportati, come `INT`, `DOUBLE` e `BOOLEAN`.
- `LPAREN` e `RPAREN`: Rappresentano rispettivamente le parentesi sinistra e destra.
- `MUL`, `DIV`, `ADD`, `SUB`, `POW`: Rappresentano gli operatori aritmetici di moltiplicazione, divisione, addizione, sottrazione e potenza.
- `LESS`, `LESSER`, `GREAT`, `GREATER`, `EQUALS`, `NEQUALS`: Rappresentano gli operatori di confronto.
- `COMMENT`: Definisce i commenti, che iniziano con `--` e continuano fino alla fine della riga.
- `WS`: Rappresenta gli spazi bianchi, come spazi, tabulazioni e nuove righe, che vengono ignorati dal lexer.

Per quanto riguarda l'implementazione in Java, il progetto è strutturato attorno a diverse classi fondamentali, in particolare `Espression`, `Environment`, `Value`, `AntlrToExpression`, `AntlrToProgram`, `EspressioneApp`, che collaborano per gestire l'interpretazione e la valutazione di espressioni matematiche complesse.
La classe astratta `Expression` rappresenta il fondamento del progetto, fornendo la struttura per tutte le espressioni. Definisce il metodo astratto `evaluate`, che ogni espressione concreta deve implementare e che permette di valutare l'espressione in un contesto specifico, fornito dalla classe `Environment`, e restituisce un oggetto di tipo `Value`; la classe `Environment` svolge il ruolo di ambiente di valutazione, mantenendo una corrispondenza tra nomi di variabili e i rispettivi valori numerici.
Le classi `VariableDeclaration` e `Var` si occupano rispettivamente della dichiarazione di variabili e della rappresentazione di variabili all'interno delle espressioni. `VariableDeclaration` consente l'aggiunta di nuove variabili all'ambiente di

valutazione e restituisce il valore di una variabile durante la valutazione, mentre `Var` permette di ottenere il valore corrente di una variabile dall'ambiente.
La classe `BooleanExpression` estende `Expression` per rappresentare espressioni booleane; essa implementa il metodo `evaluate` che valuta le operazioni booleane tra le sottoespressioni. Analogamente, la classe `NumericExpression` estende `Expression` per rappresentare espressioni numeriche, implementando il metodo `evaluate` per gestire le operazioni aritmetiche, applicare funzioni matematiche sfruttando `Functions` - un tipo enumerativo che definisce funzioni matematiche comuni come `COS`, `SIN` e `LOG` - e per trattare le costanti come `PI` ed `EULER`.
La classe `Program` rappresenta invece un programma composto da una lista di espressioni, offrendo metodi per aggiungere espressioni e ottenere l'elenco completo.
`AntlrToExpression` funge da visitor per convertire le regole definite dalla grammatica ANTLR in oggetti espressione utilizzabili nel progetto Java. Questa classe gestisce anche la validazione semantica durante la conversione, garantendo che le espressioni siano correttamente interpretate e valutate secondo le specifiche definite, mentre `AntlrToProgram` funge da visitor convertendo un programma definito nella grammatica in un oggetto di tipo `Program`, utilizzando `AntlrToExpression` per convertire singole espressioni.
`ExpressionProcessor` è responsabile della valutazione delle espressioni all'interno di un `Program`. Essa utilizza l'ambiente `Environment` per tenere traccia delle variabili e valuta ciascuna espressione, restituendo i risultati o gli eventuali errori riscontrati.
Infine, per quanto riguarda `EspressioneApp`, dopo aver verificato che venga passato il nome del file da elaborare, il programma utilizza il parser `CalcParser` per costruire l'albero di parsing a partire dalla regola `prog`; dopo il controllo per eventuali errori di sintassi - che in tal caso vengono segnalati mediante la chiamata alla classe `MyErrorListener`, la quale mostra il messaggio di errore in un dialogo grafico - un visitor `AntlrToProgram` converte l'albero di parsing in espressioni; se non vi sono errori semantici, l'oggetto `ExpressionProcessor` valuta le espressioni e stampa i risultati - altrimenti vengono stampati gli errori semantici - .
Il metodo `getParser` gestisce l'input del file, inizializzando il lexer e il parser e aggiungendo un listener per gli errori.
In sintesi, l'implementazione Java del progetto sfrutta una combinazione di classi astratte e concrete per costruire un interprete di espressioni matematiche complesso, con un forte supporto per la gestione delle variabili, delle espressioni booleane e delle funzioni matematiche.

4.4.2 Grammatica Espr

Come la grammatica `Calc`, la grammatica `Espr` è progettata per la gestione delle espressioni aritmetiche e dichiarazioni di variabili, ma con l'aggiunta di azioni corporate, di un meccanismo di gestione degli errori semantici e la costruzione di un albero sintattico astratto (AST).
Anche in questo caso, la regola principale per il parser della grammatica è `prog`,

che inizializza un oggetto di tipo `Program` il quale funge da radice per l'AST e aggiunge dichiarazioni o espressioni a questo programma.
Le dichiarazioni di variabili sono gestite dalla regola `decl`, come nella grammatica `Calc`.
Durante il processo di dichiarazione di una variabile, la grammatica verifica se l'identificatore è già stato dichiarato e, in tal caso, aggiunge un messaggio di errore alla lista `semErrors`; se l'identificatore è valido, viene aggiunto alla lista delle variabili e viene creata una nuova istanza di `VariableDeclaration` per rappresentare la dichiarazione nell'AST.
Le espressioni aritmetiche sono definite dalla regola `expr`. Se viene utilizzata una variabile non dichiarata, viene aggiunto un messaggio di errore alla lista `semErrors`.
Le regole lessicali sono le stesse della grammatica `Calc`, cosi come l'implementazione in Java, con la differenza che non sono necessarie le classi `AntlrToExpression` e `AntlrToProgram` in quanto le operazioni presenti in tali classi vengono gestite nella grammatica.

4.4.3 Grammatica Teoria

La grammatica `Teoria` è strutturata per esaminare dichiarazioni e operazioni relative a una pseudo-teoria degli insiemi.
Per quanto riguarda le regole del parser, come per le grammatiche precedenti, il punto di ingresso della grammatica è rappresentato dalla regola `prog`.
Le dichiarazioni di insiemi sono gestite dalla regola `decl`: in maniera simile a `Calc`, ogni dichiarazione è costituita da un identificatore (`ID`) seguito da una lista di elementi racchiusi tra parentesi graffe. Gli elementi in questo caso possono essere numeri (`NUM`) o stringhe (`STRING`), separati da virgole; è quindi permesso dichiarare insiemi eterogenei contenenti sia numeri che stringhe.
Le espressioni nella grammatica `Teoria` sono definite dalla regola `expr` e possono includere varie operazioni sugli insiemi, tra cui quelle binarie di unione, intersezione, differenza, differenza simmetrica e quella unaria di complemento, eseguita considerando come insieme universo tutti gli insiemi definiti in quel momento.
Fra le operazioni unarie, menzioniamo anche l'operazione di massimo `MAX` e minimo `MIN` di un insieme. Per convenzione, abbiamo assunto che queste operazioni restituissero rispettivamente il numero più grande e più piccolo contenuto in un insieme; nel caso di una stringa, per numero di una stringa si intende la sua lunghezza.
In particolare, è doveroso osservare che queste operazioni sono state definite come regole lessicali prima della regola che definisce il formato degli identificatori per gli insiemi: ciò è di fatto necessario, in quanto altrimenti il lexer e il parser considererebbero `MAX` e `MIN` come nomi di insiemi anziché come operazioni.
Nella regola `expr` sono definite, oltre alle parentesi anche le operazioni relazionali tra insiemi, come l'uguaglianza e il contenimento.
A livello di regole lessicali, menzioniamo anche che le stringhe sono necessariamente scritte in lettere minuscole, in quanto le lettere maiuscole sono utilizzate

per identificare gli insiemi.
Per quanto riguarda l'implementazione in Java, essa è simile a quella del progetto `Calc` ma presenta alcune differenze significative: le classi `Var` e `VariableDeclaration` sono sostituite rispettivamente dalle classi `Set` e `DeclarationSet`, con una modifica anche nel campo di quest'ultima: non più un campo privato *double value*, ma una $List < Object > value$, considerando una lista invece di una singola variabile numerica, concettualmente pensata come una lista di numeri e parole, implementata genericamente come un oggetto *Object*. Inoltre, non sono più presenti costanti e funzioni. Sebbene le operazioni aritmetiche siano sostituite da operazioni insiemistiche, è necessario prestare attenzione all'implementazione di `MAX` e `MIN`: a tale scopo, dato un insieme, abbiamo implementato una funzione ausiliaria `GetSize` per recuperare la lunghezza dei suoi elementi e confrontarli, invocata dalle funzioni `MAX` e `MIN` secondo la convenzione adottata.
Questa osservazione conclude il paragrafo, il capitolo e l'elaborato.

Riferimenti bibliografici

[1] Giorgio Ausiello, Fabrizio d'Amore, Giorgio Gambosi, Luigi Laura, et al. *Linguaggi modelli complessità*. F. Angeli, 2003.

[2] Terence Parr. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2013.