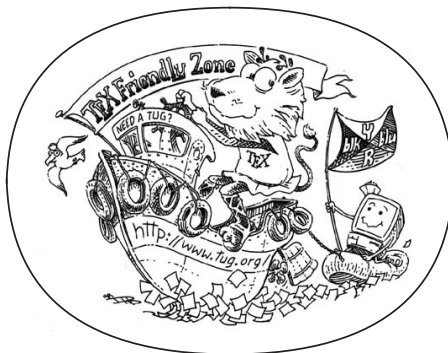


# LINGUAGGI E PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

RICCARDO CEREGHINO



Appunti

Settembre 2019 – classicthesis v4.6



## INTRODUZIONE AGLI ELEMENTI DI UN LINGUAGGIO DI PROGRAMMAZIONE

---

I motivi della creazione ed utilizzo di un linguaggio di programmazione di alto livello sono: di fornire una descrizione precisa, ovvero una specifica formale; offrire un'interpretazione tramite interprete da compilare.

Le caratteristiche principali che categorizzano i linguaggi di programmazione sono la sintassi e la semantica, la quale può essere statica o dinamica.

### 1.1 LINGUAGGI STATICAMENTE TIPATI

Nei linguaggi tipizzati staticamente il *tipo* di variabile viene stabilito nel codice sorgente, per cui si rende necessario che:

- gli operatori e le assegnazioni devono essere usati coerentemente con il *tipo* dichiarato;
- le variabili siano usate consistentemente rispetto la loro dichiarazione.

I vantaggi della staticità risiedono nella preventiva rilevazione degli errori e nell'efficienza di calcolo risultante dalla coerenza tra codice e compilatore.

### 1.2 LINGUAGGI DINAMICAMENTE TIPATI

Nei linguaggi di programmazione dinamicamente tipati le variabili sono assegnate ai *tipi* durante l'esecuzione del programma, ne consegue che:

- la semantica statica non è definita;
- un utilizzo inconsistente di variabili, operazioni o assegnazioni generano errori dinamici basati sui tipi.

I linguaggi dinamici per questi motivi risultano essere più semplici ed espressivi.

#### 1.2.1 Esempi di errori

Listing 1.1: Errore di sintassi

```
|| x = ;
```

Un errore sintattico generico a molti linguaggi è un'espressione formattata erroneamente.

Listing 1.2: Errore statico

```
int x=0;  
if(y<0) x=3; else x="three"
```

In un linguaggio statico come *Java* l'esempio precedente darebbe un errore in quanto una stringa non può essere convertita in un tipo intero.

Listing 1.3: Errore Dinamico

```
x = null;  
if(y<0) y=1; else y=x.value;
```

L'esempio, per  $y > 0$ , darebbe un errore dinamico sia in *Java*, che in un linguaggio dinamico come *Javascript*.

Parte I

SINTASSI



## STRINGHE

**Definizione 1** Un alfabeto  $A$  è un insieme finito non vuoto di simboli.

**Definizione 2** Sia una stringa in un alfabeto  $A$  la successione di simboli in  $u$ :

$$u : [1 \dots n] \rightarrow A$$

Sia:

- $[1 \dots n] = m$ , l'intervallo dei numeri naturali tale che:

$$1 \leq m \leq n;$$

- $u$  sia una funzione totale;
- $n$  sia la lunghezza di  $u : \text{length}(u) = n$ .

**Definizione 3** Un programma è una stringa in un alfabeto  $A$ .

## 2.1 ESEMPIO DI STRINGHE

### 2.1.1 Stringa vuota

$$u : [1 \dots 0] \rightarrow A$$

Esiste un'unica funzione  $u : 0 \rightarrow A$

Le notazioni standard di una stringa vuota sono:  $\varepsilon, \lambda$

### 2.1.2 Stringa non vuota

Si consideri  $A = \{ 'a', \dots, 'z' \} \cup \{ 'A', \dots, 'Z' \}$ , l'alfabeto inglese di lettere minuscole e maiuscole. La funzione  $u : [1 \dots 4] \rightarrow A$  rappresenta la stringa "Word" con:

- $u(1) = 'W'$
- $u(2) = 'o'$
- $u(3) = 'r'$
- $u(4) = 'd'$

## 2.1.3 Concatenazione di stringhe

**Definizione 4**

$$\text{length}(u \cdot v) = \text{length}(u) + \text{length}(v)$$

Per ogni  $i \in [1 \dots \text{length}(u) + \text{length}(v)]$

$$(u \cdot v)(i) = \text{if } i \leq \text{length}(u) \text{ then } u(i) \text{ else } v(i - \text{length}(u))$$

*Monoide*

La concatenazione è associativa, ma non commutativa.

La stringa vuota è l'identità dell'elemento.

*Induzione*

La definizione di  $u^n$  per induzione su  $n \in \mathbb{N}$ :

Base:  $u^0 = \lambda$

Passo induttivo:  $u^{n+1} = u \cdot u^n$  Per cui  $u^n$  si concatena con se stesso  $n$  volte.

## 2.1.4 Insiemi di stringhe

**Definizione 5** Sia  $A$  un alfabeto:

- $A^n =$  l'insieme di tutte le stringhe in  $A$  con lunghezza  $n$ ;
- $A^+ =$  l'insieme di tutte le stringhe in  $A$  con lunghezza maggiore di 0;
- $A^* =$  l'insieme di tutte le stringhe in  $A$ ;
- $A^+ = \bigcup_{n>0} A^n$ ;
- $A^* = \bigcup_{n \geq 0} A^n = A^0 \cup A^+$

## 2.2 LINGUAGGIO FORMALE

**Definizione 6 (Nozione sintattica di linguaggio)** Un linguaggio  $L$  in un alfabeto  $A$  è un sottoinsieme di  $A^*$

ESEMPIO: L'insieme  $L_{\text{id}}$  di tutti gli identificatori di variabile:

$$A = \{ 'a', \dots, 'z' \} \cup \{ 'A', \dots, 'Z' \} \cup \{ '0', \dots, '9' \}$$

$$L_{\text{id}} = \{ 'a', 'b', \dots, 'a0', 'a1', \dots \}$$

## 2.2.1 Composizione di operatori tra linguaggi

Le operazioni possono essere di concatenazione o di unione:

- **Concatenazione:**  $L_1 \cdot L_2 = \{ u \cdot w \mid u \in L_1, w \in L_2 \}$ ;
- **Unione:**  $L_1 \cup L_2$ .



### 2.2.2 Intuizione

#### Unione

$L = L_1 \cup L_2$ : qualsiasi stringa  $L$  è una stringa di  $L_1$  o di  $L_2$ .

ESEMPIO:

$$L' = \{ 'a', \dots, 'z' \} \cup \{ 'A', \dots, 'Z' \}$$

#### Concatenazione

$L = L_1 \cdot L_2$ : qualsiasi stringa  $L$  è una stringa di  $L_1$ , seguita da una stringa di  $L_2$ .

ESEMPIO:

$$\{ 'a', 'ab' \} \cdot \{ \lambda, '1' \} = \{ 'a', 'ab', 'a1', 'ab1' \}$$

$$L_{\text{id}} = L' \cdot A^* \text{ con } A = \{ 'a', \dots, 'z' \} \cup \{ 'A', \dots, 'Z' \} \cup \{ '0', \dots, '9' \}$$

### 2.2.3 Monoide

La concatenazione è associativa, ma non commutativa.

$A^0 (= \{ \lambda \})$  è l'identità dell'elemento; quindi  $A^0$  non è l'elemento neutro, l'elemento neutro è  $0 = \{ \}$ .

### 2.2.4 Passo induttivo

$L^n$  è definito per induzione su  $n \in \mathbb{N}$ : Base:  $L^0 = A^0 (= \{ \lambda \})$ ,

Passo induttivo:  $L^{n+1} = L \cdot L^n$ .

### 2.2.5 Operatori + e \*

- **Addizione:**  $L^+ = \bigcup_{n>0} L^n$ ;
- **Moltiplicazione:**  $\star$  viene chiamata *Kleen star*, *stella di Kleen*.

$$L^* = \bigcup_{n \geq 0} L^n$$

Sono equivalenti  $L^* = L^0 \cup L^+, L \cdot L^*$ .

#### Intuizione

- Qualsiasi stringa di  $L^+$  è ottenuta concatenando una o più stringhe di  $L$ ;
- Qualsiasi stringa di  $L^*$  è ottenuta concatenando 0 o più stringhe di  $L$ : Concatenando zero stringhe si ottiene la stringa vuota.



## ESPRESSIONI REGOLARI

---

Le espressioni regolari sono un formalismo comunemente utilizzato per definire linguaggi semplici.

**Definizione 7** *La definizione induttiva di un espressione regolare su un alfabeto  $A$ :*

**BASE:**

- $0$  è un espressione regolare di  $A$ ;
- $\lambda$  è un espressione regolare di  $A$ ;
- per ogni  $\sigma \in A$ ,  $\sigma$  è un espressione regolare in  $A$ .

**PASSO INDUTTIVO:**

- se  $e_1$  ed  $e_2$  sono espressioni regolari di  $A$ ,  
allora  $e_1|e_2$  è un espressione regolare di  $A$ ;
- se  $e_1$  ed  $e_2$  sono espressioni regolari di  $A$ ,  
allora  $e_1e_2$  è un espressione regolare di  $A$ ;
- se  $e$  è un espressione regolare di  $A$ ,  
allora  $e^*$  è un espressione regolarare di  $A$ .

**ESERCIZIO** Scrivere un **REGEX** che esprima i nomi di variabili permessi.

$$L_{id} = (\{ 'a', \dots, 'z' \} \cup \{ 'A', \dots, 'Z' \}) \cdot \{ 'a', \dots, 'z' \} \cup \{ 'A', \dots, 'Z' \} \cup \{ '0', \dots, '9' \}^*$$

$$e_{id} = (a| \dots |z|A| \dots |Z)(a| \dots |z|A| \dots |Z|0| \dots |9)^*$$

### 3.1 SEMANTICA

La semantica di un espressione regolare in  $A$  è un linguaggio su  $A$ :

- $\emptyset \rightsquigarrow$  insieme vuoto;
- $\epsilon \rightsquigarrow \{\epsilon\}$ ;
- $\sigma \rightsquigarrow \{''\sigma''\}, \forall \sigma \in A$ ;
- $e_1 | e_2 \rightsquigarrow$  unione delle semantiche di  $e_1$  ed  $e_2$ ;
- $e_1 e_2 \rightsquigarrow$  concatenazione delle semantiche di  $e_1$  ed  $e_2$ .

## 3.2 SINTASSI CONCRETA DELLE ESPRESSIONI REGOLARI

## 3.2.1 Precedenza ed associatività

- la **stella di Kleene** ha priorità sulla concatenazione e l'unione;
- la concatenazione ha precedenza sull'unione;
- la concatenazione e l'unione sono associative a sinistra.

## 3.2.2 Operatori derivati

- $e^+ = ee^+$ : (una o più volte  $e$ );
- $\epsilon$ : è rappresentata dalla stringa vuota:  $a|\epsilon$  diventa  $a|$ ;
- $e? = |e$ : ( $e$  è opzionale, ovvero uno o nessuno);
- $[a|B]$ : uno qualsiasi dei caratteri nella quadre ( $a|B$ );
- $[b - d]$ : uno qualsiasi dei caratteri nel range tra le quadre ( $b|c|d$ );
- $[a|B][b - d]$ : può essere scritto come  $[a|Bb - d]$ ;
- $[\wedge \dots]$ : qualsiasi carattere ad eccezioni di  $\dots$  (esempio:  $[^a|Bb - d]$  qualsiasi carattere ad eccezione di  $a, 0, B, b, c, d$ ).

## 3.2.3 Caratteri speciali in JAVA

- `.` rappresenta ogni carattere;
- `\` è il carattere di *escape*, per dare un significato speciale ai caratteri regolari, oppure un significato ordinario per caratteri speciali.

*Caratteri speciali cui dare un significato ordinario*

Esempi:

`|, *, +, ?, ., (, ), [, ], -, ^`

## SEMANTICHE:

- $\backslash. \rightsquigarrow \{".\." \}$ ,
- $\backslash\backslash \rightsquigarrow \{""\backslash\}$ ,
- $. \rightsquigarrow \{s \mid s \text{ ha lunghezza } 1, \text{ l'insieme di tutti i caratteri} \}$ ,
- $\backslash \rightsquigarrow$  non è sintatticamente corretto.
- $n \rightsquigarrow \{""n"" \}$ ,
- $\backslash n \rightsquigarrow \{""linefeed"" \}$ ,

*Caratteri cui dare un significato speciale*

- $\backslash t$ : tab;
- $\backslash n$ : newline;
- $\backslash s$ : qualsiasi spazio vuoto;
- $\backslash S$ : qualsiasi spazio non vuoto;
- $\backslash d$ : qualsiasi carattere numerico ( $[0 - 9]$ );
- $\backslash D$ : qualsiasi carattere non numerico ( $[\wedge 0 - 9]$ );
- $\backslash w$ : qualsiasi parola ( $[[a - zA - Z_0 - 9]]$ );
- $\backslash W$ : qualsiasi carattere che non sia una parola ( $[\wedge \backslash w]$ ).

**Definizione 8** *Un linguaggio si dice regolare se può essere definito da un'espressione regolare.*

### 3.3 ANALISI LESSICALE

**Definizione 9** *Un lexeme è una sottostringa considerata come un'unità sintattica.*

**Definizione 10** *L'analisi lessicale affronta il problema della decomposizione di una stringa in un lexeme.*

**Definizione 11** *Un lexer o scanner è un programma che esegue l'analisi lessicale e genera lexemes.*

**ESEMPIO IN C:** La stringa `"x2 = 042;` è decomposta nei *lexemes* seguenti:

- `"x2"`;
- `" = "`;
- `"042"`;
- `","`.

#### 3.3.1 Token

Un *token* è una nozione di *lexeme* più astratta; ad un *token* corrisponde sempre un *lexeme*.

In alcuni casi un *token* può mantenere informazioni sulla semantica, come i valori dei numeri.

Un *tokenizer* è un programma che esegue l'analisi lessicale e genera *token*.

ESEMPIO IN C: La stringa "x2 = 042;" è decomposta nei token seguenti:

- *IDENTIFIER*: con il nome "x2";
- *ASSIGN\_OP*;
- *INT\_NUMBER*: con il valore di 34;
- *STATEMENT\_TERMINATOR*.

### 3.4 LINGUAGGI REGOLARI

**Definizione 12** *Un linguaggio regolare è un linguaggio definibile con un'espressione regolare.*

*I linguaggi regolari possono essere definiti in altre maniere equivalenti:*

- *con una grammatica regolare a destra o sinistra, anche chiamata lineare;*
- *con una serie di automata non deterministica o deterministica finita (NEA o DFA).*

#### 3.4.1 Limitazioni

I linguaggi regolari sono linguaggi semplici. Esempio:

- il linguaggio degli identificatori;
- il linguaggio dei numeri.

Le espressioni regolari possono definire le unità che costituiscono la sintassi di un linguaggio di programmazione, ma non possono definire la sintassi dell'intero linguaggio.

#### 3.4.2 Esempi di linguaggi non regolari

Il linguaggio di espressioni con numeri naturali, addizione binaria e moltiplicazione e parentesi **non possono** essere definiti da un'espressione regolare: il problema è posto dalle parentesi, per cui se venissero rimosse, allora il linguaggio sarebbe regolare.

Un altro esempio di linguaggio semplice non regolare:

$$\{ "a^n b^n" \mid n \in \mathbb{N} \} = \{ "", "ab", "aabb", "aaabbb", \dots \}.$$

## ANALISI SINTATTICA

---

**Definizione 13** *L'analisi sintattica è definita sull'analisi lessicale, risolve:*

- *il riconoscimento di una sequenza di lexems/tokens come validi se rispettano alcune regole sintattiche;*
- *la costruzione, in caso di successo, di una rappresentazione astratta della sequenza riconosciuta per eseguire delle operazioni su di essa.*

### 4.1 PARSE

**Definizione 14** *Un parser è un programma che esegue l'analisi sintattica.*

#### 4.1.1 Parsers per i linguaggi di programmazione

I parser per i linguaggi di programmazione, riconoscono i *token* generati da un *tokenizer*, mentre le regole sintattiche sono definite formalmente da una *grammatica*.

I parser generano:

- un albero **parse/deviation**, una rappresentazione meno astratta della sequenza analizzata;
- un **albero sintattico astratto** (abstract syntax tree, AST), una rappresentazione più astratta della sequenza analizzata.

Inoltre i parser possono essere scritti a mano, oppure generati automaticamente da una grammatica con specifici software come *ANTLR* o *BISON*.

PRIMO ESEMPIO CON SINTASSI C/JAVA/C++    Token analizzati:

- *IDENTIFIER*: con il nome "x2";
- *ASSIGN\_OP*;
- *INT\_NUMBER*: con il valore di 34;
- *STATEMENT\_TERMINATOR*.

Il parser ritornerà errore dato che la sequenza non è riconosciuta oppure uno o più messaggi di errore sono presenti.

SECONDO ESEMPIO CON SINTASSI C / JAVA / C++ Token analizzati:

- *IDENTIFIER*: con il nome "x2";
- *ASSIGN\_OP*;
- *INT\_NUMBER*: con il valore di 34;
- *ADD\_OP*;
- *INT\_NUMBER*: con il valore di 10;
- *STATEMENT\_TERMINATOR*.

Il parser riconosce la sequenza e genera un AST.

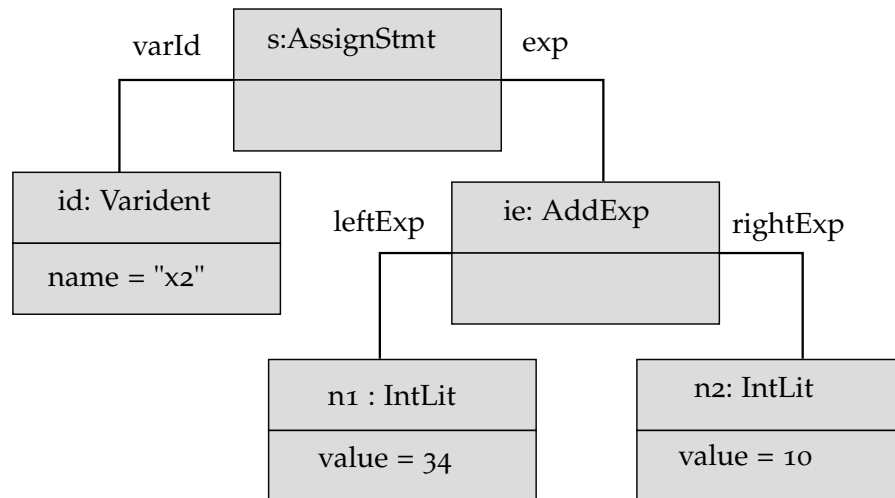


Figura 4.1: L'AST generato della sequenza precedente.



## CONTEXT FREE (CF) GRAMMARS

Le grammatiche *context free*, sono il formalismo più diffuso per definire le regole sintattiche di un linguaggio di programmazione. Sono più espressive di un'espressione regolare e sono basate sulla concatenazione di unione di più nomi e su definizioni ricorsive.

Listing 5.1: Una grammatica CF per espressioni semplici

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

DA NOTARE CHE: *Num* è definito nella grammatica solo per completezza, infatti i token *Num* sono definiti separatamente da un'espressione regolare.

Listing 5.2: Esempio rivisitato di una grammatica CF per espressioni semplici

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
NUM definito da 0|1
```

NOTAZIONE: In *Exp* è maiuscolo solo il primo carattere: è definito nella grammatica. In *NUM* tutte le lettere sono maiuscole; è definito separatamente da un'espressione regolare.

## 5.1 TERMINOLOGIA DELLE GRAMMATICHE CF

Listing 5.3: Esempio

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

5.1.1 Terminologia per la grammatica *G*

Per  $G = (T, N, P)$ :

- $\{ '+', '*', '(', ')', '0', '1' \}$  è l'insieme *T* dei **simboli terminali**;
- $\{ Exp, Num \}$  sono l'insieme *N* di simboli **non terminali**;
- $\{ (Exp, Num), (Exp, Exp, '+' Exp), (Exp, Exp, '*' Exp), (Exp, '(' Exp ')'), (Num, '0'), (Num, '1') \}$  è l'insieme *P* di produzioni.

DA NOTARE CHE:

- ogni simbolo non terminale corrisponde ad un linguaggio; i linguaggi sono definiti come unioni di concatenazioni;
- i simboli terminali sono *lexems* dei linguaggi definiti dalla grammatica;
- le produzioni hanno forma  $(B, \alpha)$ , per  $B \in \mathbb{N} \cap \alpha \in (T \cup N)^*$

## 5.2 GRAMMATICHE COME DEFINIZIONE INDUTTIVA DI LINGUAGGI

### 5.2.1 Primo esempio

Listing 5.4: Esempio

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

*Definizione induttiva di linguaggi*

$$\begin{aligned} \text{Exp} &= \text{Num} \cup (\text{Exp} \cdot \{ "+" \} \cdot \text{Exp}) \cup (\text{Exp} \cdot \{ "*" \} \cdot \text{Exp}) \cup (\{ "(" \} \cdot \text{Exp} \cdot \{ ")" \}) \\ \text{Num} &= \{ "0" \} \cup \{ "1" \} \end{aligned}$$

DA NOTARE CHE:

- $\text{Exp} = \text{Num} \cup \dots$  è il caso base per  $\text{Exp}$ : un numero è un espressione;
- $\text{Exp}$  è definito su di  $\text{Num}$ ,  $\text{Num}$  è definito esclusivamente per casi base.

### 5.2.2 Secondo esempio

Listing 5.5: Esempio

```
Exp ::= Term | Exp '+' Term | Exp '*' Term
Term ::= '(' Exp ')' | Num
Num ::= '0' | '1'
```

DA NOTARE CHE: Le definizioni di  $\text{Exp}$  e  $\text{Term}$  sono ricorsive reciprocamente.

## 5.3 DERIVAZIONI

Listing 5.6: Esempio

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

### 5.3.1 Linguaggi generati da una grammatica

- Una grammatica genera un linguaggio per ogni simbolo non terminale;
- la grammatica precedente genera due linguaggi  $L_{\text{Exp}}$  e  $L_{\text{Num}}$ ;
- il linguaggio per  $\text{Num}$  è relativamente semplice:  $L_{\text{Num}} = \{ "0", "1" \}$ .

DA NOTARE CHE: per definire  $L_{\text{Num}}$  e per dimostrare che  $"1 + 0" \in L_{\text{Exp}}$  e che  $"1 + *(" \notin L_{\text{Exp}}$  si rendono necessarie la **derivazione a passo singolo** e la **derivazione a passi multipli**.

### 5.3.2 Derivazione ad un passo

- $\text{Exp} \rightarrow \text{Exp}' *' \text{Exp}$  è usata la produzione  $(\text{Exp}, \text{Exp}' *' \text{Exp})$ ;
- $\text{Exp}' *' \text{Exp} \rightarrow \text{Num}' *' \text{Exp}$  è usata la produzione  $(\text{Exp}, \text{Num})$ ;
- $\text{Num}' *' \text{Exp} \rightarrow \text{Num}' *' \text{Num}$  è usata la produzione  $(\text{Exp}, \text{Num})$ ;
- $\text{Num}' *' \text{Num} \rightarrow '0'' *' \text{Num}$  è usata la produzione  $(\text{Num}, '0')$ ;
- $'0'' *' \text{Num} \rightarrow '0'' *' '1'$  è usata la produzione  $(\text{Num}, '1')$ ;

DA NOTARE CHE: Non esiste alcuna derivazione da  $'0'' *' '1'$  dato che nessuna produzione può essere usata;  $'0'' *' '1'$  è la stringa  $0 * 1$  che appartiene a  $L_{\text{Exp}}$ .

### 5.3.3 Definizioni di derivazione

#### Derivazione ad un passo

**Definizione 15** La derivazione ad un passo per la grammatica  $G = (T, N, P)$ :

- possiede una forma  $\alpha_1 B \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$ ;
- $\alpha_1, \alpha_2 \in (T \cup N)^*$ ;
- $(B, \gamma) \in P$  ovvero  $(B, \gamma)$  in produzione.

#### Derivazione a più passi

**Definizione 16** La chiusura transitiva di  $\rightarrow$ :

- il caso base: se  $\gamma_1 \rightarrow \gamma_2$ , allora  $\gamma_1 \rightarrow^+ \gamma_2$ ;
- caso induttivo: se  $\gamma_1 \rightarrow \gamma_2$ , e  $\gamma_2 \rightarrow^+ \gamma_3$ , allora  $\gamma_1 \rightarrow^+ \gamma_3$ .

*Linguaggio generato*

Il linguaggio  $L_B$  generato da  $G = (T, N, P)$  per i non terminali  $B \in N$ :

- tutte le stringhe di terminali che possono essere derivati in uno o più passaggi da  $B$ ;
- formalmente:  $L_B = \{u \mid B \rightarrow^+ u\}$ .

5.3.4 *Albero di derivazione (parse tree)*

OSSERVAZIONE 1 Le grammatiche CF sono utilizzate per definire linguaggi ed implementare parsers, i parsers dovrebbero generare gli alberi, ma le derivazioni non sono alberi.

OSSERVAZIONE 2 Un passo di derivazione è determinato da:

- la produzione usata;
- lo specifico simbolo non terminale che rimpiazza.

Quest'ultimo punto non influenza la stringa finale dei terminali ottenuti dalla derivazione.

INTUIZIONE Un albero di derivazione è una generalizzazione di una derivazione a più passaggi in modo che la stringa derivata contenga solo terminali e che i non terminali siano rimpiazzati in parallelo.

5.3.5 *Esempi di alberi di derivazione (ANTLR)*

Listing 5.7: Grammatica ANTLR

```
grammar SimpleExp;
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')';
Num ::= '0' | '1';
```

*Albero di derivazione per "1\*1+1"*

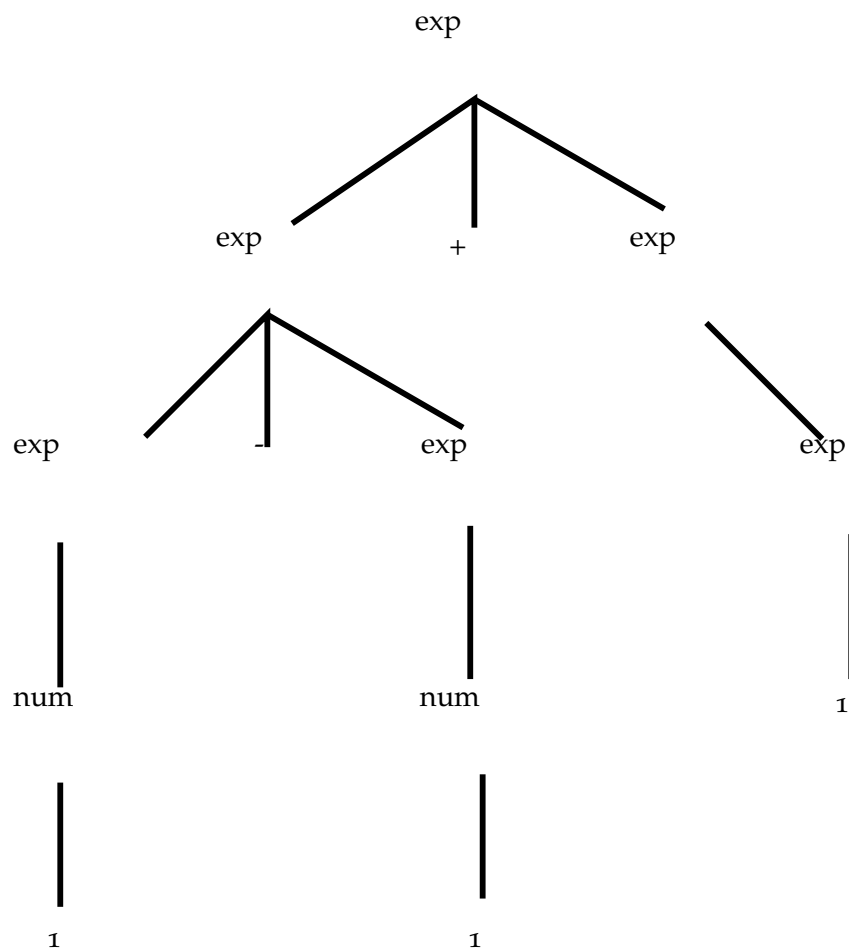


Figura 5.1: Albero di derivazione per "1\*1+1"