

# LINGUAGGI E PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

RICCARDO CEREGHINO



Appunti

Settembre 2019 – classicthesis v4.6



## INDICE

---

1	INTRODUZIONE AGLI ELEMENTI DI UN LINGUAGGIO DI PROGRAMMAZIONE	1
1.1	Linguaggi staticamente tipati	1
1.2	Linguaggi dinamicamente tipati	1
1.2.1	Esempi di errori	1
I	SINTASSI	
2	STRINGHE	5
2.1	Esempio di stringhe	5
2.1.1	Stringa vuota	5
2.1.2	Stringa non vuota	5
2.1.3	Concatenazione di stringhe	6
2.1.4	Insiemi di stringhe	6
2.2	Linguaggio formale	6
2.2.1	Composizione di operatori tra linguaggi	6
2.2.2	Intuizione	7
2.2.3	Monoide	7
2.2.4	Passo induttivo	7
2.2.5	Operatori + e *	7
3	ESPRESSIONI REGOLARI	9
3.1	Semantica	9
3.2	Sintassi concreta delle espressioni regolari	10
3.2.1	Precedenza ed associatività	10
3.2.2	Operatori derivati	10
3.2.3	Caratteri speciali in JAVA	10
3.3	Analisi lessicale	11
3.3.1	Token	11
3.4	Linguaggi regolari	12
3.4.1	Limitazioni	12
3.4.2	Esempi di linguaggi non regolari	12
4	ANALISI SINTATTICA	13
4.1	Parser	13
4.1.1	Parsers per i linguaggi di programmazione	13
5	CONTEXT FREE (CF) GRAMMARS	15
5.1	Terminologia delle grammatiche CF	15
5.1.1	Terminologia per la grammatica G	15
5.2	Grammatiche come definizione induttiva di linguaggi	16
5.2.1	Primo esempio	16
5.2.2	Secondo esempio	16
5.3	Derivazioni	16
5.3.1	Linguaggi generati da una grammatica	17

5.3.2	Derivazione ad un passo	17
5.3.3	Definizioni di derivazione	17
6	ALBERI DI DERIVAZIONE	19
6.1	Albero di derivazione (parse tree)	19
6.1.1	Esempi di alberi di derivazione (ANTLR)	19
6.1.2	Definizione di albero di derivazione in $G=(T,N,P)$	20
7	GRAMMATICHE AMBIGUE	21
7.1	Soluzioni per l'ambiguità	21
7.1.1	Notazione prefissa	21
7.1.2	Notazione postfissa	21
7.1.3	Notazione funzionale	21
7.1.4	Notazione infissa	21
7.1.5	Operatori con la stessa precedenza	22
7.1.6	Tecniche per risolvere l'ambiguità	22
7.2	Sintassi ambigua per statements	25
8	COSTRUZIONE DI UN PARSER DA UNA GRAMMATICA	27
8.1	Come costruire un parser da una grammatica	27
8.2	Grammatiche EBNF	29
8.2.1	Da una grammatica ad un parser top-down	29
9	PARADIGMI DI PROGRAMMAZIONE	33
9.1	Paradigma completamente funzionale	33
9.2	OCaml	34
9.2.1	Funzioni ed applicazioni	34
9.2.2	Regole di precedenza ed associatività	35
9.2.3	Funzioni di alto ordine	37
9.2.4	Tuple	37

## ELENCO DELLE FIGURE

---

Figura 4.1	L'AST generato della sequenza precedente.	14
Figura 6.1	Albero di derivazione per "1*1+1"	19
Figura 7.1	Primo esempio di albero di derivazione per "if(x) x=false;y=true"	25
Figura 7.2	Secondo esempio di albero di derivazione per "if(x) x=false;y=true"	25
Figura 8.1	Parse tree per "1+1*1"	29

## LISTINGS

---

Listing 1.1	Errore di sintassi	1
Listing 1.2	Errore statico	2
Listing 1.3	Errore Dinamico	2
Listing 5.1	Una grammatica CF per espressioni semplici	15
Listing 5.2	Esempio rivisitato di una grammatica CF per espressioni semplici	15
Listing 5.3	Esempio	15
Listing 5.4	Esempio	16
Listing 5.5	Esempio	16
Listing 5.6	Esempio	16
Listing 6.1	Grammatica ANTLR	19
Listing 7.1	Notazione prefissa	21
Listing 7.2	Notazione postfissa	21
Listing 7.3	Notazione funzionale	21
Listing 7.4	Grammatica ambigua	22
Listing 7.5	Associatività a sinistra non ambigua	23
Listing 7.6	Associatività a destra non ambigua	23
Listing 7.7	Associative a sinistra non ambigue	24
Listing 7.8	Associative a destra non ambigue	24
Listing 7.9	Esempio di ambiguità per gli statements	25
Listing 8.1	Una grammatica non ambigua	27
Listing 8.2	Una grammatica non ambigua	27
Listing 8.3	Grammatica rivisitata a seguito delle osservazioni precedenti	28
Listing 8.4	Soluzione completa	28
Listing 8.5	Soluzione completa utilizzando la grammatica ENBNF	29

Listing 8.6	Pseudocodice derivato dalla grammatica EBNF precedente	30
Listing 8.7	Grammatica non ambigua	30
Listing 8.8	Grammatica EBNF equivalente	30
Listing 8.9	Pseudocodice derivato dalla grammatica EBNF precedente	31
Listing 9.1	Sintassi	34
Listing 9.2	Esempio di funzioni anonime	34
Listing 9.3	Esempio di applicazione di funzioni	34
Listing 9.4	Esempio di applicazioni	35
Listing 9.5	Esempio di associatività	35
Listing 9.6	Esempio di precedenza	35
Listing 9.7	Esempio di precedenza	35
Listing 9.8	Casi limite di precedenza	35
Listing 9.9	I tipi possono essere inferiti dall'interprete	36
Listing 9.10	Grammatica BNF	36
Listing 9.11	Associatività a destra dell'operatore freccia	36

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

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

## 2.1.3.1 Monoide

La concatenazione è associativa, ma non commutativa.

La stringa vuota è l'identità dell'elemento.

## 2.1.3.2 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

#### 2.2.2.1 Unione

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

ESEMPIO:

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

#### 2.2.2.2 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  $L^* = L^0 \cup L^+, L \cdot L^*$ .

##### 2.2.5.1 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);
- $[a0B]$ : uno qualsiasi dei caratteri nella quadre ( $a|0|B$ );
- $[b - d]$ : uno qualsiasi dei caratteri nel range tra le quadre ( $b|c|d$ );
- $[a0B][b - d]$ : può essere scritto come  $[a0Bb - d]$ ;
- $[\wedge \dots]$ : qualsiasi carattere ad eccezioni di  $\dots$  (esempio:  $[^a0Bb - d]$  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.

## 3.2.3.1 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.
- $n \rightsquigarrow \{""n"" \}$ ,
- $\backslash n \rightsquigarrow \{""linefeed"" \}$ ,



3.2.3.2 *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 (NFA 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.

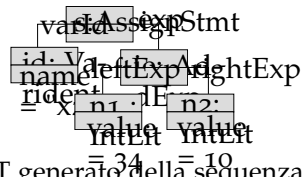


Figura 4.1: L'AST generato della sequenza precedente.

## 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'
```

#### 5.2.1.1 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

#### 5.3.3.1 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.

#### 5.3.3.2 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  $\gamma_1 \rightarrow^+ \gamma_3$ .

5.3.3.3 *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:  $L_B = \{u \mid B \rightarrow^+ u\}$ .



## ALBERI DI DERIVAZIONE

### 6.1 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.

#### 6.1.1 Esempi di alberi di derivazione (ANTLR)

Listing 6.1: Grammatica ANTLR

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

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

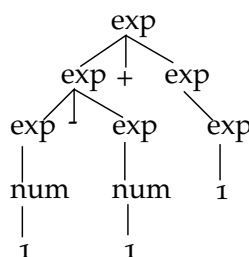


Figura 6.1: Albero di derivazione per "1\*1+1"

ESERCIZIO: Mostrare che  $"1 * 1 + 1" \in L_{\text{Exp}}$  usando la nozione di derivazione ad uno o più passi.

$$\text{Exp} \rightarrow \text{Exp}' + ' \text{Exp} \rightarrow^+ \text{Exp} \times \text{Exp}' + ' \text{Num}' \rightarrow^+ \text{Num}' * ' \text{Num}' + ' 1' \rightarrow^+ '1' '*' '1' '+' '1'$$

ESERCIZIO: Mostrare che  $"1 + (\notin L_{\text{Exp}}:$

$$\text{Exp} \rightarrow \text{Exp}' * ' \text{Exp} \rightarrow^+ \text{Num}' * ' \text{Exp}' + ' \text{Exp} \rightarrow^+ '1' '*' \text{Num}' + ' \text{Num} \rightarrow^+ '1' '*' '1' '+' '1'$$

### 6.1.2 Definizione di albero di derivazione in $G=(T,N,P)$

Albero di derivazione per  $u \in T^*$  partendo da  $B \in N$ .

- se un nodo è etichettato da  $C$  ed ha  $n$  figli  $I_1, \dots, I_n$ , allora  $(C, I_1, \dots, I_n) \in P$  (ovvero,  $(C, I_1, \dots, I_n)$  è una produzione di  $G$ ;
- la radice è etichettata da  $B$ ;
- $u$  è ottenuto dalla concatenazione da sinistra a destra di tutte le etichette terminali (nodi foglia).

#### 6.1.2.1 Definizione equivalente di un linguaggio generato

Il linguaggio  $L_B$  generato da  $G = (T, N, P)$  per un non terminale  $B \in N$  è composto da tutte le stringhe  $u$  di terminali così che esiste un albero di derivazione per  $u$  partendo da  $B$ .

## GRAMMATICHE AMBIGUE

**Definizione 17** Una grammatica  $G = (T, N, P)$  è ambigua per  $B \in N$  se esistono due differenti alberi di derivazione partendo da  $B$  per la stessa stringa.

Per esempio la grammatica  $1 + 1 + 1$  è ambigua a seconda delle parentesi.

## 7.1 SOLUZIONI PER L'AMBIGUITÀ

Per risolvere il problema dell'ambiguità si può cambiare la sintassi:

## 7.1.1 Notazione prefissa

Listing 7.1: Notazione prefissa

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

In questo caso esiste un unico albero di derivazione per  $11 + 1*$  e le parentesi non sono più necessarie.

## 7.1.2 Notazione postfissa

Listing 7.2: Notazione postfissa

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

In questo caso esiste di nuovo un unico albero di derivazione e le parentesi sono di nuovo non necessarie.

## 7.1.3 Notazione funzionale

Listing 7.3: Notazione funzionale

```
Exp ::= Num | 'add' '(' Exp ',' Exp ')' | 'mul' '(' Exp ',' Exp ')'
Num ::= '0' | '1'
```

In questo esempio esiste un unico albero di derivazione per  $add(1, mul(1, 1))$ .

## 7.1.4 Notazione infissa

Generalmente la notazione infissa è una soluzione più pratica.

- si definiscono le regole di associatività per gli operatori binari:

- addizione associativa a sinistra: " $1 + 1 + 1$ " diventa " $(1 + 1) + 1$ ";
- addizione associativa a destra: " $1 + 1 + 1$ " diventa " $1 + (1 + 1)$ ";
- si definiscono le regole di precedenza per gli operatori, usando le parentesi per sovrascriverlo:
  - la moltiplicazione ha precedenza rispetto l'addizione: " $1 + 1 * 1$ " significa " $1 + (1 * 1)$ ";
  - l'addizione ha precedenza rispetto la moltiplicazione: " $1 * 1 + 1$ " significa " $1 * (1 + 1)$ ".

#### 7.1.5 Operatori con la stessa precedenza

Gli operatori binari possono avere la stessa precedenza, in questo caso condividono la regola associativa:

- addizione e moltiplicazione hanno la stessa precedenza e sono associative a sinistra: " $1 + 1 * 1$ " diventa " $1 + (1 * 1)$ " e " $1 * 1 + 1$ " diventa " $(1 * 1) + 1$ ";
- addizione e moltiplicazione hanno la stessa precedenza e sono associative a destra: " $1 + 1 * 1$ " diventa " $1 + (1 * 1)$ " e " $1 * 1 + 1$ " diventa " $1 * (1 + 1)$ ";

NOTA CHE: le regole sull' associatività risolvono ambiguità tra operatori binari con la stessa precedenza, inoltre mischiando operatori con diverse **arità** rende l'eliminazione dell'ambiguità più complessa.

#### 7.1.6 Tecniche per risolvere l'ambiguità

- una grammatica ambigua  $G$  è trasformata in una grammatica non ambigua  $G'$ ;
- l' **equivalenza** significa che per tutti i non terminali di  $B$  e  $G$ , i linguaggi generati da  $G, G', B$  sono uguali;
- è possibile per la **trasformazione** di codificare l' associatività e le regole di precedenza nella grammatica non ambigua  $G'$ .

##### 7.1.6.1 Esempio 1: $+$ e $*$ con la stessa precedenza

Listing 7.4: Grammatica ambigua

```

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

```

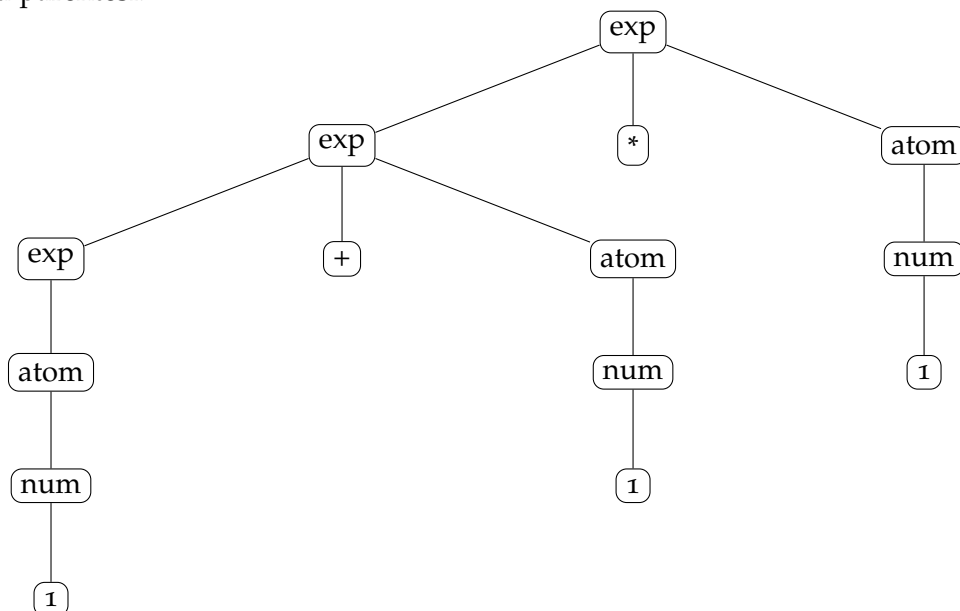
Listing 7.5: Associatività a sinistra non ambigua

```

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

```

Nota che:  $Exp' + Atom | Exp' * Atom$  significa che sul lato destro di  $+(*)$ , le addizione (e moltiplicazioni) sono permesse solo se circondate da parentesi.



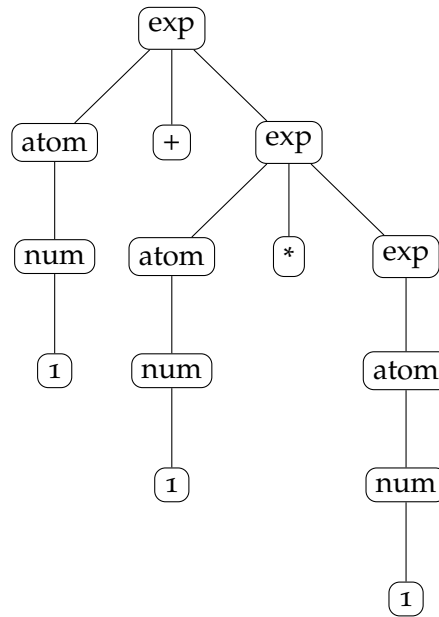
Listing 7.6: Associatività a destra non ambigua

```

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

```

Nota che:  $Atom' + Exp | Atom' * Exp$  significa che sul lato sinistro di  $+(*)$ , le addizione (e moltiplicazioni) sono permesse solo se circondate da parentesi.



### 7.1.6.2 Esempio 2: \* con la maggiore precedenza

Listing 7.7: Associative a sinistra non ambigue

```

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

Nota che:  $Mul' *' Atom$  significa che entrambi i lati delle addizione di  $*$  sono permesse solo se circondate da parentesi.

Listing 7.8: Associative a destra non ambigue

```

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

Nota che:  $Atom' *' Mul$  significa che entrambi i lati delle addizione di  $*$  sono permesse solo se circondate da parentesi.

### 7.1.6.3 Esempi rimanenti

- $*$  con precedenza ed associatività a sinistra,  $+$  associatività a destra;
- $*$  con precedenza ed associatività a destra,  $+$  associatività a sinistra;
- $+$  con precedenza ed associatività a sinistra,  $*$  associatività a destra;
- $+$  con precedenza ed associatività a destra,  $+$  associatività a sinistra;

## 7.2 SINTASSI AMBIGUA PER STATEMENTS

Listing 7.9: Esempio di ambiguità per gli statements

```

Stmt ::= ID '=' Exp | 'if' '(' Exp ')' Stmt | Stmt ';' Stmt | '{'
      Stmt '}'
Exp  ::= ID | BOOL // ID e BOOL sono definiti da espressioni
           regolari

```

Figura 7.1: Primo esempio di albero di derivazione per "if(x) x=false;y=true"

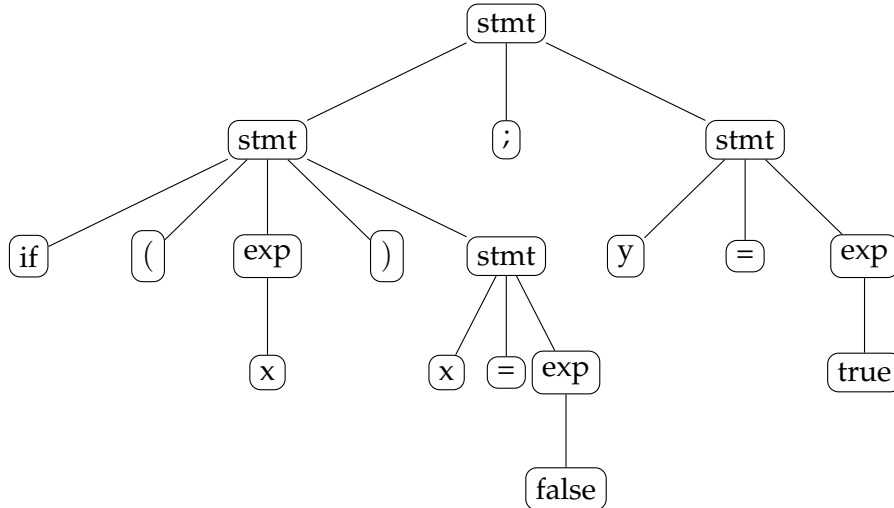
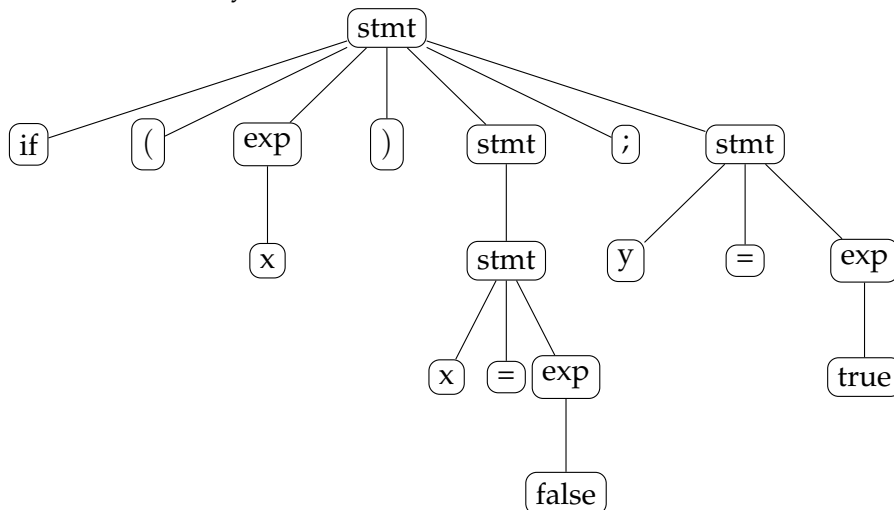


Figura 7.2: Secondo esempio di albero di derivazione per "if(x) x=false;y=true"







## COSTRUZIONE DI UN PARSER DA UNA GRAMMATICA

Individuiamo due passaggi per costruire un parser da una grammatica:

1. la grammatica non deve essere ambigua;

Listing 8.1: Una grammatica non ambigua

```

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

```

2. per ogni token di input, il parser deve scegliere una produzione unica da usare per l'albero *parse* corretto.

### PROBLEMI:

- i token sono letti dal parser da sinistra a destra;
- nella più semplice delle ipotesi, il parser è a conoscenza solo del token successivo (*lookahead token*);
- un parser con un *lookahead token* non è in grado di scegliere la giusta produzione per la grammatica di cui sopra.

### 8.1 COME COSTRUIRE UN PARSER DA UNA GRAMMATICA

Listing 8.2: Una grammatica non ambigua

```

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

```

Quindi un parser con **un** *lookahead token* non può essere costruito per la grammatica di cui sopra.

CONTROESEMPIO: Se il primo *lookahead token* è un numero, allora entrambe le produzioni di *Exp* potrebbero funzionare. A seconda del secondo *lookahead token* *t*:

- la produzione (*Exp*, *Mul*) è usata se *t* è '\*' oppure la fine dello stream di input;
- la produzione (*Exp*, *Exp '+' Mul*) è usata se *t* = '+'.

**Osservazioni:** Per costruire un albero *parse*, la produzione  $(Exp, Mul)$  deve essere usata, inoltre quando le produzioni di  $Exp$  sono usate consecutivamente, vengono ottenute stringhe della forma seguente:  $Mul$ , oppure  $Mul$  seguita dalla stringa  $' + ' Mul$  ripetuta una o più volte.

Listing 8.3: Grammatica rivisitata a seguito delle osservazioni precedenti

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

#### CONSIDERAZIONI SULLA TRASFORMAZIONE DELLA GRAMMATICA

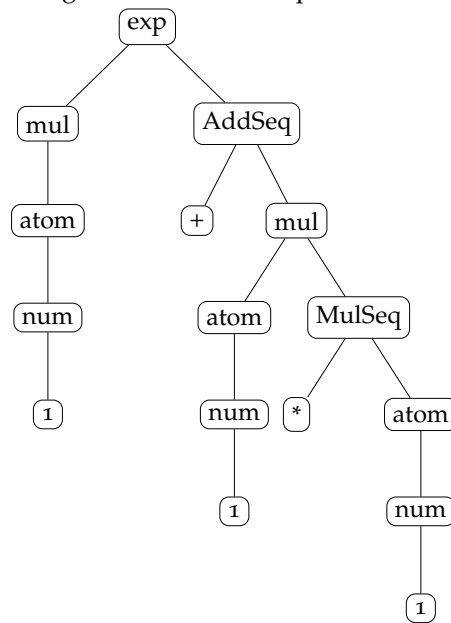
La grammatica è equivalente alla precedente, ma quando si costruisce un albero *parse* sappiamo che un nodo  $Exp$  deve sempre avere un figlio  $c$  a sinistra etichettato da  $Mul$  dopo che l'albero *parse* con radice  $c$  è stato costruito, quindi la produzione corretta è  $(Exp, Mul\ AddSeq)$  se il token *lookahead* è  $' + '$ ; altrimenti la produzione è  $(Exp, Mul)$ : un nodo  $AddSeq$  deve sempre avere un figlio a sinistra  $c_1$  etichettato da  $' + '$ , seguito da un figlio  $c_2$  etichettato da  $Mul$ .

Dopo che l'albero con radice  $c_2$  è costruito, la produzione corretta se il token *lookahead* è  $' + '$  è:  $(AddSeq, '+'\ Mul\ AddSeq)$ , altrimenti la produzione è  $(AddSeq, '+'\ Mul)$ .

Listing 8.4: Soluzione completa

```
Exp ::= Mul | AddSeq
AddSeq ::= '+' Mul | '+' Mul AddSeq
Mul ::= Atom | Atom MulSeq
MulSeq ::= '*' Atom | '*' Atom MulSeq
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Figura 8.1: Parse tree per "1+1\*1"



## 8.2 GRAMMATICHE EBNF

La notazione **BNF** estende la notazione postfissa con gli operatori:  $*$ ,  $+$ ,  $?$ . La grammatica precedente può essere trasformata in una grammatica più semplice utilizzando la notazione **EBNF**.

Listing 8.5: Soluzione completa utilizzando la grammatica EBNF

```

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

```

### 8.2.1 Da una grammatica ad un parser top-down

Per semplicità consideriamo solamente il problema del riconoscimento del linguaggio, **top-down** significa che il *parse tree* è costruito dalla radice, l'albero di generazione sarà considerato nei laboratori.

**ASSUNZIONI SUL TOKENIZER** Il tokenizer definisce le procedure seguenti:

- *nextToken()*: la lettura del successivo *lookahead token*;
- *tokenType()*: il tipo del token corrente;
- *checkTokenType()*: il tipo del token corrente è controllato un'eccezione viene sollevata se il controllo fallisce, altrimenti viene letto il token successivo.

**Linee guida** Il codice del parser proviene direttamente dalla grammatica, struttura ricorsiva inclusa, il parser consiste in una procedura principale insieme ad una procedura per ogni simbolo non terminale della grammatica.

Listing 8.6: Pseudocodice derivato dalla grammatica EBNF precedente

```

parse(){
  nextToken()
  parseExp()
  checkTokenType(EOS)
}
parseExp(){
  parseMul()
  while(tokenType()==ADD){
    nextToken()
    parseMul()
  }
}
parseMul(){
  parseAtom()
  while(tokenType()==MUL){
    nextToken()
    parseAtom()
  }
}
parseAtom(){
  if(tokenType()==OPEN_PAR){
    nextToken()
    parseExp()
    checkTokenType(CLOSE_PAR)
  }
  else
    checkTokenType(NUM)
  nextToken()
}

```

#### 8.2.1.1 Grammatica EBNF per operatori con associatività a destra

Listing 8.7: Grammatica non ambigua

```

// * con precedenza maggiore, sia + che * sono associativi a
// destra
Exp ::= Mul | Mul '+' Exp
Mul ::= Atom | Atom '*' Mul
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'

```

Listing 8.8: Grammatica EBNF equivalente

```

Exp ::= Mul | ( '+' Exp )?
Mul ::= Atom ( '*' Mul )?

```

```

Atom ::= Num | '(' Exp ')'
Num  ::= '0' | '1'

```

Listing 8.9: Pseudocodice derivato dalla grammatica EBNF precedente

```

parse(){
    nextToken()
    parseExp()
    checkTokenType(EOS)
}
parseExp(){
    parseMul()
    if(tokenType()==ADD){
        nextToken()
        parseExp()
    }
}
parseMul(){
    parseAtom()
    if(tokenType()==MUL){
        nextToken()
        parseMul()
    }
}
parseAtom(){
    if(tokenType()==OPEN_PAR){
        nextToken()
        parseExp()
        checkTokenType(CLOSE_PAR)
    }
    else
        checkTokenType(NUM)
    nextToken()
}

```



## PARADIGMI DI PROGRAMMAZIONE

---

**Definizione 18** *I paradigmi di programmazione sono lo stile/approccio nell'utilizzo di un linguaggio di programmazione.*

Il più delle volte un paradigma di programmazione è basato su un modello computazionale emergente.

### ESEMPI PRINCIPALI DI PARADIGMI

- **Imperativi:** più vicini al modello hardware, quindi basati sulle nozioni di istruzione e stato, dei linguaggi di esempio sono:
  - *procedurali:* C;
  - *orientati agli oggetti:* Java;
- **dichiarativi:** basati su un modello astratto, esempi di linguaggi sono:
  - *funzionali:* ML, basati sulla nozione di *definizione di funzione* e *applicazione di funzione*;
  - *logici:* Prolog, basati sulla nozione di *regola logica* e *query*.

Di solito i linguaggi di programmazione implementano più di un paradigma per favorire la flessibilità, *Java*, *Javascript*, *Python* supportano sia i paradigmi imperativi che quelli dichiarativi.

### 9.1 PARADIGMA COMPLETAMENTE FUNZIONALE

Caratteristiche:

- **programma:** le definizioni di funzioni matematiche ed un espressione principale;
- **computazione:** l'applicazione della funzione (*function call*);
- **nessuna nozione di stato:** nessun assegnamento di variabili, più in generale nessuno *statement*, solo *espressioni*;
- **variabili:** parametri di funzioni o variabili locali contenenti valori costanti.

Terminologia:

- **higher order functions:** funzioni che possono accettare funzioni come argomenti e possono ritornare funzioni;
- **lambda expression/function o anonymous functions:** funzioni ottenute dall'evaluazione di un'espressione.

LE FUNZIONI SONO VALORI DI PRIMA CLASSE: un'evaluazione di espressioni può dare come risultato un'altra espressione (first class values).

## 9.2 OCAML

OCaml deriva da **ML**, è un linguaggio multi-paradigma con un approccio puramente funzionale, è staticamente tipato con inferenza di tipo:

- gli errori di tipo sono rilevati staticamente;
- i tipi possono essere omessi nei programmi.

Listing 9.1: Sintassi

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp | Exp
      BOP Exp | '(' Exp ')'
Pat ::= ID // pattern semplificato
```

COMMENTI:

- **ID** rappresenta gli identificatori di variabile  $[a - zA - Z\_][\backslash w']^*$ ;
- **NUM** rappresenta i numeri naturali:

```
0[bB][01][01_]*|0[oO][0-7][0-7_]*|0[xX][0-9a-fA-F][0-9a-fA-F_]*|\d[\d_]*
;
```

- **UOP** rappresenta operatori aritmetici unari  $[+-]$ ;
- **BOP** rappresenta operatori aritmetici binari  $[+ - * \backslash]|mod$ ;
- **Pat** rappresenta i patterns, per semplicità si possono associare agli identificatori.

### 9.2.1 Funzioni ed applicazioni

- esempi di funzioni anonime:

Listing 9.2: Esempio di funzioni anonime

```
fun x -> x+1 (* funzione di incremento *)
fun x y -> x+y (* funzione di addizione *)
```

- applicazione di funzioni:

Listing 9.3: Esempio di applicazione di funzioni

```
(fun x -> x+1) 3 (* il risultato sarà 4 *)
```



Inoltre prendendo in esempio:

Listing 9.4: Esempio di applicazioni

```
|| exp1 exp2
```

- l'evaluazione di **exp1** si aspetta come valore di ritorno una funzione  $f$ ;
- l'evaluazione di **exp2** si aspetta come valore di ritorno un argomento valido  $a$ ;
- l'evaluazione di **exp1 exp2** ritorna  $f(a)$  ( $f$  applicato ad  $a$ ).

### 9.2.2 Regole di precedenza ed associatività

- Si usano le regole standard per le espressioni aritmetiche;
- l'applicazione è associativa a sinistra:

Listing 9.5: Esempio di associatività

```
|| (fun x y -> x+y) 3 4 (* equivale a ((fun x y -> x+y) 3)
  4 *)
```

- l'applicazione ha precedenza maggiore rispetto gli operatori binari:

Listing 9.6: Esempio di precedenza

```
|| (fun x -> x*2) 1+2 (* equivale a ((fun x -> x*2) 1) +2
  *)
  1+((fun x -> x*2) 1+2 (* equivale a 1+((fun x -> x*2) 1)
    +2 *)
```

- le funzioni anonime hanno priorità più bassa rispetto le applicazioni e gli operatori binari:

Listing 9.7: Esempio di precedenza

```
|| fun x -> x*2 (* equivale a fun x -> x*2 *)
  fun f a -> f a (* equivale a fun f a -> (f a) *)
```

- i casi limite:

Listing 9.8: Casi limite di precedenza

```
|| f + 3 (* addizione *)
  f (+3) (* applicazione *)
  f - 3 (* sottrazione *)
  f (-3) (* applicazione *)
  + f 3 (* equivale a +(f 3) *)
  - f 3 (* equivale a -(f 3) *)
```

## 9.2.2.1 Una sessione REPL (Read Eval Print Loop)

Listing 9.9: I tipi possono essere inferiti dall'interprete

```
# 42;;
- : int = 42
# fun x->x*2;;
- : int -> int = <fun>
# (fun x->x+1) 2;;
- : int = 3
```

Listing 9.10: Grammatica BNF

```
Type ::= 'int' | Type '->' Type
```

## 9.2.2.2 Terminologia

## SINTASSI

- **int** è il tipo primitivo degli interi;
- **int -> int** è un tipo composito;
- **->** è un tipo *costruttore*, usato per costruire tipi composti da tipi più semplici;
- i tipi costruiti con il costruttore freccia (->) sono chiamati *arrow types* o *tipi di funzione*.

SIGNIFICATO DEL TIPO FRECCIA  $t_1 \rightarrow t_2$  identifica il tipo di funzioni da  $t_1$  a  $t_2$  che:

- può essere applicato ad un singolo argomento di tipo  $t_1$ ;
- ritorna sempre un valore di tipo  $t_2$ .

## Osservazioni:

- il costruttore di tipo freccia è associativo a destra.

Listing 9.11: Associatività a destra dell'operatore freccia

```
int->int->int = int->(int->int)
```

- un tipo costruttore costruisce sempre un tipo diverso rispetto a quello dei propri componenti:

$$t_1 \rightarrow t_2 \neq t_1 \quad t_1 \rightarrow t_2 \neq t_2$$

- due tipi freccia sono uguali se sono costruiti dallo stesso tipo di componenti:

$$t_1 \rightarrow t_2 = t_3 \rightarrow t_4 \text{ se e solo se } t_1 = t_3 \cap t_2 = t_4$$

- dai punti prima deduciamo:

|| `int->int->int`  $\neq$  `int->(int->int)`

### 9.2.3 Funzioni di alto ordine

**fun** *pat*<sub>1</sub>*pat*<sub>2</sub> ... *pat*<sub>*n*</sub>->*exp*

è un abbreviazione di:

**fun** *pat*<sub>1</sub>->**fun** *pat*<sub>2</sub>-> ... **fun** *pat*<sub>*n*</sub>->*exp*

### 9.2.4 Tuple