LINGUAGGI E PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

RICCARDO CEREGHINO

Appunti

Settembre 2019 – classicthesis v4.6

[10 ottobre 2019 at 11:04 – classicthesis v4.6]

1 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* durente l'esecuzione del programma, ne consegue che:

- la semantica statica non è definita;
- un utilizzo incosistenze 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

2

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:*

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

Sia:

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

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

- *u sia una funzione totale;*
- *n sia la lunghezza di u : $length(u) = n$.*

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

2.1 ESEMPIO DI STRINGHE

2.1.1 *Stringa vuota*

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

Esiste un unica funzione $u : 0 \to A$
Le notazioni standard di una stringa vuoto sono: ε, λ

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 $u : [1 \dots 4] \to A$ rappresenta la stringa *"Word"* con:

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

2.1.3 *Concatenazione di stringhe*

Definizione 4

$$length(u \cdot v) = length(u) + length(v)$$

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

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

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

2.2 LINGUAGGIO FORMALE

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

ESEMPIO: L'insieme L_{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:** $L_1 \cdot L_2 = \{u \cdot w | u \in L_1, w \in L_2\}$;
- **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:

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

Concatenazione

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

ESEMPIO:

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

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

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 neuro 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*.

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

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

Intuizione

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

3 ESPRESSIONI REGOLARI

Le espressioni regolari sono un formalismo comunamente 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;*
- λ *è un espressione regolare di A;*
- *per ogni* $\sigma \in A$, σ *è un espressione regolare in A.*

PASSO INDUTTIVO:

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

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

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

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

3.1 SEMANTICA

La semantica di un espressione regolare in A è un liguaggio su A:

- $\emptyset \rightsquigarrow$ insieme vuoto;
- $\epsilon \rightsquigarrow \{\epsilon\}$;
- $\sigma \rightsquigarrow \{"\sigma"\}, \forall \sigma \in A$;
- $e_1 \mid e_2 \rightsquigarrow$ unione delle semantiche di e_1 ed e_2;
- $e_1e_2 \rightsquigarrow$ concatenzatione 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 concatenzazione e l'unione;
- la concatenazione ha precedemza sull'unione;
- la concatenazione e l'unione sono associative a sinistra.

3.2.2 *Operatori derivati*

- $e+ = ee*$: (una o più volte e);
- ϵ: 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.

Caratteri speciali cui dare un significato ordinario

Esempi:

$$|, *, +, ?, ., , (,), [,], -, \wedge$$

SEMANTICHE:

- $\backslash. \rightsquigarrow \{"."\}$,
- $\backslash\backslash \rightsquigarrow \{"\backslash"\}$,
- $. \rightsquigarrow \{s \mid s \text{ ha lunghezza 1, l'insieme di tutti i caratteri}\}$,
- $\backslash \rightsquigarrow$ 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* pià astratta; ad un *token* corrisponde sempre un *lexeme*.

In alcuni casi un *token* può mantenere informazioni sulla sematica, 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 linguaggui 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:

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

4 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 PARSER

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 pi 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*.

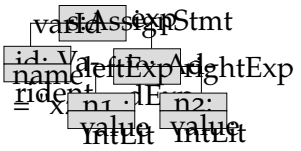


Figura 4.1: L'AST generato della sequenza precedente.

5 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 d unzione 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, α), 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

$$Exp = Num \cup (Exp \cdot \{"+"\} \cdot Exp) \cup (Exp \cdot \{"*"\} \cdot Exp) \cup (\{"("\} \cdot Exp \cdot \{")"\})$$
$$Num = \{"0"\} \cup \{"1"\}$$

DA NOTARE CHE:

- $Exp = Num \cup \ldots$ è il caso base per Exp: un numero è un espressione;
- Exp è definito su di Num, 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 Exp e $Term$ 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_{Exp} e L_{Num};
- il linguaggio per Num è relativamente semplice: $L_{\text{Num}} = \{''0'', ''1''\}$.

DA NOTARE CHE: per definire L_{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*

- $Exp \to Exp' *' Exp$ è usata la produzione $(Exp, Exp' *' Exp)$;
- $Exp' *' Exp \to Num' *' Exp$ è usata la produzione (Exp, Num);
- $Num' *' Exp \to Num' *' Num$ è usata la produzione (Exp, Num);
- $Num' *' Num \to' 0'' *' Num$ è usata la produzione $(Num,' 0')$;
- $'0'' *' Num \to' 0'' *'' 1'$ è usata la produzione $(Num,' 1')$;

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_{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 $\alpha_1 B \alpha_2 \to \alpha_1 \gamma \alpha_2$;*
- *$\alpha_1, \alpha_2 \in (T \cup N)^*$;*
- *$(B, \gamma) \in P$ ovvero (B, γ) in produzione.*

Derivazione a più passi

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

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

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: $L_B = \{u \mid B \to^+ u\}$.

5.4 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 termitale 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.

5.4.1 *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"*

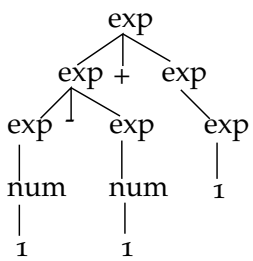


Figura 5.1: Albero di derivazione per "1*1+1""

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

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

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

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

5.4.2 *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 $I_1, \ldots, I_n$, allora $(C, I_1, \ldots, I_n) \in P$ (ovvero, $(C, I_1, \ldots, I_n)$ è una produzione di G;
- la radice è etichettata da B;
- u è ottenuto dalla concatenaziona da sinistra a destra dii tutte le etichette terminali (nodi foglia).

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.

6 GRAMMATICHE AMBIGUE

Definizione 17 *Una grammatica $G = (T, N, P)$ è ambigua per $B \in N$ se esitono 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.

6.1 SOLUZIONI PER L'AMBIGUITÀ

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

6.1.1 *Notazione prefissa*

Listing 6.1: 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.

6.1.2 *Notazione postfissa*

Listing 6.2: 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.

6.1.3 *Notazione funzionale*

Listing 6.3: Notazione funzionale

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

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

6.1.4 *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 "$1*(1+1)$".

6.1.5 *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.

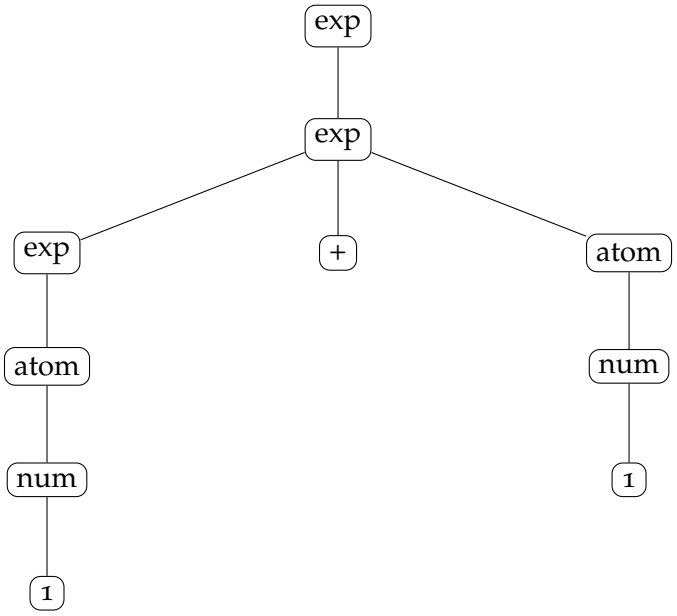
6.1.6 *Tecniche per risolvere l'ambiguità*

- 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'.

*Esempio 1: + e * con la stessa precedenza*

Listing 6.4: Grammatica ambigua

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



Listing 6.5: 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 6.6: 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.
*Esempio 2: * con la maggiore precedenza*

Listing 6.7: 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 6.8: 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.
Esempi rimanenti

- ∗ con precedenza ed associativtà a sinistra, + associatività a destra;
- ∗ con precedenza ed associativtà a destra, + associatività a sinistra;
- + con precedenza ed associativtà a sinistra, ∗ associatività a destra;
- + con precedenza ed associativtà a destra, + associatività a sinsitra;

6.2 SINTASSI AMBIGUA PER STATEMENTS

Listing 6.9: 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
```