Azzolini Riccardo 2020-11-30

CFG — Ambiguità

1 Ambiguità

Una CFG G è **ambigua** se una stringa del linguaggio $L(G)$ può essere generata da più alberi sintattici diversi.

L'ambiguità è un problema importante non solo dal punto di vista teorico, ma anche nelle applicazioni pratiche. Infatti, nella pratica le CFG vengono principalmente usate per descrivere la sintassi dei linguaggi di programmazione, attribuendo un significato ai costrutti presenti nel codice sorgente (espressioni, strutture di controllo, ecc.) in base alla struttura dell'albero sintattico. È allora fondamentale che ciascuna stringa del linguaggio corrisponda a un unico albero sintattico, in modo che essa abbia un significato univoco e prevedibile.

1.1 Esempio

Si consideri una grammatica per le espressioni formate dai numeri interi decimali senza segno (uInt, *unsigned integer*), dagli operatori + e ∗, e dalle parentesi,

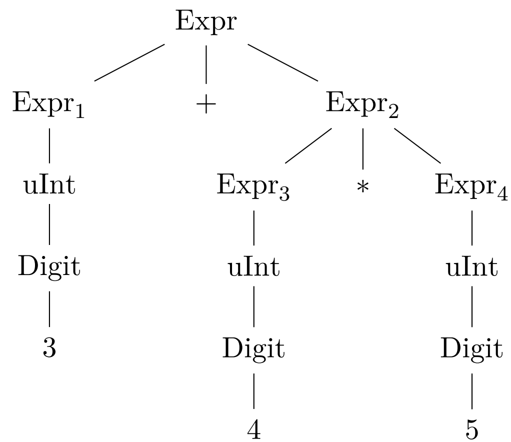
$$G = \langle \{\text{Expr}, \text{uInt}, \text{Digit}\}, \{+, *, (,), 0, 1, \dots, 9\}, \Gamma, \text{Expr} \rangle$$

in cui Γ contiene le produzioni:

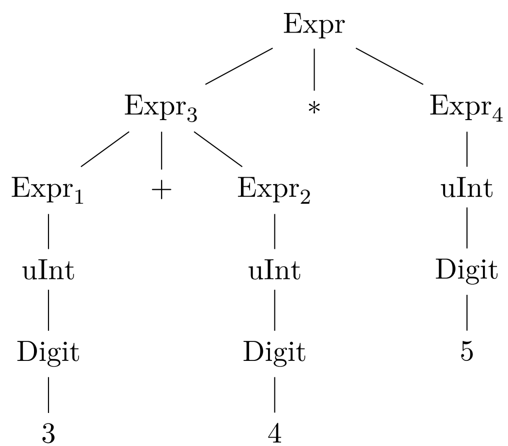
$$\begin{aligned} \text{Expr} &\to \text{Expr}_1 + \text{Expr}_2 \mid \text{Expr}_3 * \text{Expr}_4 \mid (\text{Expr}_5) \mid \text{uInt} \\ \text{uInt} &\to \text{Digit} \mid \text{uInt}\,\text{Digit} \\ \text{Digit} &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Si noti che qui i pedici sul non-terminale Expr non indicano simboli differenti, ma servono piuttosto a distinguere le diverse occorrenze del medesimo simbolo, in modo da poter riconoscere la regola dalla quale e stato generato ogni non-terminale nell'albero sintattico.

Un esempio di stringa appartenente al linguaggio di questa grammatica è $3+4*5 \in L(G)$. Essa è generata da due diversi alberi sintattici: l'albero



che corrisponde all'interpretazione della stringa come $3 + (4 * 5)$, e l'albero



che invece corrisponde all'interpretazione (3+4)∗5. Allora, per definizione, la grammatica G è ambigua. In particolare, in questo caso, il problema è che essa non esprime la precedenza di $*$ su $+$.

Così come ci sono due alberi per $3 + 4 * 5$, esistono anche due derivazioni che generano

tale stringa:

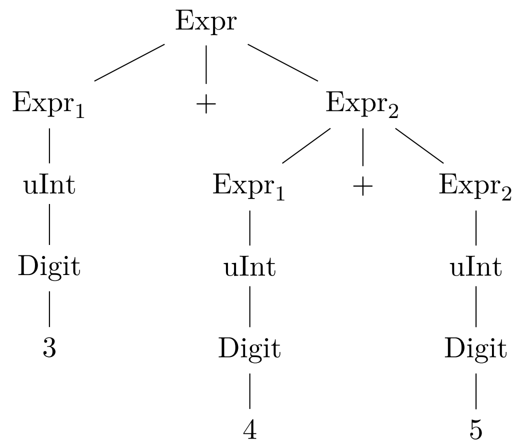
$$
\begin{aligned}
\text{Expr} &\Rightarrow \text{Expr}_1 + \text{Expr}_2 \\
&\Rightarrow \text{uInt} + \text{Expr}_2 \\
&\Rightarrow \text{Digit} + \text{Expr}_2 \\
&\Rightarrow 3 + \text{Expr}_2 \\
&\Rightarrow 3 + \text{Expr}_3 * \text{Expr}_4 \\
&\Rightarrow 3 + \text{uInt} * \text{Expr}_4 \\
&\Rightarrow 3 + \text{Digit} * \text{Expr}_4 \\
&\Rightarrow 3 + 4 * \text{Expr}_4 \\
&\Rightarrow 3 + 4 * \text{uInt} \\
&\Rightarrow 3 + 4 * \text{Digit} \\
&\Rightarrow 3 + 4 * 5
\end{aligned}
\qquad
\begin{aligned}
\text{Expr} &\Rightarrow \text{Expr}_3 * \text{Expr}_4 \\
&\Rightarrow \text{Expr}_1 + \text{Expr}_2 * \text{Expr}_4 \\
&\Rightarrow \text{uInt} + \text{Expr}_2 * \text{Expr}_4 \\
&\Rightarrow \text{Digit} + \text{Expr}_2 * \text{Expr}_4 \\
&\Rightarrow 3 + \text{Expr}_2 * \text{Expr}_4 \\
&\Rightarrow 3 + \text{uInt} * \text{Expr}_4 \\
&\Rightarrow 3 + \text{Digit} * \text{Expr}_4 \\
&\Rightarrow 3 + 4 * \text{Expr}_4 \\
&\Rightarrow 3 + 4 * \text{uInt} \\
&\Rightarrow 3 + 4 * \text{Digit} \\
&\Rightarrow 3 + 4 * 5
\end{aligned}
$$

La derivazione qui riportata a sinistra corrisponde al primo dei due alberi mostrati prima, quello per $3+(4*5)$, mentre la derivazione a destra corrisponde al secondo albero, quello per $(3+4)*5$. Si osservi che queste sono entrambe derivazioni leftmost: l'ambiguità non è dovuta alla possibilità di scegliere diverse strategie di derivazione, ma piuttosto alla scelta di quale regola applicare a ogni passo, e quindi non può essere eliminata semplicemente fissando la strategia.

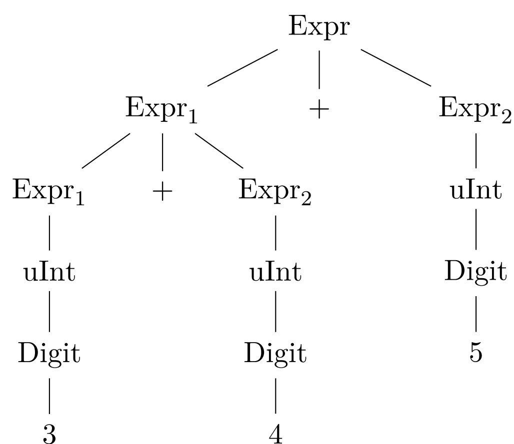
Dal punto di vista della stringa generata, entrambe le derivazioni ed entrambi gli alberi sintattici appena visti sono perfettamente validi ed equivalenti. Se però si vuole usare la grammatica G per rappresentare delle espressioni aritmetiche che verranno poi valutate, ad esempio in un linguaggio di programmazione, i due alberi danno risultati diversi,[1] perché $3+(4*5) = 23$, mentre $(3+4)*5 = 35$. Il programmatore che scrive quest'espressione nel codice potrebbe risolvere il problema aggiungendo delle parentesi esplicite, ma è opportuno che anche un'espressione senza parentesi abbia un significato prevedibile. A tale scopo, una delle due interpretazioni deve essere arbitrariamente dichiarata quella "corretta", e bisogna far sì che la grammatica produca sempre questa. Per convenzione, nel caso dell'aritmetica, si dà la precedenza al $*$ sul $+$, quindi l'interpretazione corretta di $3+4*5$ è $3+(4*5)$. Per eliminare l'ambiguità, sarà necessario di esprimere in qualche modo tale precedenza all'interno della grammatica.

Un'altra stringa generata da due alberi sintattici è $3+4+5$, che può essere interpretata come $3+(4+5)$

[1] La valutazione di un'espressione aritmetica corrisponde a una visita in postordine dell'albero sintattico: prima si valutano ricorsivamente gli operandi dell'operatore principale, e poi si valuta l'operatore stesso.



o come $(3+4)+5$



In questo caso, il risultato dell'espressione non cambia, perché l'operatore $+$ è associativo, ma è meglio eliminare comunque l'ambiguità, e per convenzione si sceglie l'associatività da sinistra a destra, cioè l'interpretazione $(3+4)+5$.

2 Eliminazione delle ambiguità

Dal punto di vista teorico, non è sempre possibile eliminare le ambiguità da una CFG. Infatti:

- *non esiste* un algoritmo generale (cioè una tecnica che funzioni sempre) per eliminare le ambiguità da una CFG;
- *non esiste* un algoritmo generale per determinare se una CFG è ambigua;

- esistono linguaggi context-free che sono *descritti esclusivamente da grammatiche ambigue*.

Esistono però delle tecniche che consentono di eliminare le ambiguità per le strutture sintattiche tipiche dei linguaggi di programmazione. Tali tecniche consistono nel trasformare la grammatica in una equivalente (che generi lo stesso linguaggio), ma appunto non ambigua, mediante l'introduzione di nuovi simboli non-terminali e nuove regole di produzione.

2.1 Esempio

Si consideri la grammatica ambigua delle espressioni vista prima:

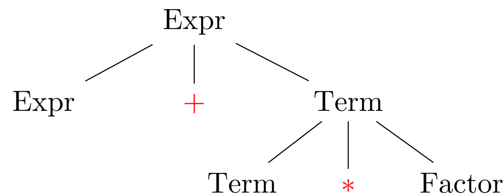
$$
\begin{aligned}
G = \langle \{\text{Expr}, \text{uInt}, \text{Digit}\}, \{+, *, (,), 0, 1, \dots, 9\}, \Gamma, \text{Expr}\rangle \\
\text{Expr} &\to \text{Expr} + \text{Expr} \mid \text{Expr} * \text{Expr} \mid (\text{Expr}) \mid \text{uInt} \\
\text{uInt} &\to \text{Digit} \mid \text{uInt}\,\text{Digit} \\
\text{Digit} &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Informalmente, le sue ambiguità possono essere eliminate modificandola in modo che, ad esempio, per generare un $*$ dopo aver introdotto un $+$, si sia obbligati a scendere di un livello nell'albero. La grammatica G' così modificata ha nuovi non-terminali (Term e Factor) e nuove regole di produzione:

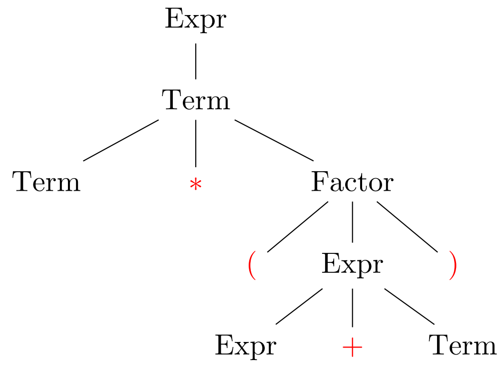
$$
\begin{aligned}
G' = \langle \{\text{Expr}, \text{Term}, \text{Factor}, \text{uInt}, \text{Digit}\}, \{+, *, (,), 0, 1, \dots, 9\}, \Gamma, \text{Expr}\rangle \\
\text{Expr} &\to \text{Term} \mid \text{Expr} + \text{Term} \\
\text{Term} &\to \text{Factor} \mid \text{Term} * \text{Factor} \\
\text{Factor} &\to \text{uInt} \mid (\text{Expr}) \\
\text{uInt} &\to \text{Digit} \mid \text{uInt}\,\text{Digit} \\
\text{Digit} &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

G' risolve le ambiguità viste prima in G, perché:

- un operando di un $+$ può contenere liberamente un $*$,

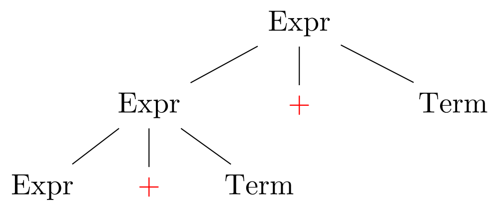


  mentre operando di un $*$ può contenere un $+$ solo se quest'ultimo è racchiuso tra parentesi esplicite,

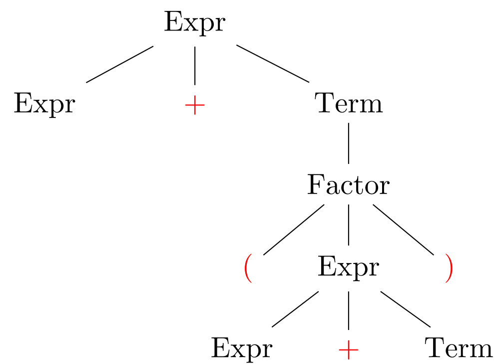


quindi il $*$ risulta avere la precedenza sul $+$;

- l'operando sinistro di un $+$ può contenere liberamente un altro $+$,

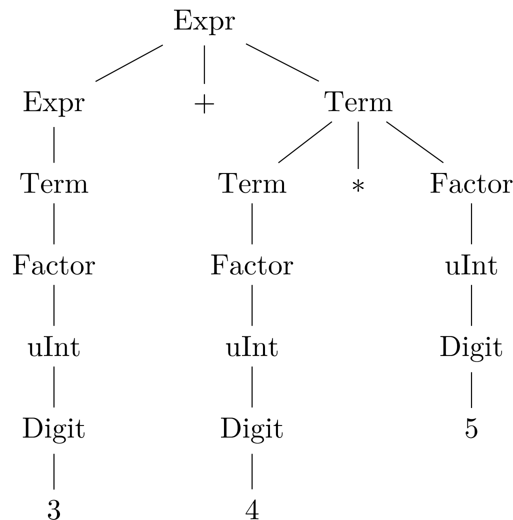


mentre l'operando destro può contenere un $+$ solo tra parentesi esplicite,



quindi il $+$ risulta associativo da sinistra a destra.

Di conseguenza, l'unico albero sintattico per $3 + 4 * 5$ in G' è



nel quale la stringa viene correttamente interpretata come $3 + (4 * 5)$, e che corrisponde alla seguente derivazione leftmost:

$$\begin{aligned}
\text{Expr} &\Rightarrow \text{Expr} + \text{Term} \\
&\Rightarrow \text{Term} + \text{Term} \\
&\Rightarrow \text{Factor} + \text{Term} \\
&\Rightarrow \text{uInt} + \text{Term} \\
&\Rightarrow \text{Digit} + \text{Term} \\
&\Rightarrow 3 + \text{Term} \\
&\Rightarrow 3 + \text{Term} * \text{Factor} \\
&\Rightarrow 3 + \text{Factor} * \text{Factor} \\
&\Rightarrow 3 + \text{uInt} * \text{Factor} \\
&\Rightarrow 3 + \text{Digit} * \text{Factor} \\
&\Rightarrow 3 + 4 * \text{Factor} \\
&\Rightarrow 3 + 4 * \text{uInt} \\
&\Rightarrow 3 + 4 * \text{Digit} \\
&\Rightarrow 3 + 4 * 5
\end{aligned}$$

Osservazione: La trasformazione di G in G' ha permesso di eliminare le ambiguità, ma ha reso la grammatica un po' più difficile da leggere. Questo effetto è in genere tanto più significativo quanto più è complessa la grammatica ambigua di partenza: per i "veri" linguaggi di programmazione (che hanno numerosi operatori, ecc.), la grammatica risultante tende a essere praticamente incomprensibile.