Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2011-2012

Linguaggi Formali e Compilatori

Grammatiche context-free

Giacomo PISCITELLI

Grammatiche libere da contesto (tipo 2)

Una **grammatica** G **libera da contesto** (non contestuale o di tipo 2) è una quadrupla (Σ, V, P, S) tale che:

- ✓ Σ (alfabeto terminale) è l'insieme dei simboli elementari del linguaggio definito dalla grammatica, alcune volte detti token
- ✓ V (alfabeto non terminale) è un insieme di metasimboli non terminali, alcune volte detti variabili sintattiche
- ✓ P è un insieme di regole o produzioni sintattiche della forma $A \rightarrow \alpha$
- ✓ $S \in V$ è l'assioma o simbolo distintivo della grammatica

Le regole sono coppie

$$A \rightarrow \alpha$$

dove $A \in V$ è detto corpo della regola sintattica e α è una stringa di simboli terminali e/o non terminali ($\alpha \in (V \cup \Sigma)^*$) detta costrutto della regola sintattica.

Se $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots . A \rightarrow \alpha_n$ sono le produzioni di G aventi la stessa parte sinistra A, esse possono essere raggruppate usando la notazione:

$A \rightarrow \alpha_1 | \alpha_2 | \ldots . | \alpha_n$ oppure $A \rightarrow \alpha_1 \cup \alpha_2 \cup \ldots . \cup \alpha_n$

che abbrevia $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots . A \rightarrow \alpha_n$

Linguaggio generato da una grammatica

Il **linguaggio generato da una grammatica partendo dal non terminale** A (notazione $L_A(G)$) è l'insieme delle stringhe di terminali prodotte da A

$$L_A(G) \equiv \{x \in \Sigma^* \mid A \rightarrow^+ x\}$$

Il **linguaggio generato da una grammatica** (notazione L(G)) è l'insieme delle stringhe di terminali prodotte dall'assioma

$$L(G) \equiv \{x \in \Sigma^* \mid S \rightarrow^+ x\}$$

$G \equiv (\{S, T\}, \{a, b\}, \{S \rightarrow aTb, T \rightarrow bSa|ab\}, S)$

- ✓ $T \rightarrow bSa \rightarrow baTba \rightarrow baabba$
- ✓ $T \rightarrow bSa \rightarrow baTba \rightarrow babSaba \rightarrow babaTbaba \rightarrow babaabbaba$
- ✓ $S \rightarrow aTb \rightarrow aabb$
- ✓ $S \rightarrow aTb \rightarrow abSab \rightarrow abaTbab \rightarrow abaabbab$

$L_T(G) \equiv \{baabba, babaabbaba, \ldots\}$

$L(G) \equiv \{aabb, abaabbab, \ldots\}$

Esempio

T = { (,), id , + , * , / , -}

V = {E}

E → E + E

E → E – E

E → E * E

E → E / E

E → - E

E → (E)

E → id

Linguaggio libero da contesto

Un linguaggio è **libero dal contesto** se esiste una grammatica libera dal contesto che lo genera.

Esempio di linguaggio context free: quello generato dalla grammatica

$$G \equiv (\{S, T\}, \{a, b\}, \{S \rightarrow aTb, T \rightarrow bSa, T \rightarrow ab\}, S)$$

Due grammatiche G e G' sono debolmente equivalenti se generano lo stesso linguaggio, cioè $L(G) \equiv L(G')$

Esempio:

$G \equiv (\{S\}, \{a, b\}, \{S \rightarrow aSb|ab\}, S)$

$G' \equiv (\{S, A, B\}, \{a, b\}, \{S \rightarrow ASB|AB, A \rightarrow a, B \rightarrow b\}, S)$

$L(G) \equiv L(G') \equiv \{ab, aabb, aaabbb, \ldots\} \equiv \{a^n b^n \mid n \geq 1\}$

Derivazioni

Derivazioni

Si dice che la stringa β diviene la stringa γ (o che dalla parola β si può derivare in un passo la parola γ) o che la stringa β produce la stringa γ per una grammatica G e si scrive

$$\beta \rightarrow \gamma$$

se esiste una produzione $(A \rightarrow \alpha) \in P$ tale che $\beta = \delta A \eta$ e $\gamma = \delta \alpha \eta$
con $\beta, \gamma \in (V \cup \Sigma)^*$

$\beta \rightarrow^n \gamma$ (β diviene γ in n passi) se esiste una catena finita di stringhe (o parole) $x_1, x_2, \ldots . x_n$ tale che $\beta = x_0 \rightarrow x_1 \ldots x_{n-1} \rightarrow x_n = \gamma$

$\beta \rightarrow^* \gamma$ (β diviene riflessivamente e transitivamente γ) se $\beta \rightarrow^n \gamma$ per qualche $n \geq 0$

$\beta \rightarrow^+ \gamma$ (β diviene transitivamente γ) se $\beta \rightarrow^n \gamma$ per qualche $n \geq 1$

$(\{S, T\}, \{a, b\}, \{S \rightarrow aTb, T \rightarrow bSa|ab\}, S)$

- ✓ $aTb \rightarrow abSab$
- ✓ $S \rightarrow aTb \rightarrow abSab$
- ✓ $S \rightarrow^2 abSab$

Ricorsione e linguaggi finiti/infiniti

Una derivazione è ricorsiva se, partendo da un non terminale, produce in almeno un passo una stringa che contiene lo stesso non terminale, cioè se è della forma

$$A \rightarrow^{+} \alpha A \beta$$

E' ricorsiva sinistra se $\alpha = \varepsilon$; ricorsiva destra se $\beta = \varepsilon$.

Il non terminale A è detto ricorsivo.

$G \equiv (\{S\}, \{a, b\}, \{S \rightarrow aSb|ab\}, S)$

$S \rightarrow aSb$

Per decidere se una grammatica ha delle ricorsioni basta esaminare la chiusura transitiva della relazione binaria fra non terminali definita da:

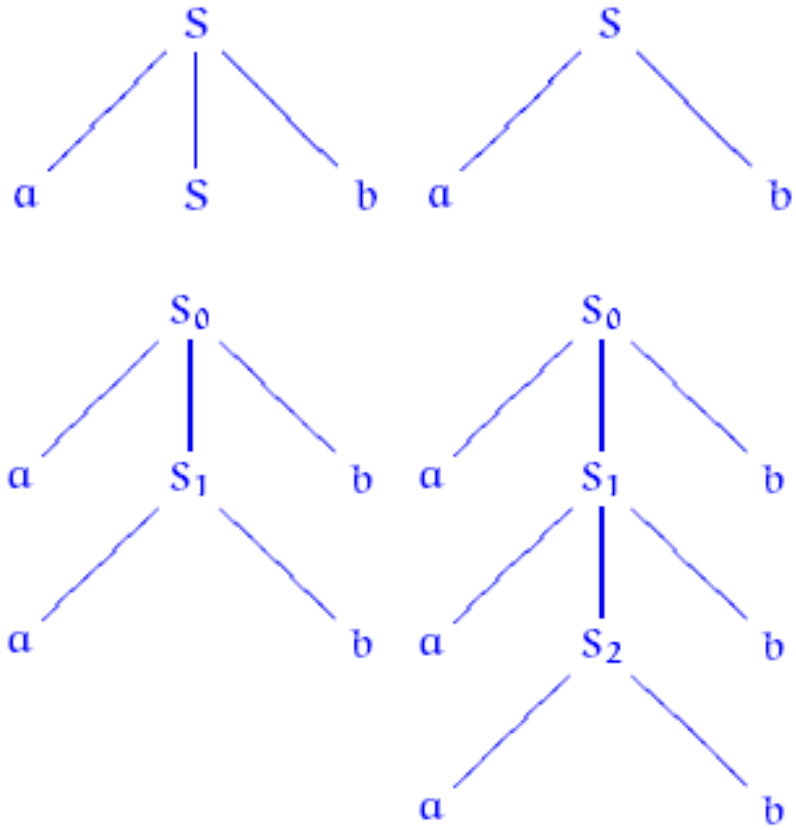
A produce B se la grammatica contiene una produzione della forma $A \rightarrow \alpha B \beta$.

La grammatica ha delle ricorsioni solo se la chiusura transitiva di questa relazione ha dei cicli.

Produzioni, derivazioni e alberi sintattici

Possiamo rappresentare produzioni e derivazioni mediante alberi:

$$G \equiv (\{S\}, \{a, b\}, \{S \rightarrow aSb|ab\}, S)$$



Parse tree

Un **parse tree** o albero sintattico di una grammatica descrive graficamente come l'assioma S della grammatica deriva una stringa nel linguaggio da essa sotteso.

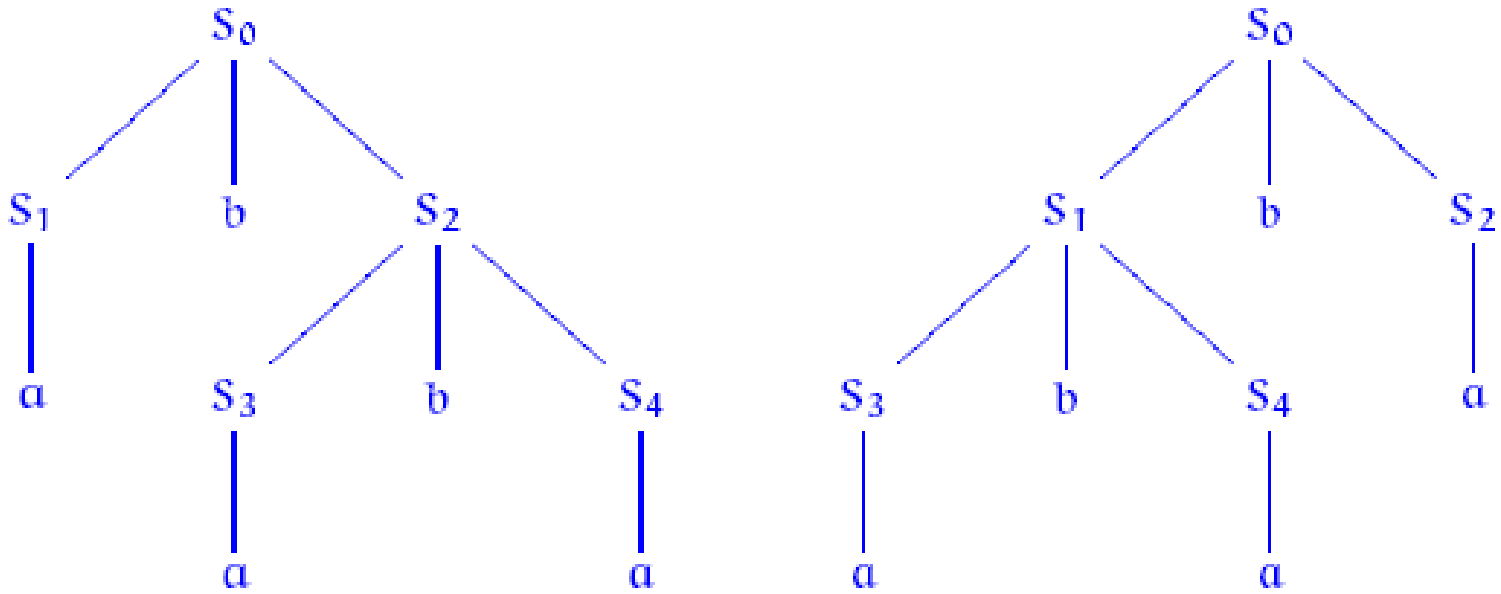
Formalmente, data una grammatica non contestuale, il suo albero sintattico avrà le seguenti caratteristiche:

- la radice dell'albero avrà come etichetta l'**assioma**;
- ogni nodo non terminale dell'albero corrisponderà ad **un elemento dell'alfabeto V**;
- ogni nodo terminale (foglia) dell'albero corrisponderà ad **un elemento dell'alfabeto Σ**;
- da ogni nodo non terminale dell'albero **derivano nodi terminali e/o non terminali**;
- come caso particolare, da un nodo non terminale può derivare **un nodo terminale costituito dal simbolo vuoto ε**.

Alberi sintattici e ambiguità

Una **frase** è **ambigua** se essa è generata dalla grammatica con due alberi sintattici distinti. In tal caso anche la **grammatica** è detta **ambigua**.

$$G \equiv (\{S\}, \{a, b\}, \{S \rightarrow SbS|a\}, S)$$



Il grado di ambiguità di una frase è il numero dei suoi alberi sintattici distinti.

Non è decidibile se una grammatica è ambigua.

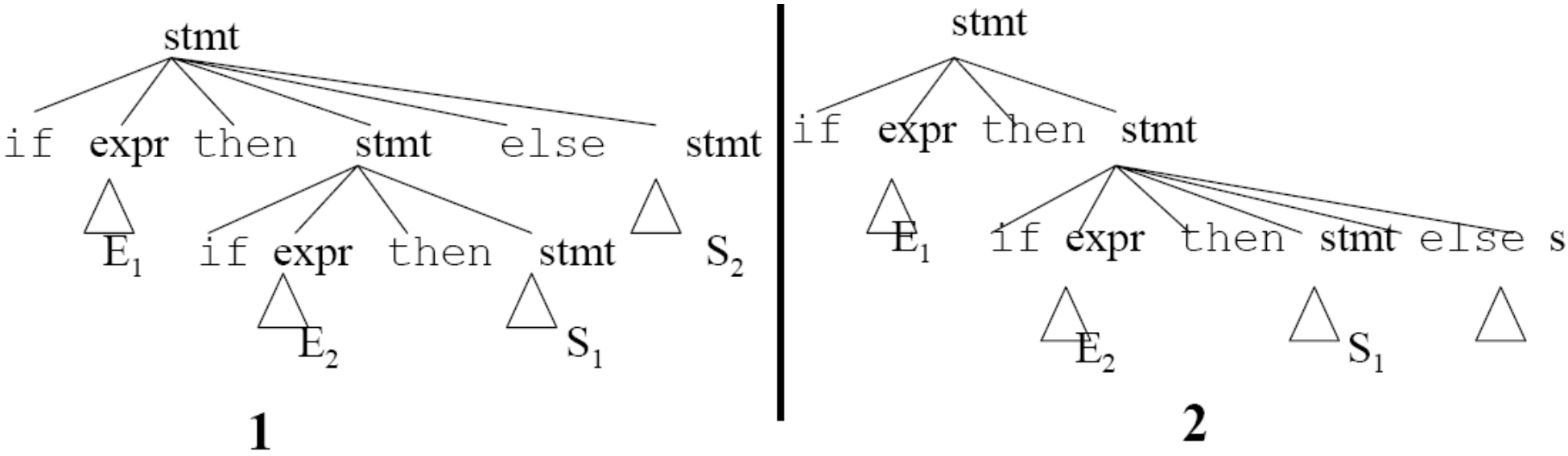
Ambiguità

Per poter costruire un parser, la grammatica non deve essere ambigua.
Le ambiguità nella grammatica devono essere eliminate durante il progetto del compilatore.

Grammatica non ambigua ➔ unica selezione nell'albero sintattico per una frase.

`stmt → if expr then stmt | if expr then stmt else stmt | otherstmts`

`if E1 then if E2 then S1 else S2`



Ambiguità

- Noi assumiamo il secondo albero sintattico (**else** corrisponde all'**if** più vicino).
- Di conseguenza dobbiamo eliminare l'ambiguità con tale obbiettivo.
- La grammatica non-ambigua sarà:

```
stmt → matchedstmt | unmatchedstmt
matchedstmt → if expr then matchedstmt else matchedstmt | otherstmts
unmatchedstmt →  if expr then stmt |
                 if expr then matchedstmt else unmatchedstmt
```

Ambiguità – Precedenza degli operatori

Grammatiche ambigue possono essere rese non-ambigue in accordo con le precedenze degli operatori e con le regole di associatività degli operatori.

L'**associatività** esprime a quali operandi si applica un operatore: nella stragrande maggioranza dei linguaggi di programmazione, i 4 operatori aritmetici (**+**, **-**, *****, **/**) associano verso sinistra, nel senso che un operando con lo stesso operatore aritmetico alla sua sinistra e alla sua destra, è associato con l'operatore a sinistra. L'operatore di esponenziazione (^) associa verso destra.

In presenza di operatori diversi, la **precedenza** regola l'ordine di applicazione degli operatori.

precedenze:

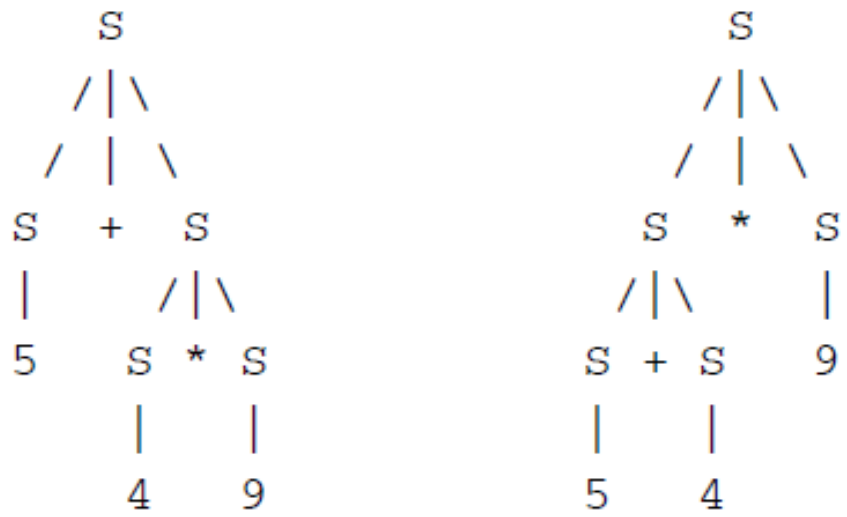
^	(right to left)
***** **/**	(left to right)
+ **-**	(left to right)

Esempio: grammatica G per espressioni aritmetiche:

$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (0 \mid 1 \mid \ldots \mid 9)^{+}$

la stringa s = 5 **+** 4 ***** 9 ammette due alberi sintattici distinti.

Ambiguità – Precedenza degli operatori



Una grammatica ambigua NON è adatta all'analisi sintattica (e traduzione automatica):

- la struttura sintattica di una frase non è unica
- l'analizzatore sintattico dovrebbe restituire tutti gli alberi sintattici corrispondenti alla stringa (oppure uno a caso)
- ambiguità sintattica → ambiguità semantica

In presenza di grammatiche ambigue non è possibile eseguire l'analisi sintattica (e la traduzione) in modo deterministico: di conseguenza, tali grammatiche sono poco utilizzate nelle applicazioni informatiche.

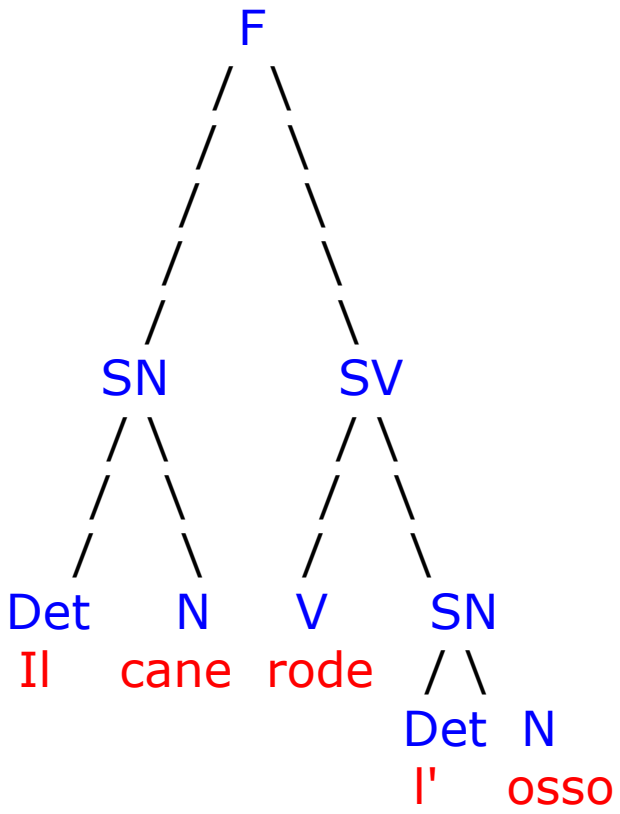
Produzioni, derivazioni e alberi sintattici nei linguaggi naturali

Il lavoro dei linguisti nell'ambito della grammatica generativa vede spesso tale albero di derivazioni come un oggetto di studio fondamentale.

Secondo questo punto di vista, una frase non è semplicemente una stringa di parole, ma piuttosto un albero con rami subordinati e sopraordinati connessi a dei nodi.

Sostanzialmente, il modello ad albero funziona in qualche modo come il seguente esempio (si è considerata una frase semplice), nel quale F è una frase (*sentence*), Det è un articolo determinativo (*determiner*), N è un sostantivo (*noun*), V è un verbo (*verb*), SN è un sintagma nominale (*noun phrase*) e SV è sintagma verbale (*verb phrase*).

Produzioni, derivazioni e alberi sintattici nei linguaggi naturali



Questo diagramma ad albero è anche chiamato un indicatore sintagmatico (*phrase marker*).

Esso può essere rappresentato più convenientemente in forma testuale, sebbene il risultato sia meno facilmente leggibile.

Produzioni, derivazioni e alberi sintattici nei linguaggi naturali

In forma testuale la suddetta frase sarebbe resa nel modo seguente:

[F [SN [Det Il] [N cane]] [SV [V rode] [SN [Det l'] [N osso]]]]

Comunque Chomsky ha affermato che anche le grammatiche a struttura sintagmatica (*phrase structure grammars*) sono inadeguate per descrivere i linguaggi naturali. Per far fronte a questa necessità, egli formulò allora il sistema più complesso della grammatica trasformazionale.

Forma normale di Chomsky

Data una grammatica, si possono costruire delle grammatiche equivalenti che abbiano particolari forme delle produzioni, dette **forme normali**.

Una grammatica è in forma normale di Chomsky se le sue regole sono caratterizzate dalle seguenti proprietà:

1. le regole sono omogenee binarie: cioè esse sono della forma

   $A \rightarrow BC$ con $B, C \in V$

   la stringa è perciò formata da due non terminali;

2. le regole sono terminali unitarie: cioè esse sono della forma

   $A \rightarrow a$ con $a \in \Sigma$

   la stringa è formata da un terminale;

3. la regola $S \rightarrow \varepsilon$ si applica solo se il linguaggio contiene ε.

La forma normale di Chomsky è caratterizzata da un albero sintattico binario.

Data una grammatica qualsiasi, si può dimostrare che esiste un semplice algoritmo per costruire una grammatica equivalente in forma normale di Chomsky.

Derivazioni Left-Most e Right-Most

Il parser può lavorare, come vedremo, in una varietà di modi – i più comuni sono il metodo ***top-down*** (nel quale l'ordine con cui il parse tree viene esplorato procede dalla radice dell'albero sintattico fino alle foglie) e quello ***bottom-up*** (nel quale l'ordine procede dalle foglie alla radice) – ma in quasi tutti esso processa l'input da sinistra a destra, un simbolo alla volta, applicando una *left-most derivation*.

Si dice, in sostanza, che le grammatiche si prestano per lo più al L-parsing.

Left-Most Derivation

Una "left-most derivation" è una derivazione nella quale durante ogni passo solo il non-terminale più a sinistra è sostituito.

E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(**id**+E) ⇒ -(**id**+**id**)

Right-Most Derivation

Una "right-most derivation" è una derivazione nella quale durante ogni passo solo il non-terminale più a destra è sostituito.

E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(E+**id**) ⇒ -(**id**+**id**)

L parser

Effettuare il riconoscimento di una stringa rispetto ad una grammatica è un'operazione intrinsecamente nondeterministica.

Questo comporta in generale tempi che possono diventare esponenziali per risolvere il problema del riconoscimento, e quindi dell'analisi sintattica.

Si comprende, perciò, che l'analizzatore sintattico deve essere molto efficiente.

In particolare:

1. tipicamente si richiede all'analizzatore sintattico di operare in tempi lineari rispetto alle dimensioni della stringa di input
2. inoltre, si richiede all'analizzatore di esplorare l'albero sintattico leggendo pochi simboli al di là dell'ultimo carattere del lessema in esame della stringa di input (tipicamente un solo simbolo, detto simbolo di lookahead)

A questo scopo, è necessario che la grammatica (in particolare, le regole di produzione) abbia delle caratteristiche che la rendono adatta all'analisi sintattica.

L parser

Le frasi

- possono essere analizzate da sinistra a destra (L parser),
- possono essere costruite con derivazioni left-most (LL(k) parser) o right-most (LR(k) parser) utilizzando k symboli di lookahead.

LL è più comunemente impiegata nei top-down parser.

LR è più comunemente impiegata nei bottom-up parser.

Per ragioni pratiche k deve essere piccolo.

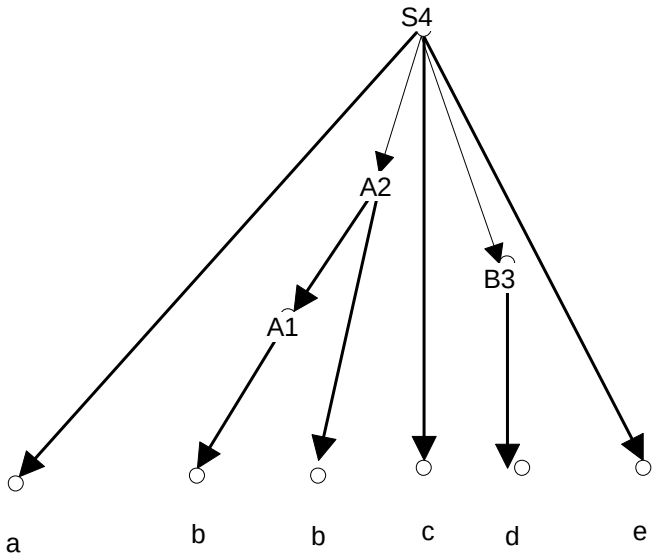
L parser

Per un compilatore è auspicabile l'uso di grammatiche che possano essere analizzate in modo deterministico con al più k symboli di lookahead.
L'assenza di ambiguità è condizione necessaria per un'analisi deterministica.

Consideriamo ad esempio un bottom up parser per **abbcde** generato dalla seguente grammatica con assioma S, eseguendo un approccio "left-most matches First".

S → aAcBe
A → Ab|b
B → d

aAbcde	applicando **A → b**
aAcde	applicando **A → Ab**
aAcBe	applicando **B → d**
S	applicando **S → aAcBe**



Ricorsione sinistra (Left Recursion)

Una grammatica è "***left recursive***" se ha un non terminale A tale che

$A \Rightarrow A\alpha$ per qualche stringa α

Le tecniche di parsing Top-down non possono gestire grammatiche left-recursive, perché ne potrebbe derivare un *loop* infinito.

Una grammatica left-recursive deve essere convertita in una non left-recursive.

La ricorsione sinistra può comparire in un singolo passo della derivazione (*immediate left-recursion*), o può comparire in più che un passo.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ dove $\beta \neq A\gamma$ e $\gamma \in (V \cup T)^*$

$\Downarrow$

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ grammatica equivalente

In generale

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ dove $\beta_i \neq A\gamma$ e $\gamma \in (V \cup T)^*$

$\Downarrow$

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ grammatica equivalente

Algoritmo generale per l'eliminazione della ricorsione sinistra

- Ordinare i non-terminali $A_1 \ldots A_n$
- for i from 1 to n do {
- for j from 1 to i-1 do {
  sostituire ogni produzione
  $A_i \rightarrow A_j \gamma$
  con
  $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$
  dove $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$
  }
- eliminare la ricorsione sinistra nelle produzioni di A_i
  }

Algoritmo generale per l'eliminazione della ricorsione sinistra

Esempio

S → Aa | b
A → Ac | Sd | f

- Ordiniamo i non-terminali: S, A

per S: non c'e una ricorsione sinistra diretta.
per A: sostituiamo A → Sd con A → Aad | bd
così avremo A → Ac | Aad | bd | f

- Eliminiamo la ricorsione sinistra in A

A → bdA' | fA'
A' → cA' | adA' | ε

- Avremo la grammatica non ricorsiva equivalente:

S → Aa | b
A → bdA' | fA'
A' → cA' | adA' | ε

Algoritmo per l'eliminazione della ricorsione sinistra

Altro esempio

S → Aa | b
A → Ac | Sd | f

- Ordiniamo i non-terminali: A, S

per A: eliminamo la ricorsione sinistra in A

A → SdA' | fA'
A' → cA' | ε

per S: sostituiamo S → Aa con S → SdA'a | fA'a
così avremo S → SdA'a | fA'a | b

- Eliminiamo la ricorsione sinistra in S

S → fA'aS' | bS'
S' → dA'aS' | ε

- Avremo la grammatica non ricorsiva equivalente:

S → fA'aS' | bS'
S' → dA'aS' | ε
A → SdA' | fA'
A' → cA' | ε

Left-Factoring (fattorizzazione sinistra)

Un parser top down o predittivo richiede una grammatica *left-factored*, una grammatica, cioè, che consenta, quando la scelta tra due produzioni alternative non sia chiara, di posticipare tale decisione a quando sia stata acquisita una parte dell'input sufficiente per effettuare la giusta scelta.

La fattorizzazione richiede dunque la trasformazione della grammatica in una ➔ grammatica equivalente.

Per esempio, se abbiamo le seguenti due produzioni:

```
istr   →      if (expr ) istr else istr |
              if (expr) istr
```

non possiamo dire, riconoscendo il primo **`if`**, quale produzione verrà impiegata per derivare `istr`.

In generale, se:

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$

dove $\alpha \in (V \cup T)^* - \{\varepsilon\}$, $\beta_1 \in (V \cup T)^* - \{\varepsilon\}$, $\beta_2 \in (V \cup T)^* - \{\varepsilon\}$ e $\beta_1 \neq \beta_2$

possiamo riscrivere la grammatica come segue

$A \rightarrow \alpha A'$

$A' \rightarrow \beta 1 \mid \beta 2$

Algoritmo di Left-Factoring

Per ogni non-terminale A con due o più alternative (produzioni) con una parte non vuota comune

$$A \rightarrow \alpha\beta_1 | \ldots | \alpha\beta_n | \gamma_1 | \ldots | \gamma_m$$

diventa

$$A \rightarrow \alpha A' | \gamma_1 | \ldots | \gamma_m$$
$$A' \rightarrow \beta_1 | \ldots | \beta_n$$

Esempio

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$
$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$
$A' \rightarrow bB \mid B$
$A'' \rightarrow g \mid eB \mid fB$

Algoritmo di Left-Factoring

Altro esempio

A → ad | a | ab | abc | b

⇓

A → aA′ | b
A′ → d | ε | b | bc

⇓

A → aA′ | b
A′ → d | ε | bA″
A″ →ε | c

```
istr   →      if (expr ) istr else istr |
              if (expr) istr
istr   →      if (expr ) istr X
X      →      else istr
X      →      ε
```

Sintassi

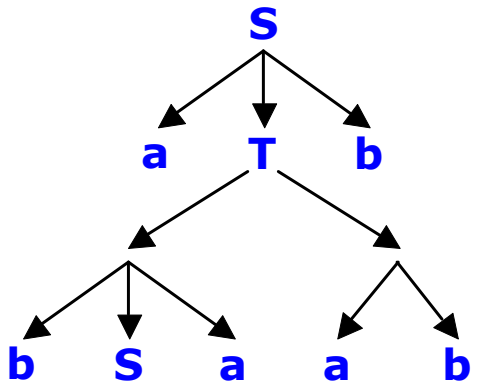
In conclusione

La sintassi è costituita da un insieme di regole che definiscono le frasi formalmente corrette e allo stesso tempo permettono di assegnare ad esse una struttura che rappresenta graficamente il processo di derivazione (albero sintattico).

Considerata la seguente grammatica:

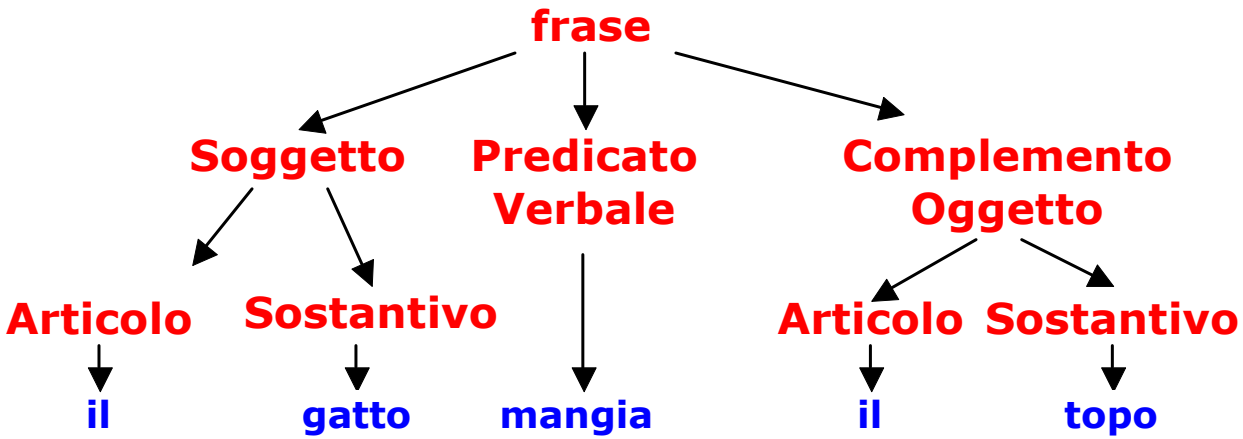
$$G \equiv (\{S, T\}, \{a, b\}, \{S \rightarrow aTb, T \rightarrow bSa, T \rightarrow ab\}, S)$$

l'insieme delle sue regole può essere rappresentato graficamente tramite il seguente albero:



Sintassi context-free

Se, analogamente, si vuole rappresentare graficamente la struttura di una **frase** del linguaggio italiano, che ha come costituenti un **Soggetto**, un **Predicato Verbale** e un **Complemento Oggetto**, ecc. , si potrà ricorrere al seguente albero sintattico:



Sintassi context-free

Se, ancora, si vuole rappresentare la struttura di un programma in **un linguaggio di programmazione**, si dovrà costruire, come in seguito esamineremo, un albero sintattico dal quale si evinca che un programma ha come costituenti **le parti dichiarative e quelle esecutive**.

Le parti dichiarative definiscono i dati usati dal programma. Le parti esecutive si articolano nelle istruzioni, che possono essere di vari tipi. I costituenti del livello più basso sono gli elementi lessicali già considerati, che dalla sintassi sono visti come atomi indecomponibili. Infatti la loro definizione spetta al livello lessicale.