

L'ANALIZZATORE SINTATTICO

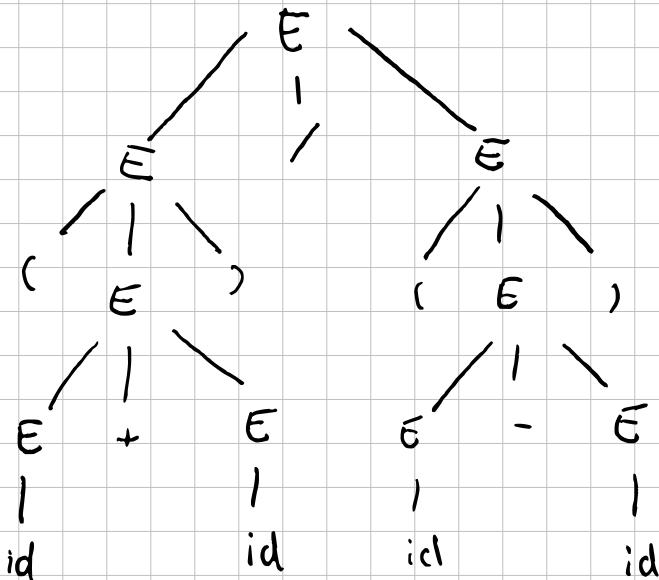
è uno delle parti fondamentali dell'analisi di codice fatta dal compilatore, e viene spesso fatta da un'ALBERO di DERIVAZIONI che viene costruito sulla base di un grammatica e di una string.

ES.

$$\begin{array}{lll} E \rightarrow E+E & E \rightarrow E-E & E \rightarrow E * E \\ E \rightarrow E/E & E \rightarrow (E) & E \rightarrow \text{id} \end{array}$$

i lessimi a e b saranno quindi sostituiti con id

data la stringa $(a+b) / (a-b)$ si può costruire il seguente albero

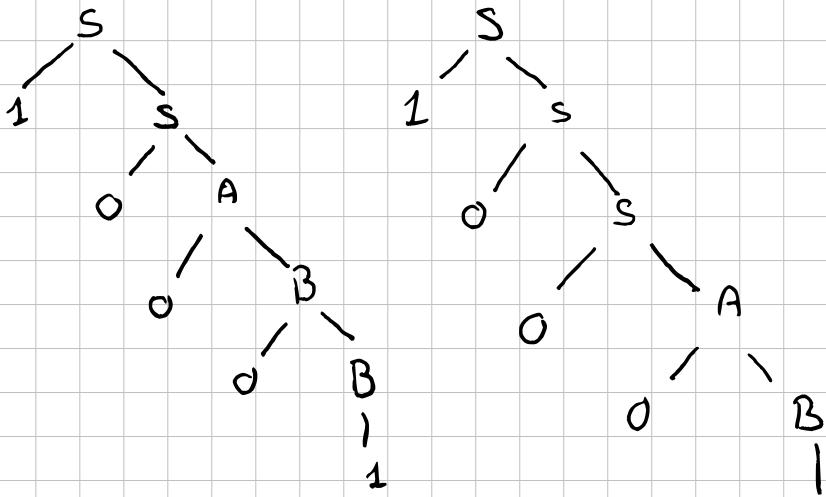


def. data una grammatica generativa $G = (V, N, S, P)$
 un albero di derivazione è un albero la cui
 radice è S (simbolo iniziale della grammatica), e
 dato ogni nodo con non terminale A , i figli del nodo
 siano i simboli della parte destra della produzione P ,
 la cui sinistra è A .

Grammatiche AMBIGUE

data la grammatica $S \rightarrow 1S \quad A \rightarrow 0$ e la stringa
 $S \rightarrow OS \quad B \rightarrow 0B$
 $S \rightarrow OA \quad B \rightarrow 1B$ 10001
 $S \rightarrow OB \quad B \rightarrow 0$
 $B \rightarrow 1$

S. possono costruire più alberi di derivazione



nel caso d. un analizzatore binario non
 è un problema, ma le ambiguità possono essere più gravi
 (es precedenza somma/prodotto)

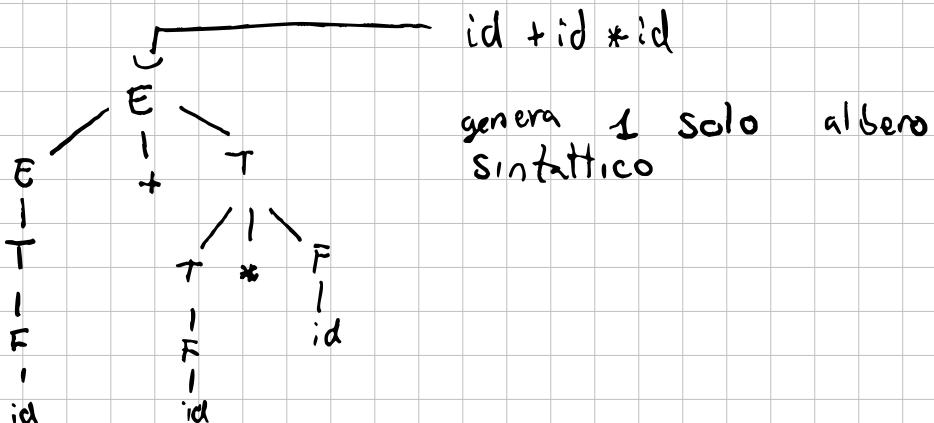
Def: Una grammatica nella quale è possibile analizzare una sequenza in 2 o più modi diversi - generando quindi più alberi sintattici - è detta **AMBIGUA**

Un modo per risolvere il problema dell'ambiguità farebbe implementare un albero sintattico con priorità (ad esempio, facendo analizzare prima la moltiplicazione della somma), oppure è possibile **riscrivere la grammatica per eliminare ambiguità**

es.

$E \rightarrow E + E$ $E \rightarrow E * E$ $E \rightarrow id$	$E \rightarrow E + T$ $T \rightarrow T * F$ $T \rightarrow T / F$ $F \rightarrow (E)$ $F \rightarrow id$
a	a

La nuova grammatica esplicita il fatto che, se si vuole usare una somma o differenza, è necessario prima chiuderle tra parentesi, r



Derivazioni destre e sinistre

data la grammatica delle espressioni aritmetiche all'inizio specificata (quella ambigua) è possibile analizzare la stringa $(id+id)/(id-id)$ in 2 mod:

Derivazione sinistra

$$\begin{aligned} E &\rightarrow E / E \\ E &\rightarrow (E) / E \\ E &\rightarrow (E+E) / E \\ E &\rightarrow (id+E) / E \\ E &\rightarrow (id+id) / E \\ E &\rightarrow (id+id) / (E) \\ &\vdots \\ &\vdots \\ E &\rightarrow (id+id) / (id-id) \end{aligned}$$

Derivazione destra

$$\begin{aligned} E &\rightarrow E / E \\ E &\rightarrow E / (E) \\ E &\rightarrow E / (E-E) \\ E &\rightarrow E / (E-id) \\ E &\rightarrow E / (id-id) \\ E &\rightarrow (E) / (id-id) \\ &\vdots \\ &\vdots \\ E &\rightarrow (id+id) / (id-id) \end{aligned}$$

Nonostante ci fossero due "strade" di derivazione, il risultato porta allo stesso albero di derivazione.

→ ogni forma sentenziale che compare in una derivazione sinistra è detta forma sentenziale sinistra, e viceversa (forma sentenziale destra)

L) Ci si limita ad analizzare i parser che generano derivazioni destre, detti parser **TOP-DOWN**

PARSER Top-Down

Un parser top-down parte dalla radice dell'albero di derivazione e cerca di ricostruire la crescita dell'albero fino a sequenza di simboli terminali, utilizzando derivazione sinistra.

I problemi d. una simile derivazione sono quelli riguardante la ricorsione, che immaneabilmente andrà a generarsi. per esempio, utilizzando la grammatica non ambigua

$$\begin{array}{l} E \rightarrow E + T \\ E \rightarrow T \\ T \rightarrow :id \end{array}$$

e la stringa
id + id + id

Un parser inizierà espandendo E con produzione $E \rightarrow E + T$, quindi di nuovo espanderà E allo stesso modo all'infinito

Come può un analizzatore scegliere la produzione corretta quando il primo carattere di produzione porta a ricorsione?

Non c'è soluzione per grammatiche d. tipo

$$A \rightarrow A\alpha$$

IMMEDIATE

$$A \rightarrow B\alpha \quad B \rightarrow A\beta$$

NON IMMEDIATE

C'è necessità di modificare la grammatica per evitare situazioni di questo tipo

Eliminazione ricorsioni sinistre

1 Rimuovere le ricorsioni sinistre immediate, si procede nel modo seguente (si assume che la grammatica originale non contenga λ -produzioni)

Per ogni NON TERMINALE A nella grammatica che presenta almeno una produzione ricorsiva sinistra immediata, si eseguono le seguenti operazioni:

- Si separano le produzioni ricorsive immediate, si supponga

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots$$

$$A \rightarrow S_1 | S_2 | \dots$$

- Si introduce un nuovo non terminale A'

- Si sostituisce ogn produzione non ricorsiva $A \rightarrow \delta$, con la produzione $A \rightarrow S_i A'$

- Si sostituisco ogn produzione immediata ricorsiva $A \rightarrow A\alpha_i$ con la produzione $A' \rightarrow \alpha_i A'$

- Si aggiunge la produzione $A' \rightarrow \lambda$

Lo scopo dell'eliminazione delle ricorsioni sinistre è avere un simbolo terminale all'inizio, così da poter immediatamente "chiudere" il ramo sinistro d. un certo nodo N.T.

ES.

$$S \rightarrow Sa$$

$$S \rightarrow b$$

esso produce stringhe della forma ba^n .
Si trasforma la grammatica nel modo seguente

$$S \rightarrow bS' \quad S' \rightarrow aS' \quad S' \rightarrow \lambda$$

Così facendo si avrà

$$S \rightarrow bS' \rightarrow bas' \rightarrow baas' \rightarrow \dots \rightarrow ba^n$$

L'eliminazione di ricorsioni non immediate è più
ardua, si rimanda alle dispense LINGUAGGI - 2020-21.pdf
pagina 44-45

Backtracking

Un modo per sviluppare parser top-down è basato sul provare tutte le strade, quindi tutte le produzioni, fino a trovarne una che produce la stringa in input

Usando la forza bruta, però porta ad alcuni problemi:

ad esempio, considerato la grammatica

$$\begin{aligned} S &\rightarrow ee \quad S \rightarrow bAc \quad S \rightarrow bAe \quad A \rightarrow d \quad A \rightarrow cA \\ &\text{e la stringa } bcd. \end{aligned}$$

Se il parser tentasse tutte le produzioni esaurivamente, incomincerai con la produzione $S \rightarrow bAc$, per poi scendere A , ma ciò porta un albero sintattico
ERRATO. Bisogna quindi tornare indietro (fare appunto, "backtrack")
fino a trovare una produzione alternativa

il backtracking corrisponde ad un' visita in profondità di un grafo, in cui si va avanti fino ad aver trovato o la soluzione o un vicolo cieco, e nel secondo caso deve tornare indietro.

Questo ha costo elevato infatti è necessario:

- distruggere l'albero creato
- ricostruire l'input della stringa di simboli terminale prima della strada sbagliata,
- "ricreare" simboli consumati

Questo può essere risolto **modificando la grammatica** per evitare di avere 2 simboli terminali data una produzione con stesso simbolo a sinistra.

S. può modificare la grammatica precedente, in modo da evitare il problema

$$S \rightarrow ee \quad S \rightarrow bAQ \quad Q \rightarrow cle \quad A \rightarrow d \quad A \rightarrow cA$$

fattorizzando il prefisso comune bA ed usando un nuovo non terminale per permettere in secondo momento la scelta tra c ed e .

Ora il parser può generare l'albero di derivazione senza backtracking, daje. **METODO DELL' FATTORIZZAZIONE SINISTRA**.

Un altro metodo consiste nell'utilizzo del PARSER PREDITTIVI

PARSER PREDITTIVI

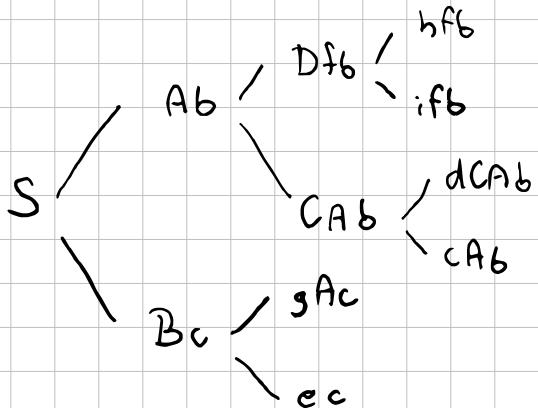
Se nella grammatica al più una produzione inizia con un simbolo non terminale, allora sarebbe semplice utilizzare la strategia del parser. Ma se si avesse una grammatica con simboli terminali:

Il backtrace potrebbe essere evitato se il parser potesse "predire", ovvero guardare avanti nella grammatica in modo da anticipare quali simbol. sono derivabili: da ciascuna delle parti destre delle produzioni.

Per esempio, si guardi la grammatica

$$\begin{array}{lll} S \rightarrow A b & A \rightarrow C A & C \rightarrow d C \\ S \rightarrow B c & B \rightarrow g A & C \rightarrow c \\ A \rightarrow D f & B \rightarrow e & D \rightarrow h \\ & D \rightarrow i & \end{array}$$

S. vogliono seguire tutte le derivazioni sinistre derivate da S, fino ad arrivare un simbolo terminale, si ha



se dobbiamo iniziare un simbolo h, i, d, c
dobbiamo usare la produzione $S \rightarrow A b$, mentre per g, e $S \rightarrow B c$

Se inizia con qualcosa di diverso, restituisci errore

DEFINIZIONE FIRST()

Data una grammatica $G = (V, N, SP)$
 v-symbol: terminali, N symbol: non terminali, definiamo la funzione $FIRST : (VN)^+ \rightarrow 2^V$:

Sia α è una sequenza di simboli terminali e non e se X è l'insieme di tutte le forme sentenziali derivabili da α mediante derivazioni sinistre allora per ogni $y \in X$ che inizia con un terminale x , x appartiene a $FIRST(\alpha)$

Proprietà:

1. Se α inizia con terminale x , allora $FIRST(\alpha) = x$
2. $first(\lambda) = \lambda$
3. Se α inizia con N.T B allora $FIRST(\alpha)$ include $FIRST(B) - \{\lambda\}$

ES. riprendo

Se ho $A \rightarrow Df \cup A \rightarrow Ca$ ottengo

$$\begin{aligned} first(Ab) &= FIRST(Dfb) \cup FIRST(Cab) \\ &= FIRST(hfb) \cup FIRST(ifb) \cup FIRST(dcab) \cup \\ &\quad \cup FIRST(cab) = \{h, i, d, c\} \end{aligned}$$

$$\begin{aligned} first(Bc) &= FIRST(gac) \cup FIRST(ec) \\ &= \{g, e\} \end{aligned}$$

$$\begin{aligned} first(s) &= first(ab) \cup first(bc) = \dots \\ &= \{h, i, d, c, g, e\} \end{aligned}$$

DEFINIZIONE FOLLOW(α)

Quando si ha a che fare con un non terminale, che tra le sue produzioni destre hanno una λ , il processo si blocca e ci è difficile sapere come riconoscere il prossimo terminale.

Se A è un non terminale, allora $V \times G V$ che può seguire A in una forma sentenziale C , x appartiene a $\text{FOLLOW}(A)$. Per ogni non terminale A , si calcola $\text{FOLLOW}(n)$ nel seguente modo:

1. Se A è il simbolo iniziale allora $\$ \in \text{Follow}(A)$

2. Si cercano nella grammatica le occorrenze di A nelle parti destre. Sia $Q \rightarrow^* A \gamma$ una di queste posizioni.

- Se γ inizia con terminale q , allora $q \in \text{Follow}(A)$
- Se γ non inizia con terminale, allora $\{\text{FIRST}(\gamma) - \lambda\} \in \text{Follow}(A)$
- Se $\gamma = \lambda$, ovvero A è in fondo, o γ è annullabile, allora includiamo $\text{Follow}(Q)$ in $\text{Follow}(A)$

GRAMMATICHE LL(1)

In un parser predittivo, l'insieme FOLLOW risulta utile, in quanto ci dice quando applicare le λ -produzioni. Si supponga di dover espandere un non terminale A.

Inizialmente si vede se il simbolo terminale appartiene al $\text{First}(A)$. Se non appartiene è in generale errore, ma non se $\text{FIRST}(A) = \lambda$. In quel caso si vede se il simbolo terminale è nel $\text{follow}(A)$. Se lo è bisogna applicare la produzione $A \rightarrow \lambda$.

Le grammatiche LL(1) utilizzano questa tecnica. (Left-to-right left 1-char) Le grammatiche LL(1) assicurano che attraverso i soli CARATTERI la scelta sia univoca.

Esempio grammatica (LL(1))

- espressione \rightarrow digit | (espressione operatore espressione)
- operatore \rightarrow '+' | '-'
- digit \rightarrow '0' | '1' | ... | '8' | '9'

FIRST(digit) = digit

FIRST((espressione operatore espressione)) = (

|
↓
utilizzo le regole FIRST per trovare produzione
da applicare (cs.)

((3+5)* 2)
|
↓
'(' applico (expr digit expr)

Esempio grammatica non LL(1)

NON LL(1) perché
leggendo il carattere
'a' non SAPPiamo
se applicare $S \rightarrow A_c$
o $S \rightarrow B_d$

- $S \rightarrow A_0 \mid B_0$
 - $A \rightarrow a$
 - $B \rightarrow a$

α	$\text{first}(\alpha)$
a	{a}
b	{b}
d	{d}
A	{a}
B	{a}
S	{a}
Ac	{a}
Bd	{a}

• Se la Grammatica è LL(1) allora possiamo costruire un parser predittivo ESAMINANDO UN UNICO SIMBOLÒ NON TERMINALE

Es $E \rightarrow (\varepsilon) \text{ lid}$

E	(E → (E))	D E → id	\$
---	---------------	-------------	----

1d NON TERMINAL

r. prendo l'esempio di prima

	(+	*	0	1	2	...
Esp	(ESP open)			id	id	id	
Open			#				
id				0	1	2	

Una volta costruita la tabella l'algoritmo è banale, la parte di costruzione tabella è difficile.

