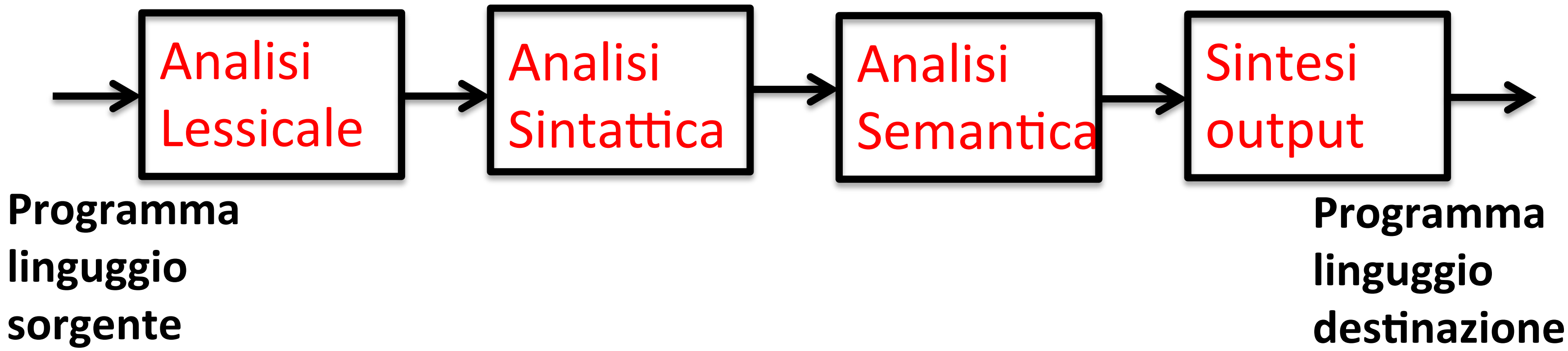
Analisi Sintattica
e
Grammatiche di tipo 2

Fabrizio d'Amore
Alberto Marchetti Spaccamela

Fondamenti di Informatica II

Struttura di un traduttore



- Le fasi di analisi lessicale e analisi sintattica dipendono **soltanto** dalla sintassi del linguaggio sorgente
- La fase di analisi semantica dipende sia dalla sintassi che dalla semantica del linguaggio sorgente
- La fase di sintesi dell'output dipende sia da sintassi e semantica del linguaggio sorgente che da sintassi e semantica del linguaggio destinazione
- Dopo la sintesi può seguire eventuale ottimizzazione

Analisi sintattica

Programma P (sequenza di token) → Analizzatore Sintattico → Albero di derivazione

modulo di analisi sintattica = analizzatore sintattico

Data una sequenza s di token, l'analizzatore sintattico verifica se la sequenza appartiene al linguaggio generato dalla grammatica G

A questo scopo, l'analizzatore sintattico cerca di costruire l'albero sintattico di P (che rappresenta le produzioni usate per verificare che P sia corretto)

– in caso positivo, restituisce in uscita l'albero sintattico per la sequenza di input nella grammatica G

– in caso negativo, restituisce un errore (errore sintattico)

Analisi sintattica

Programma P (sequenza di token) → Analizzatore Sintattico → Albero di derivazione

– la sintassi e` in genere specificata in termini di un linguaggio non contestuale (tipo 2)

– il modulo di analisi sintattica (analizzatore sintattico) viene realizzato sulla base di tale specifica

– in genere l'analizzatore sintattico è il risultato di un programma: si utilizzano dei programmi chiamati **generatori di analizzatori sintattici** che, a partire dalla specifica della sintassi, generano automaticamente l'analizzatore sintattico corrispondente

Analisi sintattica

L'albero sintattico ottenuto viene usato per le fasi successive della traduzione

La traduzione (sintesi dell'output) può essere
guidata dalla sintassi, cioè si può decomporre il processo di traduzione delle frasi del linguaggio sorgente sulla base della struttura sintattica di tali frasi

– a tal fine, è possibile estendere i formalismi di specifica della sintassi al fine di catturare alcuni aspetti "semantici" collegati alle produzioni della grammatica

– tramite le azioni semantiche, è possibile specificare la fase di traduzione in parallelo alla specifica della sintassi del linguaggio sorgente

Perché grammatiche tipo 2 per i linguaggi di programmazione?

Quali grammatiche per il linguaggi di programmazione? Due esigenze contrapposte

- Efficienza nella traduzione → linguaggi semplici
- Grammatiche sufficentemente espressive che siano in grado di descrivere linguaggi di programmazione

Linguaggi tpo 3

- Il riconoscimento per linguaggi di tipo 3 è efficiente
- Le grammatiche di tipo 3 non sono adatte
  - Abbiamo visto che il linguaggio $a^n b^n$ non è regolare
  - Quindi anche sapere se le parentesi in un'espressione aritmetica sono bilanciate ((((…))))
  - Altro esempio: le parentesi { } in Java sono bilanciate bene

Perché grammatiche tipo 2 per i linguaggi di programmazione ?

- Le grammatiche di tipo 3 non sono adatte a descrivere linguaggi di programmazione
- Le grammatiche di tipo 2 sono adatte a descrivere la sintassi dei linguaggi di programmazione (in particolare permettono di rappresentare la struttura gerarchica dei programmi)
- Sono più semplici delle grammatiche di tipo 1 e questo permette di avere parser efficienti
- Vedremo in realtà che per avere avere analizzatori sintattici (parser) efficienti dobbiamo introdurre ulteriori restrizioni sulle produzioni della grammatica

I linguaggi non contestuali (tipo 2)

I linguaggi non contestuali (in inglese context free):

- sono generati da grammatiche di tipo 2
- sono riconosciuti da automi a stati finiti non deterministici con l'ausilio di una pila (automi a pila)
- contengono i linguaggi di programmazione
- In realtà i linguaggi di programmazione appartengono a classi di linguaggi context free molto particolari

I linguaggi non contestuali (tipo 2)

I linguaggi non contestuali (in inglese context free):

- sono generati da grammatiche di tipo 2

Grammatiche di tipo 2

- tutte le produzioni sono del tipo $A \rightarrow \alpha$

  dove A è un simbolo non terminale

  α è una stringa di simboli terminali e non terminali

Ricorda grammatiche tipo 3: produzioni del tipo

$A \rightarrow a$ o del tipo $A \rightarrow aB$

- La parte sinistra di una produzione la stessa per tipo 2 e 3
- La parte destra in tipo 2 può essere qualunque stringa

I linguaggi non contestuali (tipo 2)

Molte frasi in linguaggio naturale hanno una struttura sintattica non contestuale

Esempio

In Italiano possiamo affermare che una frase è costituita da

- un soggetto seguito da un complemento
- un soggetto è un articolo seguito da un sostantivo
- il complemento è un verbo o un verbo seguito da un complemento oggetto.

I linguaggi non contestuali (tipo 2)

Esempio (continua)

Usando caratteri maiuscolo per i concetti sintattici (simboli non terminali), minuscolo per le parole (simboli terminali) possiamo scrivere

- FRASE →SOGGETTO COMPLEMENTO
- SOGGETTO → ART SOST
- COMPLEMENTO → VERBO | COMP-OGGETTO
- COMP-OGGETTO → ART SOST
- ART → il
- SOST → gatto | topo
- VERBO → mangia

Una possibile frase è “il gatto mangia il topo”

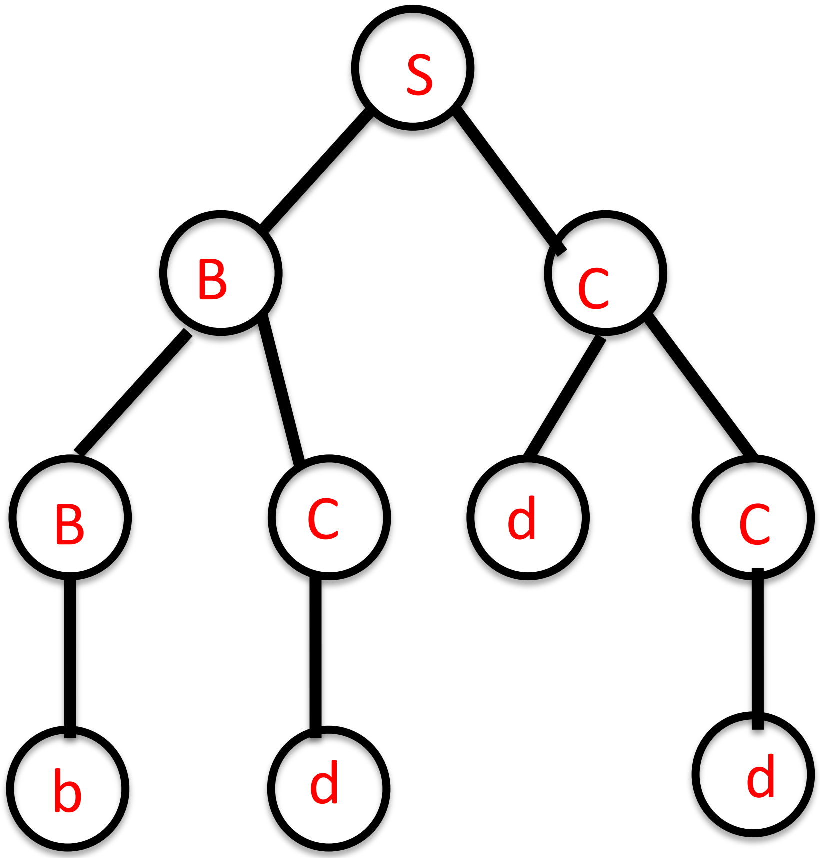
Alberi di derivazione

Per i linguaggi di Tipo 2 abbiamo una immediata corrispondenza tra derivazioni e alberi (alberi sintattici).

Esempio.
Data la grammatica

$S \rightarrow BC \qquad B \rightarrow b \mid BC$
$C \rightarrow d \mid dC$

la derivazione della stringa bddd è rappresentata con l'albero a destra



Alberi di derivazione

Data una grammatica G=(V,N,S,P), un **albero di derivazione** è un albero per cui

- la radice è etichettata con S (il simbolo iniziale)
- le foglie sono etichettate con simboli in V (simboli terminali)
- per ogni nodo con etichetta un simbolo non terminale A, i figli di tale nodo sono etichettati con i simboli della parte destra di una produzione in P la cui parte sinistra sia A

Un albero di derivazione è detto essere un albero di derivazione della stringa x se i simboli che etichettano le foglie dell'albero, letti da sinistra verso destra, formano la stringa x

Albero sintattico e traduzione

L'albero sintattico ottenuto viene usato per le fasi successive della traduzione

Esempio
Consideriamo l'istruzione a = b + c che porta alla sequenza di token id=id+id (con token id,=,+)
Un frammento di grammatica che descrive l'istruzione:
ISTRUZIONE-ASS → id = ESPRESSIONE
ESPRESSIONE → id + ESPRESSIONE
ESPRESSIONE → id

Sequenza produzioni per ottenere id=id+id
ISTRUZIONE-ASS → id = ESPRESSIONE → id = id + ESPRESSIONE → → id = id + ESPRESSIONE → id=id+id

Albero sintattico e traduzione

Semantica (semplificata) delle produzioni

1. ISTRUZIONE-ASS → id = ESPRESSIONE

   Semantica: assegna a id a sinistra di = il valore di ESPRESSIONE
2. ESPRESSIONE → id + ESPRESSIONE

   Semantica: calcola la somma di id e di ESPRESSIONE a destra
3. ESPRESSIONE → id

   Semantica: il valore di ESPRESSIONE è quello di id

Sequenza produzioni per ottenere id=id+id

ISTRUZIONE-ASS (1)→ id = ESPRESSIONE

(2)→ id = id + ESPRESSIONE (3)→ id = id + id

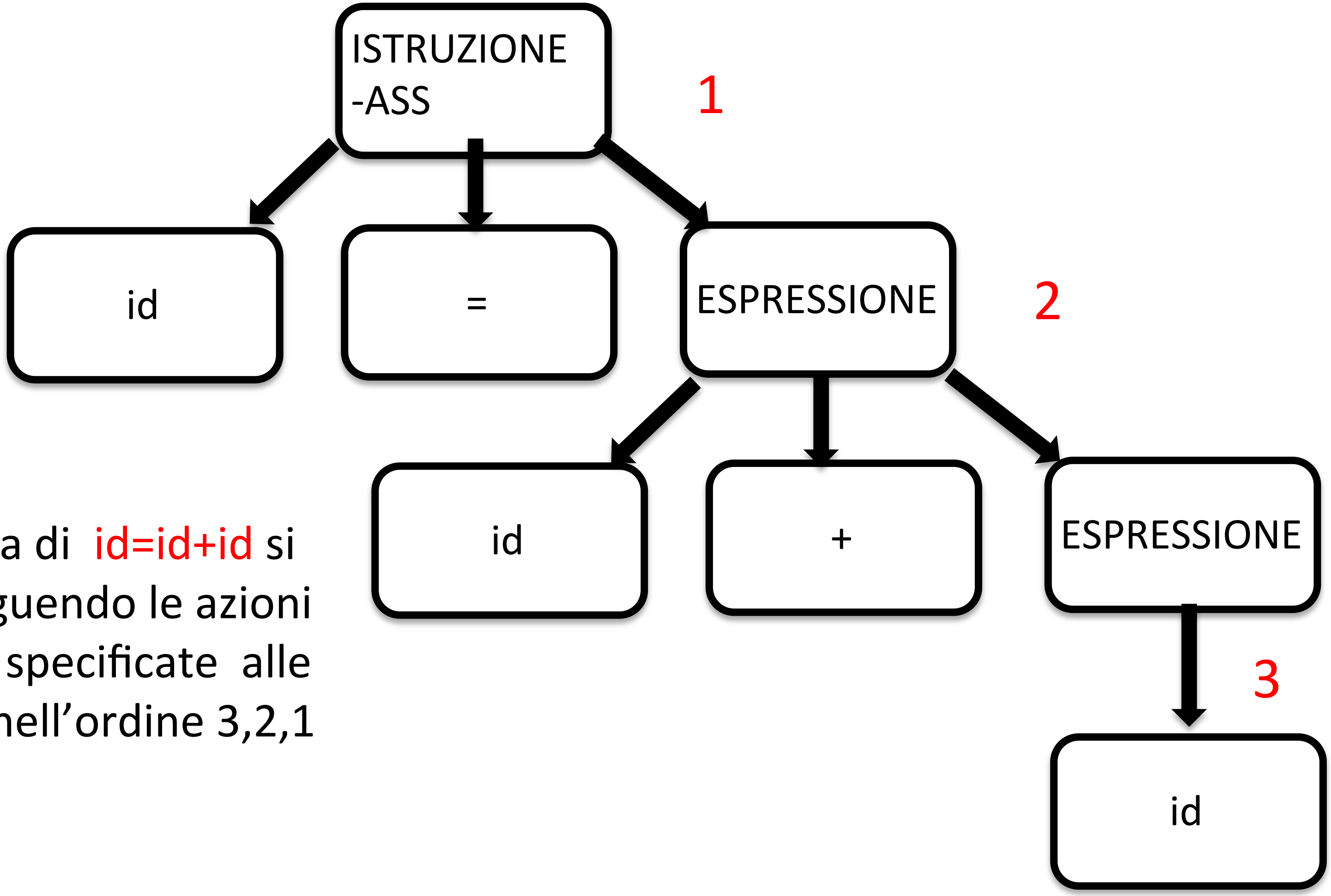
La semantica (il valore) di id=id+id si ottiene eseguendo le azioni semantiche associate alle produzioni nell'ordine 3,2,1

Albero sintattico e traduzione

Sequenza produzioni per ottenere id=id+id

ISTRUZIONE-ASS (1)→ id = ESPRESSIONE

(2)→ id = id + ESPRESSIONE (3)→ id = id + id



La semantica di id=id+id si ottiene eseguendo le azioni semantiche specificate alle produzioni nell'ordine 3,2,1

Analisi sintattica e Alberi di derivazione

Consideriamo le espressioni aritmetiche in un linguaggio di programmazione formate facendo uso delle sole operazioni di somma, sottrazione, moltiplicazione e divisione definita dalla grammatica
(con E simbolo iniziale e unico simbolo non termnale)

$E \rightarrow E+E \qquad E \rightarrow E-E \qquad E \rightarrow E*E$

$E \rightarrow E/E \qquad E \rightarrow (E) \qquad E \rightarrow id$

Analizziamo l'espressione (a+b)/(a-b)

L'analizzatore lessicale passerà quest'espressione in forma di token, sostituendo i lessemi a e b con il token id. fornendo all'analizzatore sintattico la stringa

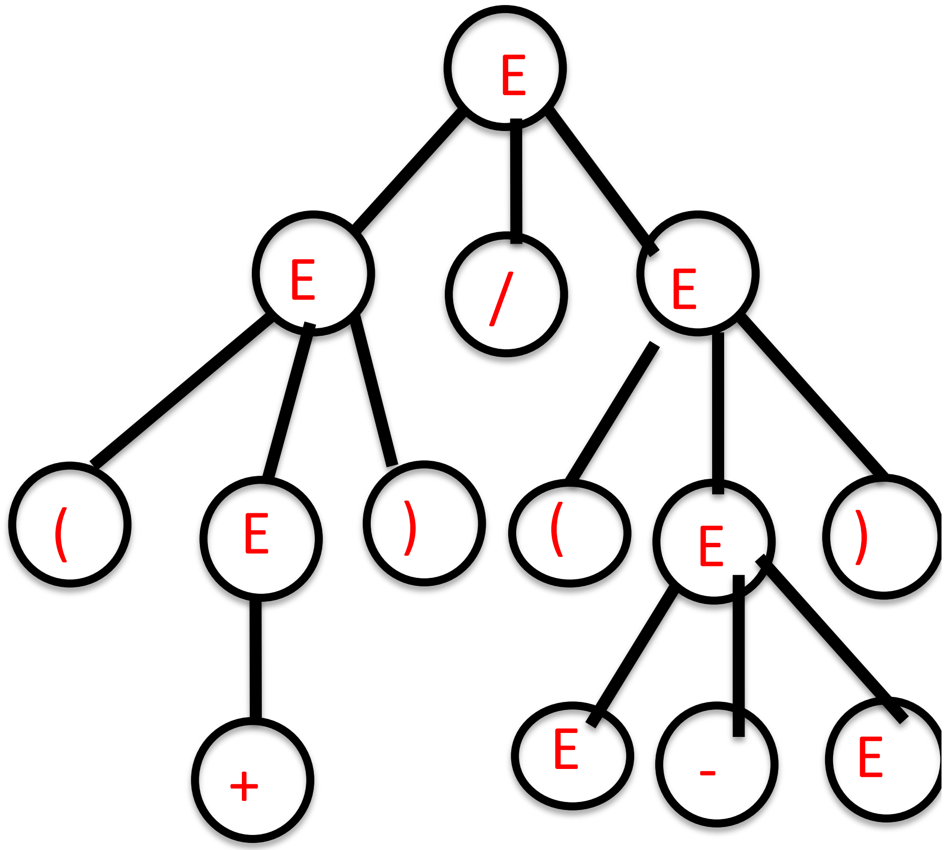
(id+id)/(id-id) - i token sono ()+/- id

Analisi sintattica e Alberi di derivazione

L'analizzatore sintattico riceve la stringa (id+id)/(id-id) deve ricostruire l'alberto di derivazione

La presenza di ( ) e del simbolo /
suggerisce che la prima
produzione da applicare
all'assioma E
sia E → E/E
e poi due volte la produzione
E → (E)

Ottenendo l'albero a destra

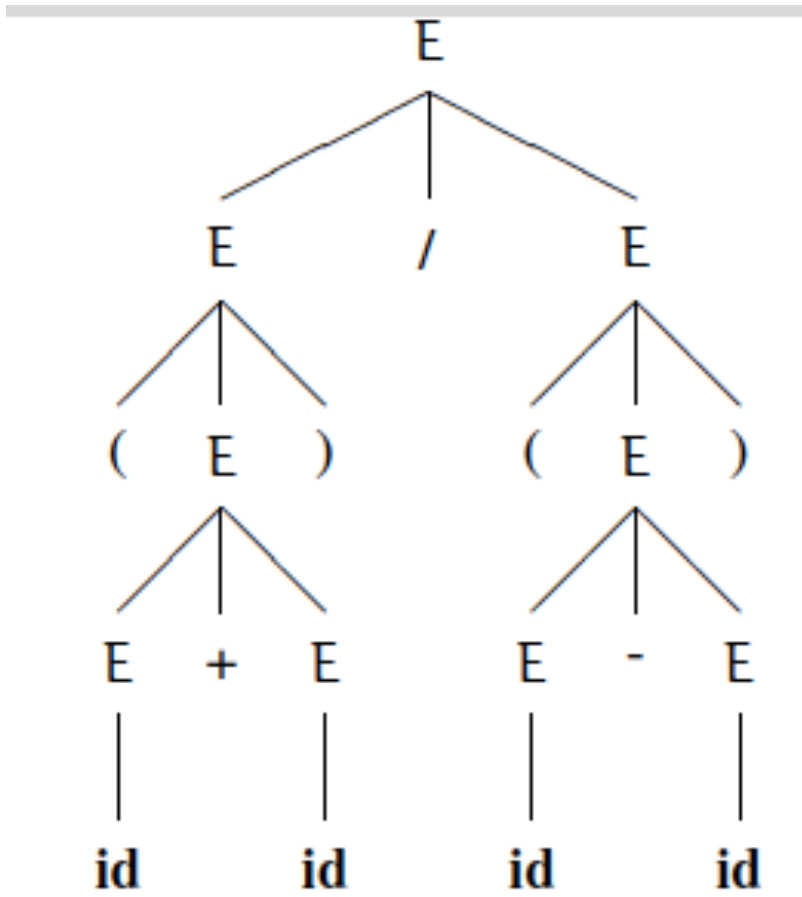


Analisi sintattica e Alberi di derivazione

L'analizzatore sintattico riceve la stringa (id+id)/(id-id) deve ricostruire l'alberto di derivazione

Si continua espandendo (E)/(E) applicando
Prima le produzioni
$E \rightarrow E+E$ e $E \rightarrow E-E$
e poi due volte la produzione
$E \rightarrow id$

Ottenendo l'albero a destra le cui foglie sono solo simboli terminali e rappresentano la stringa fornita dall'analizzatore lessicale: (id+id)/(id-id)

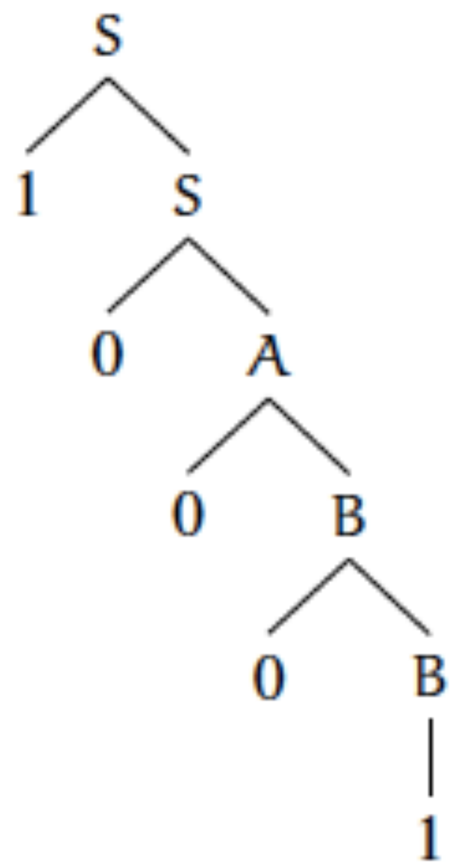


Analisi sintattica e Alberi di derivazione

Data la grammatica con simbolo iniziale S e produzioni:

S →1S S→0S S→0A A→0B A→0
B→0B B→1B B→0 B→ 1

Un possibile albero di derivazione della stringa 10001



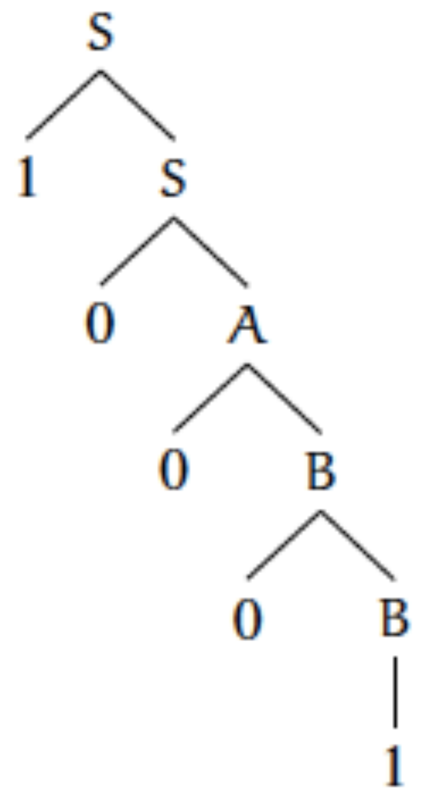
Analisi sintattica e Alberi di derivazione

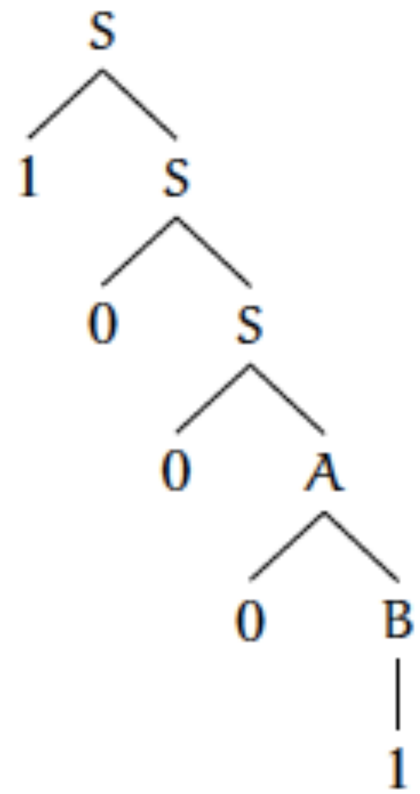
Data la grammatica con simbolo iniziale S e produzioni:

S → 1S S → 0S S → 0A A → 0B A → 0

B → 0B B → 1B B → 0 B → 1

A destra un secondo albero di derivazione della stringa 10001





Grammatiche ambigue

- Una grammatica è ambigua quando per una stessa stringa che appartiene al linguaggio definito dalla grammatica esistono due diversi alberi di derivazione
- Le grammatiche ambigue non sono adatte per i linguaggi di programmazione
- Semantica non chiara
- Complessità analisi (due alberi derivazione)

Ambiguità nei linguaggi

In italiano l'ambiguità sintattica è una proprietà di quelle frasi per le quali può essere fornita più di un'interpretazione.

- l'ambiguità semantica nasce dalla gamma di significati diversi che possiamo associare ad una parola,
- l'ambiguità sintattica si verifica quando sono possibili più strutture sintattiche diverse per interpretare la stessa frase

Esempi di ambiguità sintattica:

- Luigi ha visto Ada nel parco con il cannocchiale (chi sta usando il cannocchiale? Luigi o Ada?)
- Una vecchia legge la regola (il soggetto è una donna anziana – "una vecchia" - oppure il soggetto è "una vecchia legge"?)
- Si vendono letti di ferro per bambini con palle d'ottone (senza commento)

Ambiguità nei linguaggi

In italiano, nella maggioranza dei casi pratici, il significato di una frase sintatticamente ambigua si risolve utilizzando il contesto che chiarisce il vero significato della frase.

- Purtroppo nel caso dei linguaggi di programmazione il problema dell'ambiguità non può essere risolto dal contesto.
- Durante l'analisi sintattica di un programma non possiamo ammettere che il programma in input possa essere intrpretato in due modi diversi.

Pertanto la grammatica utilizzata per definire il linguaggio non deve permettere ambiguità

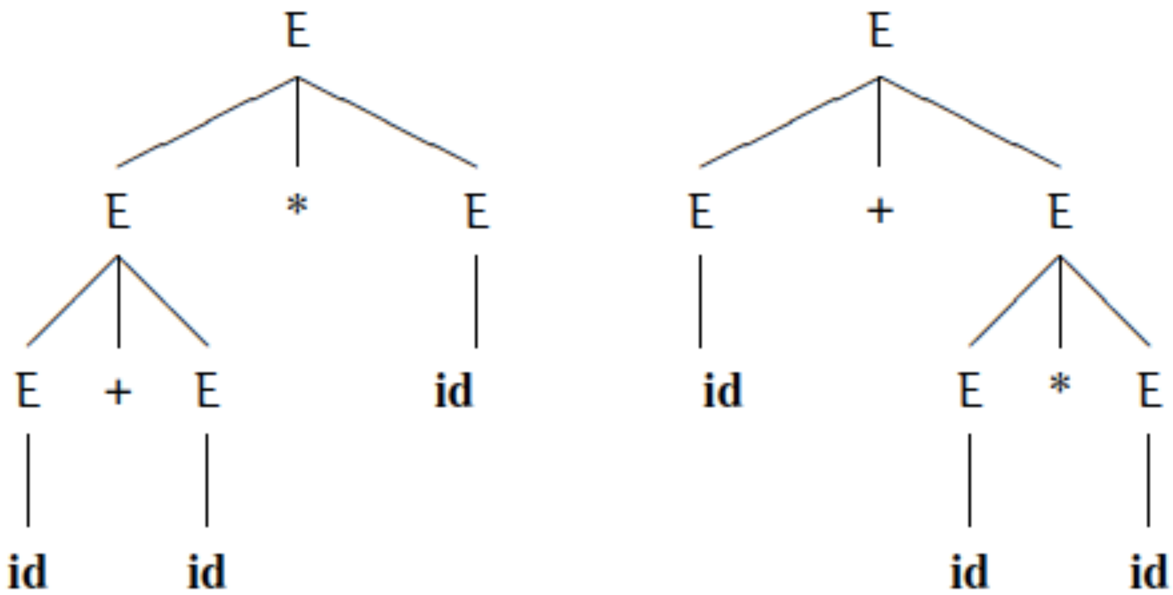
Grammatiche ambigue

Consideriamo la grammatica
(con E simbolo iniziale e unico simbolo non terminale)

E → E+E   E → E-E   E→ E*E
E → E/E   E →(E)   E → id

Analizziamo l'espressione id+id*id - che corrisponde alla espressione a+b*c
Quale produzione applichiamo per prima? E → E*E o E→ E+E ?

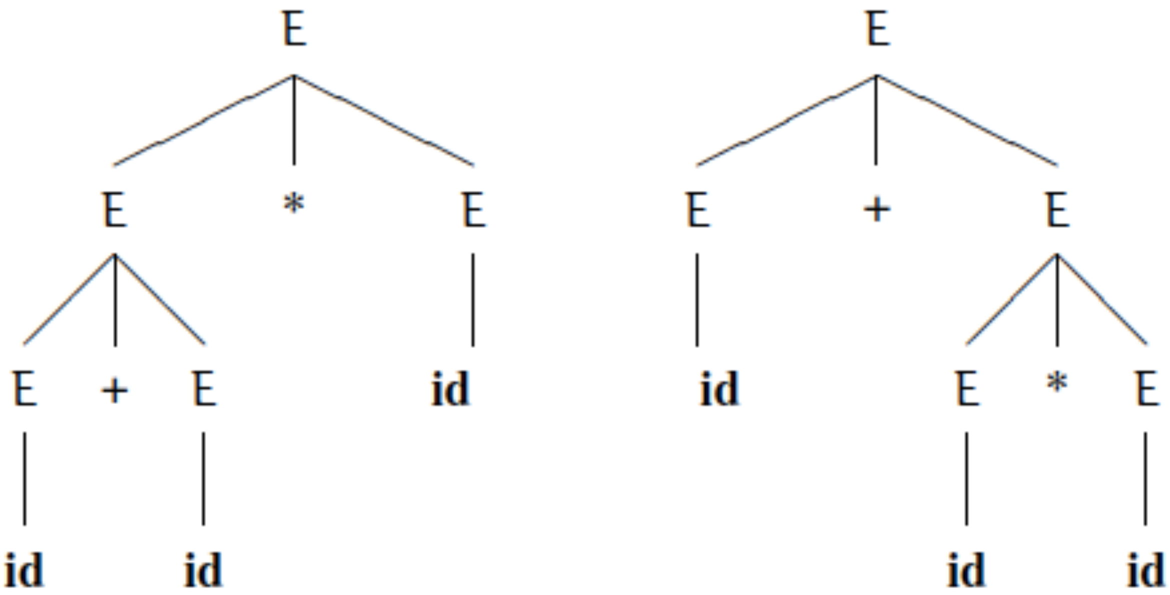
Inizia con E→ E*E



Inizia con E→ E+ E

Grammatiche ambigue

Inizia con E→ E*E



Inizia con E→ E+ E

Ricorda albero sintattico guida la traduzione

Quale produzione applichiamo per prima? E → E*E o E→ E+E ?

- Albero a sinistra: eseguiamo la moltiplicazione fra il risultato della somma dei primi due identificatori (id+id) e il terzo id
- Albero a destra: sommiamo il primo identificatore al risultato della moltiplicazione fra il secondo e terzo identificatore
- **Un albero è giusto, uno è errato**

Grammatiche ambigue

- Abbiamo una grammatica ambigua per un linguaggio L
- Le grammatiche ambigue non sono adatte per i linguaggi di programmazione
- Due alberi derivazione per un programma forniscono due interpretazioni (traduzioni del programma) diverse
- Data una grammatica ambigua dobbiamo riscrivere le produzioni per ottenere una grammatica NON ambigua
- La grammatica ottenuta deve rispettare le regole semantiche che vogliamo abbia il linguaggio
- Questo lavoro richiede di capire bene la semantica del linguaggio

Grammatiche non ambigue per espressioni aritmetiche

- Grammatica ambigua e due possibili riscritture non ambigue
- Domanda: Quale delle due non ambigue mantiene l'usuale priorità fra le operazioni di '+' e '*'?

ambigua	non ambigua -1	non ambigua -2
1 $E \rightarrow E + E$	1 $E \rightarrow E + T$	1 $E \rightarrow E * T$
2 $E \rightarrow E * E$	2 $E \rightarrow T$	2 $E \rightarrow T$
3 $E \rightarrow id$	3 $T \rightarrow T * F$	3 $T \rightarrow F + T$
4 $E \rightarrow (E)$	4 $T \rightarrow F$	4 $T \rightarrow F$
	5 $F \rightarrow id$	5 $F \rightarrow id$
	6 $F \rightarrow (E)$	6 $F \rightarrow (E)$

Ricostruire alberi di derivazione per id+id*id usando le due grammatiche

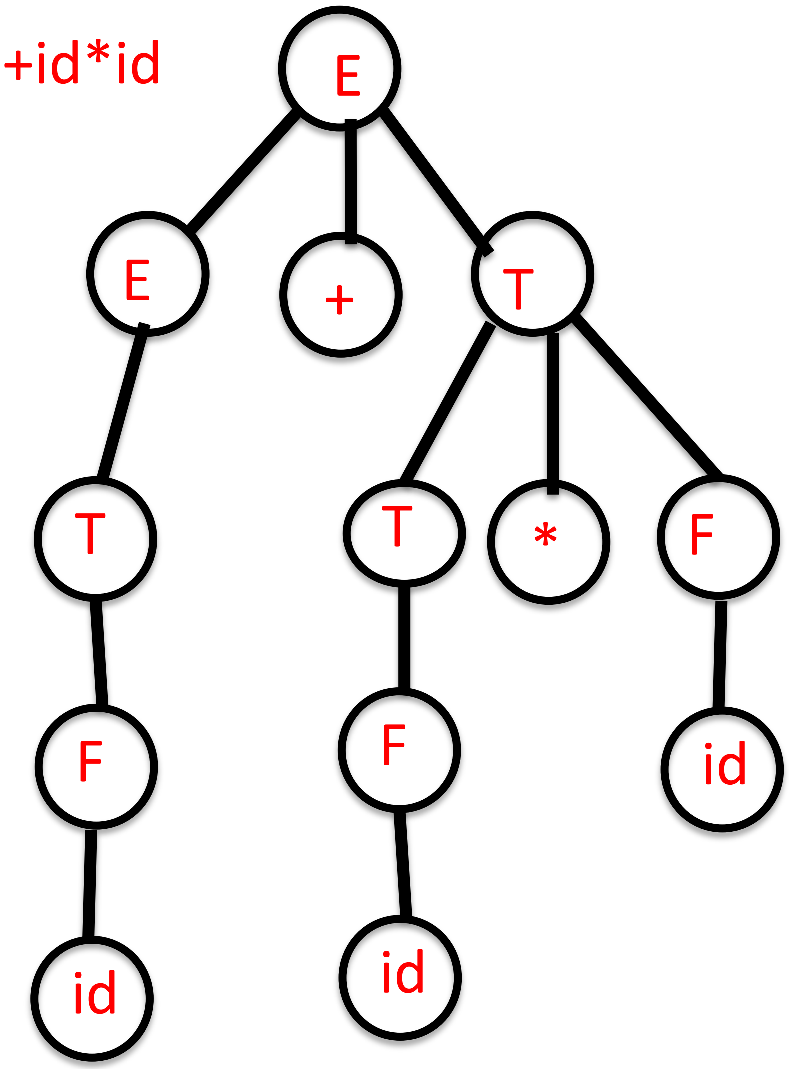
Grammatiche non ambigue per espressioni aritmetiche

Ricostruire albero di derivazione per id+id*id

non ambigua -1

1 $E \rightarrow E + T$
2 $E \rightarrow T$
3 $T \rightarrow T * F$
4 $T \rightarrow F$
5 $F \rightarrow id$
6 $F \rightarrow (E)$

E→E+T→T+T→F+T→
→ id+T →id+ T*F → id+F*F
→ id+id*F →id+id*id



Grammatiche non ambigue

Ricostruire albero di derivazione per id+id*id

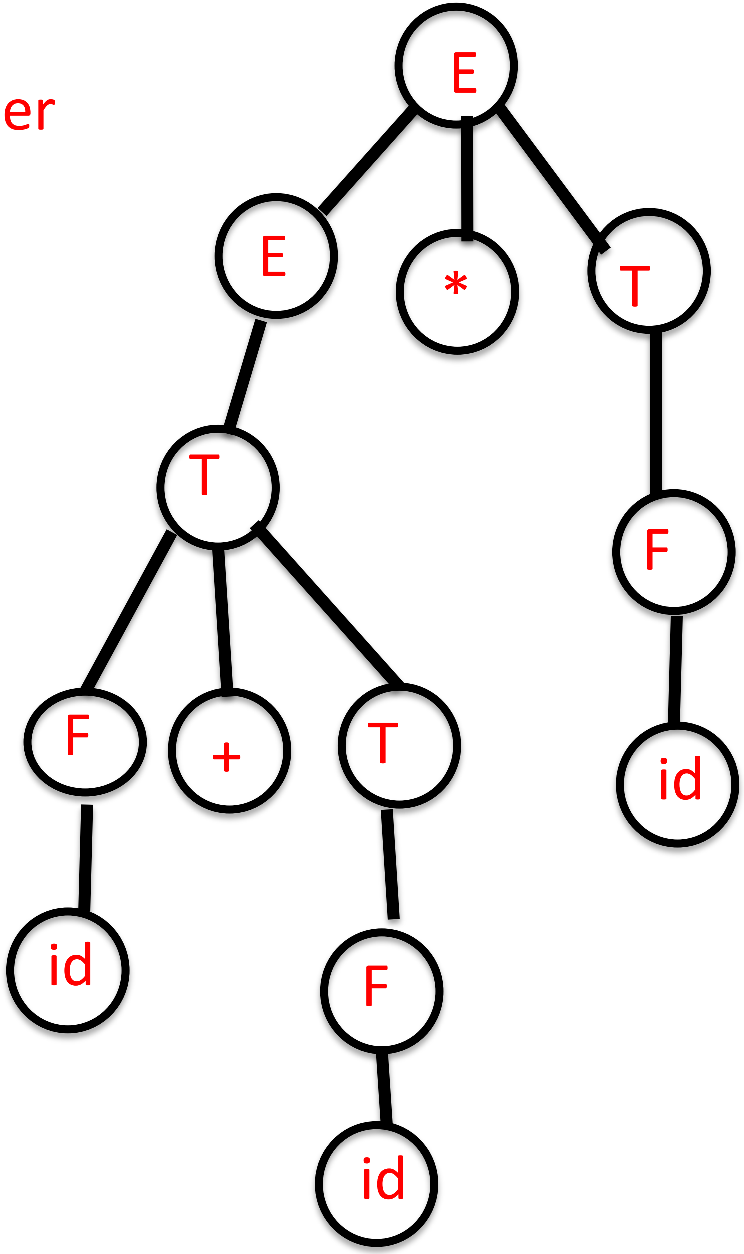
non ambigua -2

1	$E \rightarrow E * T$
2	$E \rightarrow T$
3	$T \rightarrow F + T$
4	$T \rightarrow F$
5	$F \rightarrow id$
6	$F \rightarrow (E)$

E→E*T→T*T→F+T*T→

→ id+T*T →id+ F*T → id+id*T

→ id+id*F →id+id*id



Grammatiche non ambigue per espressioni aritmetiche

- Grammatica ambigua e due possibili riscritture non ambigue
- Domanda: Quale delle due non ambigue mantiene l'usuale priorità fra le operazioni di '+' e '*'?

ambigua

1 $E \rightarrow E + E$
2 $E \rightarrow E * E$
3 $E \rightarrow id$
4 $E \rightarrow (E)$

non ambigua -1

1 $E \rightarrow E + T$
2 $E \rightarrow T$
3 $T \rightarrow T * F$
4 $T \rightarrow F$
5 $F \rightarrow id$
6 $F \rightarrow (E)$

non ambigua -2

1 $E \rightarrow E * T$
2 $E \rightarrow T$
3 $T \rightarrow F + T$
4 $T \rightarrow F$
5 $F \rightarrow id$
6 $F \rightarrow (E)$

La grammatica a destra non rispetta l'usuale priorità fra le operazioni di * e +

Grammatiche non ambigue per espressioni aritmetiche

Come passare da ambiguo a non ambiguo? Si riscrive la grammatica in modo da eliminare le l'ambiguità.

Nell'esempio visto prima distinguiamo tra espressioni, termini e fattori (grammatica non ambigua con anche operaz. – e /)

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

- Questa grammatica esplicita il fatto che, che * e / sono possibili fra un termine e un fattore (simboli non termin. T e F)
- Questo implica che - se vogliamo usare la somma o sottrazione di due addendi come fattore di una moltiplicazione o di una divisione - allora dobbiamo prima racchiudere tale somma o sottrazione tra parentesi.

Analisi sintattica - Parsing

L'analizzatore sintattico programma con

- INPUT: Sequenza di tokens
- OUTPUT: Albero di derivazione (unico perché assumiamo una grammatica non ambigua)

Per costruire l'albero di derivazione produce una sequenza di produzioni a partire dall'assioma

Altre cose fatte dall'analizzatore sintattico

- Riporta errori di sintassi (e non riesce a costruire l'albero di derivazione)
- Crea tabella dei simboli, che contiene i diversi identificatori usati nel programma

Derivazioni destre e sinistre

Consideriamo la grammatica

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

La grammatica non è ambigua → esiste un solo albero di derivazione

Però possiamo scegliere fra più produzioni

Esempio id + id *id: due fra le molte possibili sequenze

$E \rightarrow E + T$	$E \rightarrow E + T$
$\rightarrow T + T$	$\rightarrow E + T * F$
$\rightarrow T + T * F$	$\rightarrow E + F * F$
$\rightarrow F + T * F$	$\rightarrow T + F * F$
$\rightarrow id + T * F$	$\rightarrow F + F * F$
$\rightarrow id + T * id$	$\rightarrow F * id * F$
$\rightarrow id + F * id$	$\rightarrow F * id * id$
$\rightarrow id + id * id$	$\rightarrow id + id * id$

Derivazioni destre e sinistre

Invece di scegliere a caso la produzione possiamo considerare due approcci

-derivazione destra: espandi sempre il nonterminale più a destra

-derivazione sinistra: espandi sempre il non teminale più a sinistra

Esempio Consideriamo la grammatica precedente e analizziamo la stringa (id+id)/(id-id)

derivazione destra

$$
\begin{aligned}
E &\to E/E \\
&\to E/(E) \\
&\to E/(E-E) \\
&\to E/(E-\mathbf{id}) \\
&\to E/(\mathbf{id}-\mathbf{id}) \\
&\to (E)/(\mathbf{id}-\mathbf{id}) \\
&\to (E+E)/(\mathbf{id}-\mathbf{id}) \\
&\to (E+\mathbf{id})/(\mathbf{id}-\mathbf{id}) \\
&\to (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})
\end{aligned}
$$

derivazione sinistra

$$
\begin{aligned}
E &\to E/E \\
&\to (E)/E \\
&\to (E+E)/E \\
&\to (\mathbf{id}+E)/E \\
&\to (\mathbf{id}+\mathbf{id})/E \\
&\to (\mathbf{id}+\mathbf{id})/(E) \\
&\to (\mathbf{id}+\mathbf{id})/(E-E) \\
&\to (\mathbf{id}+\mathbf{id})/(\mathbf{id}-E) \\
&\to (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})
\end{aligned}
$$

Parser top-down e parser bottom -up

- La distinzione tra derivazioni sinistre e destre non è accademica infatti esistono, **due tipi di analizzatori sintattici**
- Top-down (genera derivazioni sinistre)
- Bottom-up (genera derivazioni destre)
- le differenze tra questi due tipi incide direttamente sui dettagli della costruzione del parser e sulle sue operazioni
- nel seguito ci limiteremo ad analizzare gli analizzatori del primo tipo (derivazioni sinistre) anche detti **parser top-down**

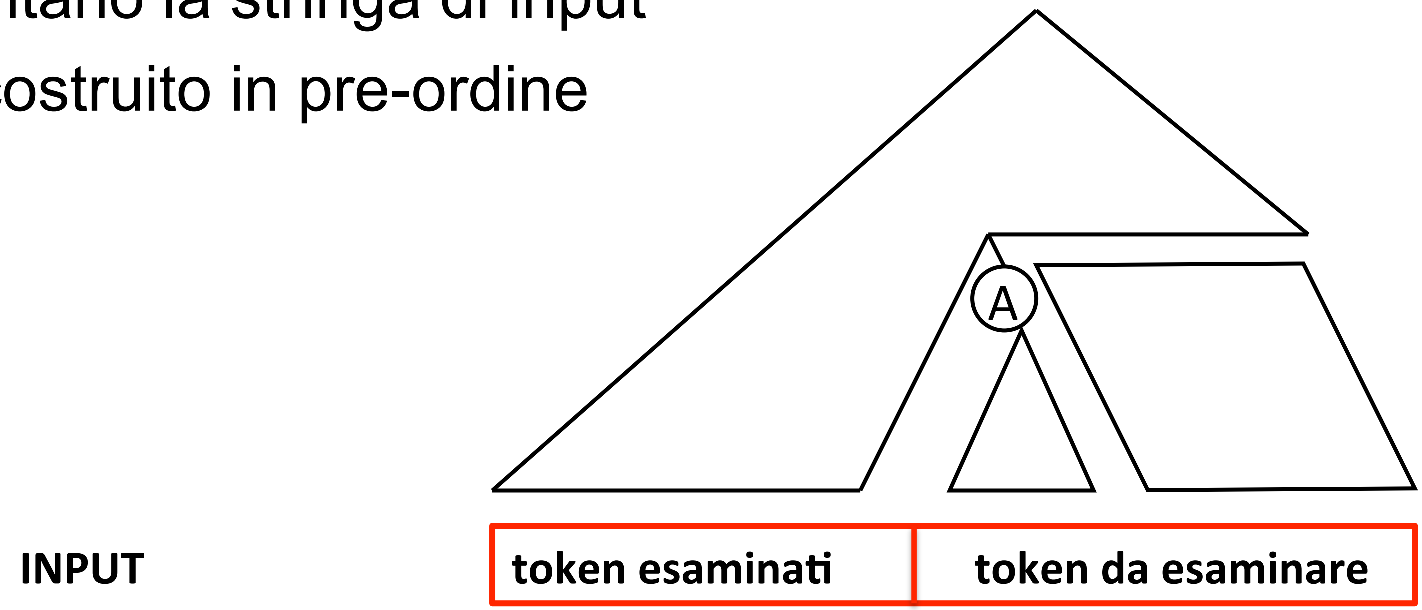
Analisi sintattica: top-down

Nel seguito ci limitiamo a considerare analizzatori

Top-Down

Si inizia con l'assioma

- Ripetutamente scegli un simbolo non terminale A dell'albero finora ottenuto e applica una produzione ad A
- Termina con successo quando le foglie dell'albero rappresentano la stringa di input
- Albero ricostruito in pre-ordine



Proprietà dei prefissi

Per ogni sequenza di token iniziale (prefisso) $t_1, t_2, \ldots, t_k$ che l'analisi individua come legale (cioè che potrebbe fornire una stringa del linguaggio)

- **devono esistere tokens $t_{k+1}, t_{k+2}, \ldots, t_n$ tali che $t_1, t_2, \ldots, t_n$ è un programma sintatticamente corretto**

Equivalentemente

- Se consideriamo un token come un singolo carattere, per ogni parola prefisso x che l'analisi considera legale
  - **esiste parola suffisso w tale che $x \cdot w$ è un programma valido**

Esempio: istruzione if… then ...else ...

- Supponiamo che il prossimo token da esaminare sia il token <if> e scelgo la produzione con parte sinitra di → che corrisponde a ISTR-IF-THEN-ELSE fra i token che seguono <if> ci deve essere il token <then> e poi il token <else>

Analisi Top-Down

Supponi che

- **input sia wxy** e **w la parte già esaminata** dell'input
- e di aver completato parte della derivazione dall'assioma S

ottenendo S $\rightarrow$ .. $\rightarrow$ ...$\rightarrow$ wAα

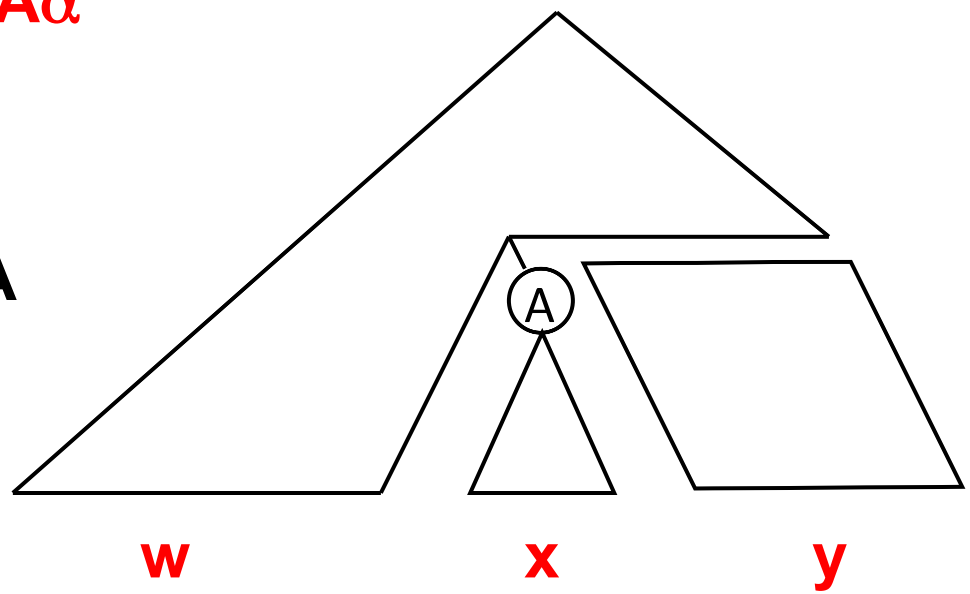
Passo da fare

Scegli una produzione su A

$$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_n$$

che è congruente con

la parte x dell'input



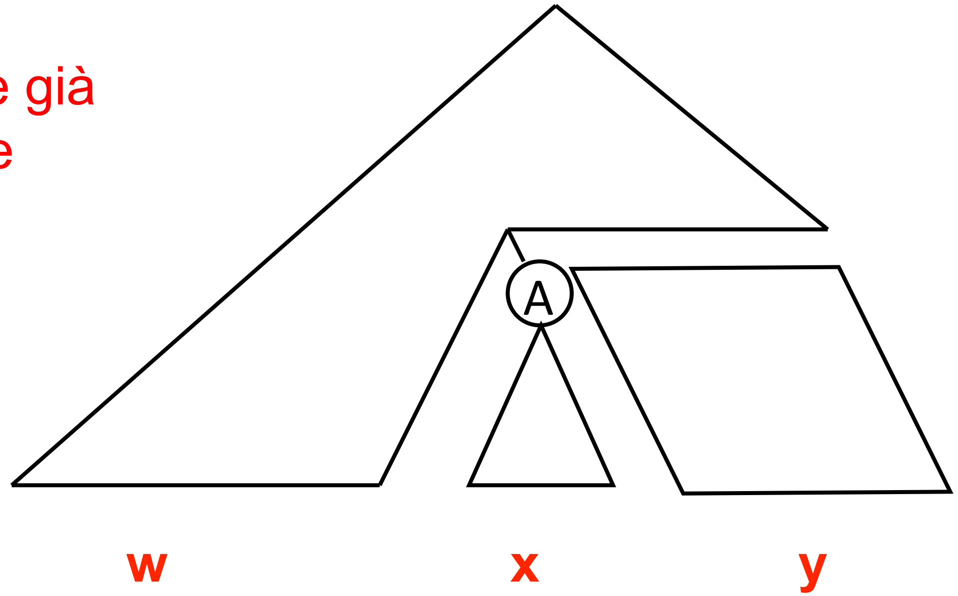
Analisi Top-Down

Supponi che input sia wxy, w la parte già esaminata, e di aver completato parte della derivazione dall'assioma S

ottenendo $S \rightarrow .. \rightarrow ... \rightarrow wA\alpha$

Passo: Scegli una produzione su A congruente con la parte x dell'input

$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_n$



Esempio

Supponiamo che una produzione possibile sia

A→ ISTRUZ-IF-THEN-ELSE

Per essere corretta allora

- il prossimo token da esaminare (primo token di x) è il token <if>
- fra i token di x che seguono <if> ci deve essere
- la sequenza di token che corrispondono a CONDIZIONE seguito
- quindi il token <then> e poi un'istruzione e poi il token <else> ecc.

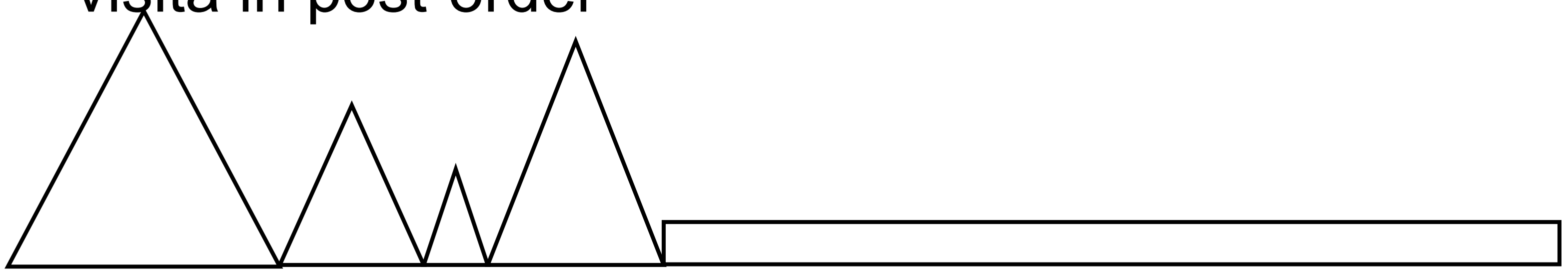
Analisi sintattica: bottom-up

Bottom-up (dal basso all'alto)

usa derivazioni destre

Costruisce l'albero di derivazione dalle foglie

- Avanza nell'input (shift) o deriva un non terminale (reduce)
- Accetta quando input è finito e hai ottenuto assioma
- Ricostruisce in effetti l'albero attraverso una sua visita in post-order



Analisi Top-Down: parser a discesa ricorsiva

- Un vantaggio dei parser top-down è che sono più semplici da implementare
- Idea fondamentale: scrivere una funzione (procedura, metodo o sottoprogramma) in corrispondenza ad ogni simbolo non terminale
- Ognuna di queste funzioni è responsabile di accoppiare il non-terminale con il resto dell’input
- Per motivi di efficienza il passo deve essere deterministico (non vogliamo avere backtracking)

Analisi Top-Down: parser a discesa ricorsiva

In generale da un simbolo terminale possiamo applicare più produzioni. Come procedere?

Soluzione semplice: algoritmo ricorsivo prova tutte le pssibilità

Esempio

- Date le produzioni $A \rightarrow \alpha \mid \beta$ **(**α**,** β sono stringhe di simboli terminali e non terminali)
- Algoritmo ricorsivo contiene una funzione che tenta dapprima di usare la produzione $A \rightarrow \alpha$
- Se ottiene successo algoritmo termina con successo, altrimenti tenta di usare la produzione $A \rightarrow \beta$
- Se ottiene successo, l'analisi di A termina con successo, altrimenti termina con fallimento.
- Se ci sono più possibiità si procede in modo analogo

Tentare di usare una produzione significa consumare porzioni di input (terminali nella parte destra) e chiamare funzioni associate a non terminali.

Analisi Top-Down: parser a discesa ricorsiva

- Tentare di usare una produzione $A \rightarrow \alpha$ significa consumare porzioni di input (simboli terminali in α) e chiamare funzioni associate ai non terminali di α

Esempio: $A \rightarrow aBcD$

Abbiamo successo se e solo se

- il primo carattere di α in input è 'a' ,
- la chiamata a B restituisce successo,
- quando B termina il carattere successivo da leggere in α è 'c'
- la chiamata D restituisce successo

Parser: backtracking

In generale, assumi che per A esistano produzioni del tipo

$A \rightarrow \alpha \mid \beta \mid \gamma \mid \delta \ldots$

($\alpha \mid \beta \mid \gamma \mid \delta$ sono stringhe di simboli)

- Bisogna provare tutte le possibili produzioni: provo $A \rightarrow \alpha$, se non funziona provo $A \rightarrow \beta$ e così via
- Nota: non so subito se $A \rightarrow \alpha$ non funziona: potrei andare avanti con input e solo alla fine accorgermi che era la scelta sbagliata (Backtracking)
- Questo porta a programmi lenti

Parser predittivi

Bisogna evitare il backtracking e non provare tutte le possibili produzioni; ma questo porta a programmi lenti (sopratutto se il programma e' sintatticamente scorretto)

- Parser predittivo: capisce sempre la produzione giusta da applicare
- Supponi di dover espandere il non terminale A e di avere due produzioni del tipo

  $A \rightarrow \alpha \qquad\qquad A \rightarrow \beta$
- Dobbiamo scegliere la produzione giusta sulla base dell'input ancora da esaminare
- Se siamo in grado di fare questo otteniamo un parser predittivo che non ha bisogno di simulare il non determinismo

Parser predittivi: Esempio

I linguaggi di programmazione sono spesso adatti al parsing predittivo

Esempio: grammatica con simbol.non terminali {istr, exp}; Assioma istr simb. terminali {id, =, ;, return, (,)}

- Possibili produzioni da assioma istr

```
istr  →   id = exp ; |
          return exp ; |
          if ( exp ) istr |
          while ( exp ) istr
```

Se la prima parte dell'input da esaminare inizia con

- il token 'id' allora dobbiamo espandere istr come `id = exp ;` (prima produzione)
- il token 'if' allora dovremmo espandere istr come `if ( exp ) istr` (terza produzione)

Parser predittivi: Esempio

Esempio (semplice)

```
Stmt →   id = exp ; |
         return exp ; |
         if ( exp ) stmt |
         while ( exp ) stmt
```

In questo caso basta analizzare un solo token per decidere la produzione; se il token è

- id la produzione è ‘id=exp;’
- return la produzione è ‘return exp;
- if la produzione è ‘if (exp) stmt
- while la produzione è ‘while (exp) stmt

Parser predittivi: Esempio

I linguaggi di programmazione sono spesso adatti al

pa

Es

st

- Purtroppo i linguaggi di programmazione portano spesso a casi più complessi che non possono essere risolti direttamente
- Abbiamo bisogno di modificare la grammatica per poter applicare i parser predittivi
- Nel seguito consideriamo alcuni dei problemi e vediamo le soluzioni adottate

Se

all

sta

Parser predittivi: produzioni ricorsive

Problema: produzioni del tipo $A \rightarrow Ab$

Esempio: versione semplificata della grammatica precedentemente introdotta

$E \rightarrow E+T \quad E \rightarrow T \quad T \rightarrow T*F$

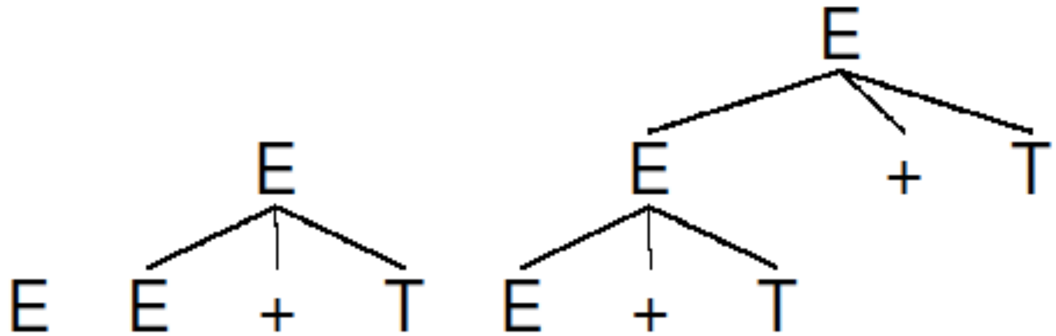
$T \rightarrow F \quad F \rightarrow (E) \quad F \rightarrow id$

Consideriamo la stringa id+id+id

Sembra una buona idea applicare la produzione $E \rightarrow E+T$

Parser predittivi: produzioni ricorsive

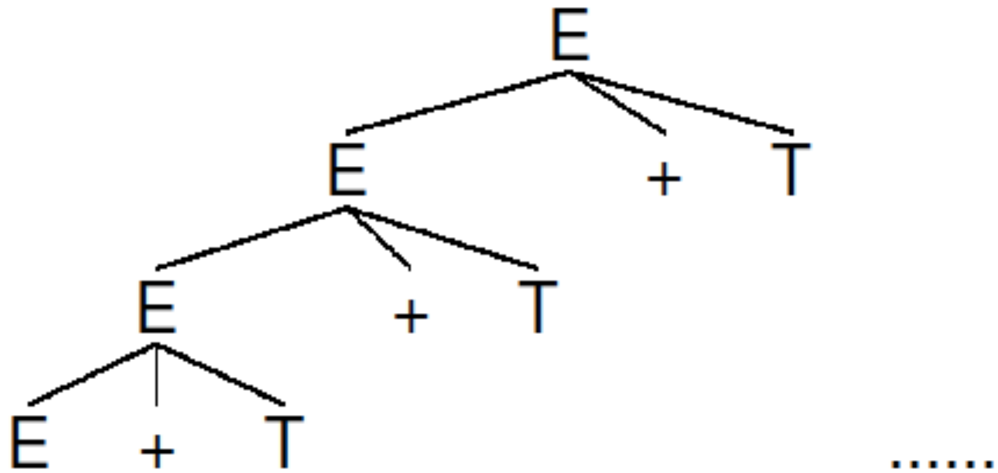
Sembra una buona idea applicare la produzione E→E+T



Problema

quando dobbiamo applicare una produzione diversa da E→ E+T?

Per decidere quando (in questo caso si applica E→T) dobbiamo vedere una lunga sequenza di token oppure si cicla



Parser predittivi: produzioni ricorsive

Produzioni della forma $A \rightarrow A\,\alpha$ (nell'esempio $E \rightarrow E+T$) sono **produzioni ricorsive a sinistra**

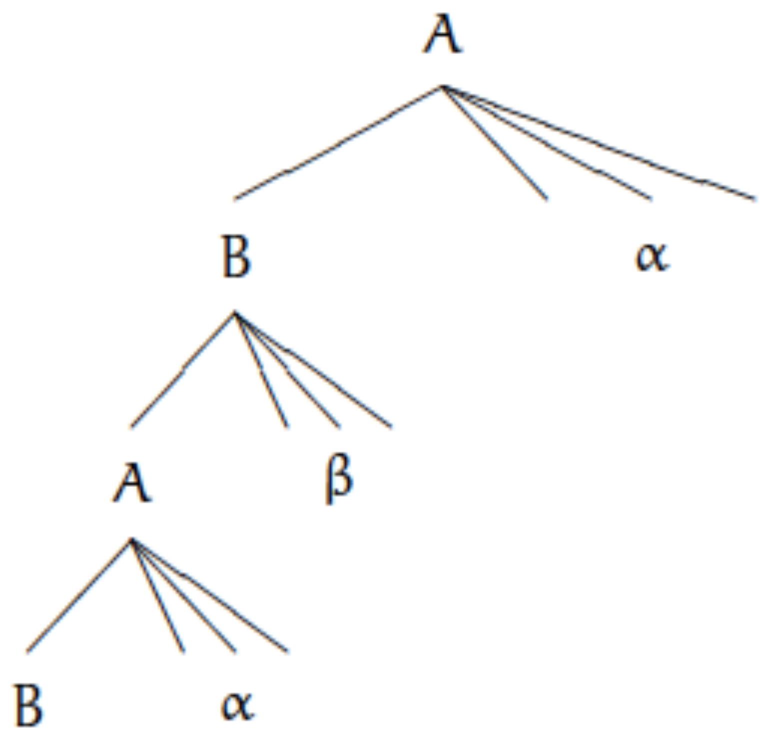
- **nessun parser top-down è in grado di gestirle.**
- Infatti il parser procede "consumando" i simboli terminali: ognuno di tali simboli guida il parser nella sua scelta di azioni.
- Quando il simbolo terminale è usato, un nuovo simbolo terminale diviene disponibile e ciò porta il parser a una diversa mossa. I simboli terminali vengono consumati quando si accordano con i terminali nelle produzioni:
- Nel caso di una produzione ricorsiva a sinistra, l'uso ripetuto di tale produzione non usa simboli terminali per cui, a ogni mossa nuova, il parser ha di fronte lo stesso simbolo terminale e quindi far`a la stessa mossa.
- Soluzione: riscrivere la grammatica

Ricorsione a sinistra

Due tipi di ricorsione

- le ricorsioni sinistre immediate generate da produzioni del tipo $A \rightarrow A\alpha$
- quelle non immediate generate da produzioni del tipo $A \rightarrow B\alpha \quad B \rightarrow A\beta$

  In quest'ultimo caso, A produrrà B, B produrrà A e così via



Eliminazione produzioni ricorsive dirette

Supponi di avere le produzioni (no prod. Ricorsive indirette)

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \ldots \quad A \rightarrow \delta_1 \mid \delta_2 \mid \delta_3 \ldots$

1. Introduciamo un nuovo non terminale A’
2. Sostituiamo ogni produzione non ricorsiva

   $A \rightarrow \delta_i$ con la produzione $A' \rightarrow \delta_i A'$
3. Sostituiamo ogni produzione ricorsiva immediata $A \rightarrow A\alpha_i$ con la produzione $A' \rightarrow \alpha_i A'$
4. Aggiungiamo la produzione $A' \rightarrow \varepsilon$

Eliminazione produzioni ricorsive dirette

Esempio: Consideriamo la grammatica: $S \rightarrow Sa$ $S \rightarrow b$

- Tutte le derivazioni in questa grammatica hanno la forma

  $S \rightarrow Sa \rightarrow Saa \rightarrow Saaa \rightarrow \ldots \rightarrow ba^n$

  (il processo ha termine quando viene scelta la produzione $S \rightarrow b$)

La grammatica trasformata è la seguente:

$S \rightarrow bS'$ $S' \rightarrow aS'$ $S' \rightarrow \varepsilon$ (ε stringa vuota)

- Tutte le derivazioni in questa grammatica hanno la forma

  $S \rightarrow bS' \rightarrow baS' \rightarrow baaS' \rightarrow baaaS' \rightarrow \ldots \rightarrow ba^n$

  (in questo caso il processo ha termine quando scegliamo $S' \rightarrow \varepsilon$)

Eliminazione produzioni ricorsive dirette

Esempio: Supponi di avere le produzioni (no prod. ricorsive indirette) $A \rightarrow A\alpha_1 \mid A\alpha_2$ e $A \rightarrow \delta_1 \mid \delta_2$

Da queste produzioni otteniamo stringhe del tipo

$A \rightarrow A\alpha_1 \rightarrow A\alpha_1\alpha_1 \rightarrow A\alpha_1\alpha_1\alpha_2 \rightarrow \ldots \rightarrow \delta_1\alpha_1\alpha_2\alpha_2\alpha_1\alpha_1$

1. Introduciamo un nuovo non terminale A'
2. Sostituiamo $A \rightarrow \delta_1 \mid \delta_2$ con le produzioni
   $A' \rightarrow \delta_1 A' \mid \delta_2 A'$
3. Sostituiamo le produzione ricorsive immediate
   $A \rightarrow A\alpha_1 \mid A\alpha_2$ con $A' \rightarrow \alpha_1 A' \mid \alpha_2 A'$
4. Aggiungiamo la produzione $A' \rightarrow \lambda$

Eliminazione produzioni ricorsive dirette

1. Crea una lista ordinata A1, A2,…Am di tutti i simboli non terminali
2. Per j = 1,2,…m, esegui le seguenti operazioni:
   1. per h = 1; . . . ; j-1, sostituiamo ogni produzione del tipo $A_j \rightarrow A_h \beta$ con l'insieme delle produzioni $A_j \rightarrow \gamma \beta$ per ogni produzione del tipo $A_h \rightarrow \gamma$ (facendo riferimento alle produzioni di A_h già modificate);
   2. facendo uso della tecnica descritta in precedenza, rimuovi le eventuali ricorsioni sinistre immediate a partire da A_j (i nuovi non terminali che eventualmente vengono introdotti in questo passo non sono inseriti nella lista ordinata).

Eliminazione produzioni ricorsive non dirette

Esempio: Consideriamo la grammatica (S assioma):

$S \rightarrow aA$ $S \rightarrow b$ $S \rightarrow cS$ $A \rightarrow Sd$ $A \rightarrow e$

- S è assioma e non è coinvolto in prod. ricorsive dirette; per questo le produzioni per S non richiedono alcuna modifica.
- Per il simbolo non terminale A , abbiamo una parte destra che inizia con S. Sostituiamo tale produzione con

  $A \rightarrow aAd$ (usiamo $S \rightarrow aA$) $A \rightarrow bd$ (usiamo $S \rightarrow b$)

  $A \rightarrow cSd$ (usiamo $S \rightarrow cS$)

NOTA 1 uso ripetuto di $A \rightarrow Sd$ e $S \rightarrow aA$ porta a stringhe del tipo

$A \rightarrow Sd \rightarrow aAd \rightarrow aSdd \rightarrow \ldots \rightarrow aSddddd$

prima o poi abbiamo una 'e'; però per decidere quando non applicare $A \rightarrow Sd$ e applicare $A \rightarrow e$ dobbiamo contare quante d ci sono alla fine

NOTA2 questo equivale a inserire i passi iniziali di tutte le possibili derivazioni sinistre a partire da A tramite S nelle nuove parti destre. Infatti, a partire da A otteniamo Sd e quindi al passo successivo solo le seguenti forme: $A \rightarrow Sd \rightarrow aAd$ $A \rightarrow Sd \rightarrow bd$ $A \rightarrow Sd \rightarrow cSd$

Backtracking

Un modo per sviluppare un parser top-down consiste semplicemente nel fare in modo che il parser tenti esaustivamente tutte le produzioni applicabili fino a trovare l'albero di derivazione corretto (metodo della "forza bruta")

Tale metodo può essere inefficiente.

Esempio, consideriamo la grammatica :

$S \rightarrow ee \quad S \rightarrow bAc \quad S \rightarrow bAe \quad A \rightarrow d \quad A \rightarrow cA$

E ci chiediamo se bcde appartiene al linguaggio generato dalla grammatica

Backtracking

Esempio: consideriamo la grammatica

S→ee | bAc | bAe   A→d | cA

bcde appartiene al linguaggio?

- Il primo token è b; questo esclude la produzione S→ee
- Quindi proviamo con S→ bAc (primo token è ok)
- Token succesivo è c; questo esclude di applicare la produzione A→d; quindi otteniamo S→ bAc → bcAc (primi due token di input sono ok)
- Token successivo è d; possiamo applicare solo A→ d e otteniamo S→ bAc → bcAc → bcdc
- bcdc NON è la stringa di input: fallimento!

Backtracking

Esempio: consideriamo la grammatica

S→ee | bAc | bAe A→d | cA

bcde appartiene al linguaggio?

1. Il primo token è b; questo esclude la produzione S→ee; quindi proviamo con S→ bAc (primo token è ok)
2. Token succesivo è c; questo esclude di applicare la produzione A→d; quindi otteniamo S→ bAc → bcAc (primi due token di input sono ok)
3. Token successivo è d; possiamo applicare solo A→ d e otteniamo S→ bAc → bcAc → bcdc
4. bcdc NON è la stringa di input: fallimento!
5. Dobbiamo fare bactracking: ritornare al passo 1, considerare il primo token (b) e provare la produzione S→bAe

Backtracking

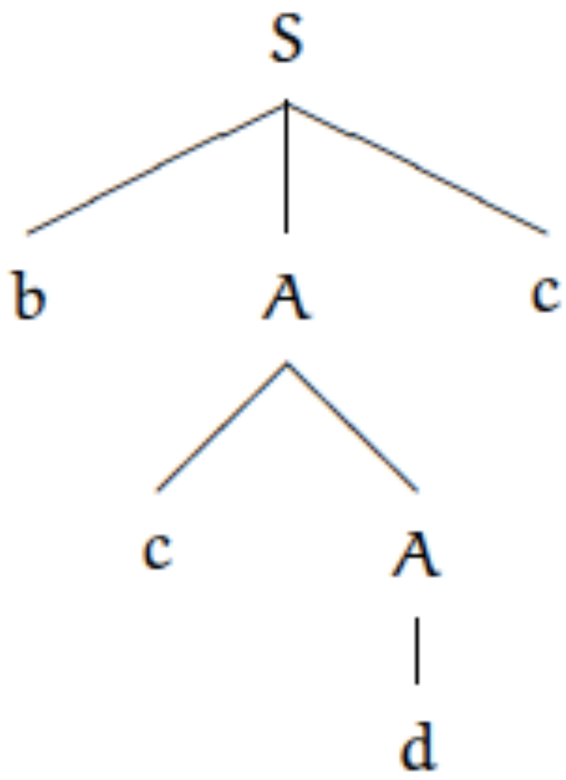
Esempio: consideriamo la grammatica

S→ee | bAc | bAe

A→d | cA

bcde appartiene al linguaggio?

Se consideriamo le produzioni possibili in ordine generiamo il seguente albero: ERRATO!



Bisogna provare più volte

Backtracking: problemi

Backtracking richiede di tornare indietro

- Nell'esempio siamo andati avanti nella stringa e abbiamo "*consumato*" token di input
- Quindi tornare indietro non vuol dire solo cancellare parti dell'albero ma anche tornare indietro nella lista dei token al punto dove aveva sbagliato. Compito non sempre facile e veloce
- Inoltre bactaracking richiede che analisi lessicale fatta prima analisi sintattica (lentezza) e non on-line (altrimenti dobbiamo rifarla per quelle parti)

Conclusione: il backtracking rende il compilatore lento (soprattutto nei programmi sintatticamente scorretti)

Backtracking: problemi

La grammatica seguente genera lo stesso linguaggio

$S \rightarrow ee \mid bAQ$ $\quad\quad Q \rightarrow c \mid e$ $\quad\quad A \rightarrow d \mid cA$

Stringa di input: bcde

1. Il primo token dell'input è b; escludo $S \rightarrow ee$ e quindi posso solo applicare la produzione $S \rightarrow bAQ$
2. Secondo token dell'input è c; quindi da A posso solo applicare $A \rightarrow cA$ e ottenere $S \rightarrow bAQ \rightarrow bcAQ$
3. Terzo token dell'input è d; quindi posso solo applicare $A \rightarrow d$ e ottenere $S \rightarrow bAQ \rightarrow bcAQ \rightarrow bcdQ$
4. Quarto token dell'input è e; quindi posso solo applicare $Q \rightarrow e$ e ottenere $S \rightarrow bAQ \rightarrow bcAQ \rightarrow bcdQ \rightarrow bcde$

La grammatica genera lo stesso linguaggio di quella precedente ma il parser può ora generare l'albero di derivazione della stringa 'bcde' senza backtrack

Backtracking: problemi

Il problema del backtracking è in parte nello sviluppo del parser e in parte nello sviluppo del linguaggio.

- L'esempio che abbiamo usato illustra come modificare la grammatica per avere un parser predittivo
- Infatti avevamo una produzione che sembrava promettente ma che aveva una trappola alla fine.
- la grammatica seguente genera lo stesso linguaggio
  $S \rightarrow ee \quad S \rightarrow bAQ \quad Q \rightarrow c \quad Q \rightarrow e \quad A \rightarrow d \quad A \rightarrow cA$
- In questo caso abbiamo fattorizzato il prefisso comune bA e usato un nuovo non terminale Q per permettere la scelta finale tra c ed e
- Questa grammatica genera lo stesso linguaggio di quella precedente ma il parser può ora generare l'albero di derivazione della stringa 'bcde' senza backtrack

Parser predittivi

Adesso vediamo come possiamo fornire al parser (almeno in alcuni casi) la capacità di predire la produzione da applicare

Partiamo da una semplice regola:

- Se nella grammatica al più una produzione inizia con un simbolo non terminale, allora la strategia del parser potrebbe essere facile: tenta le parti destre che iniziano con un terminale e se fallisce tenta quella che inizia con il non terminale.
- Questa strategia è ragionevole ma se abbiamo più produzioni in cui la parte destra inizia con un nonterminale (come, ad esempio $S \rightarrow Ab$ $S \rightarrow Bc$) quale delle due produzoni dobbiamo scegliere?

Parser predittivi

Supponiamo di avere la grammatica seguente con due produzioni da S (S→Ab e S→Bc)

S→Ab S→Bc A→Df A→CA

B→gA B→ e C→dC C→c D→h D→i

La stringa “gchfc” appartiene al linguaggio generato

- Un’analisi intelligente nota che la stringa finisce con ‘c’ quindi devo scegliere S→Bc invece di S→Ab
- Ma questo ragionamento richiede di andare a vedere l’ultimo carattere della stringa e quindi è adatto ad un’analisi bottom-up (non top down!!)

Parser predittivi

Supponiamo di avere la grammatica seguente con due produzioni da S ($S \rightarrow Ab$ e $S \rightarrow Bc$)

$S \rightarrow Ab$ $S \rightarrow Bc$ $A \rightarrow Df$ $A \rightarrow CA$

$B \rightarrow gA$ $B \rightarrow e$ $C \rightarrow dC$ $C \rightarrow c$ $D \rightarrow h$ $D \rightarrow i$

Supponi di scegliere le produzioni nell'ordine come sono scritte

Prima di trovare la sequenza corretta

($S \rightarrow Bc \rightarrow gAc \rightarrow gCAc \rightarrow gcAc \rightarrow gcDfc \rightarrow gchfc$)

Devo fare tanti passi avanti e poi tornare indietro di molti passaggi (quindi analisi sintattica lenta)

Parser predittivi

Come evitare il backtracking?

- Il backtrack potrebbe essere evitato se il parser avesse la capacità di guardare avanti nella grammatica in modo da anticipare quali simboli terminali sono derivabili (mediante derivazioni sinistre) da ciascuno dei vari simboli non terminali nelle parti destre delle produzioni.
- Per esempio, se seguiamo le parti destre della grammatica $S \rightarrow Ab$ $S \rightarrow Bc$ $A \rightarrow Df$ $A \rightarrow CA$ $B \rightarrow gA$ $B \rightarrow e$ $C \rightarrow dC$ $C \rightarrow c$ $D \rightarrow h$ $D \rightarrow i$

  possibili derivazioni sinistre da S sono

  $S \rightarrow Ab \rightarrow Dfb \rightarrow hfb$ $S \rightarrow Ab \rightarrow Dfb \rightarrow ifb$

  $S \rightarrow Ab \rightarrow Cab \rightarrow dCab$ $S \rightarrow Ab \rightarrow Cab \rightarrow cab$ …

Parser predittivi

Data la grammatica

S→Ab S→Bc

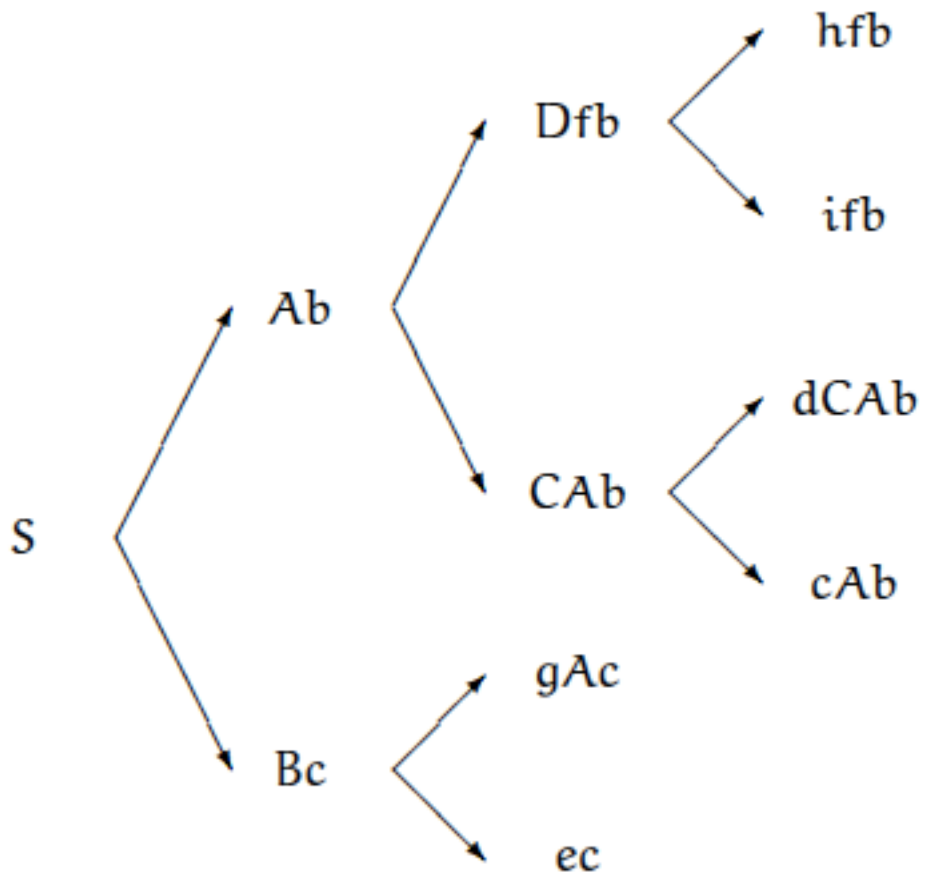
A→Df A→CA

B→gA B→ e C→dC

C→c D→h D→i

Se consideriamo tutte le possibili derivazioni sinistre, troviamo le possibilità mostrate in figura

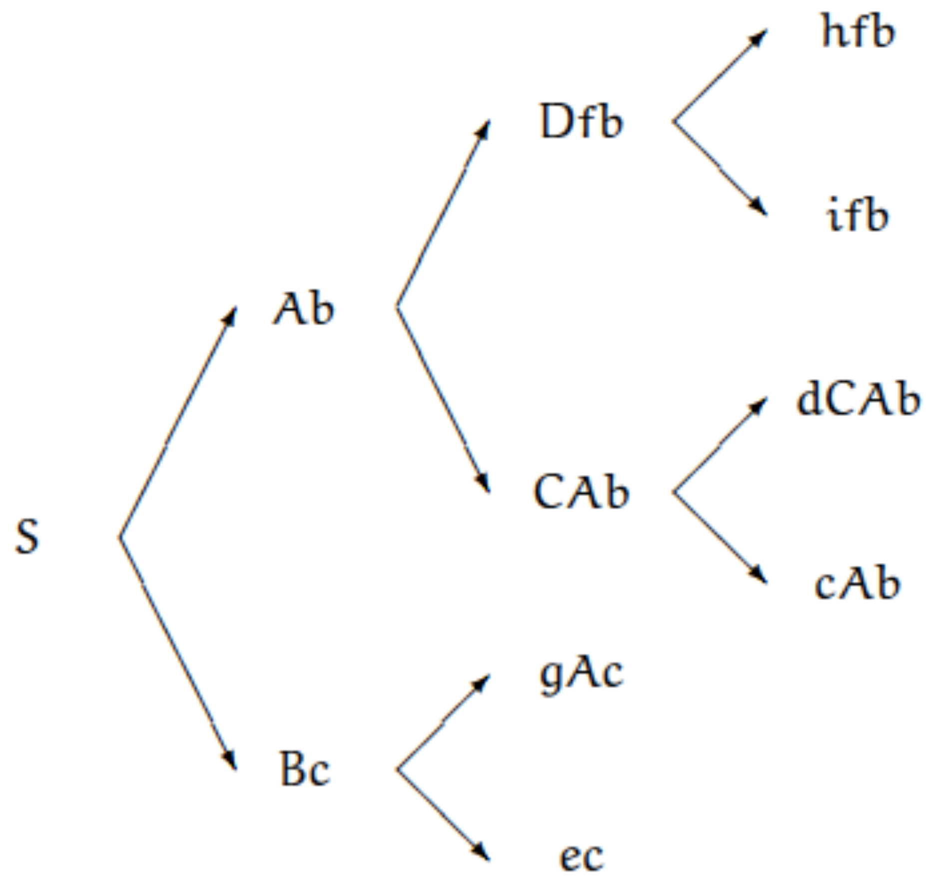
Nota la figura non è un albero di derivazione, ma un albero che mostra tutte le possibili derivazioni a partire dal simbolo S



Parser predittivi

La figura evidenzia che

- se la sequenza di input inizia con c, d, h oppure i dobbiamo scegliere S→Ab
- se inizia con e oppure g dobbiamo scegliere S→Bc
- se inizia con qualcosa di diverso, dobbiamo dichiarare un errore.



Parser predittivi

La figura evidenzia che

- se la sequenza di input inizia con c, d, h oppure i dobbiamo scegliere S→Ab
- se inizia con e oppure g dobbiamo scegliere S→Bc
- se inizia con qualcosa di diverso, dobbiamo dichiarare un errore.

Formalizziamo quanto osservato come segue.
Dato un simbolo nonterminale (nell'es. S) a cui possiamo applicare due produz. (S→Ab | Bc) definiamo FIRST(Ab)= {c,d,h,i} FIRST(Bc)= {e, g}

NOTA: FIRST(Ab) e FIRST(Bc) sono disgiunti
Questo permette al parser di scegliere la produzione da applicare a S:

- S→Ab se il simbolo terminale in input appartiene a FIRST(Ab)
- S→Bc se appartiene a FIRST(Bc)
- Se tale simbolo terminale non appartiene a FIRST(Ab) o a FIRST(Bc), allora la sequenza è grammaticalmente scorretta e può essere rifiutata.

Parser predittivi

Regola Date produzioni $A \rightarrow \alpha$ $A \rightarrow \beta$ $A \rightarrow \gamma$ ….

Calcoliamo FIRST(α), FIRST(β), FIRST(γ)….

- Se questi insiemi sono a due due disgiunti possiamo decidere quale produzione applicare a A esaminando un solo token t (il token corrente)
- Altrimenti non possiamo decidere quale produzione scegliere; se t appartiene a FIRST(α) e a FIRST(β) in presenza di t quale produzione scegliere $A \rightarrow \alpha$ o $A \rightarrow \beta$?

Cosa vedremo nel seguito

1. Come calcolare FIRST()
2. Cosa possiamo fare quando gli insiemi FIRST() **NON** sono a due due disgiunti

Definizione insieme First()

Definizione: Data una grammatica G=(V,N, S,P) - V simboli terminali, N simboli non terminali, definiamo la funzione

$$FIRST : (V \cup N)^{+} \rightarrow 2^{V}$$

- FIRST definisce un insieme di simboli terminali a partire da una stringa α (sequenza di simboli terminali e non)
- In particolare dato α, se X è l'insieme di tutte le forme β derivabili da α mediante derivazioni sinistre, allora, per ogni stringa β in X che inizia con un terminale x, x appartiene a FIRST(α).
- se la stringa λ (stringa vuota) è generabile a partire da α allora λ appartiene a FIRST(α)
- Negli esempi precedenti

  FIRST(Ab)= {c,d,h,i}   FIRST(Bc)= {e, g}

Definizione insieme First()

Esempio

Supponiamo che la grammatica includa le seguenti produzioni:

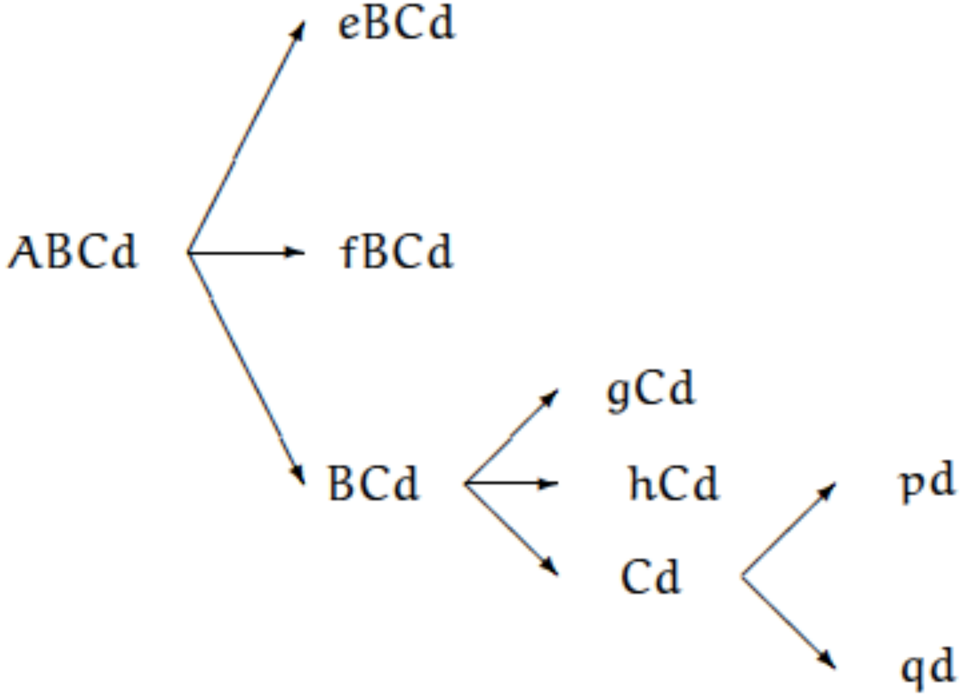
$S \rightarrow ABCd$ $A \rightarrow e$ $A \rightarrow f$ $A \rightarrow \lambda$

$B \rightarrow g$ $B \rightarrow h$ $B \rightarrow \lambda$ $C \rightarrow p$ $C \rightarrow q$

Vogliamo trovare
FIRST(S) = FIRST(ABCd)

Esplorando ABCd
troviamo FIRST(ABCd) =
{e,f,g,h,p,q}

Vedi figura



Definizione insieme First()

Definizione: Data una grammatica G=(V,N, S,P) - V simboli terminali, N simboli non terminali, definiamo la funzione

FIRST : $(V \cup N)^{+} \rightarrow 2^{V}$

Problema: Vogliamo calcolare FIRST in modo automatico
A tale scopo utilizziamo le seguenti proprietà di FIRST:

1. se inizia con un terminale x, allora FIRST(α) = x;
2. FIRST(λ) = {λ} (λ stringa vuota)
3. se α inizia con un non terminale A, allora FIRST(α) include FIRST(A) - {λ}

Intuizione: 1 e 3 sopra implicano che:
Se $\alpha = A\alpha$ e abbiamo le produz. $A \rightarrow b\beta_1 \mid c\beta_2 \mid D\beta_3 \mid E\beta_4 \ldots$
(b,c simboli terminali, D e E non terminali) allora

FIRST(α)= {b} U {c} U FIRST(D) U FIRST(E) U…

Parser predittivi

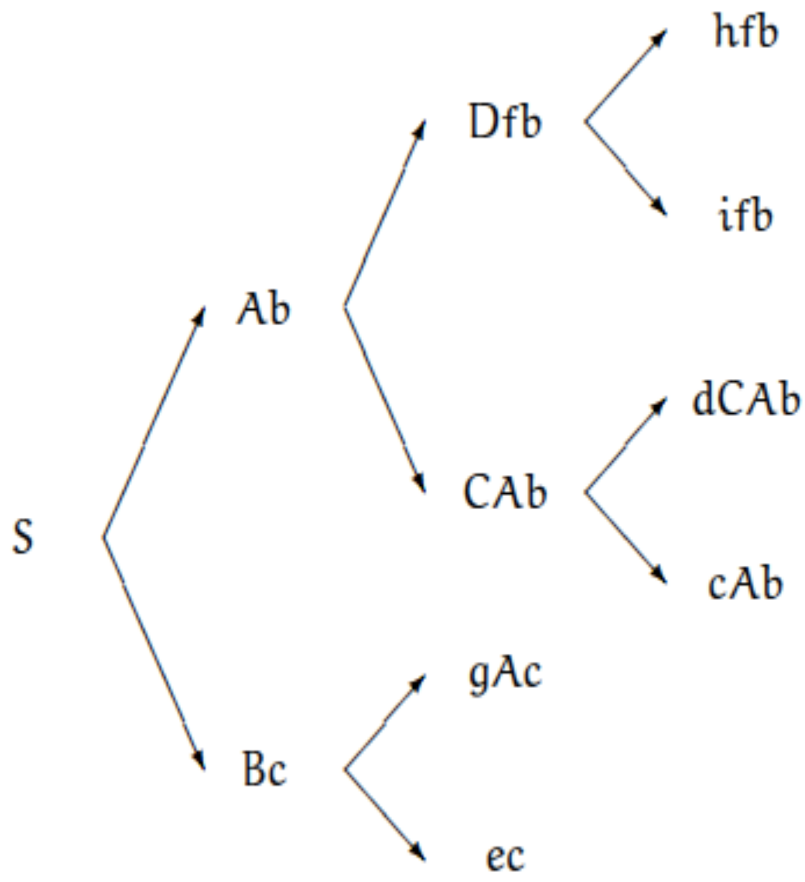
1. se inizia con un terminale x, allora FIRST(α) = x;
2. FIRST(λ) = {λ}
3. se α inizia con un non terminale A, allora FIRST(α) include FIRST(A) - {λ}

Applicando le regole otteniamo un algoritmo per il calcolo di FIRST

Esempio: Calcolo di

- FIRST(Ab)=FIRST(Dfb) U FIRST(Cab)= FIRST(hfb) U FIRST(ifb) U FIRST(dCAb) U FIRSTA(cAb) ={h,i,d,c}
- FIRST(Bc)= FIRST(gAc) U FIRST(ec)= FIRST(gAc) U FIRST(ec)= {g,e}
- FIRST(S)= FIRST(Ab) U FIRST(Bc)= … {h,i,d,c,g,e}

Nota ritroviamo quanto visto in precedenza



Definizione insieme First()

A tale scopo utilizziamo le seguenti proprietà di FIRST:

1. se inizia con un terminale x, allora FIRST(α) = x;
2. FIRST(λ) = {λ} (λ stringa vuota)
3. se α inizia con un non terminale A, allora FIRST(α) include FIRST(A) - {λ}

NOTA La terza proprietà contiene una trappola nascosta. Supponiamo che α=ABδ (δ stringa di simboli terminali e non) e che sia possibile generare λ a partire da A.

Allora, per calcolare FIRST(α), dobbiamo anche seguire le possibilità a partire da B

Inoltre, se è possibile generare λ anche a partire da B, allora dobbiamo anche seguire le possibilità di partire da δ

Definizione insieme First()

Esempio

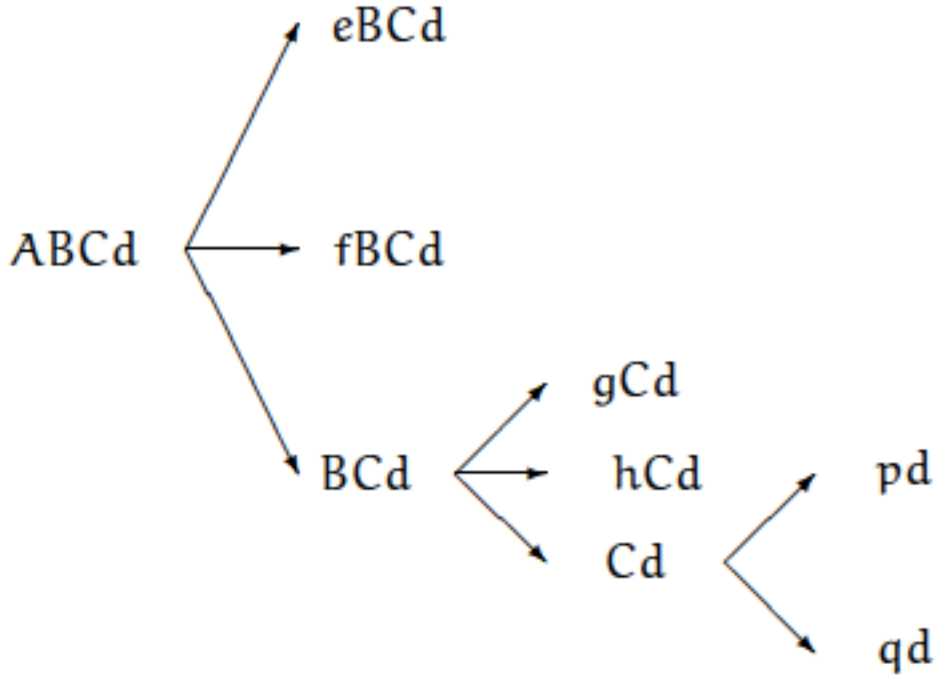
Supponiamo che la grammatica includa le produzioni:

S→ ABCd

A→e   A→f   A→ λ

B→g   B→h   B→ λ

C→p   C→q



Vogliamo trovare FIRST(S) =FIRST(ABCd)

Esplorando questa forma sentenziale, abbiamo (vedi figura)

FIRST(ABCd) = {e} U {f} U FIRST(B) = {e, f} U FIRST(B)

={e, f} U {g} U {h} U FIRST(C) = {e,f,g,h} U {p} U {q}= {e,f,g,h, p, q}

Parser LL(1)

Parser LL(): Parsing predittivo top-down che analizza l'input da sinistra (Left) a destra e costruisce una derivazione sinistra (Leftmost)

- LL(k) usa **k token per decidere quale regola usare;** in pratica usiamo LL(1), k=1
- Data una produzione $A \rightarrow \alpha$, sia FIRST(α) l'insieme dei simboli terminali che possono essere all'inizio di α
- Una grammatica è LL(1) se, per ogni simbolo non terminale A, con produzioni $A \rightarrow \alpha \mid \beta$, allora $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- In questo modo leggendo un solo simbolo terminale decidiamo quale produzione applicare

Grammatiche LL(1)

- Se una grammatica è LL(1) allora possiamo costruire un parser predittivo esaminando un unico simbolo!
- dato $A \rightarrow \alpha|\beta$ possiamo decidere se applicare la prima o la seconda produzione a seconda se il simbolo di input in esame appartenga a FIRST(α) o a FIRST(β)
- **Problema** se esistono λ produzioni $A \rightarrow \lambda$ possiamo avere problemi (tra poco)

Esempio grammatica LL(1)

Esempio (banale)

Consideriamo il caso di espressioni aritmetiche completamente parentesizzate con numeri composti da una sola cifra

- Espressione → digit | ‘(‘espressione operatore espressione ‘)’
- operatore → ‘+’ | ‘*’
- digit → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

- ((3+5)*2) e (2+3) appartengono al linguaggio
- 3+5, (3+5)*2 (123 + 12) NON appartengono al linguaggio

Questa grammatica è LL(1) :
se simbolo che leggi è ‘(‘ la produzione da applicare è
Espressione → ‘(‘Espressione operatore Espressione ‘)’
Altrimenti la produzione da applicare è Espressione → digit

Esempio Grammatica non LL(1)

Esempio (banale)
Grammatica non LL(1)

- $S \rightarrow A\ c \mid B\ d$
- $A \rightarrow a$
- $B \rightarrow a$

Da S posso ottenere ac oppure ad

Questa non è LL(1) perché leggendo il carattere a non sappiamo decidere se applicare S→ Ac o S→Bd

α	First(α)
a	{a}
c	{c}
d	{d}
A	{a}
B	{a}
S	{a}
Ac	{a}
Bd	{a}

Approccio imperativo

- Si può basare il parser su una tabella di parsing avente
  $|V_N|$ linee e $|V_T|$ colonne
- Ciascuna linea è associata a un non terminale, ciascuna colonna a un token: elemento (A,t) rappresenta la produzione da applicare al simbolo terminale A quando token in lettura è t
- Nella casella (A, t) si inserisce la produzione

  $A \rightarrow \alpha$ se $t \in TABLE(A \rightarrow \alpha)$
- Caselle vuote corrispondono a errori sintattici

un semplice esempio

$$E \rightarrow (E) \mid id$$

	(	)	ID	$
E	E → (E)		E → id	

Un altro semplice esempio

Simboli nonterminali: esp, oper digit

esp → digit | (espr oper espr) oper → + | *

digit → 0 | 1| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

	(	+	*	0	1	2	3	…
esp	(esp oper esp)			id	id	id	id	
oper		+	*					
digit				0	1	2	3	…

Righe: simboli non terminali **Colonne: simboli terminali**

- Elemento (Esp, ')') : indica che se il simbolo corrente è '(' e il non terminale è Esp esegui esp → (esp oper esp)
- Elemento (oper, +): indica che se il simbolo corrente è + e il non terminale è oper esegui oper → +
- Elemento (esp, +) manca indica che se il simbolo corrente è esp e il non terminale è + allora errore

Parser Top-down efficienti

Per decidere se una grammatica è LL(1) dobbiamo calcolare gli insiemi First()

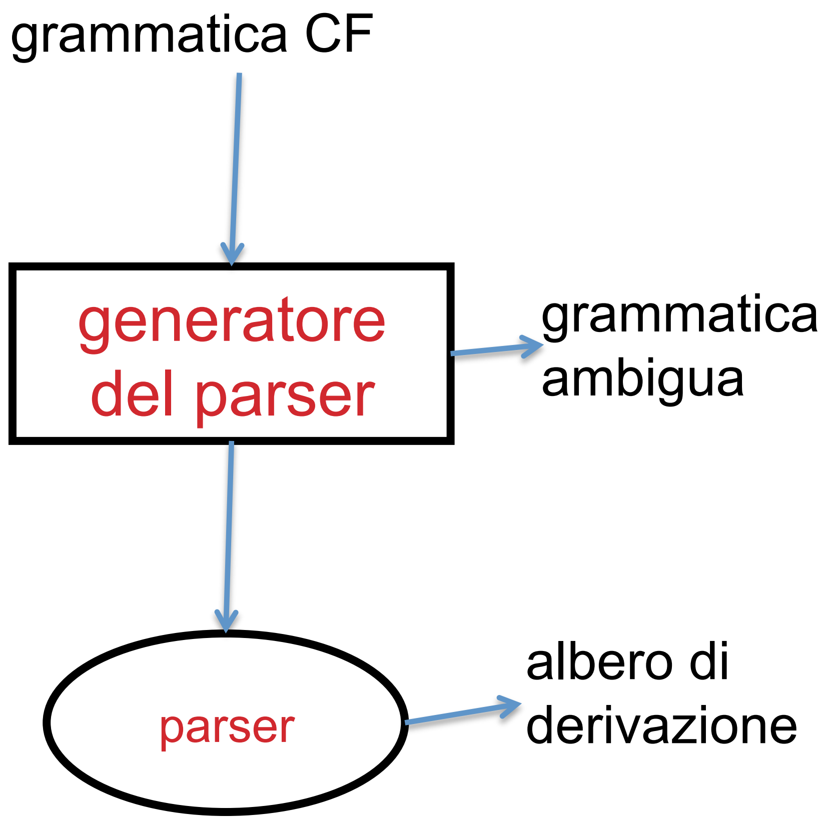
Questa operazione è complicata nel caso di produzioni del tipo

- $A \rightarrow \lambda$ (produzioni vuote)
- $A \rightarrow A\alpha$

  (produzioni con ricorsione)

Ricorda

- Per avere parser top-down dobbiamo eliminare produzioni ricorsive
- Per eliminare produzioni ricorsive si introducono produzioni vuote.



LL(1) parser: primo tentativo

1. Se la grammatica non è LL(1) segnala errore (omesso)
2. Costruisci gli insiemi FIRST()
3. Procedi costruendo un parser a discesa ricorsiva
   - definisci un metodo per ciascun non terminale A
   - per ogni token $t \in FIRST(\alpha)$ usa la regola $A \rightarrow \alpha$

- La tecnica basata sul calcolo della funzione FIRST() che essa verifichi opportune condizioni. Ad esempio se $S \rightarrow A|B$ e il terminale 'a' appartiene sia a FIRST(A) che a FIRST(B) quando abbiamo 'a' in input cosa scegliamo $S \rightarrow A$ o $S \rightarrow B$?
- **Problema** Se la stringa vuota $\lambda \in FIRST(\alpha)$ si possono "perdere" predizioni.

Parser predittivi

Problema Se la stringa vuota $\lambda \in$ FIRST(α) si possono "perdere" predizioni.
Infatti se $\lambda \in$ FIRST(α) il parser dovrebbe poter usare la produzione $A \rightarrow \alpha$ anche nel caso in cui il prossimo carattere in input appartiene all'insieme dei simboli che potrebbero seguire A

Esempio: $S \rightarrow Ab \mid Bc$ $\quad A \rightarrow e \mid \lambda$ $\quad B \rightarrow f$
Se consideriamo S abbiamo FIRST(Ab)={e} FIRST(Bc)={f}; ma se da A otteniamo λ allora dobbiamo scegliere $S \rightarrow Ab$ anche quando il token corrente è b! (gli insiemi FIRST non ci dicono quando scegliere $A \rightarrow \lambda$)

- Conclusione: quando ci sono λ produzioni (produzioni del tipo $A \rightarrow \lambda$, λ stringa vuota) insiemi FIRST non bastano
- Ricorda: la rimozione della ricorsione sinistra, necessaria per parser top-down, introduce λ produzioni

Insieme FOLLOW

- Abbiamo visto che la rimozione della ricorsione sinistra introduce λ produzioni (produzioni del tipo $A \rightarrow \lambda$, λ stringa vuota
- In questi casi per decidere quando scegliere $A \rightarrow \lambda$ dobbiamo calcolare quali simboli possono seguire A
- FOLLOW(A) denota l'insieme dei simboli che possono seguire A

Esempio

Dato $S \rightarrow aAb$ $A \rightarrow cD \mid \lambda$

Abbiamo FIRST(A)= c e FOLLOW(A)= b

Se siamo su A e abbiamo in input c allora applichiamo $A \rightarrow cD$

se abbiamo in input b allora applichiamo $A \rightarrow \lambda$

Calcolo di insieme Follow

Insiemi FOLLOW si calcolano solo per i non terminali

- inizializzazione: $FOLLOW(S) = \{ \$ \}$ (S simbolo iniziale, \$ denota fine stringa), $FOLLOW(A) = \emptyset$ per $A \neq S$
- per calcolare $FOLLOW(B)$, B non terminale, individua le produzioni ove B compare nella parte destra
- per ciascuna produzione del tipo $X \rightarrow \alpha B \beta$
  $FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) \setminus \{ \lambda \})$
- per ciascuna produzione del tipo $X \rightarrow \alpha B \beta$ se $FIRST(\beta)$ può generare la stringa vuota λ
  $FOLLOW(B) = FOLLOW(B) \cup FOLLOW(X)$
- per ciascuna produzione del tipo $X \rightarrow \alpha B$
  $FOLLOW(B) = FOLLOW(B) \cup FOLLOW(X)$

Condizioni per grammatica LL(1)

Funzione TABLE()

Per ogni produzione $A \rightarrow \alpha$, definiamo TABLE ($A \rightarrow \alpha$) come

- FIRST(α) U FOLLOW(A), se $\lambda \in$ FIRST(α) [α si può annullare]
- FIRST(α), se $\lambda \notin$ FIRST(α) [α non si può annullare]

Definizione di grammatica LL(1)

- Una grammatica CF è LL(1) se, per ogni coppia di produzioni sullo stesso non terminale $A \rightarrow \alpha$ e $A \rightarrow \beta$,
- TABLE($A \rightarrow \alpha$) $\cap$ TABLE ($A \rightarrow \beta$) = $\emptyset$

  (Generalizza quanto visto prima per FIRST)

LL(1): esempio

Consideriamo la grammatica delle espressioni riformulata dopo avere eliminato le ricorsioni sinistre.

$E \rightarrow TQ \quad Q \rightarrow +TQ \quad Q \rightarrow -TQ \quad Q \rightarrow \lambda \quad T \rightarrow FR$

$R \rightarrow *FR \quad R \rightarrow /FR \quad R \rightarrow \lambda \quad F \rightarrow (E) \quad F \rightarrow id$

Esempio input id+id Abbiamo $E \rightarrow TQ \rightarrow FRQ \rightarrow id\ R\ Q \rightarrow id\ Q \rightarrow id + TQ \rightarrow id + FRQ \rightarrow id+id\ RQ \rightarrow id+id\ Q \rightarrow id+id$

Gli insiemi FIRST e FOLLOW sono:

FIRST(E) = FIRST(T) = FIRST(TQ) = FIRST(FR) = {(, id}

FIRST(Q) = {+,-, ε} FIRST(R) = {*,/ , ε} FIRST(+TQ) = {+}

FIRST(-TQQ) = {-} FIRST(*RF) = {*} FIRST(/RF) = {/}

FOLLOW(E) = FOLLOW(Q) = { $,)} FOLLOW(F) = {+,-,*, /,$,)}

FOLLOW(T) = FOLLOW(R) = {+,-,*, $,)}

Dati gli insiemi FIRST e FOLLOW :

FIRST(E) = FIRST(F) = FIRST(T) = FIRST(TQ) = FIRST(FR) = {(, id}

FIRST(Q) = {+,-, ε} FIRST(R) = {*,/ , ε} FIRST(+TQ) = {+}

FIRST(-TQQ) = {-} FIRST(*RF) = {*} FIRST(/RF) = {/}

FOLLOW(E) = FOLLOW(Q) = { $,)} FOLLOW(F) = {+,-,*, /,$,)}

FOLLOW(T) = FOLLOW(R) = {+,-,*, $,)}

Costruiamo la seguente tabella TABLE

Simbolo terminale letto (token)

Sim-bolo non term.

	id	+	-	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	∗FR	/FR		ϵ	ϵ
F	**id**					(E)		

LL(1) parser: primo tentativo

1. Costruisci i FIRST-SET
2. Se grammatica non LL(1) errore
3. Procedi costruendo un parser predittivo a discesa ricorsiva
   - **un metodo per ciascun non terminale A**
   - **per ogni token t $\in$ FIRST(α) usa la regola A $\rightarrow$ α**

- Problema: se $\epsilon \in$ FIRST(α) si possono "perdere" predizioni.
- In pratica: il parser dovrebbe poter usare la produzione A $\rightarrow$ α se $\epsilon \in$ FIRST(α) e il prossimo carattere in input appartiene all'insieme dei simboli che potrebbero seguire A

LL(1) parser: primo tentativo

Racchiudere ogni insieme di produzioni a partire da un simbolo non terminale in una funzione Booleana.

Si scrive una funzione per ogni non terminale della grammatica. La funzione tenta ciascuna parte destra fino a che una corrispondenza non viene trovata

Nel caso di un parser LL(1) ciascuna di queste funzioni sceglie la parte destra in base agli insiemi FIRST e FOLLOW.

Algoritmo

1. Costruisci i insiemi FIRST e FOLLOW
2. Se grammatica non LL(1) dai errore
3. Procedi costruendo un parser predittivo a discesa ricorsiva
   - **un metodo per ciascun non terminale A**
   - **per ogni token t $\in$ FIRST(α) usa la regola A $\rightarrow$ α**

esempio "Statements"

Grammatica LL(1)

stmt → id = exp ;
    | return exp ;
    | if (exp) stmt
    | while (exp) stmt

metodo

```
// parse stmt → id=exp; | …
void stmt( ) {
  switch(nextToken) {
     RETURN: returnStmt(); break;
     IF:  ifStmt(); break;
     WHILE: whileStmt(); break;
     ID: assignStmt(); break;
  }
}
```

esempio (cont)

while

```
// parse while (exp) stmt
void whileStmt() {
    // skip “while (”
    getNextToken();
    getNextToken();
    // parse condition
    exp();
    // skip “)”
    getNextToken();
    // parse stmt
    stmt();
}
```

return

```
// parse return exp ;
void returnStmt() {
    // skip “return”
    getNextToken();
    // parse expression
    exp();
    // skip “;”
    getNextToken();
}
```

Possibili problemi

Ricorsioni sinistre

- es., $E \rightarrow T \mid E + T \mid \ldots$

Parti destre di produzioni sullo stesso non terminale aventi un prefisso comune

- es., $A \rightarrow aBC \quad A \rightarrow aCD \mid \ldots$

Rendono le grammatiche non LL(1)!

ricorsione sinistra

produzioni

expr → expr + term
    | term

codice (errato!)

```
// parse expr → ...
void expr () {
    expr() ;
    if (current token is
            PLUS) {
        getNextToken();
        term();
    }
}
```

tentativo di soluzione

- sostituire la ricorsione sinistra con una destra

  $E \rightarrow T + E \mid T$ prefisso comune! (non LL(1))

- diviene

  $E \rightarrow T\,Q$

  $Q \rightarrow +\,T\,Q \mid \lambda$ codificabile direttamente!

risoluzione di ricorsioni sinistre

- Produzioni del tipo $A \rightarrow A\alpha \mid \beta$, ove β non inizia con A, costruiscono stringhe del tipo $\beta\alpha\alpha\ldots\alpha$
- Sono generabili da
- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A' \mid \epsilon$

- Più in generale
- $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_r \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_s$
- si risolvono con
- $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_s A'$
- $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_r A' \mid \epsilon$

codice per le espressioni (1)

```
// parse
//  expr →  term { + term }*
void expr() {
   term();
   while (next symbol is
    PLUS)
    {
    getNextToken();
     term()
    }
}
```

```
// parse
//  term → factor { * factor }*
void term() {
   factor();
   while (next symbol is
    TIMES)
    {
    getNextToken();
    factor()
    }
}
```

Codice per le espressioni（2）

```
// parse
//    factor → int | id | ( expr )
void factor() {
  switch(nextToken) {
    case INT:
        process int constant;
        getNextToken();
        break;
    …
    case ID:
        process identifier;
        getNextToken();
        break;
    case LPAREN:
        getNextToken();
        E();
        getNextToken();
    }
}
```

prefissi comuni

- Se esistono due produzioni sullo stesso non terminale con un prefisso comune
- $A \rightarrow \alpha\ \beta$ $\qquad$ $A \rightarrow \alpha\ \gamma$
- si può usare la fattorizzazione

- $A \rightarrow \alpha\ A'$
- $A' \rightarrow \beta \mid \gamma$

esempio istruzione if

- grammatica originale

  ifStmt → if (expr) stmt |

  if (expr) stmt else stmt

- grammatica fattorizzata

  ifStmt → if (expr) stmt ifTail

  ifTail → else stmt | λ

Parsing di istruzioni if

può essere più semplice codificare direttamente la regola "else assegnato all'ultimo if aperto"

```
// parse
//    if (expr) stmt [ else stmt ]
void ifStmt (){
    getNextToken();
    getNextToken();
    expr();
    getNextToken();
    stmt();
    if (next symbol is ELSE) {
        getNextToken();
        stmt();
    }
}
```

Progetto Analizzatore sintattico

Con il metodo visto dobbiamo scrivere una funzione per ogni produzione. Se la grammatica cambia dobbiamo riprogrammare una o più di queste funzioni.

Approccio più semplice: utilizzare una procedura di controllo che utilizza una tabella table

- La tabella ha una riga per ogni simbolo non terminale e una colonna per ogni simbolo terminale e per $
- La tabella dice quale parte destra scegliere e i simboli terminali sono usati nel modo naturale.

La tabella può essere costruita a mano o mediante un programma per grammatiche grandi. Se la grammatica cambia, solo la tabella deve essere riscritta

Algoritmo di analisi sintattica

Algoritmo utilizza una tavola table[,] e una pila

1. Inizializzazione: Inserire in pila $ (fine stringa) e poi simbolo iniziale e $ alla fine della sequenza di input;
2. Fintantochè la pila non è vuota
   Sia x l'elemento in cima alla pila e a il simbolo terminale in input.
   – Se $x \in V$ (x simbolo terminale) allora:
   se x = a (input coerente) estrai x dalla pila e avanza di un simbolo terminale, altrimenti segnala ERRORE
   – Se $x \in N$ (x simbolo non terminale), allora:
   se table[x, a] non è vuoto, estrai x dalla pila e inserisci table[x,a] nella pila ***in ordine inverso***, altrimenti segnala ERRORE

Esempio: parser per espressioni aritmetiche

Se A è simbolo non terminale in cima alla pila allora bisogna applicare una produzione ad A (ricorda derivazione sinistra)
Applicare la produzione $A \rightarrow \alpha$ equivale sostituire α ad A

La Tavola fornisce le informazioni su quale produzione applicare esaminando solo il simbolo terminale in input (assumiamo che la grammatica sia LL(1))

Possibili casi di casella non vuota

- la casella ha sequenza di simboli (terminali e non) : inserisci gli elementi in ordine inverso (in questo modo il primo simbolo di α e' in cima alla pila)
- la casella ha $\varepsilon \rightarrow$ elimina elemento dalla pila (applichiamo una λ produzione) (ε nessun carattere)

Esempio: parser per espressioni aritmetiche

Consideriamo la grammatica ottenuta dopo aver avere eliminato le ricorsioni sinistre:

$E \rightarrow TQ$	$Q \rightarrow +TQ$	$Q \rightarrow -TQ$	$Q \rightarrow \lambda$
$T \rightarrow FR$	$R \rightarrow * FR$	$R \rightarrow / FR$	$R \rightarrow \lambda$
$F \rightarrow (E)$	$F \rightarrow id$		

Vediamo come usare la tabella (poi come costrurla)

Simbolo terminale letto

Sim-bolo non term.

	id	**+**	**-**	*****	**/**	**(**	**)**	**$**
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*$FR	/FR		ϵ	ϵ
F	**id**					(E)		

Esempio: parser per espressioni aritmetiche

- Ogni elemento non vuoto della tabella indica quale produzione scegliere trovandosi a dover espandere il simbolo non terminale che etichetta la riga e leggendo in input il simbolo terminale che etichetta la colonna della tabella.
- Es. per espandere Q leggendo + scelgo Q→+TQ

Simbolo terminale letto

Sim-bolo non term.

	id	+	-	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	∗FR	/FR		ϵ	ϵ
F	id					(E)		

Esempio: parser per espressioni aritmetiche

- La tabella indica la produzione da scegliere per espandere il simbolo non terminale della riga e leggendo in input il simbolo terminale della colonna
- Se non e' presente ERRORE
- Es. espandere Q leggendo + scelgo Q→+TQ

  espandere Q leggendo * è → ERRORE

Simbolo terminale letto

Sim-bolo non term.

	id	+	-	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	∗FR	/FR		ϵ	ϵ
F	id					(E)		

Simbolo terminale letto

Sim-bolo non term.

	id	+	-	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	∗FR	/FR		ϵ	ϵ
F	id					(E)		

ogni elemento non vuoto della tabella indica quale produzione debba essere scelta dal parser trovandosi a dover espandere il simbolo non terminale che etichetta la riga della tabella e leggendo in input il simbolo terminale che etichetta la colonna della tabella.

Esempi

- se devo espandere non terminale E e in input ho id scelgo $E \rightarrow TQ$
- se devo espandere non terminale E e in input ho + alora Errore!
- se devo espandere non terminale R e in input ho + scelgo $R \rightarrow \lambda$

ESEMPIO Supponiamo che la sequenza in input sia (id+id)*id. Lo stato iniziale sarà il seguente:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	(id+id)*id $		

(PRODUZIONE indica quale produzione usiamo ogni volta; DERIVAZIONE illustra ad ogni passo la derivazione ottenuta finora)
La sequenza ha $ in fondo e la pila ha $ e il simbolo iniziale inseriti (la pila cresce da sinistra verso destra).

Ciclo principale del parser:
Il simbolo in cima alla pila è E e table[E; (] = TQ : quindi applichiamo la produzione E→TQ e otteniamo

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	(id+id)*id $	E→TQ	E→TQ
$QT	(id+id)*id $		

(E è stato estratto dalla pila e la parte destra della produzione TQ inserita nella pila in ordine inverso.

PILA	INPUT
$QT	(id+id)*id $

Ora abbiamo un non terminale T in cima alla pila e il simbolo terminale in input è ancora (

Dato che table[T, ( ] = FR la produzione è T→FR

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	(id+id)*id $	E→TQ	E→TQ
$QT	(id+id)*id $	T →FR	→FRQ
$QRF	(id+id)*id $		

Proseguendo abbiamo table[F, (] = (E) e la produzione è F→(E) :

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	(id+id)*id $	E→TQ	E→TQ
$QT	(id+id)*id $	T →FR	→FRQ
$QRF	(id+id)*id $	F→(E)	→(E)RQ
$QR)E(	(id+id)*id $		

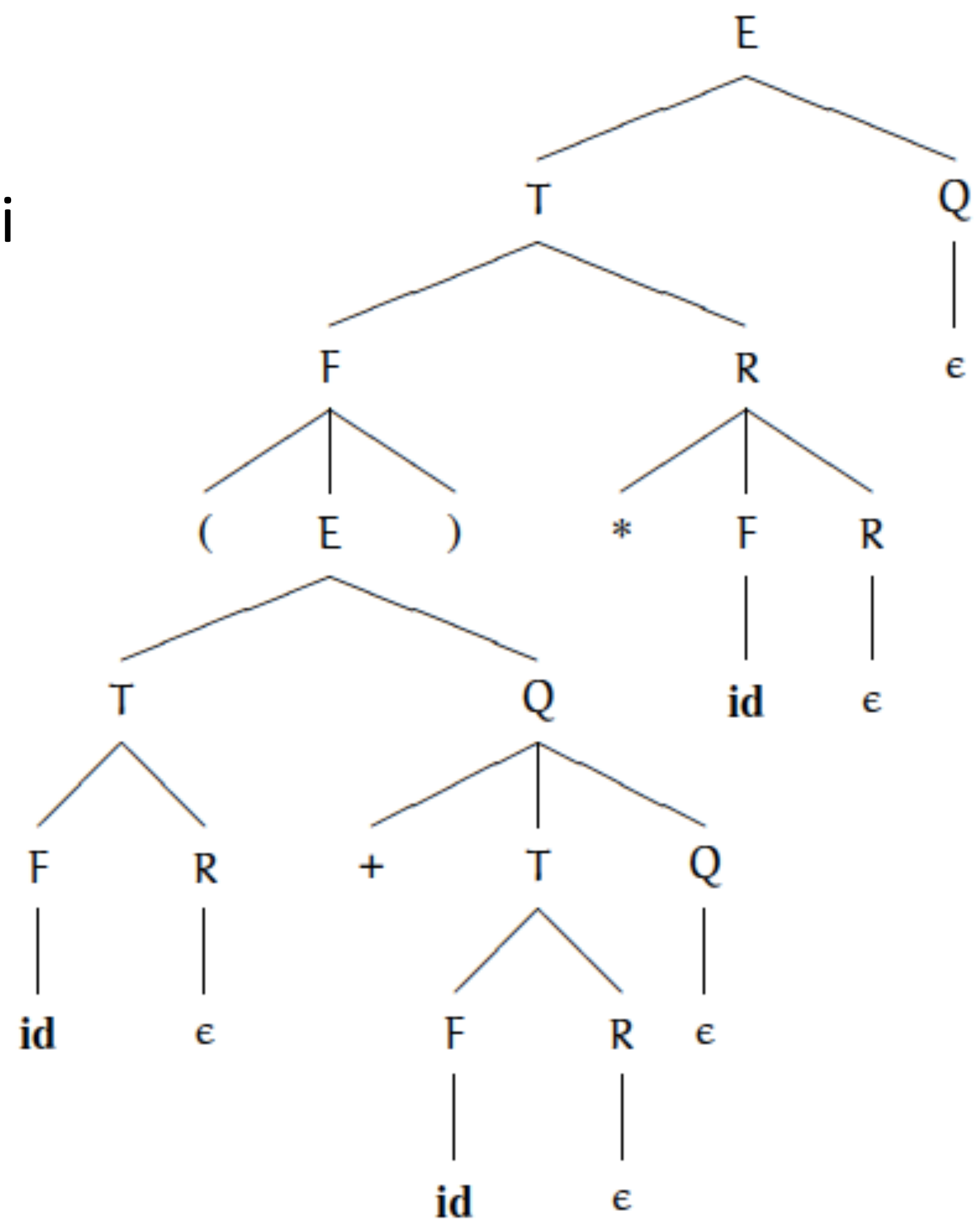
PILA	INPUT
$QR)E(	(id+id)*id $

Ora abbiamo un terminale in cima alla pila. Lo confrontiamo con il simbolo in input e, poiché coincidono, estraiamo il simbolo terminale dalla pila e ci spostiamo al prossimo simbolo della sequenza in input:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	(id+id)*id $	E→TQ	E→TQ
$QT	(id+id)*id $	T →FR	→FRQ
$QRF	(id+id)*id $	F→(E)	→(E)RQ
$QR)E(	(id+id)*id $		
$QR)E	id+id)*id $		

Proseguendo si consuma l’intero input con la pila vuota e possiamo annunciare il successo dell’analisi.
NOTA: alla fine rimane in pila $ e in input $.

Albero di derivazione di (id+id) *id



Costruzione della tabella del parser predittivo

- Supponiamo che X sia in cima alla pila e che a sia il simbolo terminale correntemente in input.
- Vogliamo selezionare una produzione da X la cui parte destra inizia con a oppure possa portare a una forma sentenziale che inizia con a

Nell'esempio all'inizio avevamo E in pila e (come input.

Avevamo bisogno di una produzione della forma E→ (

ma una tale produzione non esiste nella grammatica.

- Poiché non era disponibile, avremmo dovuto tracciare un cammino di derivazione che porta ad una stringa di simboli che inizia con (
- L'unico tale cammino è E→ TQ→ FRQ→ (E)RQ

Vogliamo selezionare una parte destra se il token appartiene a FIRST();

Costruzione della tabella del parser predittivo

Vogliamo selezionare una parte destra se il token appartiene a FIRST();

- quindi per una riga A e una produzione X → α , la tabella deve avere α in ogni colonna etichettata con un terminale in FIRST().
- Ciò funziona in tutti i casi eccetto quello in cui FIRST() include λ - la tabella non ha una colonna etichettata λ
- Per questi casi, seguiamo gli insiemi FOLLOW.

La regola per costruire la tabella è dunque la seguente.

Esamina tutte le produzioni. Sia X→β una di esse.

- Per tutti i terminali a in FIRST(β), poni table[X;a] = β
- Se FIRST(β) include λ , allora, per ogni a in FOLLOW(X), table[X;a] = ε

Costruzione della tabella del parser predittivo

La regola per costruire la tabella è dunque la seguente.

Esamina tutte le produzioni. Sia $X \rightarrow \beta$ una di esse.

- Per tutti i terminali a in FIRST(β), poni table[X;a] = β
- Se FIRST(β) include λ , allora, per ogni a in FOLLOW(X), table[X;a] = ε

Ricorda: definizione di grammatica LL(1)

- Una grammatica di tipo 2 è LL(1) se, per ogni coppia di produzioni sullo stesso non terminale $A \rightarrow \alpha$ e $A \rightarrow \beta$,
- Table[$A \rightarrow \alpha$] $\cap$ table[$A \rightarrow \beta$] = $\emptyset$

Equivalentemente: una grammatica è LL(1) se ogni elemento di table[,] corrisponde ad una sola produzione

La forma di Backus e Naur

- I manuali che descrivono i linguaggi di programmazione di solito usano la notazione nota come BNF (Backus-Naur Form) per descrivere la sintassi di un linguaggio.
- La forma di Backus e Naur è un modo di tipo generativo per definire linguaggi ed `e molto diffusa per specificare la sintassi dei linguaggi di programmazione.
- Nel seguito tra parentesi angolari “<” e “>” indichiamo i simboli non terminali e
- Al posto di → usiamo ::=, e usiamo + per indicare la ripetizione; il significato di | e’ lo stesso (oppure)

La forma di Backus e Naur

Definizione di identificatore

- <lettera>::= a | b | ... z | A | B | ... | Z
- <cifra> ::= 0 | 1 | . . .| 9
- <carattere finale> ::= <lettera> | <cifra>
- <trattino >::= _
- <carattere> ::= <lettera> | <cifra> | <trattino>
- <variabile> ::= <lettera>
- <variabile> ::= <lettera> <carattere> <carattere finale>

Posso dire che 007JamesBond non e' una variabile

Mentre lo è Agente007_James_Bond

NOTA: risulta evidente la similitudine fra la forma di Backus-Naur e la definizione di grammatica che abbiamo considerato