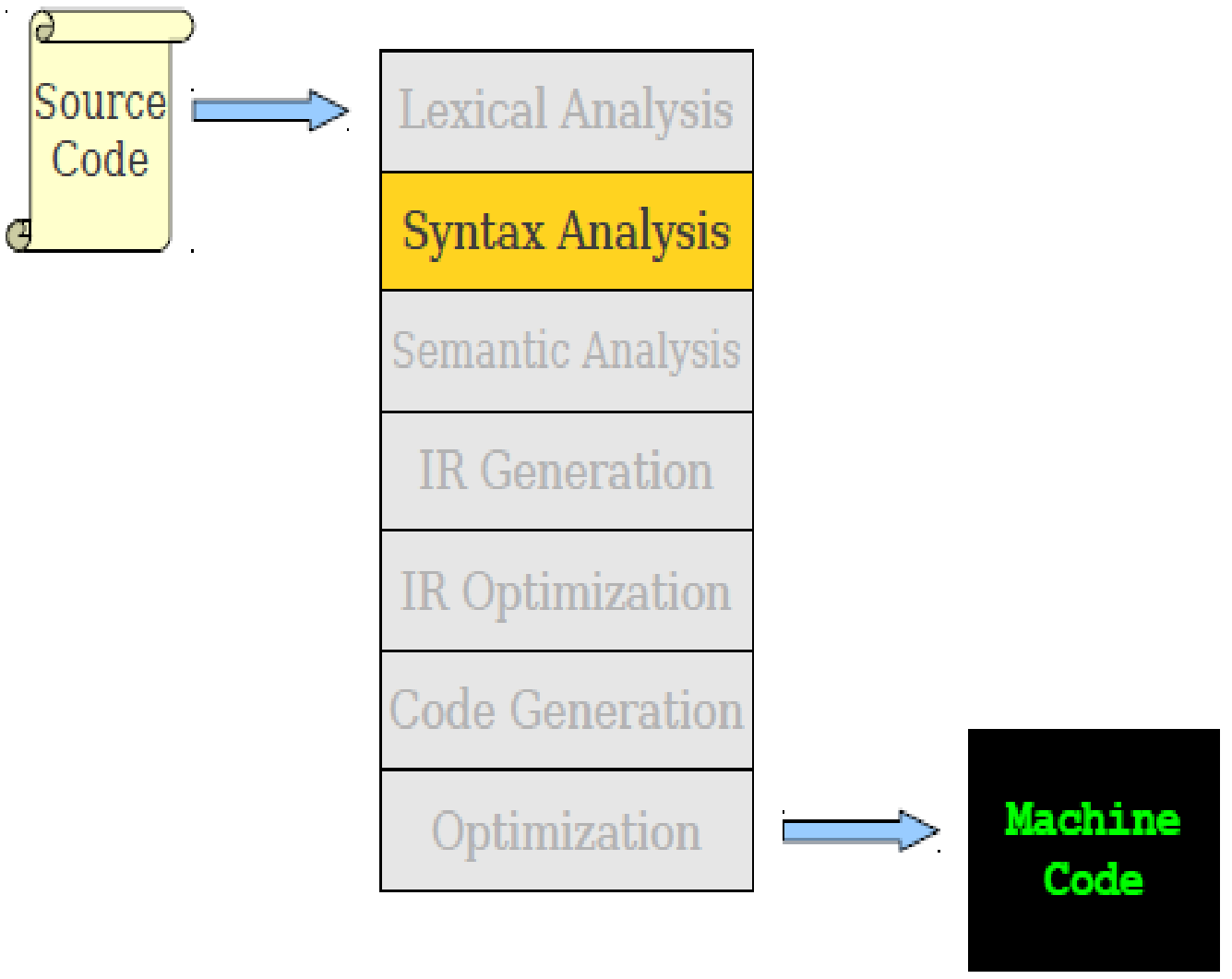
Parser discendenti

Lunedì 12 Novembre

Dove siamo?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Input e output

- INPUT: Sequenza di token prodotti dall'analizzatore lessicale
- OUTPUT: Albero sintattico se la sequenza è generata dalla grammatica CF, altrimenti produce errori sintattici

Esempio di CFG per un linguaggio di programmazione

```
BLOCK  → STMT
       | { STMTS }

STMTS  → ε
       | STMT STMTS

STMT   → EXPR;
       | if (EXPR) BLOCK
       | while (EXPR) BLOCK
       | do BLOCK while (EXPR);
       | BLOCK
       | ...

EXPR   → identifier
       | constant
       | EXPR + EXPR
       | EXPR – EXPR
       | EXPR * EXPR
       | ...
```

Derivazione leftmost o rightmost

- Ad ogni passo si espande il simbolo non terminale più a sinistra, a differenza della rightmost in cui si espande il non terminale più a destra.

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | ...

EXPR  → identifier
      | constant
      | EXPR + EXPR
      | EXPR - EXPR
      | EXPR * EXPR
      | EXPR = EXPR
      | ...
```

```
  STMTS
⇒ STMT STMTS
⇒ EXPR; STMTS
⇒ EXPR = EXPR; STMTS
⇒ id = EXPR; STMTS
⇒ id = EXPR + EXPR; STMTS
⇒ id = id + EXPR; STMTS
⇒ id = id + constant; STMTS
⇒ id = id + constant;
```

Altro esempio di CFG

$E \rightarrow \texttt{int}$

$E \rightarrow E\ Op\ E$

$E \rightarrow \texttt{(}E\texttt{)}$

$Op \rightarrow \texttt{+}$

$Op \rightarrow \texttt{-}$

$Op \rightarrow \texttt{*}$

$Op \rightarrow \texttt{/}$

Esempio

$E \rightarrow \texttt{int} \mid E\ Op\ E \mid (E)$

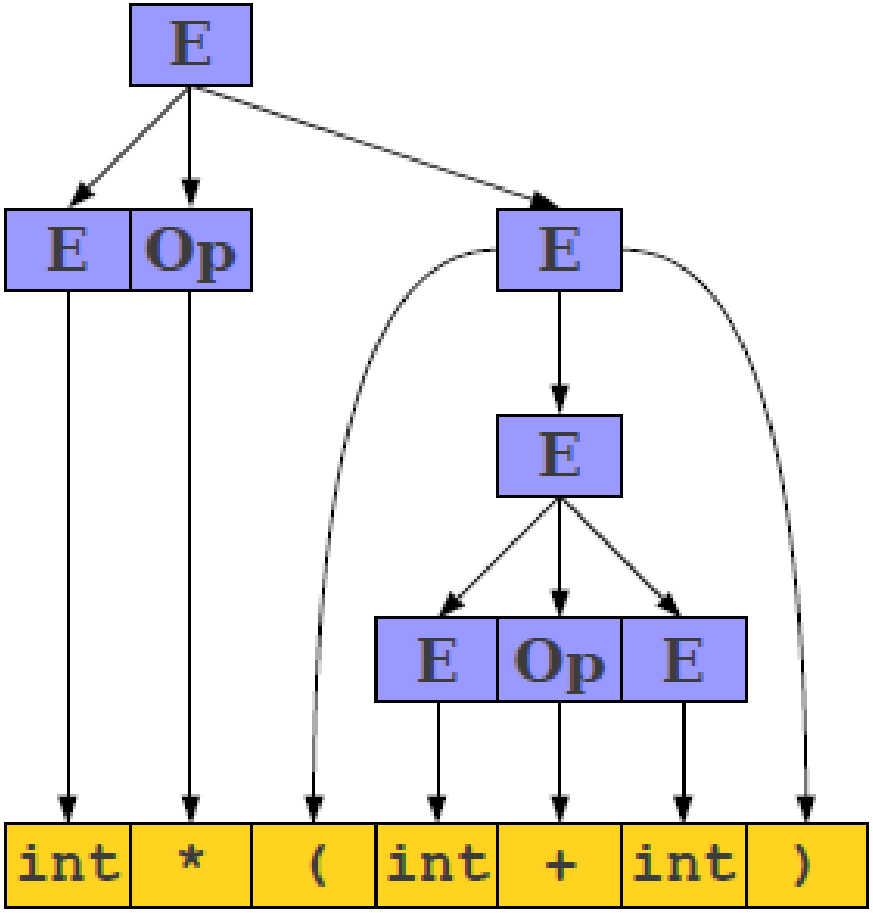
$Op \rightarrow + \mid - \mid * \mid /$

E
$\Rightarrow E\ Op\ E$
$\Rightarrow \texttt{int}\ Op\ E$
$\Rightarrow \texttt{int} * E$
$\Rightarrow \texttt{int} * (E)$
$\Rightarrow \texttt{int} * (E\ Op\ E)$
$\Rightarrow \texttt{int} * (\texttt{int}\ Op\ E)$
$\Rightarrow \texttt{int} * (\texttt{int} + E)$
$\Rightarrow \texttt{int} * (\texttt{int} + \texttt{int})$

E
$\Rightarrow E\ Op\ E$
$\Rightarrow E\ Op\ (E)$
$\Rightarrow E\ Op\ (E\ Op\ E)$
$\Rightarrow E\ Op\ (E\ Op\ \texttt{int})$
$\Rightarrow E\ Op\ (E + \texttt{int})$
$\Rightarrow E\ Op\ (\texttt{int} + \texttt{int})$
$\Rightarrow E * (\texttt{int} + \texttt{int})$
$\Rightarrow \texttt{int} * (\texttt{int} + \texttt{int})$

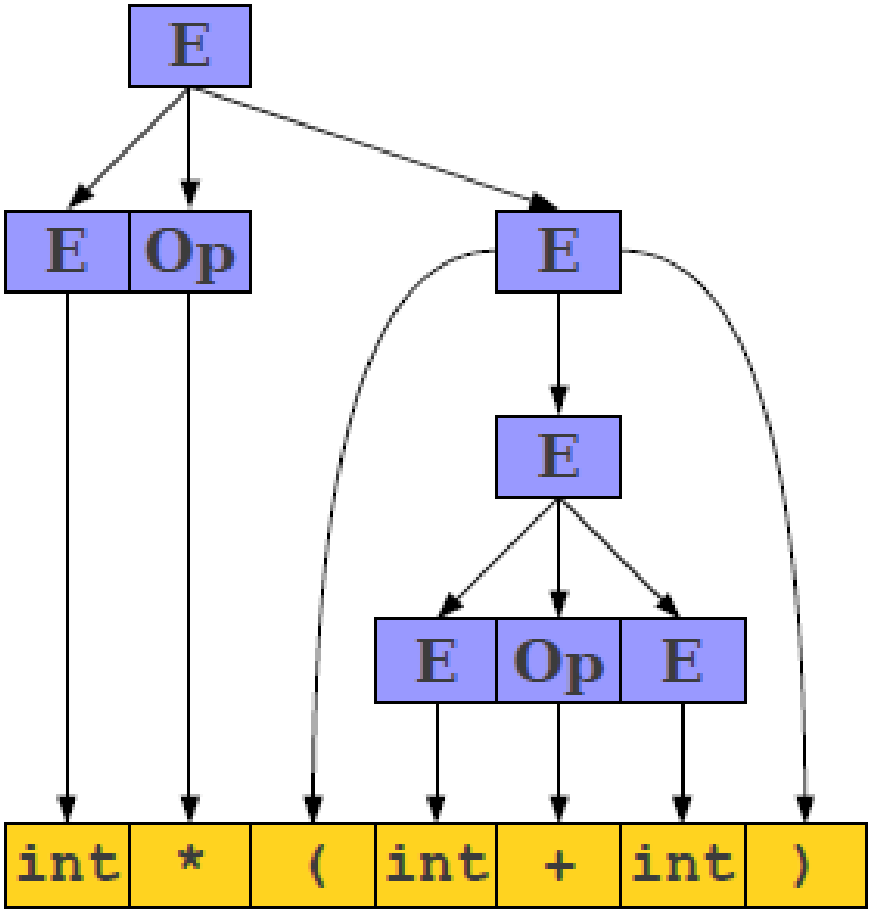
Producono lo stesso syntax tree?

E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)
⇒ int * (int + int)



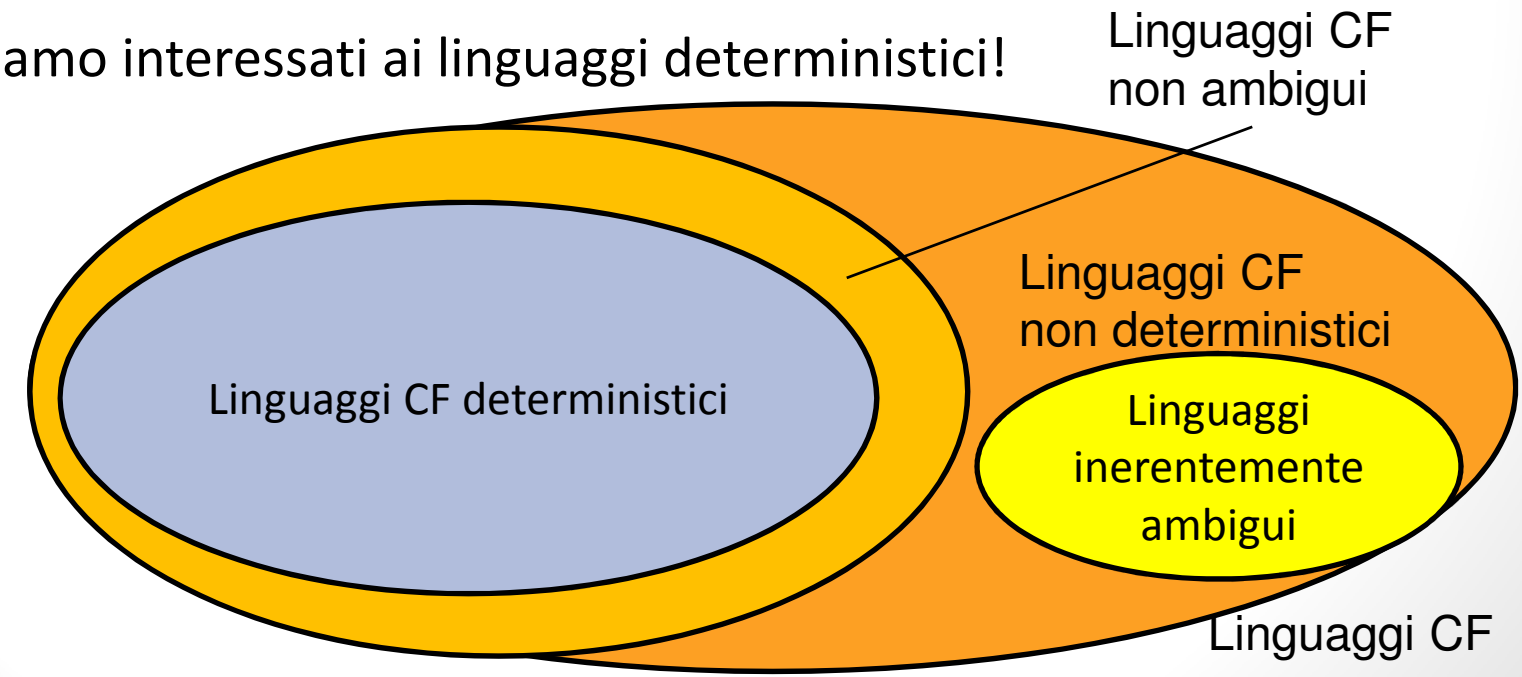
Si!

E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E Op (E Op int)
⇒ E Op (E + int)
⇒ E Op (int + int)
⇒ E * (int + int)
⇒ int * (int + int)



Obiettivo del parser in un compilatore

- Costruire il syntax tree, ovvero quali produzioni vengono applicate piuttosto che l'ordine con cui si applicano.
- Se il linguaggi è non ambiguo, per ogni sequenza esiste un unico syntax tree
- Per l'insieme dei linguaggi non ambigui, il parser deve produrre un unico oggetto, ma potrebbe essere non deterministico.
- Siamo interessati ai linguaggi deterministici!



Parser discendenti e ascendenti

- Considereremo due classi di parser:
  - Discendenti o TOP-DOWN: si costruisce la derivazione partendo dall'assioma; l'albero di derivazione si costruisce dalla radice alle foglie;
  - Ascendenti o BOTTOM-UP: si costruisce la derivazione ma nell'ordine riflesso, cioè l'albero si costruisce dalle foglie alla radice.

Esercizio

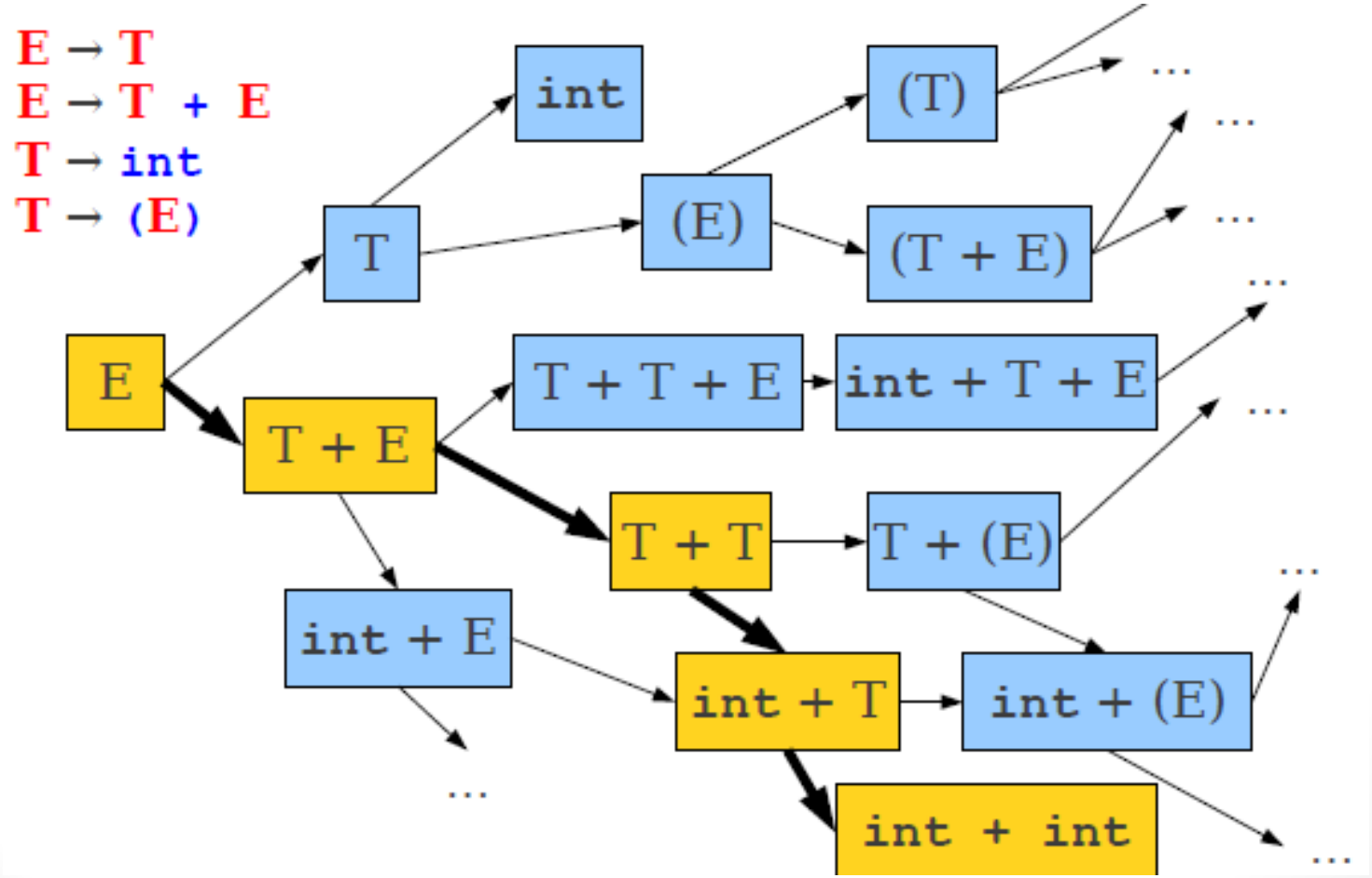
1. S->aSAB
2. S->b
3. A->bA
4. A->a
5. B->cB
6. B->a

La stringa $a^2b^2a^4$ è generata dalla grammatica.
Costruire l'albero sintattico.

Parser top-down: problematiche

- I parser top-down iniziano il loro lavoro senza alcuna informazione iniziale.
- Cominciano con l'assioma, che va bene per tutti I programmi. Quale produzione applicare?
- Si può scommettere su una produzione, se il tentativo si rivela sbagliato si ritorna indietro e si ritenta (backtracking)
- Come si sceglie la produzione?

Il parsing top down è come ricercare un path in un grafo



Due strategie di parser discendenti (o top down)

- PARSER a discesa ricorsiva (possono essere deterministici o non)
- PARSER LL(1) (parser deterministici)

Parser a discesa ricorsiva

L'idea: le regole grammaticali per un non-terminale A sono viste come costituenti una procedura che riconosce un A

- la parte destra di ogni regola specifica la struttura del codice per questa procedura
- la sequenza di terminali e non terminali nelle varie regole corrisponde a un controllo che i terminali siano presenti nell'input e a invocazioni delle procedure dei simboli non terminali
- la presenza di diverse regole per A è modellata da **case** o **if**
- può richiedere backtracking (può richiedere di leggere più di una volta parte della stringa in ingresso, ovvero se l'applicazione di una produzione fallisce può tornare indietro).

Procedura tipica per un parser top-down a discesa ricorsiva

void A(){

scegli, per A, una produzione A->$X_1X_2…X_k$

for (da i a k) {

if (X_i è un non terminale)

richiama X_i();

else if (X_i è uguale al simbolo in input corrente)

procedi al simbolo successivo;

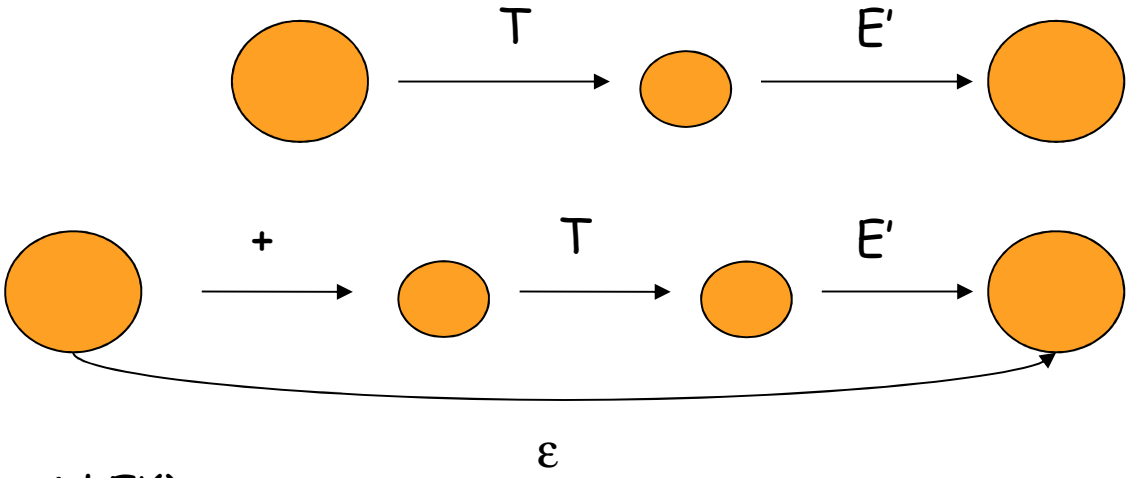
else si è verificato un errore;

}

}

Costruzione delle procedure ricorsive

$F \rightarrow (E)$ |**id**
$E \rightarrow TE'$
$E' \rightarrow +TE'|\varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'|\varepsilon$




```
void E();
{
T();
E'();
}
```

```
void E'();
{
If (tok==+) then
        {avanza input;
         T();
         E'();}
}
```

esempio:

S -> if E then S else S
S -> begin S L
L -> end
L -> ; S L
S -> print E
E -> num = num

token { IF, THEN, ELSE, BEGIN, PRINT, PUNTVIRG, NUM, EQ } ;

```
void S () { /* funzione per S */
if (tok = = IF) then {
avanza(IF) ; E() ; avanza(THEN) ; S() ; avanza(ELSE) ; S() ;
} else if (tok = = BEGIN) {
avanza(BEGIN) ; S() ; L() ;
} else if (tok = = PRINT) {
avanza(PRINT) ; E() ;
} else error() ;
}
void L () { /* funzione per L */
if (tok = = END) then
avanza(END) ;
else if (tok = = PUNTVIRG) {
avanza(PUNTVIRG) ; S() ; L() ;
} else error() ;
}
void E () { /* funzione per E */
avanza(NUM) ; avanza(EQ) ; avanza(NUM) ;
}
```

Trasformazione della grammatica per l'analisi top-down

Due aspetti rendono una grammatica inadatta all'analisi top-down: la ricorsione sinistra e la presenza di prefissi comuni in più parti destre di regole associate allo stesso simbolo non terminale.

- $A \rightarrow y\alpha^1 | \ldots | y\alpha^n$

Rimedio: fattorizzazione sinistra

- $A \rightarrow A$a (A non terminale).

Rimedio: eliminazione ricorsione sinistra

Può essere possibile adattare una grammatica in modo che sia applicabile un parser top-down.

Fattorizzazione sinistra

Idea: quando non è chiaro quale produzione usare per espandere un non terminale *A*, si possono riscrivere le produzioni in modo da "posticipare" la scelta, introducendo un non terminale supplementare.

ES.:

$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \gamma$

fattorizzazione sinistra:

$A \rightarrow \alpha\,A' \mid \gamma$ *A' è un nuovo simbolo non terminale*

$A' \rightarrow \beta_1 \mid \beta_2$

<stmt**>** → **if** <expr> **then** <stmt>
| **if** <expr> **then** <stmt> **else <**stmt**>**

fattorizzazione sinistra:

<stmt> → **if** <expr> **then** <stmt><S>
<S> → ε | **else** <stmt>

Eliminazione ricorsione sinistra

Metodo: date le produzioni "ricorsive sinistre" e non, di un non terminale A:

$A \rightarrow A\ \alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_n$

si sostituiscono con (eliminazione ricorsione sinistra immediata)

$A \rightarrow \beta_1 A' | \ldots | \beta_n A'$

$A' \rightarrow \alpha_1\ A' | \ldots | \alpha_n\ A' | \varepsilon$

Es. espressioni aritmetiche

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

NOTA: Non è detto che le ricorsioni sinistre siano solo immediate

S-> Aa | b

A-> Ac | Sd | ε

Algoritmo per eliminare le ricorsioni sinistre

- INPUT: Grammatica senza cicli né ε-produzioni
- OUTPUT: Una grammatica equivalente senza ricorsioni sinistre (Tale grammatica potrebbe contenere ε-produzioni)

Algoritmo:

1. Ordina i simboli non terminali $A_1, A_2, \ldots, A_n$;
2. For (i=1 to n) {

   for (j=1 to i-1) {

   sostituire ogni produzione $A_i \rightarrow A_j\gamma$ con le produzioni

   $A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \ldots | \delta_k\gamma$ dove $A_j \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k$ sono tutte le A_j produzioni

   }

   eliminare le ricorsioni sinistre immediate

   };

Nell'esempio: S-> Aa | b
A-> Ac | Sd | ε

Si ottiene: S->Aa |b
A->bdA' | A'
A'-> cA'| ad A' |ε

Parser predittivi

- Basandosi sull'input che resta da leggere, predice quale produzione usare senza fare uso di backtracking.
- Nella tecnica "a discesa ricorsiva" l'analisi sintattica viene effettuata attraverso una cascata di chiamate ricorsive.
- I parser discendenti deterministici possono anche essere guidati da una tabella. In tal caso si parla di parser predittivi.

- Nel parser LL(1), particolari parser predittivi, la pila delle chiamate ricorsive viene esplicitata nel parser, e quindi non si fa più uso di ricorsione

Parser LL(1)

Il termine LL(1) ha il seguente significato:

- 1. la prima L, significa che l'input è analizzato da sinistra verso destra
- 2. la seconda L, significa il parser costruisce una derivazione leftmost per la stringa di input
- 3. il numero 1, significa che l'algoritmo utilizza soltanto un solo simbolo dell'input per risolvere le scelte del parser (ci sono varianti con k simboli)

Esempio

Il linguaggio delle parentesi bilanciate

$S \rightarrow (S) S$

$S \rightarrow \varepsilon$

e vediamo come opera il parser **LL(1)**

per riconoscere la stringa "( )"

Il parser consiste di una ***pila***, che contiene inizialmente il simbolo "$" (fondo della pila), ed un ***input***, la cui fine è marcata dal simbolo "$"

(EOF generato dallo scanner)

pila	**input**	**azione**
$	**( ) $**	

-il parsing inizia inserendo il simbolo iniziale in testa alla pila

pila	input	azione
$ S	( ) $	

- il parser **accetta** una stringa di input se, dopo una sequenza di azioni, la pila contiene "$" e la stringa di input è "$"

pila	input	azione
…	… …	
$	$	accept

- ogni volta che in testa alla pila c'è un simbolo non terminale X, lo si espande secondo una produzione X → γ, che viene scelta a seconda del simbolo in testa all'input e ai valori di una tabella (la parte destra della produzione viene invertita sulla pila)

pila	input	azione
$ S	( ) $	S → (S) S

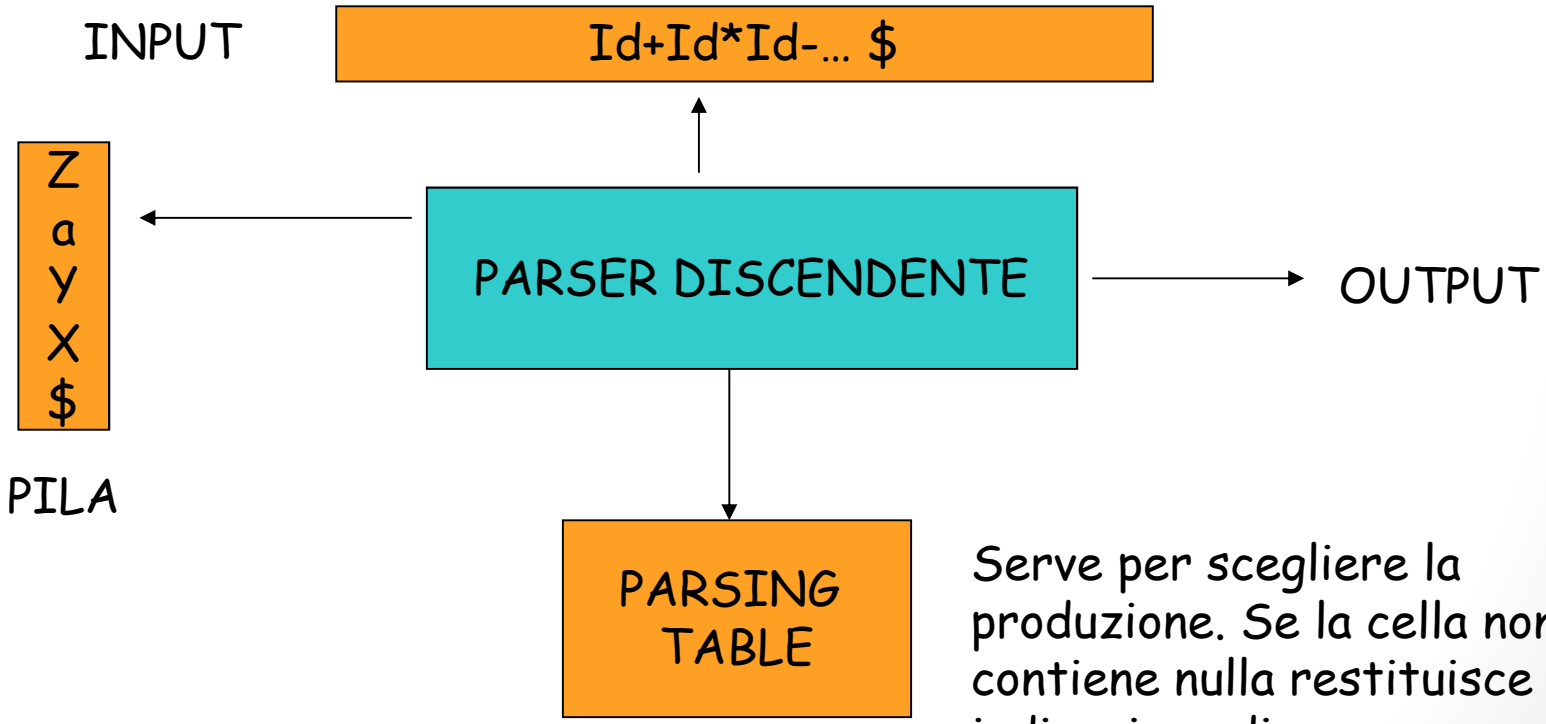
-ogni volta che sulla pila c'è un simbolo terminale **t**, si controlla che in testa all'input ci sia anche lo stesso simbolo, nel qual caso lo si elimina sia dalla pila che dall'input; altrimenti è **errore**

per costruire un parser **LL(1)**, bisogna costruire una tabella – la ***tabella LL(1)*** – che determina la regola da usare per l'espansione, dati il simbolo non-terminale e il carattere in input

pila	input	azione
$ S	( ) $	S → (S) S
$ S) S (	( ) $	match
$ S) S	) $	S → ε
$ S)	) $	match
$ S	$	S → ε
$	$	accept

Se l'input è generato dalla grammatica questo parsing fornisce una derivazione leftmost, altrimenti produce un'indicazione d'errore.

I Parser LL(1) sono parser discendenti non ricorsivi



Serve per scegliere la produzione. Se la cella non contiene nulla restituisce una indicazione di errore.