PARSER:COMPITI PRINCIPALI

GESTIONE DEGLI ERRORI SINTATTICI

GESTIONE DEGLI ERRORI IN UN PARSER

Un parser deve essere in grado di *scoprire*, *diagnosticare* e correggere gli errori in maniera efficiente, per *riprendere* l'analisi e scoprire nuovi errori.

Alcuni parser (LL e LR) hanno la proprietà "Viable prefix": sono in grado di rilevare un errore non appena si presenta perché sono in grado di riconoscere i prefissi validi del Linguaggio

STRATEGIE DI RIPARAZIONE

• "Panic mode": scoperto l'errore il parser riprende l'analisi in corrispondenza di alcuni token sincronizzanti predefiniti e facilmente riconoscibili (es.: delimitatori *begin end*) scartando alcuni caratteri. Svantaggi: può essere scartato molto input.

• "Phrase level": correzioni locali ottenute inserendo, modificando, cancellando alcuni terminali per poter riprendere l'analisi (es. trasformare il simbolo ',' in ';') Svantaggi: difficoltà quando la distanza dall'errore è notevole.

• "Error productions": uso di produzioni che estendono la grammatica per generare gli errori più comuni. Metodo efficiente per la diagnostica.

• "Global correction": si cerca di "calcolare" la migliore correzione possibile alla derivazione errata (minimo costo di interventi per inserzioni/cancellazioni). Metodo globale poco usato in pratica, ma tecnica usata per ottimizzare la strategia "phrase level".

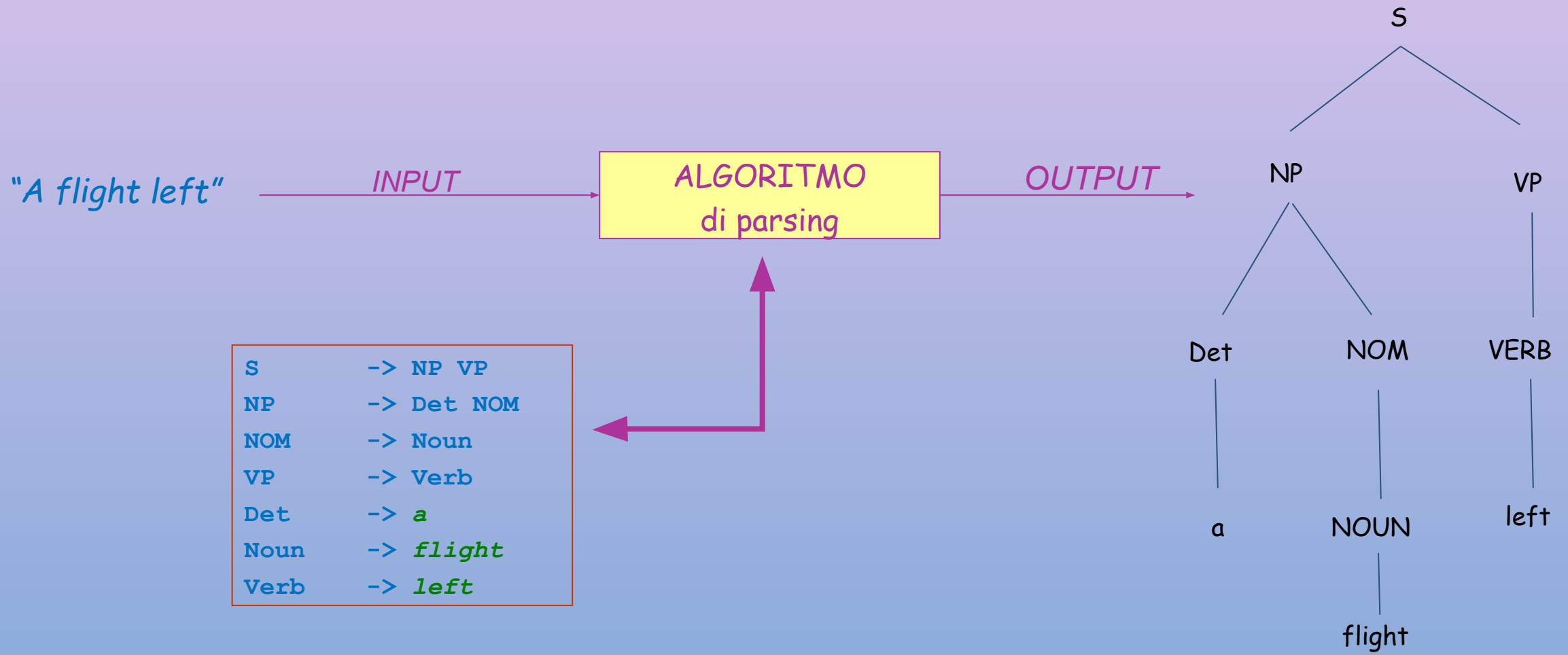
ANALISI SINTATTICA DEL COMPILATORE

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

ALGORITMO DI PARSING



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

```
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

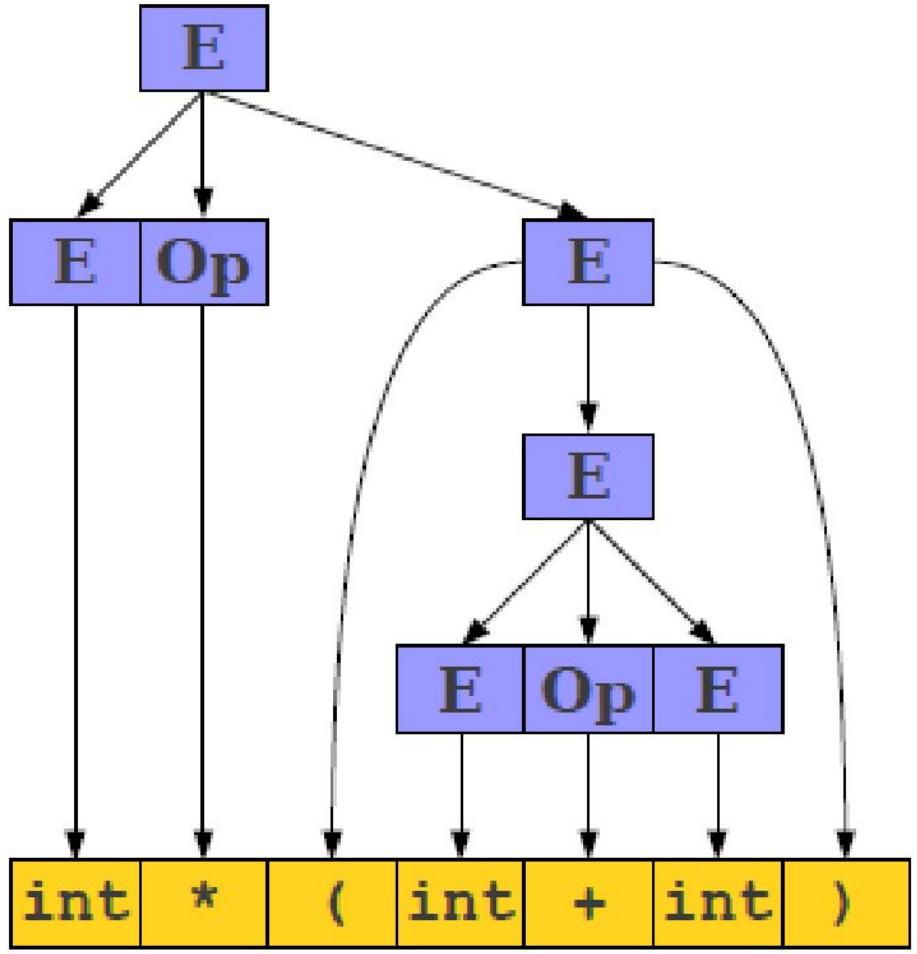
$E \rightarrow \texttt{int} \mid E\ Op\ E \mid (E)$

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

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

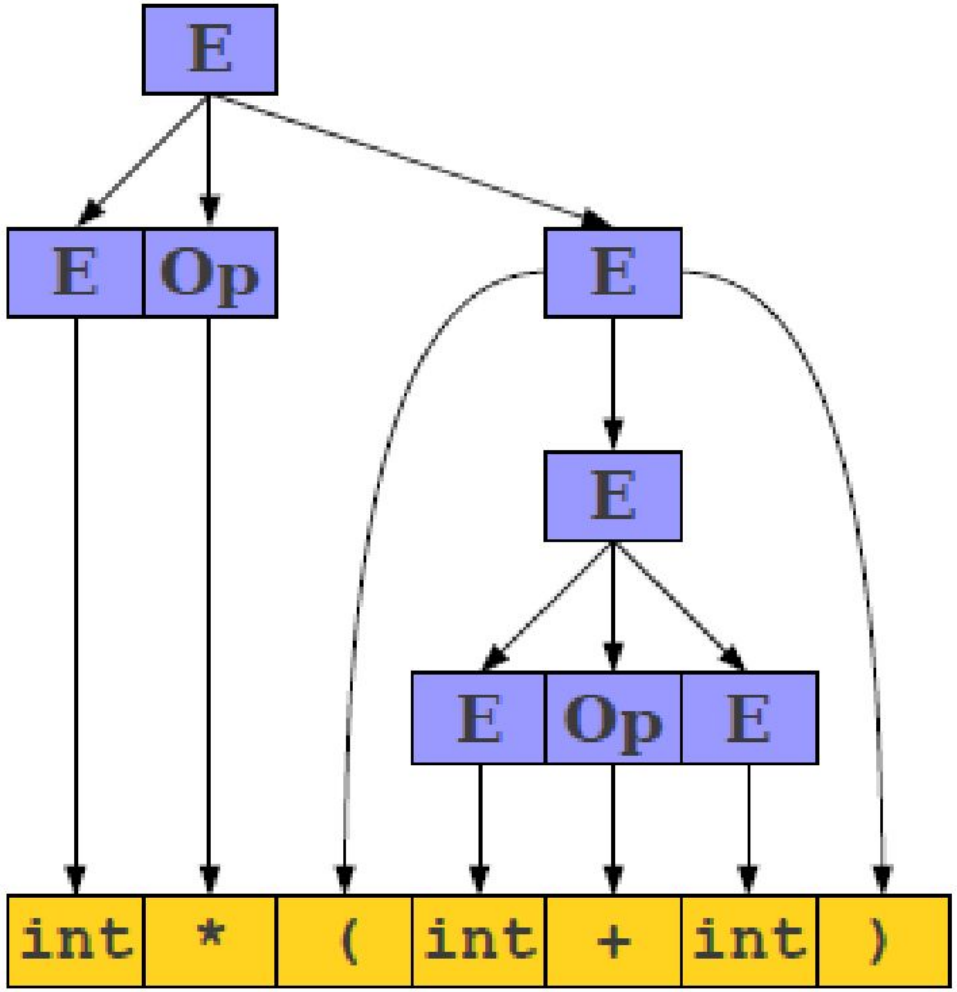
PRODUCONO LO STESSO SYNTAX TREE?

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

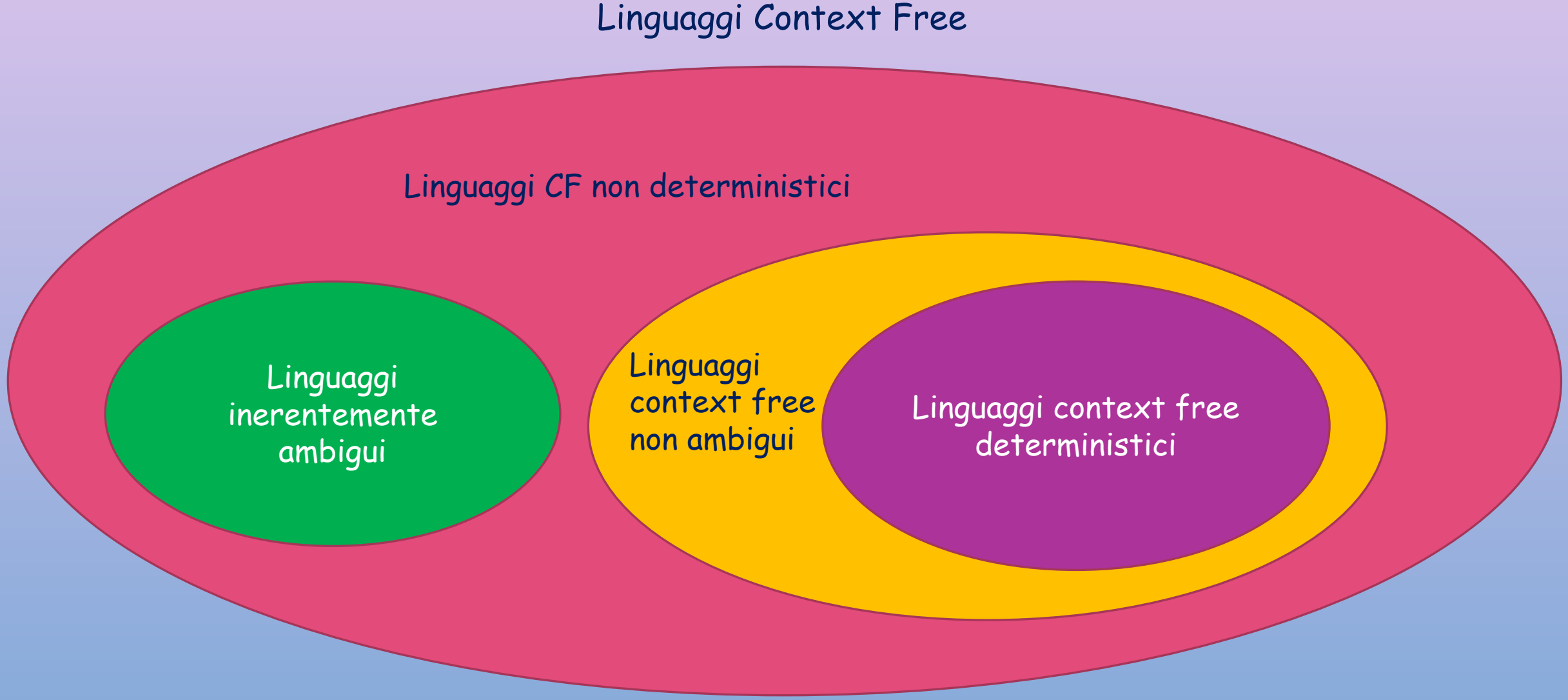
Costruire il syntax tree, ovvero quali produzioni vengono applicate piuttosto che l'ordine con cui si applicano.

Se il linguaggio 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 in generale potrebbe essere non deterministico.

Siamo interessati ai linguaggi deterministici!

GERARCHIA DEI LINGUAGGI CF



PARSER ASCENDENTI E DISCENDENTI

Considereremo due classi di parser:

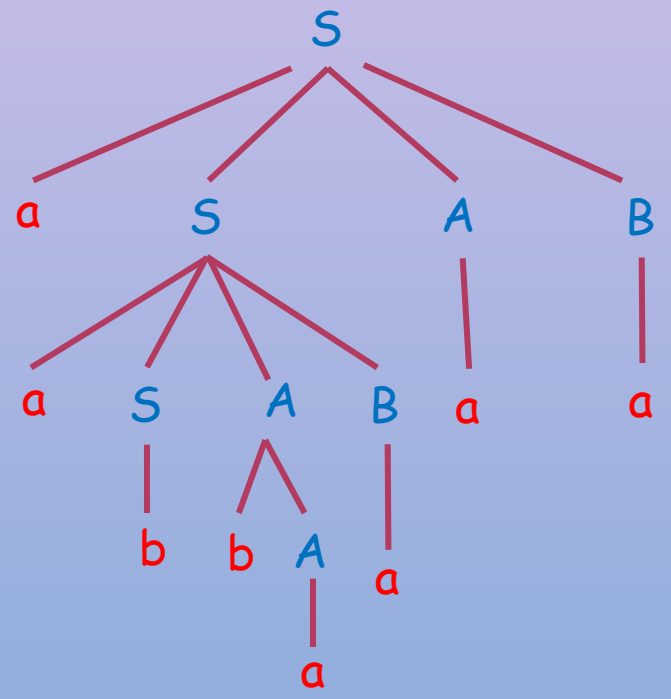
Discendenti o top-down: si costruisce la derivazione partendo dall'assioma; l'albero di derivazione si costruisce dalla radice alle foglie;

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.

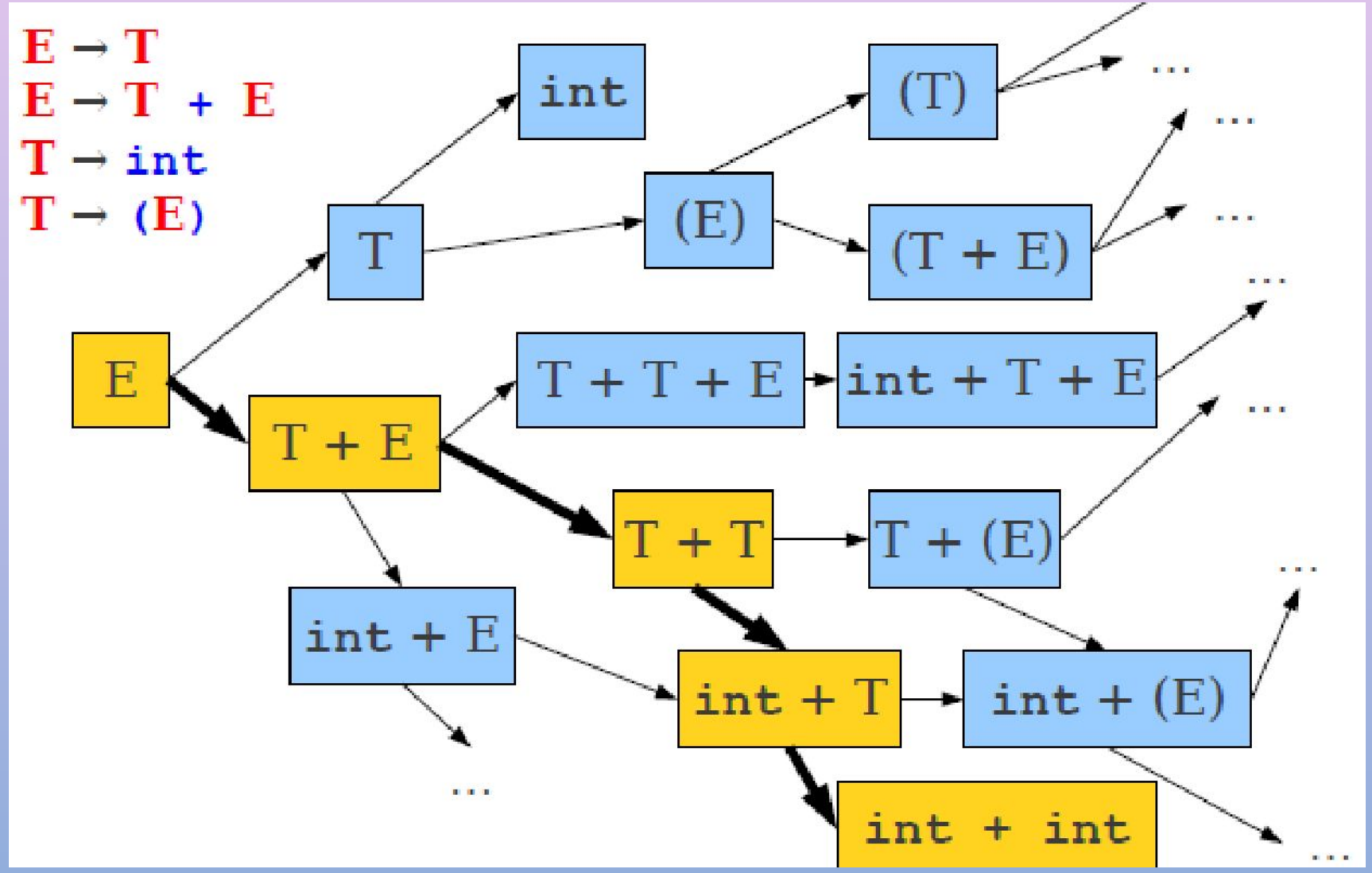


PARSER TOP DOWN-PROBLEMATICHE

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 consiste nel cercare un cammino nell'albero delle possibili scelte



DUE STRATEGIE DI PARSER 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 nelle regole corrisponde a un controllo che i terminali siano presenti nell'input

La sequenza di non terminali nelle regole corrisponde a invocazioni delle procedure.

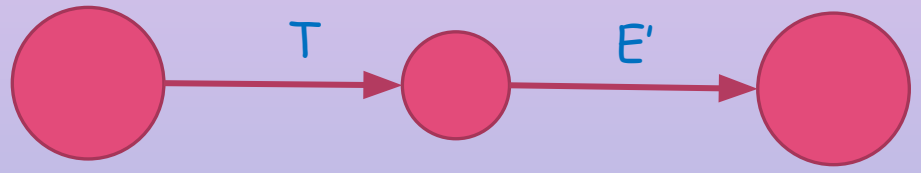
La presenza di diverse regole per A è modellata da **case** o **if**

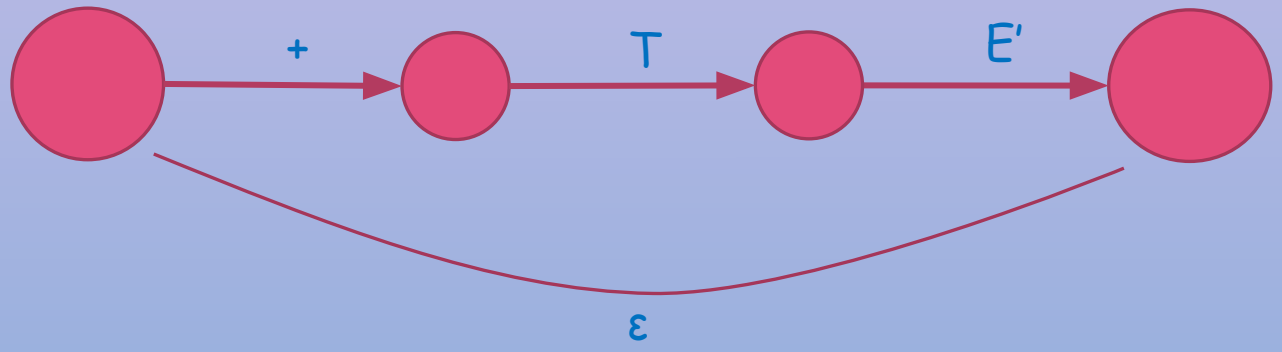
Può richiedere backtracking (può richiedere di leggere più di una volta parte della stringa in ingresso, ovvero se l'applicazione di una produzione fallisce può tornare indietro).

PROCEDURA TIPICA PER UN PARSER TOP-DOWN A DISCESA RICORSIVA

```
void A(){
   scegli, per A, una produzione A->X1X2…Xk
   for (i da 1 a k) {
      if (Xi è un non terminale)
         richiama Xi();
      else if (Xi è uguale al simbolo in input corrente)
         procedi al simbolo successivo;
            else si è verificato un errore;
   }
}
```

COSTRUZIONE DELLE PROCEDURE RICORSIVE






```
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
S -> PRINT E
L -> END
L -> ; S L
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 presenza di prefissi comuni in più parti destre di regole associate allo stesso simbolo non terminale e la ricorsione sinistra.

1. PREFISSI COMUNI A PIU' PARTI DESTRE ASSOCIATE ALLO STESSO NON TERMINALE

$A \rightarrow \gamma\alpha_1 | \ldots | \gamma\alpha_n$

Rimedio: ***fattorizzazione sinistra***

2. RICORSIONE SINISTRA

$A \rightarrow A\alpha$ (A non terminale).

Rimedio: ***eliminazione ricorsione sinistra***

FATTORIZZAZIONE SINISTRA

Produzioni provenienti dallo stesso simbolo e che iniziano con lo stesso prefisso.

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

ESEMPIO:

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

Fattorizzazione sinistra:

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

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

ESEMPIO

<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 IMMEDIATA

Metodo: date le produzioni ricorsive sinistre immediate e tutte le altre relative a un non terminale *A*:

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$ ⟶

$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A'$
$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$

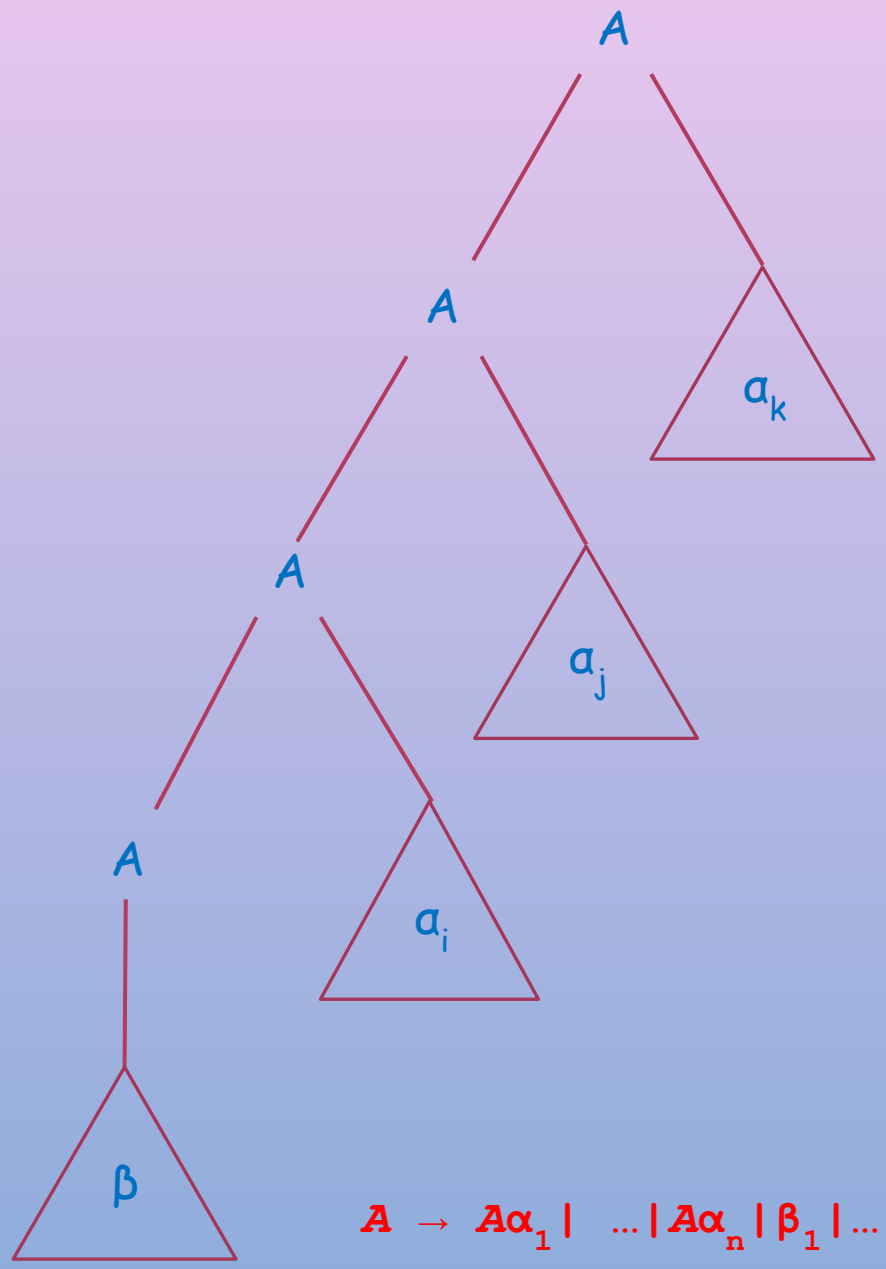
Si posticipa l'utilizzo del terminale *A*. Le produzioni devono necessariamente terminare con regole non ricorsive $\beta_1 \mid \ldots \mid \beta_m$ per produrre stringhe finite.

NOTA: non è detto che le ricorsioni sinistre siano solo immediate. Esiste un metodo più generale per risolvere il problema, che non studieremo. Per esempio:

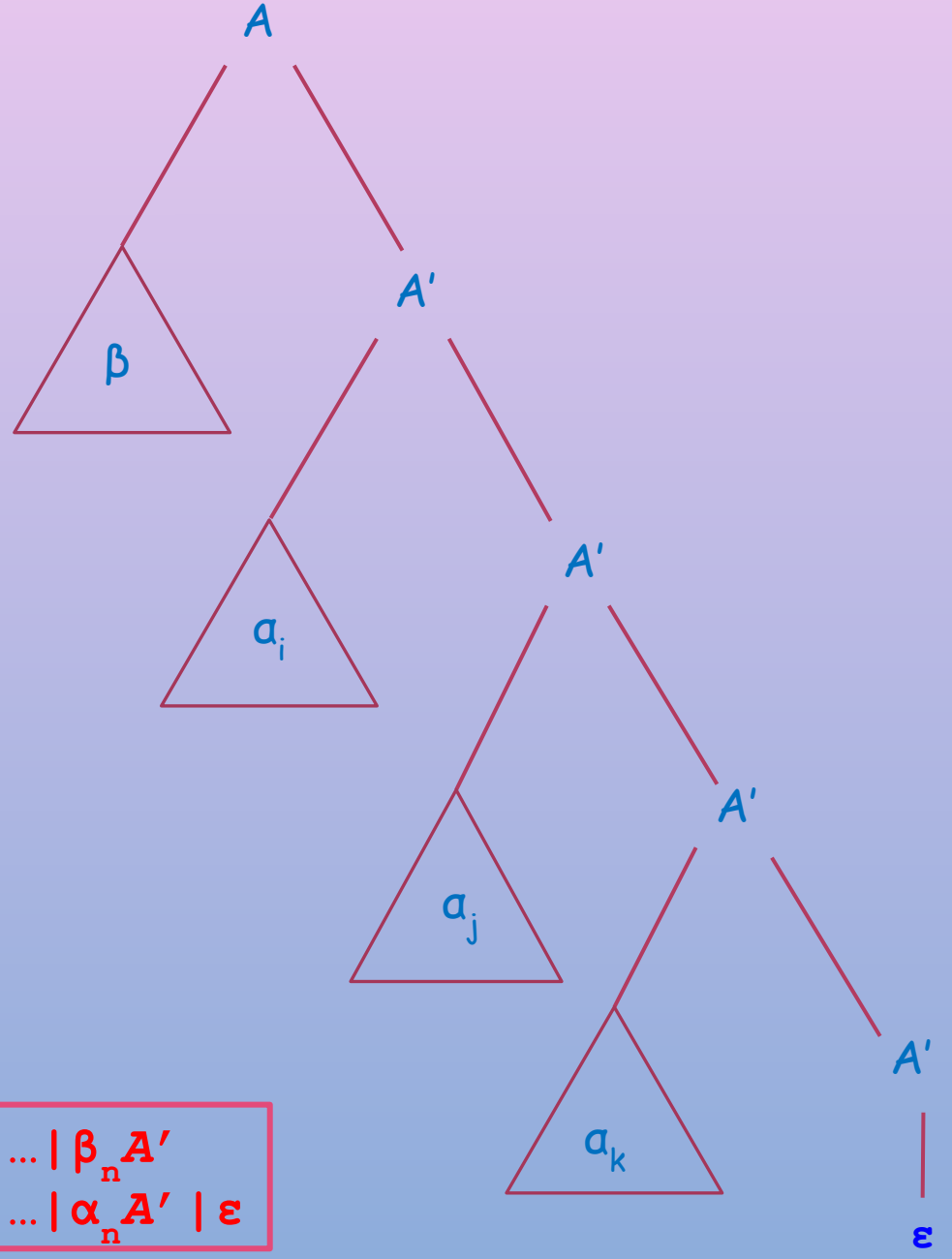
S-> Aa | b

A-> Ac | Sd | ε

S ha una ricorsione sinistra non immediata



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



$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

PARSER PREDITTIVI

Nella tecnica "a discesa ricorsiva" l'analisi sintattica viene effettuata attraverso una cascata di chiamate ricorsive che possono anche effettuare backtracking.

I parser discendenti deterministici possono anche essere guidati da una tabella. In tal caso si parla di parser predittivi.

Basandosi sull'input che resta da leggere, predice quale produzione usare senza fare uso di backtracking.

Nei parser LL(1) che vedremo in seguito e che sono particolari parser predittivi, la pila delle chiamate ricorsive viene esplicitata nel parser, e quindi non si fa più uso di ricorsione.