PARSER LL(1)

PARSER LL(1)

Il termine LL(1) ha il seguente significato:

1. La prima L, sta per Left to right, 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 guarda un solo simbolo dell'input in avanti per risolvere le scelte del parser (ci sono varianti con k simboli)

ESEMPIO

IL LINGUAGGIO DELLE PARENTESI BILANCIATE

S →(S)S

S → ε

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)

Per comodità mettiamo il fondo della pila a destra e impiliamo a sinistra

Pila	Input	azione
$	()$	

PARSING LL(1)

Il parsing **inizia** inserendo il simbolo iniziale S in testa alla pila

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

Ogni volta che in testa alla pila c'è un simbolo non terminale **x**, lo si espande secondo una produzione $x \rightarrow \gamma$, che viene scelta a seconda del simbolo in testa all'input e ai valori di una tabella

Ogni volta che sulla pila c'è un simbolo terminale **t**, si controlla se il simbolo dell'input che si sta leggendo sia lo stesso, nel qual caso lo si elimina sia dalla pila e la testina di lettura avanza sull'input; altrimenti si è riscontrato un **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$	()$	

Pila	Input	azione
…	…	…
…	…	…
$	$	

Pila	Input	azione
…	…	…
S$	()$	S→(S)S

ESEMPIO

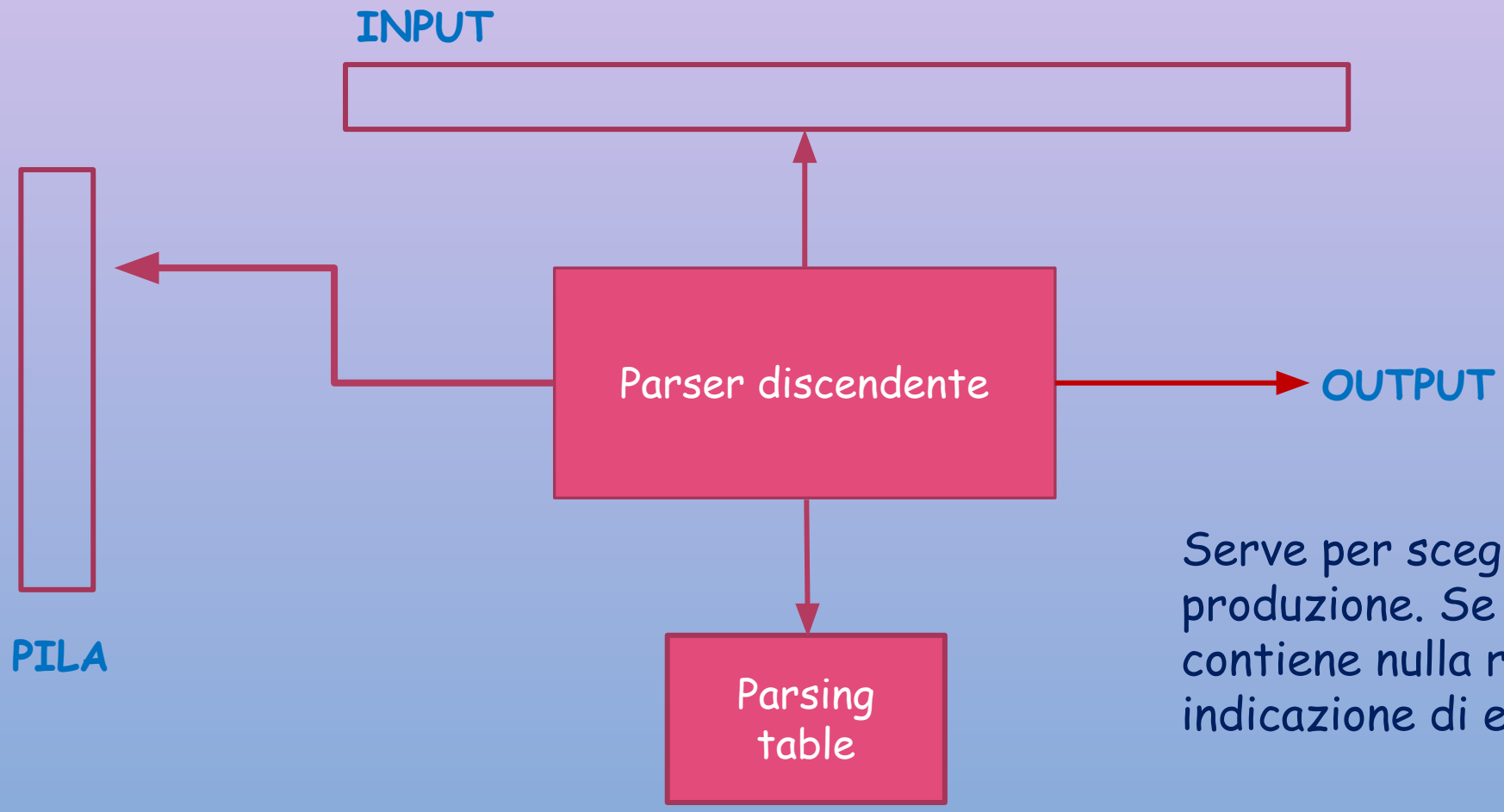
IL LINGUAGGIO DELLE PARENTESI BILANCIATE

$S \rightarrow (S)S$
$S \rightarrow \varepsilon$

Il linguaggio genera «( )»?
Se l'input è generato dalla grammatica questo parsing fornisce una derivazione leftmost, altrimenti produce un'indicazione d'errore.

Pila	Input	Azione
$S\$$	( ) $	$S \rightarrow (S)S$
$(S) S \$$	( ) $	match
$S) S \$$	) $	$S \rightarrow \varepsilon$
$) S \$$	) $	match
$S \$$	$	$S \rightarrow \varepsilon$
$\$$	$	accept

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

COSTRUZIONE DELLA PARSING TABLE DI UN LL(1)

1. La tabella ha simboli non-terminali come righe, e simboli terminali come colonne
2. La tabella contiene regole di produzione
3. Ogni regola $X \to \gamma$ di G per cui $\gamma \Rightarrow^* t\beta$ **(t appare come primo simbolo in una derivazione di γ)**, si inserisce nella casella `(X,t)` (per ogni t per cui questo avviene)
4. Ogni regola $X \to \gamma$ di G, per cui $\gamma \Rightarrow^* \varepsilon$, si inserisce nella casella `(X,t)`, per ogni `t` tale che $S \Rightarrow^* \beta X t \alpha$

(Bisogna conoscere questi simboli `t` in grado di far effettuare la scelta della produzione)

Le regole 2 e 3 sono difficili da implementare: gli algoritmi per risolverle sono discussi in seguito, vedi FIRST e FOLLOW

ESEMPIO

Data la grammatica:

S->(S)S,

S->ε

Parsing Table

	(	)	$
S	S->(S)S	S->ε	S->ε

FIRST E FOLLOW

FIRST e FOLLOW sono due funzioni, utilizzate sia dai parser top down che dai parser bottom-up, associate a una grammatica G.

In particolare nel parser top-down permettono di effettuare una scelta fra diverse produzioni in base al solo simbolo di input immediatamente successivo alla posizione della testina di lettura

FIRST

E' una funzione che consente di costruire le entries della tabella, quando possibile.

First(α): insieme dei simboli **terminali** che si trovano all'inizio delle stringhe derivabili da α (α è una stringa di simboli terminali e non).

Se **α=>*ε** allora anche ε appartiene a **FIRST(α)**

Nella slide successiva l'algoritmo per la costruzione di **FIRST(α)**

FIRST

1. Se **x** è un terminale, allora FIRST(**x**) = {**x**},

2. Se **X->ε** appartiene alla grammatica, allora aggiungi ε a FIRST(X),

3. Se **X -> $Y_1Y_2...Y_k$** appartiene alla grammatica, allora:

   - Se **a** ∈ **FIRST(Y_i)** per qualche i ed ε sta in

   FIRST(Y_1), FIRST(Y_2),..,FIRST(Y_{i-1}), aggiungi **a** a **FIRST(X)** ;

   - se tutti gli insiemi **FIRST(Y_1), FIRST(Y_2),… ,FIRST(Y_k),** contengono ε, aggiungi ε a **FIRST(X)**;

4. Per definire l'insieme **FIRST(γ)**, dove **γ = $X_1X_2...X_k$** (una stringa di terminali e non), si procede reiterando le regole seguenti:

   - Aggiungi **FIRST(X_1)\{ε}** a **FIRST(γ)**

   - Se per qualche **i<k** , tutti gli insiemi **FIRST(X_1),...,FIRST(X_i)** contengono **ε**, allora aggiungi **FIRST(X_{i+1})\{ε}** a **FIRST(γ)**

   - Se tutti gli insiemi **FIRST(X_1),…,FIRST(X_k)** contengono **ε**, allora aggiungi **ε** a **FIRST(γ)**

NOTA: Se **X->*ε** allora **ε ∈ FIRST(X)** (**X** è detto annullabile)

ESEMPIO

Nel caso della grammatica delle espressioni aritmetiche senza ricorsioni sinistre:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Simboli terminali

FIRST(id)={id}

FIRST(()={(}

FIRST(+)={+}

FIRST(*)={*}

FIRST())={)}

Simboli nonterminali

FIRST(F)={id,(}

FIRST(T)=FIRST(E)=FIRST(F)

FIRST(E')={+, ε}

FIRST(T')={*, ε}

Per quanto riguarda le stringhe nelle produzioni

Per esempio, FIRST(TE')=FIRST(T)

FIRST(+TE')={+}, ...

PSEUDO CODICE PER FIRST

```
for all terminals x and ε do FIRST(x)={x} ;
for all non-terminals X do FIRST(X) = { } ;
while (there are changes to any FIRST(X)) do
    for each production X->Y1Y2...Yk do
    {
    i:= 1 ; continue := true ;
    while (continue == true && i<=k) do
        {
        add FIRST(Yi)\{ε} to FIRST(X);
        if (ε is not in FIRST(Yi)) continue := false;
        i := i+1 ;
        }
    if (continue == true) add ε to FIRST(X) ;
    }
```

FIRST PUÒ NON BASTARE

Il nostro obiettivo è stabilire una regola per costruire una tabella che, in base al simbolo successivo a quello letto è in grado di stabilire in maniera univoca quale regola usare

A questo proposito FIRST non basta.

Se infatti abbiamo delle regole che producono in uno o più passi la parola vuota $(A \rightarrow^* \varepsilon)$, non siamo in grado mediante la solo funzione first di conoscere qual è il prossimo simbolo prodotto, e non possiamo quindi fare la scelta in base al prossimo simbolo di input.

In questo caso occorre utilizzare l'insieme FOLLOW

FIRST PUÒ NON BASTARE

Il nostro obiettivo è stabilire una regola per costruire una tabella che, in base al simbolo successivo a quello letto è in grado di stabilire in maniera univoca quale regola usare

A questo proposito FIRST non basta. Per esempio, se la grammatica contiene due produzioni

X -> γ_1

X -> γ_2

(*Stesso simbolo non-terminale a sinistra, due sequenze differenti a destra*)

e **FIRST(γ_1) ∩ FIRST(γ_2)** è non vuota.

Allora la grammatica non può essere analizzata col parsing predittivo top-down:

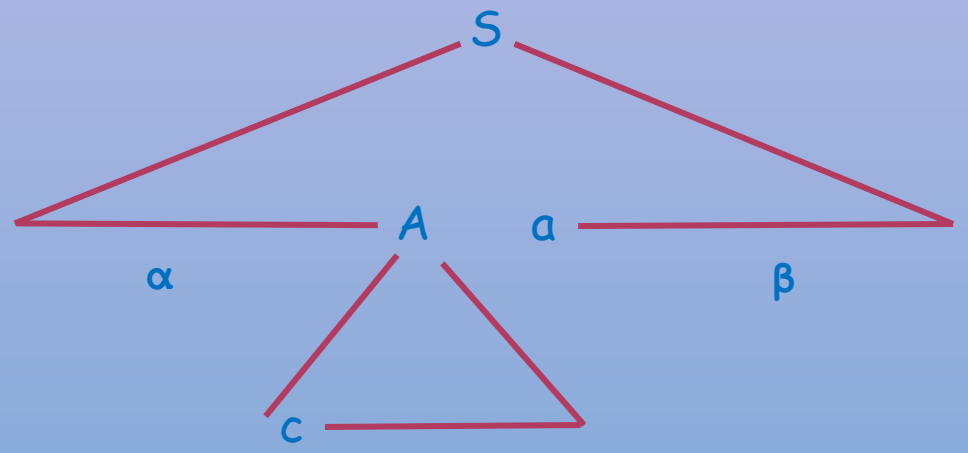
Infatti se **t∈FIRST(γ_1) ∩ FIRST(γ_2)** allora, il parsing discendente non saprà cosa fare quando il primo simbolo dell'input è **t**

In questo caso occorre utilizzare l'insieme FOLLOW

FOLLOW

FOLLOW(A) con **A** non terminale è l'insieme dei terminali che possono apparire immediatamente alla destra di **A** in una forma sentenziale, cioè l'insieme dei terminali **a** tali che esiste una derivazione della forma **X-> αAaβ**, con **α** e **β** generiche forme sentenziali. Tra *A* e **a** potrebbero anche esserci altri simboli che generano la parola vuota. Inoltre se *A* appare come ultimo simbolo allora $ appartiene a FOLLOW(*A*)

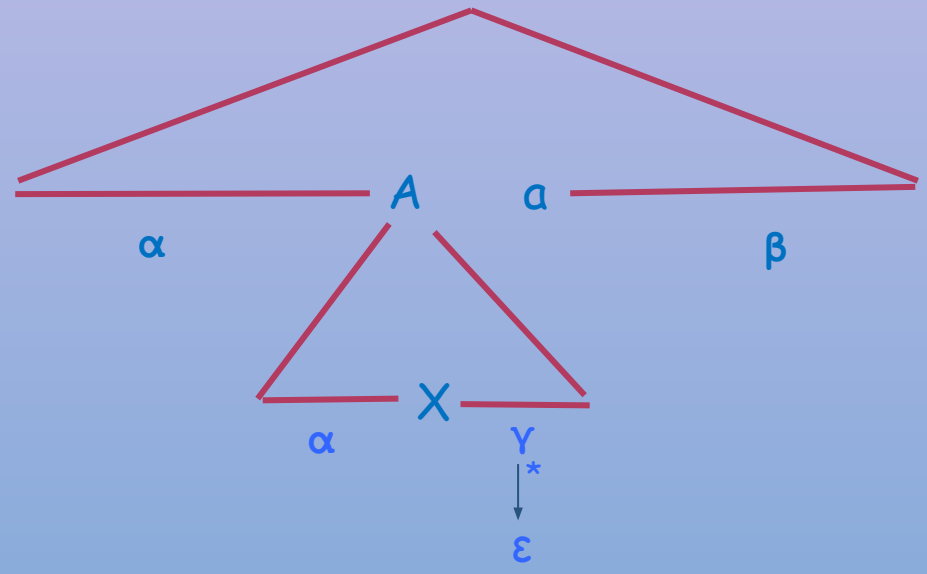
Nella figura a fianco, **c** appartiene a FIRST(*A*)
e **a** appartiene a FOLLOW(*A*)



FOLLOW

Dato un **non terminale** X, l'insieme **FOLLOW(X)** è l'insieme dei simboli **terminali**, eventualmente $, che appaiono alla destra di X in qualche forma sentenziale ed è costruito come segue:

1. Se **X** è l'assioma, allora aggiungi **$** a **FOLLOW(X)**;
2. Se c'è una produzione **A->αXγ**, allora aggiungi **FIRST(γ)\{ε}** a **FOLLOW(X)**
3. Se c'è una produzione **A->αX** oppure una produzione **A->αXγ** per cui **ε ∈ FIRST(γ)**, allora aggiungi **FOLLOW(A)** a **FOLLOW(X)**



FOLLOW - OSSERVAZIONI

1. Quando l'assioma non compare a destra delle produzioni, il "$" è l'unico simbolo nel suo insieme **FOLLOW**
2. L'insieme **FOLLOW** non contiene mai "ε";
3. **FOLLOW(X)** è definito soltanto per non-terminali: potremmo generalizzare la definizione ma ciò è inutile per la tabella **LL(1)**;
4. La definizione di **FOLLOW** lavora "alla destra" di una produzione, mentre quella di **FIRST** lavora "alla sinistra": una produzione **X->α** non ha alcuna informazione su **FOLLOW(X)**, se **X** non è presente in **α**

PSEUDO CODICE PER FOLLOW

```
FOLLOW(axiom) = {$} ;
for all (nonterminal(X) && X!=axiom) do
    FOLLOW(X)= {};
while (there are changes to any FOLLOW set) do
    for each production X->Y1Y2...Yk do
    for each nonterminal(Yi) do {
    add FIRST(Yi+1... Yk)\{ε} to FOLLOW(Yi) ;
/* note: if i=k then Yi+1... Yk=ε */
if ε is in FIRST(Yi+1... Yk)
add FOLLOW(X) to FOLLOW(Yi);
}
```

ESEMPIO

Nel caso della grammatica delle espressioni aritmetiche senza ricorsioni sinistre:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

FIRST(id)={id}
FIRST('(')={(}
FIRST('+')={+}
FIRST(*)={*}
FIRST())={)}
FIRST(T)=FIRST(E)=FIRST(F) ={id,'('}
FIRST(E')={+, ε}
FIRST(T')={*, ε}

FOLLOW(E)={$,)}
FOLLOW(E')={$,)}
FOLLOW(T)={$,),+}
FOLLOW(T')={$,),+}
FOLLOW(F)={$,),+,*}

COME COSTRUIRE LA TABELLA DI PARSING PREDITTIVO?

1. Per ogni regola **X->α** della grammatica, e per ogni terminale **a** in **FIRST(α)**, si aggiunga la regola **X->α in M[X,a]**.
2. Per ogni regola **X->α**, se **ε∈FIRST(α)** (come caso particolare la regola **X->ε**), per ogni **b ∈ FOLLOW(X)**, si inserisca nella casella **M(X,b)** la regola **X->α**.

   Se **ε ∈ FIRST(α)** e **$ ∈ FOLLOW(X)**, si inserisce la regola **X->α** in **M(X,$)**.

Le caselle non definite individuano un errore.

NOTA: Se G è ricorsiva sinistra o ambigua, la tabella avrà caselle con valori multipli.

A->Ac
A->b

Entrambe le regole contengono b nell'insieme first, dunque entrambe vanno nella stessa casella (A,b)

GRAMMATICA LL(1)

Definizione:

Una grammatica in cui ogni casella della tabella di parsing non contiene più di un elemento è detta **LL(1)**

Osservazione 1: per costruzione una grammatica **LL(1)** non è ambigua, né ricorsiva sinistra. Quindi in questi casi la grammatica non può essere LL(1).

Osservazione 2: Per stabilire se una grammatica è LL(1) bisogna costruire la tabella e verificare che in ogni casella ci sia al più un elemento.

GRAMMATICA LL(1)

Una grammatica è LL(1) se e solo se per ogni produzione del tipo **A->α|β** si ha:

- α e β non derivano stringhe che cominciano con lo stesso simbolo **a**.
- al più uno tra i due può derivare la stringa vuota.
- Se **β->*ε** allora α non deriva stringhe che cominciano con terminali che stanno in **FOLLOW(A)**. Analogamente per α.

Equivalentemente affinché una grammatica sia LL(1) deve avvenire che per ogni coppia di produzioni **A->α|β**

1. **FIRST(α)** e **FIRST(β)** devono essere disgiunti
2. Se **ε** è in **FIRST(β)** allora **FIRST(α)** e **FOLLOW(A)** devono essere disgiunti.

PARSER LL(1)

E' sempre possibile realizzare un parser predittivo con una grammatica LL(1) perché ad ogni passo la produzione per un terminale può essere decisa, analizzando la tabella, in maniera unica analizzando solo il carattere di input successivo alla posizione della testina.

PARSER LL(1)

```
LL1_parser(stack p, input i, LL1_table M, initial S ) {
/* p, i sono pile, S è l'assioma*/
int error=0; p=push(p,$);
p = push(p,S) ;
while (top(p)≠$ && top(i)≠$ && !Error){
    if isterminal(top(p)) {
        If top(p) == top(i) {p=pop(p);i=avanza(i);}
        else error = 1;}
    else {/* top(p) è un non-terminale, bisogna espandere */
        if isempty(M[top(p),top(i)]) error=1;
        else { /* M[top(p),top(i)] == X->X1...Xn; */
            p = pop(p);
            for (j = n; j > 0; j=j-1)
            p = push(p,Xj);}}
        }
If (!error) accept() ;
else raise_error() ;
}
```

COMPLESSITÀ

La complessità dell'algoritmo del parser LL(1) è lineare nella lunghezza n della stringa sorgente, poiché viene consumato un carattere per volta.

ESERCIZI

Stabilire quali delle seguenti grammatiche sono LL(1)

1. G: S->aSb|ε
2. G: S->+SS| *SS| id
3. G: S->aSb|aSc| ε
4. G: S->aSa | bSb | a | b
5. G: S->iEtS|iEtEeS|a

   E->b

ESEMPIO

Costruire il parser **LL(1)** per la grammatica

S ->(S)

S -> [S]

S -> { S }

S -> ε

	First	FOLLOW
S	(,[,{,ε	),],},$

La tabella **LL(1)**:

	(	[	{	)	]	}	$
S	S->(S)	S->[S]	S->{S}	S->ε	S->ε	S->ε	S->ε

Esercizio:
Effettuare il parsing della stringa ([{}]) e il parsing di ([)]

ESERCIZIO 1

G: S->aSb|ε

FIRST(S)={a,ε}

FOLLOW(S)={b, $}

Tabella LL(1)

	a	b	$
S	S->aSb	S->ε	S->ε

ESERCIZIO 2

G:

S->+SS|*SS|id

FIRST(S)={+,*,id}

FOLLOW(S)={+,*,id,$}

	+	*	id	$
S	S->+SS	S->*SS	S->id	

ESERCIZIO 3

G: S->aSb|aSc| ε

Fattorizzazione sinistra

S->aSA| ε
A->b|c

La grammatica presenta due regole con prefissi in comune.

FIRST(S)={a, ε}

FIRST(A)={b,c}

FOLLOW(S)={b, c, $}

FOLLOW(A)={b, c, $}

	a	b	c	$
S	S->aSA	S->ε	S->ε	S->ε
A		A->b	A->c	

ESERCIZIO 4

G: S->aSa | bSb | a | b

Fattorizzazione sinistra

S->aA|bB
A->Sa|ε
B->Sb|ε

FIRST(S)={a,b}
FIRST(A)={a,b,ε}
FIRST(B)={a,b,ε}

FOLLOW(S)={a,b,$}
FOLLOW(A)={a,b,$}
FOLLOW(B)={a,b,$}

	a	b	$
S	S->aA	S->bB	
A	A->Sa A->ε	A->Sa A->ε	A->ε
B	A->Sb B->ε	A->Sb B->ε	B->ε

Possiamo dedurre che
non è una grammatica LL(1)

ESEMPIO

G: S->iEtS|iEtSeS|a

E->b

S->iEtSS'|a
S'->eS | ε
E->b

FIRST (S)={i, a}
FIRST(S')={e, ε}
FIRST(E)={b}

FOLLOW(S)={e,$}
FOLLOW(S')={e,$}
FOLLOW(E)={t}

	i	a	b	t	e	$
S	S->iEtSS'	S->a				
S'					S'->eS S'->ε	S'->ε
E			E->b			

2 possibili scelte: la grammatica è ambigua

COME OTTENERE GRAMMATICHE LL(1)

- Verificare, se è possibile, che non sia ambigua. In caso contrario, si tenti, se possibile, di rimuovere l'ambiguità;
- Controllare che non presenti ricorsioni sinistre. In caso contrario trasformarle in ricorsioni destre.
- Se un simbolo non terminale ammette più derivazioni con lo stesso prefisso applicare il metodo della fattorizzazione sinistra.
- Dopo aver calcolato First e Follow per ogni simbolo, costruire la tabella di parsing predittivo. Se ogni casella della tabella ammette al più una derivazione allora la grammatica è LL(1)
- In alternativa, può essere necessario allungare la lunghezza della prospezioni. (Equivale a considerare parser LL(k), k>1)

LIMITI DELLA FAMIGLIA LL(K)

Non tutti i linguaggi verificabili da parser deterministici sono generabili da grammatiche LL(k).

Per esempio:

L={$a^*a^nb^n$ |n>=0} linguaggio deterministico ma non LL(k)

È generato dalla grammatica

S-> A|aS

A-> aAb|ε

Altro esempio:

S-> R|(S)

R-> E=E

E-> a|(E+E)

Definisce relazioni di uguaglianza tra espressioni aritmetiche additive. Una stringa che inizia con "(" può essere una relazione R parentesizzata oppure un'espressione.

Il linguaggio non è LL(k) ma è deterministico.

RELAZIONE FRA GRAMMATICHE LL(K)

E' possibile generalizzare la nozione di **FIRST** nel Parsing predittivo in modo da restituire i primi ***k*** token in input, e costruire una tabella predittiva in cui le righe sono i simboli nonterminali e *le colonne sono sequenze di* ***k*** *terminali* .

Le grammatiche non ambigue corrispondenti sono le ***LL(K)***