Grammatica
```
A -> aBc | bDd
B -> ε | e
D -> ε | f
```

FIRST
```
A: a, b
B: e, ε
D: f, ε
```

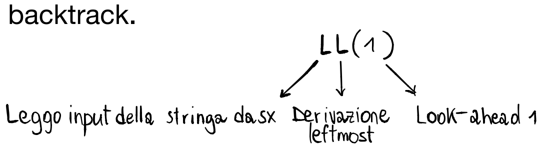
FOLLOW
```
A: $
B: c
D: d
```

Come riempire la tabella di parsing?
Input: Grammatica G
Output: Tabella di parsing per parsing top-down predittivo
```
    foreach (A -> α ∈ Produzioni) do
        ∀ b ∈ FIRST(α), inserire A -> α nella entry [A, b] della tabella

        if ε ∈ FIRST(α), ∀ x ∈ FOLLOW(A), inserire A -> α nella entry [A, x] della tabella

    end foreach
    inserire error() in tutte le entry vuote della tabella
```

Grammatica LL(1): se la tabella di parsing top-down per G non ha entry multiply-defined, allora la grammatica è LL(1), cioè non farà uso del backtrack.



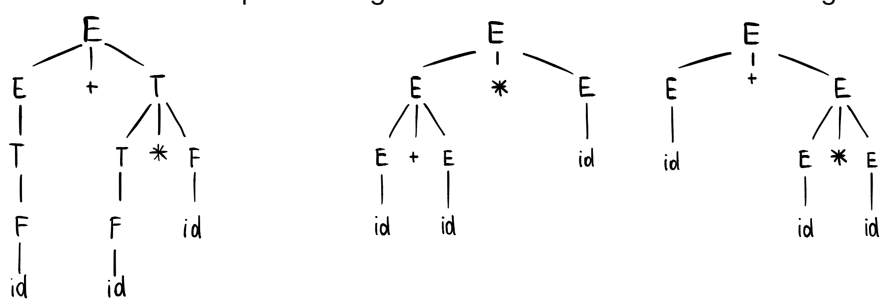
Look-ahead 1: consulto un solo simbolo "in avanti" per decidere quale produzione utilizzare nell'espansione di ciascun non-terminale. Il look-ahead 1 si visualizza notando che la tabella di parsing ha un solo simbolo sulle colonne.

`S -> aSb | ab` è LL(1)?

	a	b	$
S	S→aSb S→ab		

Entry multiply-defined: non so quale produzione applicare nel momento in cui devo espandere la `S` vedendo `a...$`.
Mentre invece, se potessi consultare 2 simboli e leggessi `ab...$`, saprei scegliere in maniera opportuna...ma non questa non sarebbe LL(1).

Vediamo ora la seguente grammatica non ambigua, una "queen grammar":
```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```
Che disambigua la seguente grammatica delle espressioni aritmetiche:
```
E -> E + E | E * E | ( E ) | id
```
L'albero di derivazione per la stringa `id + id * id` usando le due seguenti grammatiche sarebbe:



– queen grammar –   – due alberi della grammatica ambigua –

La “queen grammar” è LL(1)? Per rispondere, dobbiamo controllare che la corrispondente tabella di parsing non contenga entry multiply-defined.
Cominciamo calcolando FIRST e FOLLOW della grammatica:
FIRST:

```
E: (, id
T: (, id
F: (, id
```

FOLLOW:

```
E: $, +, )      : $, +, )
T: *            : $, +, ), *
F:              : $, +, ), *
```

Riempiamo la tabella:

	id	+	*	(	)
E	E → E + T E → T				
T					
F					

È analizzabile in top-down? No, ha alcune caratteristiche fastidiose, in particolare la left-recursion: comincia proprio con A $A \Rightarrow^{+} A\alpha$
La libertà di scelta fa sì che potenzialmente scegliamo la produzione (?) errata e la complessità diverge.

```
E => E + T => E + T + T
T => T * F
```

Per eliminare la left-recursion, si manipola la grammatica.

Una grammatica G = (V, S, T, P) è **ricorsiva a sinistra** (anche detto: mostra ricorsione a sinistra) sse, per qualche A ∈ V\T e per qualche α ∈ V*, $A \Rightarrow^{+} A\alpha$ in un certo numero di passi
Ad esempio, $\begin{cases} S \Rightarrow B \mid e \\ B \Rightarrow Sa \end{cases}$

Si dice che la ricorsione a sinistra è **immediata** se, per qualche A ∈ V\T e per qualche α ∈ V*, A -> Aα ∈ P (è a tutti gli effetti una produzione della grammatica).