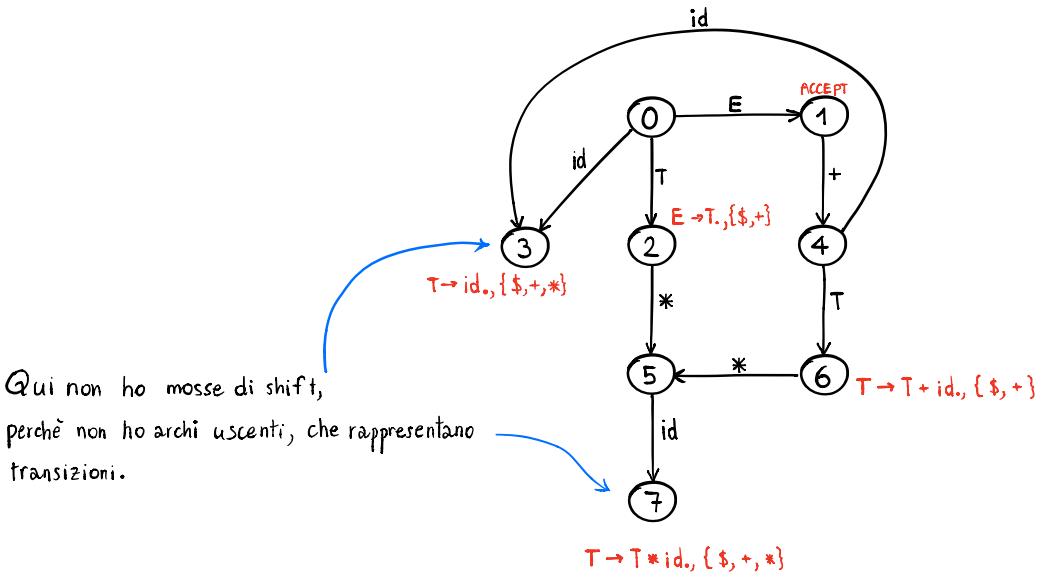
Grammatica:
```
E -> E + T | T
T -> T * id | id
```

Automa caratteristico:



R/R conflict: la stessa entry della tabella contiene “reduce `A -> β`” e “reduce `A -> γ`” per `A -> β` diverso da `A -> γ`.
Condizione necessaria per trovare nella tabella di parsing un conflitto R/R è di avere in un certo stato dell’automa caratteristico più reducing item (è l’unica situazione per la quale, nella tabella, andrei a mettere due reduce).
Poi, bisognerebbe anche avere che il lookahead set associato a ciascun reducing item abbiano intersezione non nulla.

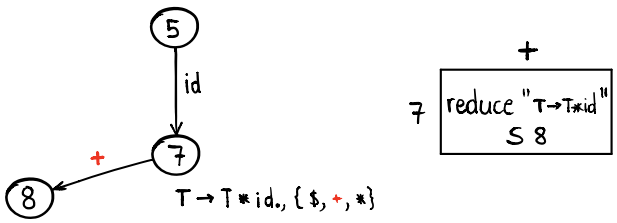
S/R conflict: la stessa entry della tabella contiene “shift P” (P è uno stato) e “reduce `A -> B`” ,
per qualche P e qualche `A -> B`.
Condizione necessaria per trovare nella tabella di parsing un conflitto S/R è di avere in un certo stato dell’automa caratteristico una transizione uscente etichettata con un certo terminale ed un reducing item il cui lookahead set contiene quel terminale.

Sulla base di queste considerazioni, possiamo concludere che la tabella, che si ricaverebbe dal precedente automa caratteristico e dalle informazioni appuntate in rosso sui reducing item e i rispettivi lookahead set, non avrà né conflitti R/R né conflitti S/R.

Tabella di parsing:

	id	+	*	$	E	T
0	S3				1	2
1		S4		ACC		
2		r “E->T”	S5	r “E->T”		
3		r “T->id”	r “T->id”	r “T->id”		
4	S3					6
5	S7					
6		r “E->E+T”	S5	r “E->E+T”		
7		r “T->T*id”	r “T->T*id”	r “T->T*id”		

Se l’automa caratteristico fosse stato così, avrei avuto un S/R conflict:



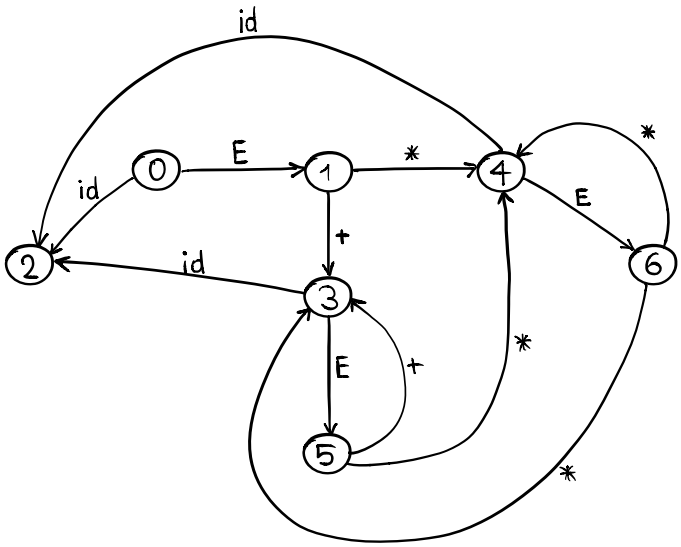
Tutt'altro paio di maniche se prendiamo una grammatica ambigua:
`E -> E + E | E * E | id`

È ambigua perché non dice che il + è associativo a sinistra, che il * è associativo a sinistra, che il * ha la precedenza sul +.
La grammatica di prima lo diceva.

Stato 0:
```
E' -> .E, {$}
- - - - - - -
E -> .E+E, {$,+,*}
E -> .E*E, {$,+,*}
E -> .id, {$,+,*}
```

Stato 1:
```
E' -> E., {$}
E -> E.+E, {$,+,*}
E -> E.*E, {$,+,*}
```

Automa caratteristico:



Stato 2:
```
E' -> id., {$,+,*}
```

Stato 3:
```
E -> E+E, {$,+,*}
- - - - - - - - -
E -> .E+E, {$,+,*}
E -> .E*E, {$,+,*}
E -> .id, {$,+,*}
```

Stato 4:
```
E -> E*E.E, {$,+,*}
- - - - - - - - - -
E -> .E+E, {$,+,*}
E -> .E*E, {$,+,*}
E -> .id, {$,+,*}
```

Stato 5:
```
E -> E+E., {$,+,*}
E -> E.+E, {$,+,*}
E -> E.*E, {$,+,*}
```

Stato 6:
```
E -> E*E., {$,+,*}
E -> E.+E, {$,+,*}
E -> E.*E, {$,+,*}
```

Tabella di parsing:

	id	+	*	$	E
0	S2				1
1		S3	S4	ACC	
2		E->id	E->id	E->id	
3	S2				5
4	S2				6
5		**S3, E->E+E**	**S4, E->E+E**	E->E+E	
6		**S3, E->E*E**	**S4, E->E*E**	E->E*E	

Ci sono 4 S/R conflict marcati in grassetto.

Ora scegliamo quale, in ciascuno dei quattro casi di conflict S/R, è la mossa appropriata da prendere.

Supponiamo di dover riconoscere la stringa `id1 + id2 + id3`.

Quando inizio, nell'automa sono nello stato 0.

State stack	0
Symbol stack	

Nell'input buffer leggo `id1`.

La tabella, alla entry [0, `id`], indica "shift `2`". Metto `id1` nel symbol stack e 2 nello state stack. Nell'input buffer leggo `+`.

State stack	0 2
Symbol stack	id_1

La tabella, alla entry [2, `+`], indica "reduce `E->id`". Tolgo 1 elemento dagli stack. Metto `E` nel symbol stack.
Nello state stack metto lo stato indicato dalla entry [0, `E`], cioè `1`.

State stack	0 1
Symbol stack	E

La tabella, alla entry [1, `+`], indica "shift 3". Metto `+` nel symbol stack e 3 nello state stack. Nell'input buffer leggo `id2`.

State stack	0 1 3
Symbol stack	E +

La tabella, alla entry [3, `id`], indica "shift 2". Metto `id2` nel symbol stack e 2 nello state stack. Nell'input buffer leggo `+`.

State stack	0 1 3 2
Symbol stack	E + id_2

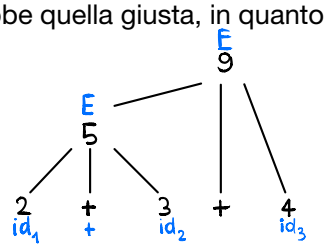
La tabella, alla entry [2, `+`], indica "reduce `E->id`". Tolgo 1 elemento dagli stack. Metto `E` nel symbol stack.
Nello state stack metto lo stato indicato dalla entry [3, `E`], cioè `5`.

State stack	0 1 3 5
Symbol stack	E + E

La tabella, alla entry [5, `+`], contiene un **S/R conflict**. Come facciamo?

La mossa di reduce sarebbe quella giusta, in quanto noi assumiamo che la grammatica sia associativa a sinistra.

Es.: 2 + 3 + 4

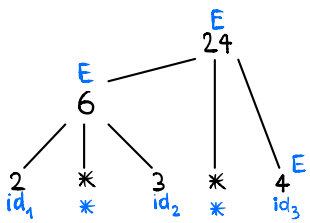


Pertanto

	+
5	~~S3~~
	reduce "E→E+E"

Supponiamo di dover riconoscere la stringa `id1 * id2 * id3`. La tabella, alla entry [6, `*`], contiene un **S/R conflict**.

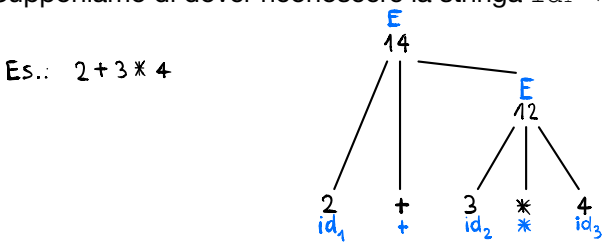
Es.: 2 * 3 * 4



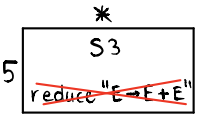
Per esprimere l'associatività a sinistra del `*`, bonifichiamo in maniera analoga la entry [6, `*`]:

	*
6	~~S4~~
	reduce "E→E*E"

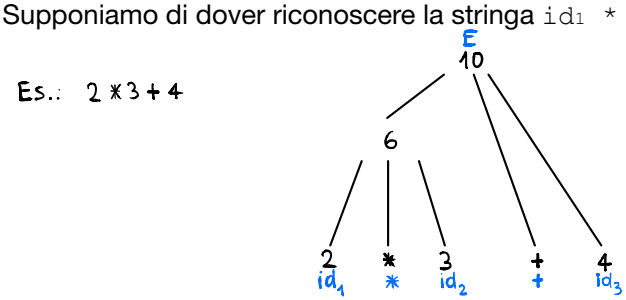
Supponiamo di dover riconoscere la stringa `id1 + id2 * id3`. La tabella, alla entry [5, *], contiene un **S/R conflict**.



Per esprimere la precedenza del *, bonifichiamo la entry [6, *]:



Supponiamo di dover riconoscere la stringa `id1 * id2 + id3`. La tabella, alla entry [6, è], contiene un **S/R conflict**.



Per esprimere la precedenza del *, bonifichiamo la entry [6, +]:

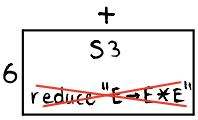


Tabella di parsing "depurata":

	id	+	*	$	E
0	S2				1
1		S3	S4	ACC	
2		E->id	E->id	E->id	
3	S2				5
4	S2				6
5		E->E+E	S4	E->E+E	
6		S3	E->E*E	E->E*E	

Qual è il vantaggio clamoroso che le grammatiche ambigue hanno rispetto alle altre?
Sono estremamente sintetiche e human-readable.