Iniziamo con un sommario di quello che abbiamo visto con il parsing canonical LR(1).

Se la tabella associata non contiene conflitti, allora la grammatica è LR. Se invece essa contiene una entry multiply-defined, cioè contenente più direttive, il parsing LR non potrebbe mai essere deterministico, e quindi la grammatica non è LR.

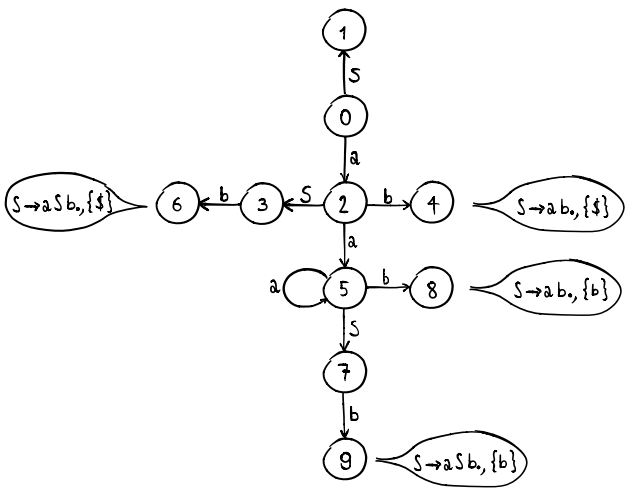
Un po' di terminologia
Item: ha la forma `A -> α.β`
Proj(`[A->α.β, Δ]`): è la prima componente, cioè `A -> α.β`.
Con proj(`I`) = $\cup_{i \in I}$ proj(`i`) intendiamo la proiezione di un certo insieme di item `I`.
I kernel item servono a discriminare se, durante la costruzione, uno stato è già stato inserito. Sono kernel item tutti quelli che non compaiono nella chiusura. Sono considerati kernel item quello iniziale: `S' ->.S` e tutti quelli dalla forma `A->α.β`, con $\alpha \neq \beta$.
Con kernel(`I`) intendiamo l'insieme dei kernel item di un certo insieme di item `I`.

Un po' di osservazioni
Supponiamo di avere due stati `I` e `J`, cioè due collezioni di item, di un qualche automa caratteristico.
Suppniamo anche che proj(kernel(`I`)) = proj(kernel(`J`)), ossia, l'insieme delle prime componenti dei kernel item di `I` è esattamente uguale all'insieme delle prime componenti dei kernel item di `J`.
Cosa possiamo dire di `I` e di `J`? Sono uguali? **No, dipende dal lookahead set** Δ.
Banalmente, uno prende `S -> aSb, Δ1` e `S -> aSb, Δ2`. Se Δ_1 e Δ_2 sono diversi, allora `I` è diverso da `J`.
Quindi, cosa possiamo dire di `I` e di `J`? È invece vero o no che proj(`I`) = proj(`J`)?
Vediamo. Supponiamo di avere la grammatica `S -> aSb | ab`. `I` nostri I e `J` sono:

```
S -> a.Sb, Δ1        S -> a.Sb, Δ2
- - - - - - -        - - - - - - -
S -> .aSb, {b}       S -> .aSb, {b}
S -> .ab, {b}        S -> .ab,{b}
```

Se proj(kernel(`I`)) = proj(kernel(`J`)), allora proj(closure$_1$(kernel(`I`))) e proj(closure$_1$(kernel(`J`))).

Riprendiamo la grammatica `S -> aSb | ab`. L'automa caratteristico era:



Durante la costruzione di questo automa caratteristico, era emerso che gli stati 2 e 5 hanno la caratteristica di avere la medesima proiezione:

Stato 2:

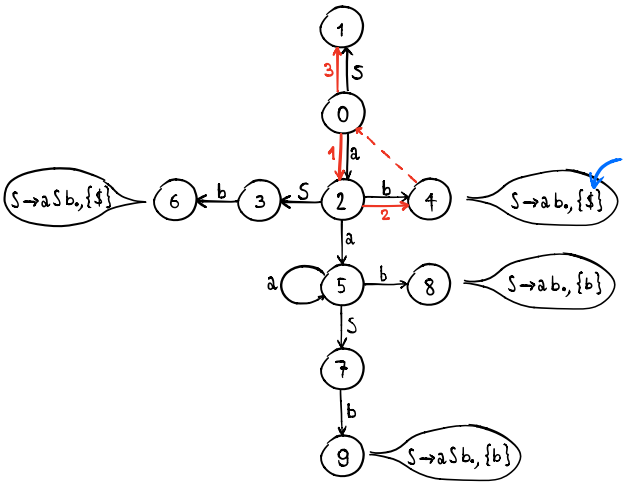
```
S -> a.Sb, {$}
S -> a.b, {$}
- - - - - - -
```

Stato 5:

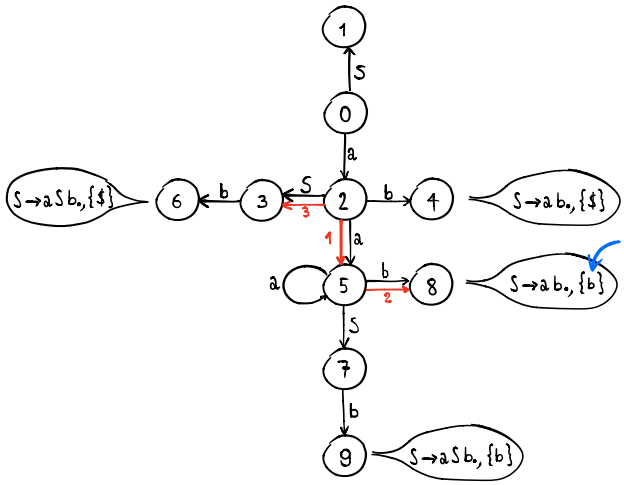
```
S -> a.Sb, {b}
S -> a.b, {b}
- - - - - - -
```

La maggior parte dei linguaggi di programmazione sono scritti in una qualche grammatica che può essere analizzata nella cosiddetta tecnica LALR. Questa tecnica si può introdurre in modo naturale come un derivato del canonical LR, in cui si vanno ad unificare gli stati che hanno la medesima prima proiezione.

In effetti, se guardiamo l'automa caratteristico di questa grammatica, ci accorgiamo dei seguenti "pattern", in due casi chiaramente diversi:



QUESTO PATTERN CORRISPONDE AL RICONOSCIMENTO DELLA PAROLA "ab". NOTIAMO CHE DIETRO HO IL '$'



QUESTO PATTERN CORRISPONDE AL RICONOSCIMENTO DELLA PORZIONE CENTRALE DI UNA PAROLA COME "aaaabbbb" (IN BLU). NOTIAMO CHE DIETRO HO LA 'b'

Il nostro obiettivo è ridurre il numero di stati dell'automa caratteristico.
Per conseguire ciò, possiamo scrivere grammatiche di classe LALR.
Le grammatiche di classe LALR hanno tabelle di parsing più piccole.
Tali tabelle possono essere generate in maniera non efficiente: prima facendo l'automa caratteristico LR e la relativa tabella di parsing LR, poi facendo l'unione degli stati con le stesse proiezioni.

Nel nostro esempio, dallo stato 2 e dallo stato 5 costruisco uno stato che ha la prima proiezione di 2 e di 5 e come lookahead set l'unione dei lookahead set di 2 e di 5. Sto dicendo che il 2 e il 5 diventano un unico stato. Dopodiché, visto che lo stato 2 e lo stato 5 avevano una b-transizione rispettivamente allo stato 4 e allo stato 8, cosa succede all'automa caratteristico? Ragioniamo: se 2 e 5 hanno la stessa proiezione, allora 4 e 8 hanno la stessa proiezione, in quanto 4 e 8 erano stati generati da due stati con la stessa proiezione del kernel.

Un ragionamento analogo ci conduce a mettere insieme le seguenti coppie di stati: 2 e 5, 4 e 8, 3 e 7, 6 e 9.

Guardiamo la tabella di parsing:

	a	b	$	S
0	S 2			1
1			acc	
2	S 5	S 4		3
3		S 6		
4			r S->ab	
5	S 5	S 8		7
6			r S->aSb	
7		S 9		
8		r S -> ab		
9		r S -> aSb		

Abbiamo detto, lo stato 2 e lo stato 5 hanno la stessa proiezione del kernel.
E infatti, se guardiamo la tabella, 2 e 5 hanno in comune shift-moves per gli stessi elementi.

Visualizziamo le prime proiezioni in comune:
6: `S -> aSb., {$}`
9: `S -> aSb., {b}`

3: `S -> aS.b, {$}`
7: `S -> aS.b, {b}`

Notiamo che stati con la medesima prima proiezione hanno il '.' davanti al medesimo simbolo.

4: `S -> ab., {$}`
8: `S -> ab., {b}`

Procediamo a fare il merge degli stati simili.
Nella tabella, facciamo il join delle informazioni che abbiamo.

	a	b	$	S
0	S 2			1
1			acc	
25	S 25	S 48		37
37		S 69		
48		r S->ab	r S->ab	
69		r S->aSb	r S->aSb	

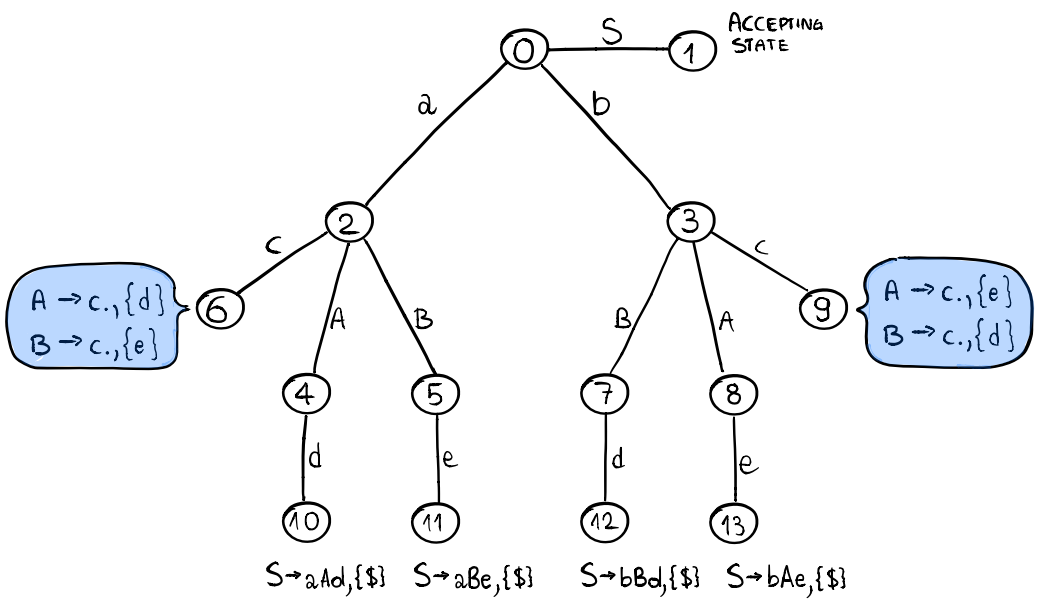
Sostanzialmente,
I: A -> α., Δ_1
J: A -> α., Δ_2
IJ: A -> α., Δ_1 U Δ_2

Abbiamo notato che la tabella si è ridotta di molto.
Ribadiamo che i linguaggi di programmazione si scrivono in modo tale che rientrino nella classe di grammatiche analizzabili con la tecnica LALR. Si dice che *una grammatica è LALR* se la tabella di ottenuta con la modalità LALR non contiene conflitti.

Tuttavia, noi abbiamo fatto l'esercizio nella maniera più inefficiente possibile, cioè prima costruendo la tabella di parsing.
Con calma, arriveremo a vedere algoritmi un po' più complessi che arrivano direttamente a generare la tabella LALR.

Ora prendiamo la seguente grammatica, che abbiamo visto parzialmente qualche giorno fa.

```
S -> aAd | bBd | aBe | bAe
A -> c
B -> c
```



C'è qualcosa che unifichiamo? Sì, lo stato 6 e lo stato 9, che hanno le stesse prime proiezioni. E come è fatta la tabella alla riga 6 e 9?

	e	d
6	r B->c	r A->c
9	r A->c	r B->c

Cosa ottengo quando metto insieme? Due R/R conflict! Nella tabella otterrei:

	e	d
69	r B->c	r A->c
	r A->c	r B->c

Quindi, questa grammatica è LR, ma non è LALR.

Visualizziamo questo fatto.

Se io leggo `ac`, io non so se la `c` che sto vedendo è un derivato della `A` o della `B`, a meno che non guardi dietro di me nell'input buffer. Se guardo nell'input buffer, questo dubbio me lo tolgo subito.
Se nell'input buffer leggo una `d`, allora so che la `c` è un derivato della `A`; se leggo una `e`, allora la `c` è un derivato della `B`.

Se leggo `bc`, il problema è analogo. A seconda di quello che leggo nell'input buffer (`e` oppure `d`),
so che la `c` è un derivato di `A` oppure di `B`.

Avere due stati separati nella tabella, lo stato 6 e lo stato 9, fanno questo servizio di capire da dove sto arrivando.

Se invece metto insieme, perdo la nozione della provenienza che avevo prima.

Disquisizione sui lookahead set nel parsing LR

Nel parsing LR, i lookahead set che noi trasportiamo da stato a stato servono, in sostanza, per raffinare gli insiemi dei follow del non terminale per il quale facciamo la riduzione, secondo un principio di località; cioè, appunto, di qual è il percorso che abbiamo seguito per arrivare allo stato nel quale facciamo la riduzione.

Quando facciamo il merge degli stati e otteniamo una tabella LALR, l'azione di unire i lookahead set di item che hanno la stessa proiezione significa che aumentiamo il numero di elementi per cui siamo disposti a fare una riduzione per una certa produzione.

Per qual motivo si vanno a scomodare queste tecniche così sofisticate?
Banalmente, potremmo semplificarci di molto la vita, mettendo d'ufficio, a ogni passo, i follow dei driver della produzione, anziché portarci avanti i lookahead set. Questa sarebbe la tecnica SLR (Simple LR).
Peccato che, se così facciamo, il nostro linguaggio non può avere i puntatori e una serie di altre feature, che con SLR non c'è verso di esprimere.

E c'è un'altra serie di casi simili a questo.

Grammatica:

```
S -> aAd | cAd
A -> e
```