Rivediamo quello che abbiamo cominciato ieri, cioè la costruzione dell'automa caratteristico. Usiamo un esempio che è un classico, in quanto include la nozione dei puntatori.

Partiamo dalla grammatica G:

```
S -> L = R | R
L -> * R | id
R -> L
```

`L = R` è l'assegnamento.
`L` può essere un puntatore ad `R`: `L -> R`, oppure un `id`.

Facciamo l'automa caratteristico per la grammatica, poi vediamo come da questo, facendo il merging degli stati, possiamo ottenere l'automa per il caso LALR.

Stato 0:

```
S' -> .S, {$}
- - - - - - -
Chiudo per S rispetto al $:
S -> .L=R, {$}
S -> .R, {$}
Chiudo sia per la L che per la R portandomi dietro il lookahead $
L -> .*R, {=}
L -> .id, {=}
R -> .L, {$}
La chiusura non è completata, in quanto ho inserito un item R -> .L, {$}. Aggiorno i seguenti item:
L -> .*R, {=,$}
L -> .id, {=,$}
```

Stato 1: τ(0, S): si legge "è la transizione dello stato 0 rispetto alla S":

```
S' -> S., {$}
```

Stato 2:

```
S -> L.=R, {$}
R -> L., {$}
```

Stato 3: τ(0, R)

```
S -> R., {$}
```

Stato 4: τ(0, *): ...e, vedi dopo, τ(4, *):

```
L -> *.R, {=,$}
- - - - - - - -
Dobbiamo chiudere la R rispetto al lookahead set {=,$}:
R -> .L, {=,$}
Dobbiamo chiudere la L rispetto al lookahead set {=,$}:
L -> .*R, {=,$}
L -> .id, {=,$}
```

Stato 5: τ(0, id): ...e, vedi dopo, τ(4, id):

```
L -> id., {=,$}
```

Dallo stato 2 abbiamo una transizione rispetto ad `=`.

Stato 6: τ(2, =):

```
S -> L=.R, {$}
- - - - - - - -
Dobbiamo chiudere la R:
R -> .L, {$}
Dobbiamo chiudere la L:
L -> .*R, {$}
L -> .id, {$}
```

Dallo stato 4 abbiamo una transizione rispetto alla `R`, una rispetto alla `L`, una rispetto ad `*`, una rispetto ad `id`:

Stato 7: τ(4, R):

```
L -> *R., {=,$}
```

Stato 8: τ(4, L):
`R -> L., {=,$}`

Quando consideriamo la transizone dallo stato 4 rispetto ad `*`, abbiamo un kernel item `L -> *.R, {=$}`, che è esattamente quello dello stato 4. Quindi, il target della `*`-transizione dallo stato 4 è lo stato 4 medesimo. Aggiungiamo allo stato 4: τ(4, *).

Quando consideriamo la transizione dallo stato 4 rispetto ad `id`, abbiamo un kernel item `L -> id., {=,$}`, che è esattamente quello dello stato 5. Aggiungiamo allo stato 5: τ(4, id).

Dallo stato 6 abbiamo una transizione rispetto alla `R`, una rispetto alla `L`, una rispetto ad `*`, una rispetto ad `id`:

Stato 9: τ(6, R):
`S -> L=R., {$}`

Stato 10: τ(6, L): ...e, vedi dopo, τ(11, L):
`R -> L., {$} //Noto che la proiezione è uguale a quella dello stato 8, ma il lookahead set è diverso`

Stato 11: τ(6, *): ...e, vedi dopo, τ(11, *) e τ(11, id).
```
L -> *.R, {$} //Noto che la proiezione è uguale a quella dello stato 7, ma il lookahead set è diverso
- - - - - - - -
Dobbiamo chiudere la R:
R -> .L, {$}
Dobbiamo chiudere la L:
L -> .*R, {$}
L -> .id, {$}
```

Stato 12: τ(6, id):
`L -> id., {$} //Noto che la proiezione è uguale a quella dello stato 5, ma il lookahead set è diverso`

Dallo stato 11 abbiamo una transizione rispetto alla `R`, una rispetto alla `L`, una rispetto a *

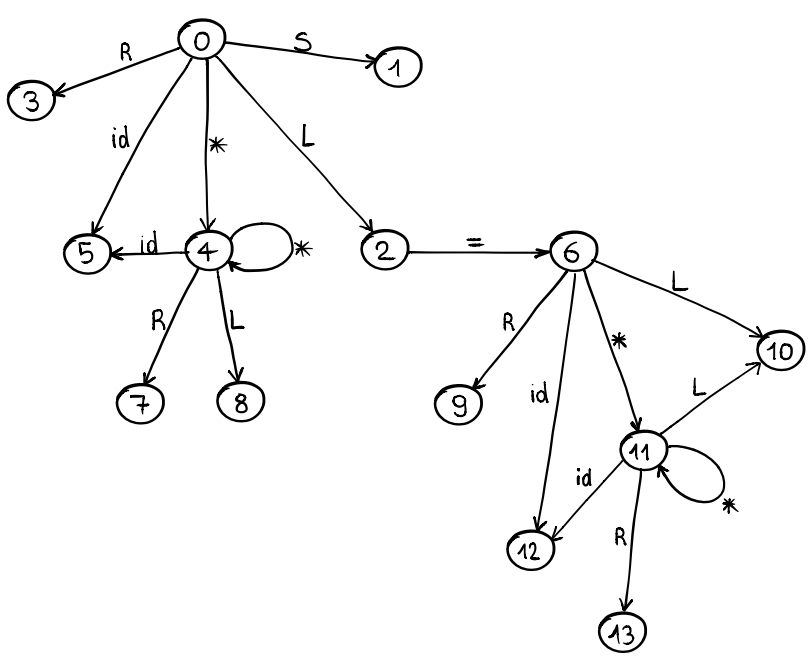
Stato 13: τ(11, R):
`L -> *R., {$} //Noto che la proiezione è uguale a quella dello stato 7, ma il lookahead set è diverso`

Quando consideriamo la transizione dello stato 11 rispetto alla `L`, abbiamo un kernel item `R -> L., {$}`, che è esattamente quello dello stato 10. Aggiungiamo allo stato 10: τ(11, L).

Quando consideriamo la transizione dello stato 11 rispetto a `*`, abbiamo un kernel item `L -> .*R, {$}`, che è esattamente quello dello stato 11 stesso. Aggiungiamo allo stato 11: τ(11, *).

Quando consideriamo la transizione dello stato 11 rispetto a `id`, abbiamo un kernel item `L -> id., {$}`, che è esattamente quello dello stato 12. Aggiungiamo allo stato 12: τ(11, id).

Automa:



Per costruire l'automa unificato (o *merged automa*), e quindi arrivare a costruire la tabella LALR di questa grammatica, dobbiamo innanzitutto individuare gli *stati equivalenti*.

Il linguaggio generato dalla grammatica comprende:
- Stringhe del tipo `***...*id`,
- Stringhe contenenti uno e un solo `=`.

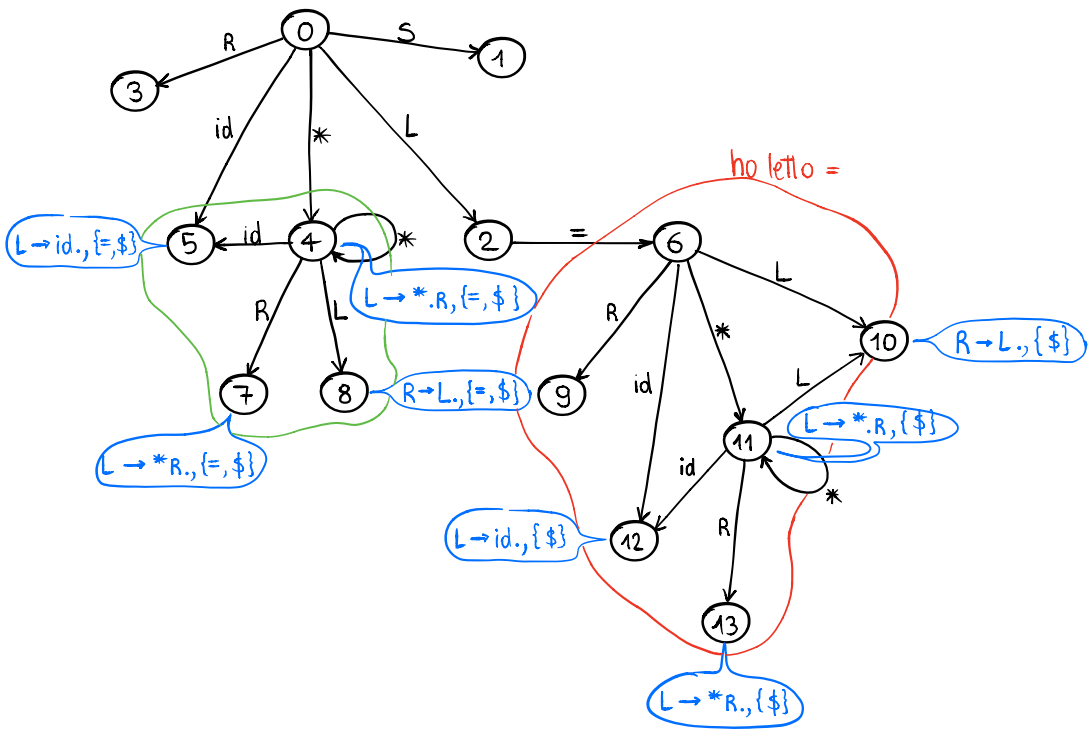
Poniamo l'attenzione sugli stati 5 ed 8.

Lo stato 5, con l'item `L -> id., {=,$}`, dice: "Sono disposto a dire che questo `id` viene dalla `L` se dopo di me vedo o l'`=` o il `$`".
Questo vuol dire che, se sono nello stato 5, posso o trovarmi alla fine della stringa, o essere l'`id` immediatamente a sinistra dell'`=`.

Lo stato 12, con l'item `L -> id., {$}`, dice: "Sono disposto a dire che questo `id` viene dalla `L` solo se dopo di me vedo il `$`".
Se sono nello stato 12 vuol dire che ho già avuto occasione di scoprire lo stato 5.

Abbiamo che gli stati 5 ed 8 parlano della stessa cosa, ma in momenti distinti del riconoscimento di una stringa.

Questo ragionamento vale per tutte le seguenti coppie di stati: `10-8`, `11-4`, `12-5`, `13-7`.



Questi stati sono quelli che metteremo insieme per fare il merged automa.

Dunque, l'idea è quella di ottenere partizioni di stati equivalenti in cui la nozione di equivalenza riguarda la prima proiezione.

Merged automata

Sia `A` l'automa caratteristico per il parsing LR(1).
Sia `Am` l'automa ottenuto dal merging degli stati poter ragionare sulle tabelle di parsing LALR(1).

Dobbiamo rispondere a tre diverse domande.

Chi sono gli stati di `Am`*?*

Stati di `Am`: ogni stato di `Am` rappresenta la classe di stati di `A` con medesima prima proiezione.

Gli stati 0, 1, 2, 3, 6, 9 non hanno proiezioni equivalenti.

Due stati come il 10: `R -> L., {$}` e l'8: `R -> L., {=, $}` hanno medesima proiezione,
e confluiscono in un unico stato con `R -> L., {$}` ∪ `{=, $}` ossia `R -> L., {=, $}`.

Quali di sono le transizioni di `Am`*?*

Siccome stiamo facendo un automa, dobbiamo descrivere come sono le transizioni, che entrano ed escono nei nuovi stati che abbiamo costruito. Come facciamo a rispondere?
Possiamo innanzitutto affermare che le transizioni entranti ed uscenti in e da due stati che hanno la stessa proiezione non dipendono certamente dal lookahead set. (Il lookahead set fa una discriminazione dei possibili FOLLOW dei non terminali che dipendono dal cammino dal quale proveniamo).
Le transizioni uscenti si calcolano andando a guardare chi c'è alla destra del '`.`' all'interno della proiezione dell'item.
Se uno stato ha una certa transizione, allora per forza la ha anche uno stato equivalente.
Chi è il target? Da due stati che hanno la medesima proiezione (ma lookahead differente), posso: o finire in un unico stato (non c'erano altri stati con la stessa proiezione), o finire in due stati con medesima proiezione (ma lookahead differente), cioè due stati di una stessa classe.
Ribadiamo: le transizioni dell'automa giocano sulla proiezione, il lookahead set non c'entra nulla!

Nell'automa dell'**esempio**:
Né lo stato 10 né lo stato 8 hanno transizioni e quindi target.
Lo stato 11 e lo stato 4 sono ricchi di transizioni:

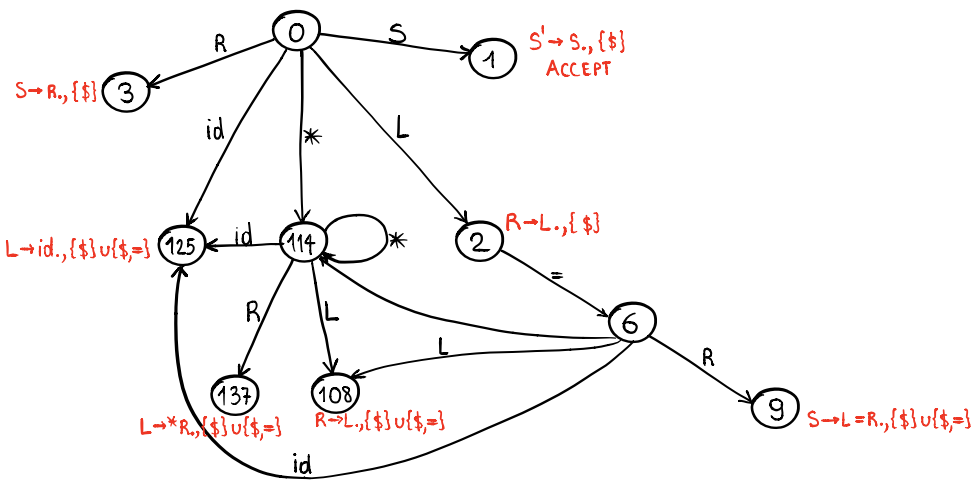
- L'11 con una id-transizione va in 12. Il 4 con una id-transizione va in 5. Verifichiamo che c'è (necessariamente) una coppia 12-5. Quindi, lo stato 114 avrà una id-transizione allo stato 125.
- L'11 con una R-transizione va in 13, il 4 con una R-transizione va in 7. Verifichiamo che c'è una coppia 13-7. Quindi, lo stato 114 avrà una R-transizione allo stato 137.
- L'11 con una L-transizione va in 10, il 4 con una L-transizione va in 8. Verifichiamo che c'è una coppia 10-8. Quindi, lo stato 114 avrà una L-transizione allo stato 108.
- L'11 con una *-transizione va in 11, il 4 con una *-transizione va in 4. Verifichiamo che c'è una coppia 11-4. Quindi, lo stato 114 avrà una *-transizione allo stato 114.

Il merging è un'operazione di unione che facciamo per risparmiare sugli stati che sono più o meno simili, ma che hanno lookahead set diversi.

Definiamo formalmente le transizioni di `Am`: se lo stato `M` di `Am` è tale che la proj(`M`) è uguale a proj(`L`) con `L` stato di `A` (l'automa originario), e se `L` ha una `Y`-transizione ad `L'`, allora `M` ha una `Y`-transizione allo stato `M'` tale che proj(`M'`) è uguale a proj(`L'`).

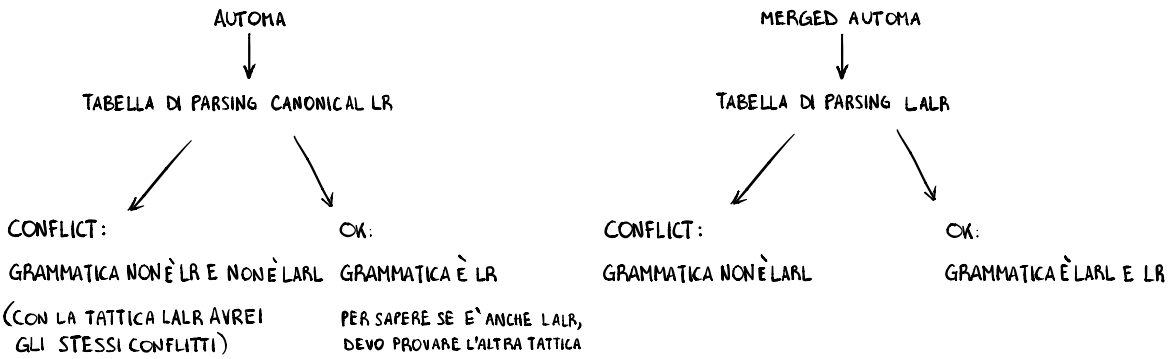
Per capire questa definizione, prova con `M` = 114, `L` = 4, `Y` = `R`, `L'` = `7`.

Come trattiamo i reducing item?
Una volta capito l'automa, siamo in grado di collocare nella tabella le mosse di shift e le mosse di goto.
Ci resta da determinare come collocare le mosse di reduce, che dipendono dai reducing items e dai loro lookahead set.
Disegniamo l'automa che si ottiene, evidenziamo i reducing items.



Da questo automa, con quelle informazioni sui reducing item e sui lookahead set, si costruisce, con lo stesso criterio che si adopera per costruire la tabella di parsing LR(1), la tabella di parsing LALR(1).

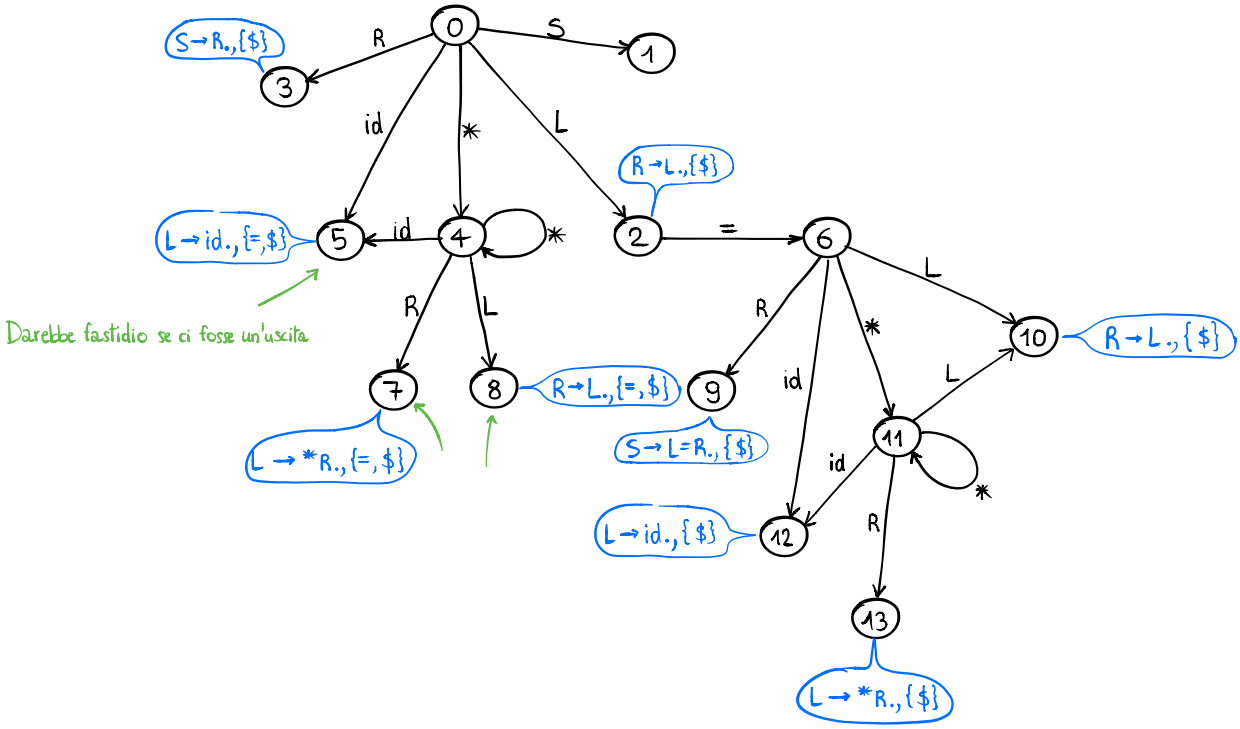
Quali conflitti possono essere introdotti dall'operazione di merge degli stati?
È possibile che l'operazione di merge degli stati causi l'introduzione di conflitti reduce-reduce, ma mai di shift-reduce.
Ribadiamo: se facciamo questa stessa operazione direttamente sulla tabella, sostituendo le righe, non possiamo mai andare a inserire dei conflitti di shift-reduce, perché l'operazione che qui facciamo, il merge degli stati, è mettere insieme tutti e quanti gli item che appartengono a stati con la stessa proiezione. La mossa dello shift dipende dalle transizioni dell'automa, quindi dipende dalla produzione dell'item. Se noi abbiamo unito due stati, l'unico possibile tipo di conflitto che possiamo generare, che non era possibile nella tabella LR e invece c'è nella tabella LALR, è un conflitto reduce-reduce.



All'inizio del corso eravamo in grado di determinare se la grammatica era context free, context dependent, regolare.
Ciò che rende fattibile questa prima grande suddivisione è il fatto che sono proprietà visibili a colpo d'occhio.

L'analizzabilità di una grammatica, invece, non è normalmente visibile. È visibile se la grammatica è a misura d'uomo, ossia se consiste di poche produzioni. Nessuno riesce a dire se una grammatica con numerose produzioni è, ad esempio, LALR.
Si parla di classi di grammatiche rispetto alla tecnica che funziona per la loro analisi.
Per dimostrare che una grammatica sta in una certa classe, bisogna verificare che la tabella di parsing per quella classe non contenga entry multiply-defined.
Quindi, si dice che una grammatica è LALR se la tabella di parsing LALR non ha entry multiply-defined.

La grammatica che abbiamo visto oggi è LR?
Possiamo rispondere anche dando un'occhiata all'automa:



Non ho stati che hanno più di un reducing item, con proiezioni diverse e lookahead set uguali → No conflitti reduce/reduce.
Non ho stati che hanno sia un reducing item il cui lookahead set contiene un certo simbolo `s`
che una transizione uscente per s → No conflitti shift/reduce.

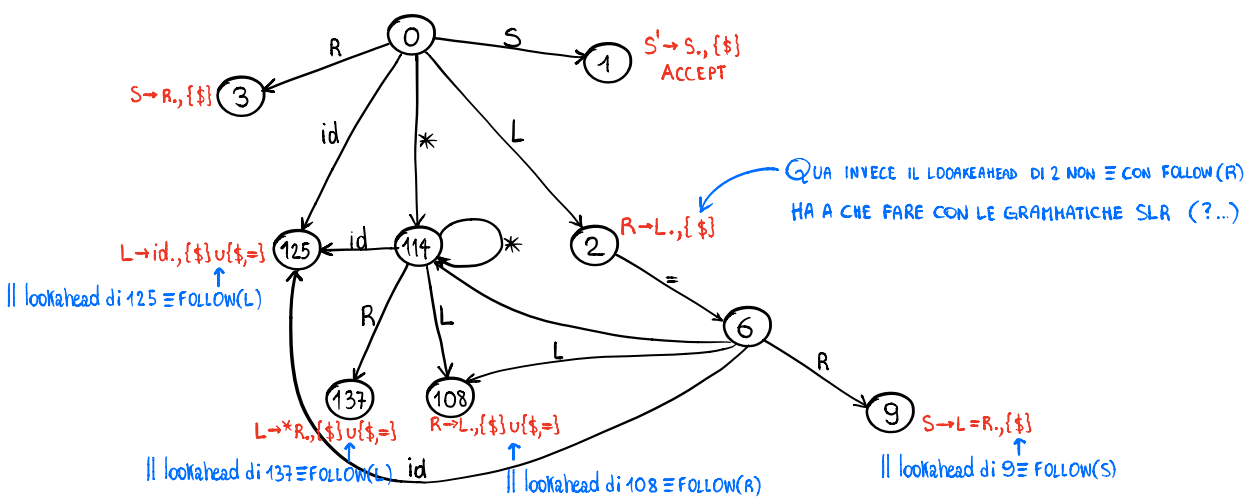
La risposta è sì: l'automa è LR.

Abbiamo detto che, portandoci dietro negli item le informazioni sui lookahead set, quello che stiamo facendo è andare a guardare proprio quale produzione specifica è stata utilizzata, e quindi dare un flavour di località alla computazione dei FOLLOW.

Detto questo, sia nel caso della computazione dell'automa caratteristico per il parsing canonical LR,
che nel caso dell'automa per il parsing LALR,
quei lookaehad set sono dei sottoinsiemi dei FOLLOW del driver della produzione per la quale riduciamo.

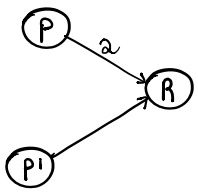
Nella grammatica di oggi, abbiamo i seguenti FOLLOW:
```
S:  $
L:  $, =
R:  $, =
```

Diamo un'occhiata all'automa. C'è una stranezza:

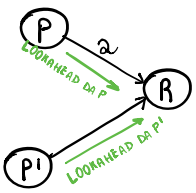


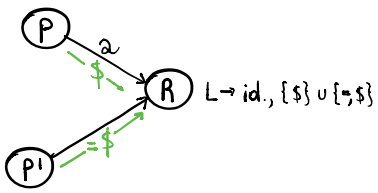
Nello stato 2, che rimane tale sia nell'automa LR(1) che nell'automa LALR, noi trovavamo l'item `R -> L., {$}`.
Sappiamo che un altro possibile FOLLOW di `R` è =.
Osservo che possiamo raggiungere lo stato 2 esattamente quando dallo stato 0 abbiamo visto la `L`, non ancora l'=.
E infatti, quali sono le stringhe per le quali lì faremo una riduzione? Quelle per le quali abbiamo scelto la produzione `S -> R`.

Anteprima della lezione successiva: costruzione di un automa caratteristico simbolico che fa l'automa merged on the fly.
Il procedimento che abbiamo finora seguito era il seguente.
Parto con lo stato iniziale, per poi cominciare a considerare tutte le transizioni da quello stato. Dallo stato iniziale genero altri stati. Per ognuno dei nuovi stati, considero chi sono i target secondo le possibili transizioni. Ma non mi viene detto in che ordine farlo. Infatti, durante il procedimento, capita spesso che venga riconosciuto che uno stato precedentemente generato è il target della transizione da un altro.
Supponiamo di essere partiti da uno stato `P`. Abbiamo considerato la sua transizione rispetto alla `a`, e per questo aggiungiamo alla collezione di stati lo stato `R`. Procediamo e aggiungiamo molti altri stati... Ad un certo punto, ci accorgiamo che `R` è anche target di un certo stato `P'`.



Per come funziona l'operazione di merging, l'item in R prende i lookahead set che arrivano da `P` e da `P'`:





Il nuovo sistema che vedremo consiste nel fare al volo la costruzione di un automa, che non è istanziato, ma che lascia informazioni simboliche, che riguardano i lookeahad set.
Si inizia esattamente come prima, con le solite elucubrazioni sugli item, ma si lasciano in maniera simbolica le informazioni sui possibili lookahead set. Quando si è finita la costruzione, cioè si è certi di aver raccolto tutti i possibili contributi da tutte le possibili parti, perché tutte le transizioni sono state guardate, si prendono le informazioni simboliche che riguardano i lookehad set e si fanno processare ad sistema di equazioni...