Per farci un'idea dell'algoritmo annunciato alla fine della lezione precedente, possiamo subito vederlo in azione.

Questo algoritmo prevede che il lookahead set dei kernel item di uno stato contengano variabili, che, durante l'operazione di chiusura dello stato, sono da considerare come simboli non terminali.

Nell'esempio, sviluppiamo in contemporanea l'automa caratteristico e il sistema di equazioni sulle variabili.

Prendiamo una semplice grammatica LALR, e quindi LR.

```
S -> aSb | ab
```

Stato iniziale 0:

```
S' -> .S, {x0}
- - - - - - -
S -> .aSb, {x0}
S -> .ab, {x0}
```

Nel sistema di equazioni, aggiungo $x_0 = \{\$\}$

Dallo stato 0 avrò due transizioni, una rispetto alla `S` e una rispetto alla `a`.

Rispetto alla `S`, andiamo nello stato 1:

```
S' -> S., {x1}
```

Variabile nuova. La chiusura LR(1) avrebbe prescritto di mettere $\{x_0\}$ come lookahead set $x_1 = x_0$

Rispetto alla `a`, andiamo nello stato 2:

```
S -> a.Sb, {x2}
S -> a.b, {x3}
- - - - - - - -
S -> .aSb, {b}
S -> .ab, {b}
```

x_2 eredita ciò che viene da x_0: $x_2 = x_0$

x_3 eredita ciò che viene da x_0 $x_3 = x_0$

$b = \text{FIRST}(bx_2)$

$b = \text{FIRST}(bx_3)$ Nell'operazione di chiusura, x_2 e x_3 sono trattati come terminali

Per lo stato 0 abbiamo considerato tutte le possibili transizioni. Lo stato 1 non ha transizioni.

Dallo stato 2 abbiamo tre transizioni, una per la `S`, una per la `b` e una per la `a`.

Rispetto alla `S`, andiamo nello stato 3:

```
S -> aS.b, {x4}
```

$x_4 = x_2$

Rispetto alla `b`, andiamo nello stato 4:

```
S -> ab., {x5}
```

$x_5 = x_3$

Rispetto alla `a`, la proiezione del kernel sarebbe:

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

Dovrei portare dietro l'informazione che il lookahead set è {b}

Anche da questa parte

$x_2 = x_0 \cup \{b\}$

$x_3 = x_0 \cup \{b\}$

Abbiamo già uno stato che ha questa proiezione del kernel, lo stato 2.
A questo punto, non vogliamo rifare tutta quanta la chiusura: semplicemente, lo riconosciamo, e diciamo che la transizione dallo stato 2 rispetto alla a, torna nello stato 2, ma provoca l'aggiunta di elementi all'insieme rappresentato da `x2` e dall'insieme rappresentato da `x3`.

Per lo stato 2 abbiamo considerato tutte le possibili transizioni.

Dallo stato 3 abbiamo una transizione, per la `b`.

Rispetto alla `b`, andiamo nello stato 5:

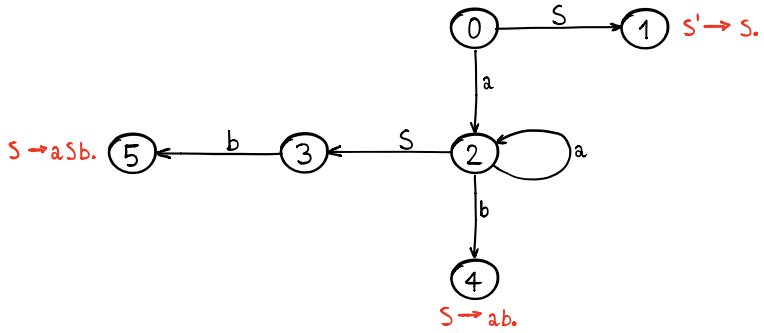
```
S -> aSb., {x6}
```

$x_6 = x_4$

Per lo stato 3 abbiamo considerato tutte le possibili transizioni. Lo stato 4 non ha transizioni, e anche lo stato 5.

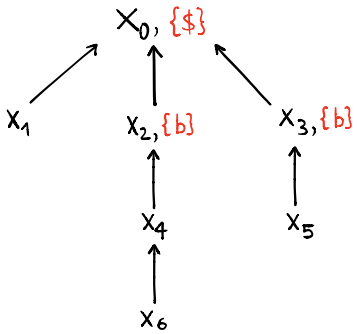
Abbiamo finito.

Disegniamo il layout dell'automa.



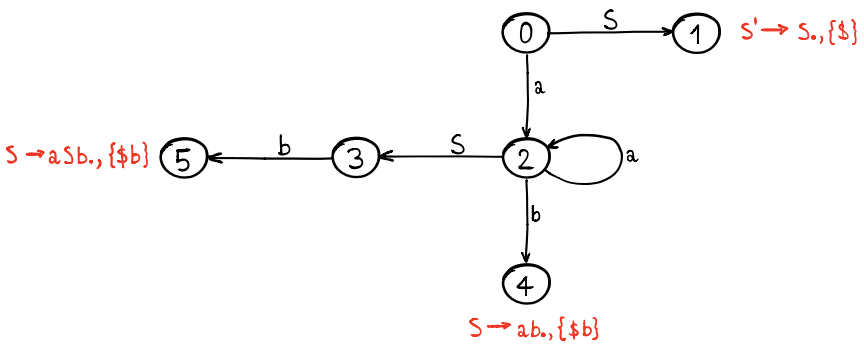
Ora dobbiamo capire quale è il valore da mettere nei lookahead set.

Si tratta di realizzare un grafo delle dipendenze. "Dipende" := "La sua definizione usa".
Nei nodi metto anche gli elementi *ground*, cioè quelli non variabili.



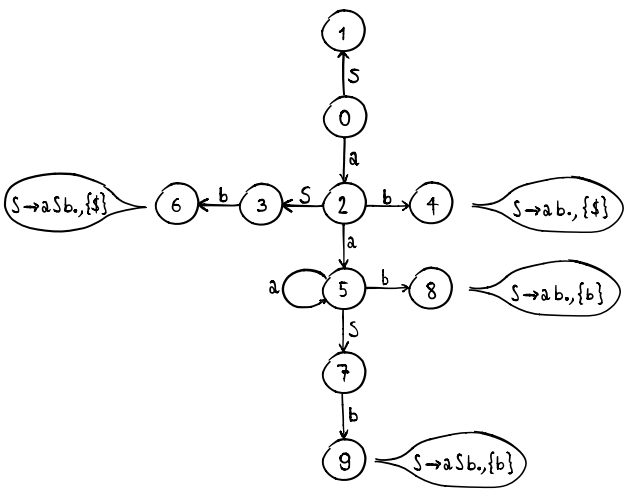
Abbiamo:
$X_0 = \{\$\} = X_1$
$X_2 = X_4 = X_6 = X_3 = X_5 = \{\$, b\}$

Ora possiamo riempire i lookahead set dell'automa:



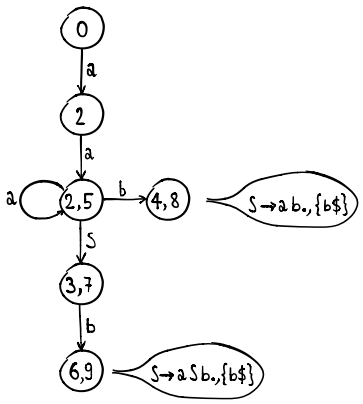
Ora potrei riempire la tabella di parsing: le transizioni per i terminali danno le mosse di shift,
le transizioni per i non-terminali danno le mosse di goto, per i passi di riduzione uso le informazioni dei reducing item
e i loro lookahead set.

Per la medesima grammatica, durante lo studio del parsing LR(1), avevamo ottenuto il seguente automa:



Li stati con la medesima proiezione, cioè quelli che andremmo ad unire se facessimo la costruzione prevista dalla tattica LALR, sono: 3 e 7, 4 e 8, 2 e 5, 6 e 9.

Il layout dell'automa merged è:



Illustriamo brevemente l'algoritmo.

Costruzione dell'automa caratteristico simbolico per parsing LALR(1)
Inizializzare la collezione di stati con closure$_1$({`[S -> .S, {x0}]`}) con `x0` nuova variabile
Inizializzare il sistema di equazioni con `x0` = {$}
Tag `P0` come non marcato
while c'è uno stato `P` non marcato nella collezione **do**
    tag `P` come marcato
    foreach `Y` che compare a destra del marker in un qualche item di `P` **do** % Fin qua, schema identico a quello del parsing LR(1)
        % Quello che cambia è la gestione delle variabili
        % Computazione temporanea del kernel del Y-target di P
        tmp = ∅
        foreach `[A -> α.Yβ, Δ]` ∈ `P` **do** aggiungere `[A -> αY.β, Δ]` a tmp
        % Distinguiamo i due casi
        if proj(tmp) = proj(kernel(`Q`)) per qualche `Q` già nella collezione **then** % Abbiamo già generato uno stato con questa proiezione
            foreach `[A -> αY.β, Δ]` ∈ `P`, `[A -> αY.β, {x}]` ∈ `Q` **do**
                % Aggiungiamo, a tutte quante le equazioni che sono associate ai vari item del kernel dello stato
                % che abbiamo trovato come target, le componenti che vengono da Δ.
                aggiornare l'equazione x = `Δ'` a x = `Δ'` ∪ `Δ`
                τ(P,Y) = `Q`
        else % Caso in cui generiamo un nuovo stato
            foreach `[A -> αY.β, Δ]` ∈ tmp **do**
                rimpiazzare `Δ` in tmp con una nuova variabile `x`
                inserire `x` = `Δ` nel sistema di equazioni
                generare come non marcato il nuovo stato `Q` = closure$_1$(tmp)
                τ(P,Y) = `Q`

Anticipazione della prossima lezione
La prossima lezione vedremo le **syntax directed definitions**, cioè daremo un senso al nostro studio fatto fino ad ora.
Vedremo la compilazione nella sua forma in assoluto più semplice, che consiste nell'arricchire le grammatiche con delle azioni semantiche. Se noi sappiamo appropriatamente organizzare la grammatica e scrivere le azioni semantiche associate alle produzioni, possiamo conseguire la generazione di codice intermedio, ottenendo così il front-end della compilazione.
Quale è il ruolo delle azioni semantiche? Nell'ambito dell'analisi sintattica bottom-up, ogni qual volta avviene una riduzione di una certa proiezione, viene contestualmente eseguito il codice associato con l'azione semantica di quella produzione. Se noi sappiamo capire quando è che capita un passo di riduzione, sappiamo anche gestire la generazione del codice intermedio.