Il canonical LR(1) parsing utilizza l'algoritmo shift-reduce.

L'algoritmo shift-reduce è usato da diverse tecniche di parsing bottom-up. Quello che differenza le varie tecniche è il modo con cui viene costruita la tabella. Nel caso del canonical LR(1) prasing, la tabella viene costruita utilizzando un automa caratteristico che si basa sugli LR(1)-item.

Gli LR item sono coppie `A -> α.β, Δ`:
- La componente `A -> α.β` si dice anche LR(0)-item, o semplicemente item. Un *item* è un elemento che contiene almeno una produzione della grammatica, con end-marker denotato da un '`.`' nel body della produzione.
- La componente `Δ` è il lookahead-set, che è un sottoinsieme `Δ ⊆ T ∪ {$}`, dove `$` è utilizzato come end-marker per le stringhe che voglio analizzare.

Terminologia per gli item
Un item `I = [A -> α.β, Δ]` si dice:
- Item Iniziale, se `A -> α.β` è l'item speciale `S' -> .S`; ricordiamo che S' è un non-terminale nuovo rispetto alla grammatica data. Questa produzione, `S' -> .S`, arricchisce la grammatica.
- Ciascun item indica rispetto alla procedura di parsing, che si è già vista/analizzata una certa porzione dell'input buffer, e che questa porzione, rispetto alla produzione indicata dall'item, è tale per cui c'è una parte alfa per la quale è stato tolto il matching con l'input che è stato dato. Quindi questo item è iniziale in quanto essendo il punto immediatamente alla sinistra dello start symbol della grammatica, sta a indicare che ancora non è stato visto nulla della stringa da analizzare.
- Item finale, se la componente `A -> α.β` è tale per cui il '`.`' è esattamente alla destra della `S`, ed indica la situazione opposta: è stato matchato tutto l'input visto rispetto ad una possibile derivazione della `S`. Serve a capire che siamo arrivati in uno stato.
- Kernel item, se `I` è iniziale, oppure se `α` è diverso da epsilon. Pertanto, sono kernel item tutti quelli in cui il '`.`' non è esattamente all'inizio della produzione, con un unica eccezione per l'item iniziale, in cui il '`.`' capita alla sinistra del body della produzione.
- Closure item, o *di chiusura*, se non è kernel item.
- Reducing item, o *di riduzione*, se nella produzione il '`.`' capita all'estrema destra del body, ossia se `β=ε` oppure se `I` non è finale.

Ci rimane da dire cos'è l'operazione di closure che applichiamo ad un insieme di item: $\text{closure}_1(\{[S' \to .S, \{\$\}]\})$.

Nella costruzione dell'automa, nel caso della subset construction, seguivamo questi passi:
"Capivamo come ci muovevamo con un determinato simbolo. Una volta capito ciò, fissavamo l'insieme degli stati possibili dopo aver letto quel simbolo. Una volta fissato questo nucleo di base, calcolavamo l'espilon closure, cioè andavamo a vedere dove, da questi stati, ci potevamo ancora muovere, utilizzando dei passi di tipo epsilon."

Quello che facciamo qui è:
Stabilire qual è l'insieme nucleo di item che deve appartenere ad uno stato target rispetto ad una certa transizione dell'automa, e una volta capito quale è questo insieme di item, la chiusura si occupa di dire quali sono gli item che rappresentano delle condizioni equivalenti rispetto alla procedura di parsing. E cioè, supponendo per esempio di partire da un item di questo genere: `A -> α.Bβ, Δ`, con un certo lookahead `Δ`, il minimo che si vuole è dire:
Se io, durante il parsing, mi trovo in uno stato in cui ho già visto la porzione di stringa che corrisponde all'espansione di `α`, dopo la quale, per poter vedere `A`, dovrò vedere l'espansione della `B` e l'espansione della `β`, allora è equivalente dire, dal punto di vista dell'algoritmo che noi poi eseguiamo per effettuare il parsing, che se la grammatica contiene *n* produzioni per `B` che hanno la forma `B -> γ1, ..., B -> γn`, allora, dopo aver visto l'espansione di `α`, visto che dobbiamo ancora vedere l'espansione di `Bβ`, potremmo anche trovarci in una qualunque delle posizioni iniziali per queste produzioni della `B`. Quindi, dimenticando per intanto la `Δ`, a un insieme di questo tipo, uno ci aggiunge comunque tutti gli item che sono *di chiusura* per `B`, cioè che hanno il punto esattamente all'inizio del body delle produzioni della `B`: `B -> .γ1, Δ1; ...; B -> .γn, Δn`, per qualche `Δ1,...,Δn`.
Naturalmente, qui la ricorsione è regina: chi sono questi `γ1 ... γn`? Per esempio, potrebbe essere che `γ1` è una cosa scritta così: `γ1 = cδ1`. Ma allora, se io sono all'inizio di `δ1` e se `δ1` comincia con `c`, è come dire che una delle possibilità è che io veda l'espansione della `c` seguita dall'espansione di `δ1`, e allora non basta l'aver aggiunto questo, ma bisogna comunque aggiungere tutto ciò che risulta equivalente rispetto all'algoritmo che dobbiamo eseguire poi per lo shift reduce, per tutte e quante le produzioni che sono state elencate all'interno dell'insieme che stiamo costruendo, e in cui capita che ci sia il '`.`' esattamente davanti ai non-terminali (perché noi sappiamo che, per ogni non-terminale, c'è una produzione).
Quindi uno continua a mettere questo produzioni all'interno dell'insieme; in questo caso specifico, supponendo che per `c` ci sia la produzione `c -> a`, uno metterebbe anche `c -> .a`, con un qualche lookahead `Δm`. E così via per tutti quei casi in cui uno ha inserito una produzione che ha un '`.`' davanti a un non-terminale.
Se invece il '.' è davanti un terminale, la chiusura non deve aggiungere niente, perché quello che uno aspetta di vedere è esattamente quel terminale nell'input-buffer, che non può essere travestito da altro: se è il terminale `c`, niente lo può cambiare.
Ora la prof spiega (non molto chiaramente) come sistemare i lookahead set; ometto questa parte. Andiamo direttamente agli esercizi.

Esempio

Grammatica:
`S -> aAd | cAb`
`A -> e`
Calcoliamo l'automa caratteristico, considerando che, per quanto riguarda la closure, non ne abbiamo ancora dato una definizione formale, ma ne abbiamo più o meno l'idea del funzionamento.

Stato iniziale:
Lo stato iniziale dell'automa caratteristico ha un kernel set con l'item `S' -> .S, {$}`, in cui `S' -> .S` è una produzione che arricchisce la grammatica.
Dobbiamo ora computare il closure set del kernel set.
Un item che vogliamo chiudere ha la forma `A -> α.Bβ`. Il lookahead di un item prodotto dalla chiusura è $\Delta_1 = \bigcup_{d \in \Delta} FIRST(\beta d)$
Dove β sono le stringhe che seguono `B`.
Siccome qui β=ε e Δ ha solo un elemento, il `$`, è molto facile e abbiamo che Δ1 = FIRST(`$`) = `$`.
Pertanto, il closure set include gli item `S -> .aAd, {$}` e `S -> .cAb, {$}`.
Ora ci chiediamo se abbiamo aggiunto item che devono essere chiusi, ma non è questo il caso, in quanto il punto sta, per ognuno dei due item raggiunti, davanti a un non-terminale. Quindi, questo è lo stato iniziale:

```
S' -> .S, {$}     } KERNEL SET
- - - - - - - -
S -> .aAd, {$}    }
S -> .cAb, {$}    } CLOSURE SET
```

Ogni stato ha tante transizioni quanti sono gli elementi del vocabolario che capitano esattamente alla destra del '`.`'.
Quindi, per il primo stato, ho tre transizioni: una per `S`, una per `a` e una per `c`.

Per costruire lo stato di una transizione, si prendendo tutti e quanti gli item che hanno un '`.`' davanti all'elemento per cui si sta costruendo la transizione, spostandolo da davanti a dietro.

Stato 1:
Dallo stato iniziale mi muovo con una `S`-transizione nello stato 1 che contiene nel kernel set l'item `S -> S., {$}`.
La chiusura non ha niente da aggiungere, in quanto il '`.`' non sta alla sinistra di un non-terminale. Quindi, lo Stato 1 è:

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

Stato 2:
Dallo stato iniziale mi muovo con una `a`-transizione nello stato 2 che contiene nel kernel set l'item `S -> a.Ad, {$}`.
Dal momento che il punto sta davanti ad A, un un non-terminale, dobbiamo aggiungere item alla chiusura.
La grammatica ha una sola produzione per la `A` che è `A -> e`.
Il closure set è quindi costituito dall'item `A -> .e, {d}` dove `d` è il FIRST di ciò che segue la `A`, quindi FIRST(`d$`) = `d`.
La chiusura non ha niente da aggiungere, in quanto il '`.`' non sta alla sinistra di un non-terminale. Quindi, lo stato 2 è:

```
S -> a.Ad, {$}
- - - - - - - -
A -> .e, {d}
```

Stato 3:
Dallo stato iniziale mi muovo con una `c`-transizione nello stato 3 che contiene nel kernel set l'item `S -> c.Ab, {$}`.

```
S -> c.Ab, {$}
- - - - - - - -
A -> .e, {b}
```

Dallo stato 2 abbiamo due transizioni: una per la `A`, una per la `e`.
Stato 4:

```
S -> aA.d, {$}
```

Stato 5:

```
A -> e., {d}
```

Dallo stato 3 abbiamo due transizioni: una per la `A`, una per la `e`.
Stato 6:

```
S -> cA.b, {$}
```

Stato 7:

```
A -> e., {b} //Va bene, non lo ho già: ha un lookahead diverso da quello dell'item nello stato 5.
```

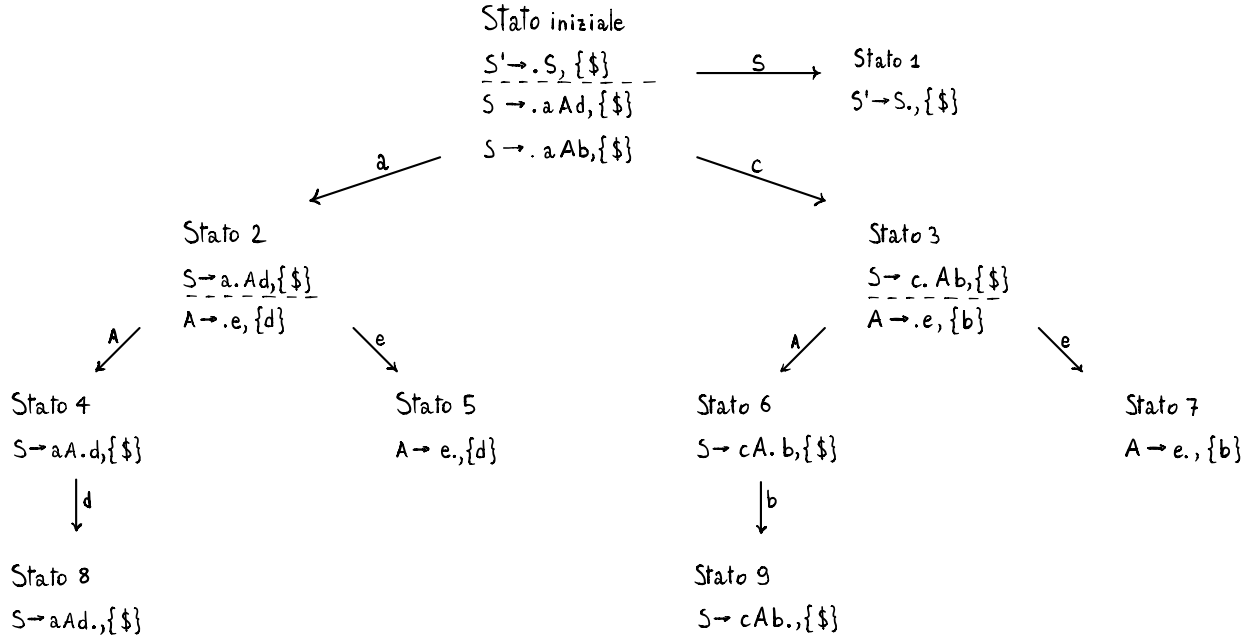
Dallo stato 4 abbiamo una `d`-transizione verso lo stato 8.
Stato 8:

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

Dallo stato 6 abbiamo una `b` transizione verso lo stato 9.
Stato 9:

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

Ora abbiamo finito.



Vediamo ora l'**algoritmo per il calcolo della chiusura di un set P di LR(1)-item**.

function closure1(P)
    tag every item in P as unmarked;
    while there is an unmarked item I in P do
        mark I;
        if I has the form [A -> α.Bβ, Δ] **then**
            Δ1 ← ⋃ d∈Δ FIRST(βd); // Δ1 = FIRST (βd1) ∪ . . . ∪ FIRST(βdn).
            foreach B→γ ∈ G do
                if B→.γ ∉ prj(P) **then** // Se ancora non c'è un item che ha B→.γ come prima componente
                    add [B -> .γ, Δ1] as an unmarked item to P;
                else
                    if [B -> .γ, Γ] ∈ P and Δ1 ⊄ Γ **then** // Se mi sto dimenticando qualche cosa
                        update [B -> .γ, Γ] to [B -> .γ, Γ ∪ Δ1] in P;
                        tag [B -> .γ, Γ ∪ Δ1] as unmarked;
    return P;

L'**if** in azzurro corrisponde alla situazione che abbiamo visto prima nello stato 2.
Avevamo la grammatica `S -> aAd | cAb, A -> e` e dovevamo chiudere `S -> a.Ad, {$}`.
Dobbiamo quindi controllare che per la produzione `A -> e`, dalla forma `B -> γ`,
non vi sia in prj(P) un item la cui prima componente abbia la forma `B -> .γ`, qui `A -> .e`.
Questa condizione era verificata, e quindi avevamo aggiunto l'item `A -> .e, {d}`.

L'**else** in azzurro corrisponde alla seguente situazione.
Ho la grammatica `E -> E+E | E*E | (E) | id`.
Lo stato iniziale ha, come kernel item, `E' -> .E, {$}`.
Siccome nel set non ho un item che abbia la prima componente di chiusura per la `E`, ci metto

```
E -> .E+E, {$}
E -> .E*E, {$}
E -> .(E), {$}
E -> .id, {$}
```

Sembra un insieme saturato? No! Noto che vi sono vari non-terminali preceduti dal '`.`' davanti.

Pertanto:
In `E -> . E + E, {$}` devo chiudere la **E** in grassetto portandomi avanti la nozione di chi è il FIRST di `+E, {$}`.
In `E -> . E * E, {$}` devo chiudere la **E** portandomi avanti la nozione di chi è il FIRST di `* E, {$}`.
Mentre gli item `E -> .(E), {$}` e `E -> .id, {$}` sono a posto.
Questa cosa si cattura con l'**else**:

update `[B -> .γ, Γ]` to `[B -> .γ, Γ ∪ Δ1]` in `P`;

Esempio

Ripetiamo pian piano l'applicazione dell'algoritmo per la grammatica `E -> E+E | E*E | (E) | id`.
Parto dall'item `E' -> .E, {$}`.
Quando calcolo `Δ1`, tecnicamente dovrei fare $\bigcup_{d \in \{\$\}} \text{FIRST}(\$)$, cioè $\Delta_1 = \{\$\}$.
Segue ora il foreach. Vengono aggiunti:

`E -> .E+E, {$}` *nm* (non marcato)
`E -> .E*E, {$}` *nm*
`E -> .(E), {$}` *nm*
`E -> .id, {$}` *nm*

Riparto dal while
Quando considero l'item `E -> .E+E, {$}`, chi è `Δ1`? È il FIRST della stringa `+E$`, cioè `+`. Quindi, `Δ1` = {+}.
Entro nell'else. L'else dice che, per tutti gli item che sono di chiusura per la E, venga aggiunto al lookahead set il `Δ1`, e cioè `+`.
Ricordo che *item di chiusura per la* `E` vuol dire che è tale da avere il '`.`' davanti al body di una qualunque delle produzioni per la `E`.
Pertanto, aggiorno gli item a:

`E -> .E+E, {$,+}`
`E -> .E*E, {$,+}` *nm*
`E -> .(E), {$,+}` *nm*
`E -> .id, {$,+}` *nm*

Riparto dal while
Quando considero l'item `E -> .E*E, {$,+}`, chi è `Δ1`? Devo andare a guardare i FIRST di due stringhe: `*E$` e `*E+`.
I FIRST sono `*`, quindi `Δ1` = {*}.
Entro nell'else, ed aggiorno gli item a:

`E -> .E+E, {$,+,*}`
`E -> .E*E, {$,+,*}`
`E -> .(E), {$,+,*}` *nm*
`E -> .id, {$,+,*}` *nm*

Quando riparto dal while per gli item `E -> .(E), {$,+,*}` e `E -> .id, {$,+,*}`, non devo fare niente.
L'insieme finale è costituito da stati marcati:

`E -> .E, {E}`
`- - - - - - - - - -`
`E -> .E+E, {$,+,*}`
`E -> .E*E, {$,+,*}`
`E -> .(E), {$,+,*}`
`E -> .id, {$,+,*}`

Vediamo ora come si fa, dato un automa caratteristico di questo tipo, a riempire la tabella di parsing.

Tabella di parsing canonical LR(1)
Q: insieme degli stati dell'automa caratteristico.
τ: funzione di transizione dell'automa caratteristico.
La tabella è una matrice di dimensione |Q| • |V ∪ {$}|.
∀ `Y` ∈ V (vocabolario della grammatica):

- Se `Y` è un terminale e τ(P,Y) = R, allora inserire "shift R" nella entry (P,Y).
- Se P contiene il reducing item `[A -> B., Δ]`, allora inserire "reduce A -> B" nella entry (P,Y) tali che `Y` ∈ `Δ`.
- Se P contiene l'item finale, allora inserire "accept" in (P, $).
- Se Y è un non-terminale e τ(P,Y) = R allora inserire "goto R" (o anche semplicemente "R") nella entry (P,Y).