05-parsing-bottom-up-intro-and-SLR

Intro

Queste tipologi di parsing condividono le stesse tecniche fondamentali.

- La P delle nostre grammatiche va sempre estesa a P' aggiungendo la produzione $S' \to S$ con S' un non-terminale *fresh*.
- Lo stesso algoritmo shift/reduce per il parsing.
- La costruzione di un automa caratteristico come controllore dell'algoritmo di parsing.

Automi LR(0)

Iniziamo col dire che LR(0) signica le leggiamo la parola da sinistra a destra L, compiamo derivazioni rightmost R e non abbiamo simboli nel lookahead 0.
Gli automi LR(0) sono formati da stati, i quali sono insiemi di LR(0)-items: $A \to \alpha \cdot \beta$.

LR(0)-items

Consideriamo l'item $S' \to \cdot S$, vuol dire che dobbiamo ancora leggere la parola da parasare e la parola risulterà accettata se deriverà da S.
Possiamo affermare che il nostro item $S' \to \cdot S$ dovrà essere nello stato iniziale chiamato P_0.
Anche altri item potrebbero essere in P_0.

Chiusura di un insieme di LR(0)-items

Definizione

Sia P un insieme di LR(0)-items, allora $closure_0(P)$ è il più piccolo insieme che soddisfa la seguente equazione:

$$closure_0(P) = P \cup \{B \to \cdot\gamma | A \to \alpha \cdot B\beta \in closure_0(P) \land B \to \gamma \in P'\}$$

In partica la chiusura aggiunge ad uno stato, per tutti gli item con il punto davanti ad un non-terminale, tutte le derivazioni di quei non-terminali.

Esempio

Prendiamo la grammatica:

$$\begin{cases} E' \to E \\ E \to E + T | T \\ T \to T * F | F \\ F \to (E) | id \end{cases}$$

Come si può vedere abbiamo aggiunto il nuovo start symbol *fresh*.
Computiamo $closure_0(\{E' \to \cdot E\})$ passo per passo:

1. Iniziamo con $closure_0(\{E' \to \cdot E\}) = \{E' \to \cdot E\}$
2. Abbiamo il marker davato al non-terminale E quindi aggiungiamo le sue derivazioni
    1. Aggiungo $\{E \to \cdot E + T\} \cup \{E \to \cdot T\}$
3. Abbiamo il marker davanti al non-terminale T, possiamo ignorare E avendolo già inserito.
    1. Aggiungo $\{T \to \cdot T * F\} \cup \{T \to \cdot F\}$
4. Abbiamo il marker davanti al non terminale F, come prima ignoro T.
   1. Aggiungo $\{F \to \cdot (E)\} \cup \{F \to \cdot id\}$
   Concludiamo che quindi:
   $$closure_0(\{E' \to \cdot E\}) = \{\{E' \to \cdot E\}, \{E \to \cdot E + T\}, \{E \to \cdot T\}, \{T \to \cdot T * F\}, \{T \to \cdot F\}, \{F \to$$

Algoritmo

```
function closure0(P)
foreach item ∈ P do
        item.unmarked = True;
while ∃item ∈ P : item.unmarked == True do
        item.unmarked = False;
        if item has the form A → α · Bβ then
                foreach B → γ ∈ P′ do
                        if B → ·γ ∉ P then
                                add B → ·γ as an unmarked item to P ;
return P ;
```

Costruzione automa LR(0)

Dobbiamo costruire l'automa popolando un insieme di stati mentre definiamo la funzione di transizione. (non è così difficile come sembra, basta trovare esercizi facili in esame)

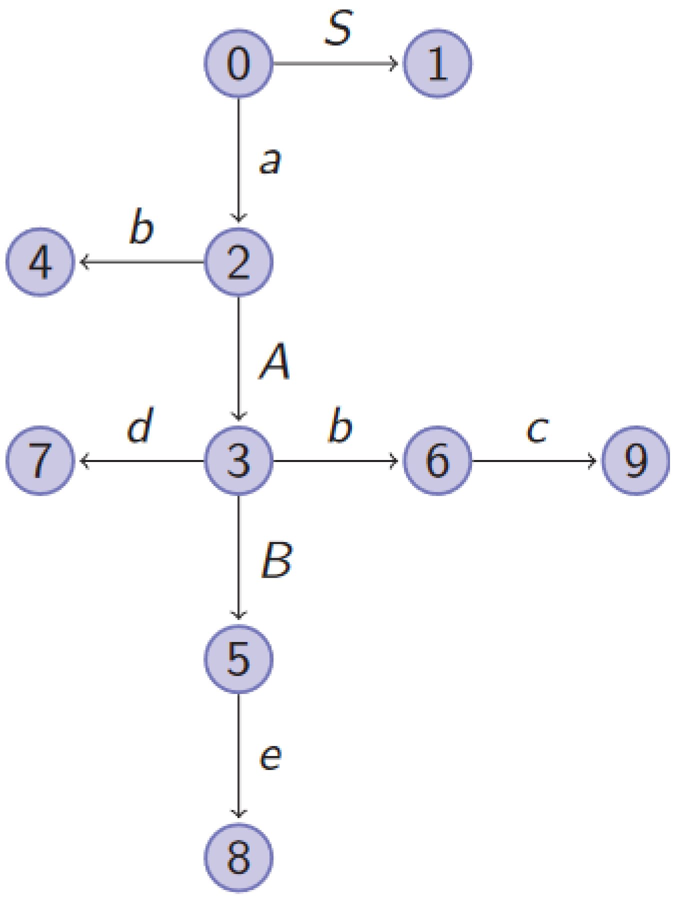
- **Inizio:** per prima cosa mettiamo nel kernel dello stato iniziale P_0 la produzione $S' \to \cdot S$.
- **Ripetizione:** ripetiamo questo procediamento per ogni stato non ancora visitato:
  Costruiamo $closure_0$(kernel), con kernel intendiamo il kernel di quello stato.
  Ora gli item nella collezione avranno la forma $A \to \alpha \cdot Y\beta$, signica che nello stato attuale ho già letto α e posso compiere una Y-transizione.
  * Creiamo ora uno stato P' attraverso la transizione $A \to \alpha Y \cdot \beta$, che ne comporrà il kernel.
  Se Y è un terminale abbiamo compiuto un'operazione di shift.
  E' possibile che il kernel del nostro stato P' sia il kernel di uno stato pre-esistente detto Q, allora la Y-produzione andrà in Q e non dovremmo creare P'.

Esempio

Costruire l'automa caratteristico per il parsing LR(0) per la seguente grammatica:

$$\begin{cases} S \rightarrow aABe \\ A \rightarrow Abc|b \\ B \rightarrow d \end{cases}$$

- Iniziamo con lo stato 0:
  - Kernel: $\{S' \rightarrow \cdot S\}$
  - Corpo: $\{S \rightarrow \cdot aABe\}$
- $\tau(0, S)$ e definisco lo stato 1:
  - Kernel: $\{S' \rightarrow S \cdot\}$
- $\tau(0, a)$ e definisco lo stato 2:
  - Kernel: $\{S \rightarrow a \cdot ABe\}$
  - Corpo: $\{A \rightarrow \cdot Abc$, $A \rightarrow \cdot b\}$
- $\tau(2, A)$ e definisco lo stato 3:
  - Kernel: $\{S \rightarrow aA \cdot Be$, $A \rightarrow A \cdot bc\}$
  - Corpo: $\{B \rightarrow \cdot d\}$
- $\tau(2, b)$ e definisco lo stato 4:
  - Kernel: $\{A \rightarrow b \cdot\}$
- $\tau(3, B)$ e definisco lo stato 5:
  - Kernel: $\{S \rightarrow aAB \cdot e\}$
- $\tau(3, b)$ e definisco lo stato 6:
  - Kernel: $\{A \rightarrow Ab \cdot c\}$
- $\tau(3, d)$ e definisco lo stato 7:
  - Kernel: $\{B \rightarrow d \cdot\}$
- $\tau(5, e)$ e definisco lo stato 8:
  - Kernel: $\{S \rightarrow aAbe \cdot\}$
- $\tau(6, c)$ e definisco lo stato 9:
  * Kernel: $\{A \rightarrow Abc \cdot\}$
  Graficamente ottengo il seguente automa:



Algoritmo

```
initialize the collection Q to contain P0 = closure0({S’ → ·S});
p0.unmarked = True;
while ∃ P ∈ Q : p.unmarked == True do
        P.unmarked = False;
        foreach Y a destra del marker in qualche item di P do
                Tmp = kernel(P, Y-transione); //il kernel da P con una Y-
transizione
                if Q contien già uno stato R il cui kernel è Tmp then
                        Q = Y -target of P;
                else
                        Q.add(closure0(Tmp)); //come stato unmarked
                        closure0(Tmp) = Y -target of P;
```

Automi LR(1)

Piccolo inciso per introdurre un concetto che serve nelle tabelle di parsing.
Questi automi sono più "ricchi" di informazioni di un automa LR(0), gli stati sono composti da insiemi di items LR(1).
LR(1)-item: $[A \rightarrow \alpha \cdot \beta, \Delta]$ dove $\Delta \subseteq T \cup \{\$\}$.
La funzione di lookahead $\mathcal{LA} : P \times P \rightarrow 2^{V \cup \{\$\}}$, tutto un casino la vedremo più in la.

Costruzione tabella di parsing LR(0)/SLR(1)/LR(1)

Tabella di parsing LR(0)/LR(1)

Dobbiamo riempire una matrice M nella quale le entry hanno la forma $M[P, Y]$ con P uno stato e $Y \in V \cup \{\$\}$.
Prendiamo la transizione $\tau(P, Y) = Q$, allora la tabella va riempita attraverso le seguenti regole:

- Se Y è un terminale inserisco la mossa `shift Q`.
- Se P contiene una produzione del tipo $A \to \beta\cdot$
  - Nel caso LR(0)-item $[A \to \beta\cdot]$ allora inserisco `reduce` $A \to \beta$.
  - Nel caso LR(1)-item, $[A \to \beta\cdot, \Delta]$ e inseriremo `reduce` $A \to \beta$ in ogni item contenuto in Δ.
- Se P contiene l'accempting item e $Y = \$$ allora inserisco `accept`.
- Se Y è un terminale o $ e non vale nessuan delle precedenti inserisco `error`.
- Se Y è un non-terminale inserisco `goto Q`.

Conflitti

La tabella può avere *entry multipli-defined*, in questo caso si parla di conflitti:

- **s/r conflict:** nel caso in cui almeno una entry $M[P, Y]$ contenga un'operazione `shift` e una `reduce`.
- **r/r conflict:** nel caso in cui almeno una entry $M[P, Y]$ contenga due operazioni `reduce` distinte.
  Appena verifichiamo la presenza di un conflitto possiamo dire che la nostra grammatica non è LR(0), SLR(1), LR(1) o LALR(1).

Tabella di parsing SLR(1)

E' il livello di informazioni intermedio tra LR(0) e LR(1).
Queste tabelle si ottengono prendendo:

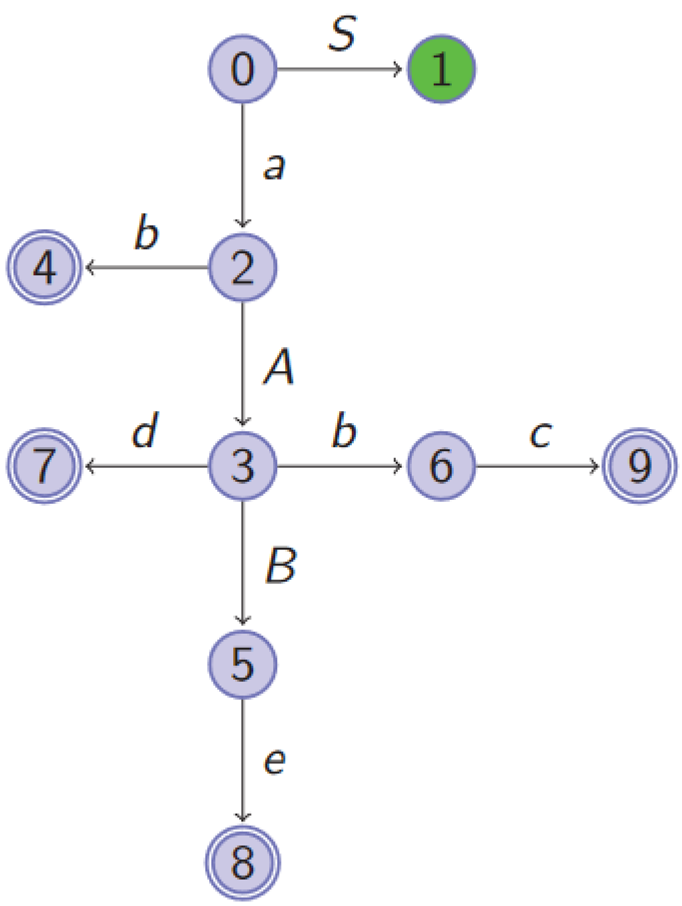
- Un automa caratteristico LR(0).
- e una funzione di lookahead molto semplice $\mathcal{LA}(P, A \to \beta) = follow(A)$ per ogni $A \to \beta\cdot \in P$
  Come prima vale che una grammatica è SLR(1) se e solo se la corrispondente atbella SLR(1) non ha conflitti.

Esempio

Costruiamo la tabella di parsing SLR(1) per la seguente grammatica

$$\begin{cases} S \to aABe \\ A \to Abc|b \\ B \to d \end{cases}$$

Ci serve l'automa LR(0) ma lo abbiamo già da un paio di esempi fa, aggiungiamoci però il colore verde allo stato di `accept` e marchiamo come finali gli stati dove abbiamo una `reduce`.



Ora dobbiamo calcolare i $follow$ e di conseguenza i $first$.

	$first$	$follow$
S	a	$
A	b	b,d
B	d	e

Una volta calcolati procediamo così:

- Prendiamo tutti gli stati P che contengono un reducing item $A \to \beta\cdot$, andiamo ad inserire la mossa di `reduce` nella casella $M[P, Y]$ con $Y = \mathcal{LA}(P, A \to \beta) = follow(A)$.
- Inseriamo in $M[1, \$]$ la mossa di `accept`.
- Inseriamo un `goto Q` in tutte le celle $M[P, Y]$ dove Y è un non-terminale per il quale esiste $\tau(P, Y) = Q$.
- Infine inseriamo `error` in tutte le altre celle.

Costruiamo quindi la tabella, per indicare lo `shift` in uno stato n scriveremo sn, per indicare il `goto` in uno stato n scriveremo Gn, per le `reduce` di una produzione n scriveremo rn ed infine per l'`accept` scriveremo Acc.

---

	a	b	c	d	e	$\$$	S	A	B
0	s2						G1		
1						Acc			
2		s4						G3	
3		s6		s7					G5
4		r3		r3					
5					s8				
6			s9						
7					r4				
8						r1			
9		r2		r2					

Shift/reduce parsing

- **Input:** una stringa w ed una tabella di parsing M per la grammatica $\mathcal{G} = \{V, T, S, P\}$.
- **Output:** la derivazione rightmost di w in ordine inverso se $w \in L(\mathcal{G})$ altrimenti `error`.
- **Inizializzazione:**
  - P_0 nello state-stack stSt.
  - nulla sullo symbol-stack symSt.
  - $w\$$ nel buffer di input.

Algoritmo

```
b = input_buffer[0];
stSt.push(0);
while true do
        S = stSt.top();
        if M[S,b] = “Shift T” then
                symSt.push(b);
                stSt.push(T);
                b = input_buffer.nextChar();
        else if M[S,b] = “Reduce A → β” then
                for i = 1; i <= |β|; ++i; do
                        symSt.pop();
                symSt.push(A);
                for i = 1; i <= |β|; ++i; do
                        stSt.pop();
                tmp = stSt.top();
                stSt.push(T); where T is such that M[temp, A] = “Goto T ”;
```

```
            output “A → β”;
        else if M[S, b] = “Accept” then
            return;
        else
            error();
```

Esempio

Troppo lungo da scrivere fanculo, non è difficile da capire.

Caso studio SLR(1)

Proviamo a costruire la tabella di parsing SLR(1) per la grammatica:

$$E \to E + E | E * E | id$$

Aggiungiamo la produzione $E' \to E$ e iniziamo con la costruzione dei vari stati.

- Stato 0:
  - Kernel: $\{E' \to \cdot E\}$
  - Corpo: $\{E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot id\}$
- $\tau(0, E)$ e definisco lo stato 1:
  - Kernel: $\{E' \to E \cdot,\ E \to E \cdot + E,\ E \to E \cdot * E\}$
- $\tau(0, id)$ e definisco lo stato 2:
  - Kernel: $\{E \to id \cdot\}$
- $\tau(1, +)$ e definisco lo stato 3:
  - Kernel: $\{E \to E + \cdot E\}$
  - Corpo: $\{E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot id\}$
- $\tau(1, *)$ e definisco lo stato 4:
  - Kernel: $\{E \to E * \cdot E\}$
  - Corspo: $\{E \to \cdot E + E,\ E \to \cdot E * E,\ E \to \cdot id\}$
- $\tau(3, E)$ e definisco lo stato 5:
  - Kernel: $\{E \to E + E \cdot,\ E \to E \cdot + E,\ E \to E \cdot * E\}$
- $\tau(4, E)$ e definisco lo stato 6:
  - Kernel: $\{E \to E + E \cdot,\ E \to E \cdot + E,\ E \to E \cdot * E\}$
- $\tau(3, id) = 2$
- $\tau(4, id) = 2$
- $\tau(5, +) = 3$
- $\tau(5, *) = 4$
- $\tau(6, *) = 4$
- $\tau(6, +) = 3$
  Possiamo quindi costruire la tabella di parsing, la rappresentazione dell'automa caratteristico è opzionale.

	+	*	id	$	E
0			s2		G1
1	s3	s4		Acc	
2	r3	r3		r3	
3			s2		G5
4			s2		G6
5	s3, r1	s4, r1		r1	
6	s3, r2	s4, r2		r2	

Abbiamo quindi dei conflitti cha vanno risolti "a mano":

- In $M[5,+]$ tengo r1 così da avere il + associativo a sinistra.
- In $M[5,*]$ tengo s4 così * avrà la precedenza sugli altri operatori.
- In $M[6,+]$ tengo r2 per dare la precedenza a *.
- In $M[6,*]$ tengo r2 per rendere * associativo a sinistra.

  Avremmo potuto modificare la grammatica in:

$$\begin{cases} E \to E + T \mid T \\ T \to T * id \mid id \end{cases}$$

Bisogna fare attenzione perchè l'ordine è importante, infatti se avessi usato, nella prima produzione, la forma $E \to T + E$ non avrei avuto l'associatività a sinistra perchè la serie di somme si espande a destra, invece se avessi inserito $E \to E * T$ avrei datto la priorità alla somma che si sarebbe trovavata più in fondo nell'albero di derivazione.
Esistono casi in cui risolvere i conflitti non è possibile, per cui abbiamo bisogno degli LR(1)-item.