Docente: Paola Quaglia
Docente Laboratorio: Dr. Ercolani
Sito: inviare mail a sympa@list.disi.unitn.it con “Subscribe formal_languages_compilers” nel testo del mesaggio.

Libri:

Autori: Aho, Lam, Sethi, Ullman (o ALSU in breve. Su esse3 si trova così)
Versione inglese “Compilers: Principles, Techniques and Tools”, Prentice Hall, 2 Ed.
Versione italiano “Compilatori: Principi, Tecniche e Strumenti”, Pearson, 2 Ed.
Link:
http://www.informatik.uni-bremen.de/agbkb/lehre/ccfl/Material/ALSUdragonbook.pdf

Autore: Kozen (o K in breve. Su esse3 si trova così)
“Automata and Computability”
Link:
https://merascu.github.io/links/SS2017FLAT/Dexter%20C.%20Kozen%20-%20Automata%20and%20Computability.pdf

Esame: 13 domande aperte: 12 normali e 1 difficile. NOTA: max 3 consegne su 5 appelli consecutivi, partendo a contare dalla prima volta che ci si presenta.
Date presunte esame: 17 gennaio (con due punti bonus!)
Altra data che non ha detto/non ho capito

Simboli presi da: https://www.matematicamente.it/staticfiles/formulario/1-Simboli.pdf

Il corso si divide in Analisi Lessicale, Analisi Sintattica e LEX & YACC
L’analisi lessicale si occupa di separare la frase in parti più piccole (soggetto, verbo etc nel caso dell’italiano). L’analisi sintattica si occupa di vedere se i pezzi sono nell’ordine giusto, o comunque in un ordine sensato. Per farlo, confronta la frase in questione con una grammatica generativa, ovvero la “formula” che genera tutte le possibili frasi che possono essere scritte in un certo linguaggio.

Esempio: position = init + rate * 60
Questa istruzione, una volta compilata, diventa una cosa del tipo:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

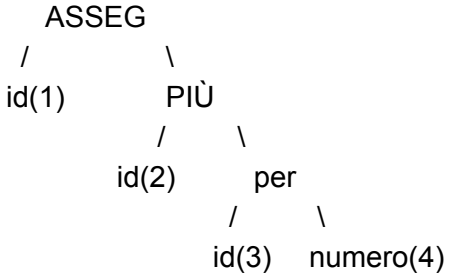
L’output dell’analisi **lessicale** di una cosa di questo tipo diventa:

<id, →1> ASSEG <id, →2> PIÙ <id, →3> PER <numero, →4> (le frecce indicano puntatori alle righe della tabella dei simboli, qui sotto)

ptr	id	type

1	position	float
2	init	float
3	rate	float
4	60	float

A questo punto si può fare l'analisi **sintattica**. Controlla per esempio se prima dell'assegnazione c'è effettivamente un identificatore, se il per ha un operando sia a sinistra che a destra, etc.
Per farlo, si deve creare un abstract syntax tree, ovvero si cerca di vedere se partendo dalla grammatica si può arrivare alla frase che abbiamo noi. Se questo non può succedere, allora c'è un syntax error. Un abstract syntax tree è fatto così:



Da qui si fa l'analisi **semantica**, per vedere per esempio se bisogna fare dei cast ad alcuni operatori per convertirli da naturali a float e cose del genere. Questo viene fatto magari per matchare un pattern di overload di una certa funzione. Questa operazione aggiungerebbe una piccola parte al syntax tree, sostituendo "numero" con "INTtoREAL(numero)", a cui si collegherebbe subito sotto "60".
Dopo questa procedura viene generato del codice intermedio

```
temp1 = INTtoREAL(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Quando si scrive un compilatore si trasforma quindi dal linguaggio originale al codice intermedio. Un altro team si occuperà di convertire il codice intermedio ad un altro linguaggio per la macchina 1, poi in un altro linguaggio per la macchina 2, e così via. Questo evita di dover rifare il linguaggio intermedio per ogni macchina, che sarebbe inutile. Noi studieremo la prima parte, ovvero dal codice originale al linguaggio intermedio.

14/09/17

Grammatiche e Linguaggi

Un alfabeto è un insieme finito e definito di simboli. Con l'alfabeto possiamo scrivere una stringa (o parola), che è una sequenza finita o nulla di simboli dell'alfabeto, ottenuta per giustapposizione di simboli. Una parola vuota è denotata dalla lettera ε. La lunghezza della parola è data dal numero di simboli dell'alfabeto che la compongono ed è invece zero se la parola è vuota.

Grammatica generativa G = (V, T, S, P).
V : vocabolario, insieme di simboli, finito e non vuoto.
T : insieme dei simboli terminali, dove T è strettamente contenuto in V (T c V)
S : simbolo iniziale, e S appartiene al vocabolario ma non fa parte dei simboli terminali ($S \in V \setminus T$)
P : insieme di produzioni, che in generale hanno la forma $\alpha \rightarrow \beta$, dove α è una stringa non vuota su V che contiene almeno un elemento non terminale e β è una stringa su V (oppure è ε).

Es: G = ({S, a, b}, {a, b}, S, {S → a S b, S → ε})
V T S P
Nota: Simboli in V \ T denotati da lettere maiuscole. Simboli in T denotati da lettere non maiuscole. X, Y si usano per denotare un generico simbolo in V (terminale o non). α, β, γ etc si usano per denotare delle parole su V*.
Immaginiamo di avere {S → a S b, S → A, S → ε}. Lo start symbol potrebbe essere sia S che A, ma per convenzione è S perché è quello che compare da solo prima della freccia.

Una grammatica generativa è libera da contesto (o libera, o context free) se ogni sua produzione ha la forma $A \rightarrow \beta$.

Al posto di scrivere S → a S b e S → ε, si può scrivere S → a S b | ε, per dire che sono alternative. E' come combinarle in una riga sola.
$S \Rightarrow \varepsilon$ vuol dire che ε è generata dalla grammatica G
$S \Rightarrow a\ S\ b \Rightarrow ab$ vuol dire che "ab" è generata dalla grammatica G
$S \Rightarrow a\ S\ b \Rightarrow a\ aSb\ b \Rightarrow aabb$ è valido
Questa grammatica genera quindi $\{a^n b^n \mid n >= 0\}$. Questo può essere usato per esempio per vedere se è stato messo un numero uguale di parentesi aperte (a) e parentesi chiuse (b), poiché da questa grammatica è impossibile generare una sequenza del tipo aab o abb, ma solamente aabb, ab, aaabbb etc.

Derivazione in un passo: $\mu \Rightarrow \gamma$(γ deriva in un passo da μ, data la grammatica G) se $\mu = \mu_1 \alpha \mu_2$ e $\alpha \rightarrow \beta$ è una produzione di G e $\gamma = \mu_1 \beta \mu_2$.
Scriviamo che $\mu \Rightarrow^+ \gamma$ (γ deriva da μ in uno o più passi, data la grammatica G) se $\mu \Rightarrow_G \delta_0 \Rightarrow_G \delta_1 \Rightarrow_G \ldots \Rightarrow_G \varepsilon$

$L(G) = \{w \mid w \in T^* \text{ e } S \Rightarrow_G^+ w\}$

Es:
G_1: S → a A b
aA → aa A b

$A \rightarrow \varepsilon$

Non è libera da contesto, e genera il linguaggio $L = \{a^n b^n \mid n > 0\}$

Es:

G_2: $S \rightarrow a\ S\ b \mid ab$

Genera lo stesso linguaggio dell'esempio di prima, ovvero $L = \{a^n b^n \mid n > 0\}$, ma hanno un alfabeto diverso (qui manca A)

Dato il linguaggio L possono esistere più grammatiche diverse tra loro che generano lo stesso linguaggio L.

E' indecidibile il linguaggio generato da una grammatica G, per G arbitraria.

Es:

G_3: $S \rightarrow AB$

$A \rightarrow aA$

$A \rightarrow a$

$B \rightarrow Bb$

$B \rightarrow b$

Genera $L(G_3) = \{a^n b^j \mid n, j > 0\}$, ovvero almeno una a e una b, ma non devono essere in numero uguale (quindi aab o abbb sono casi validi). Volendo questo stesso linguaggio si può ottenere con due grammatiche separate

G_4: $A \rightarrow aA \mid a$

G_5: $B \rightarrow Bb \mid b$

e infatti $L(G_3) = \{w_4 w_5 \mid w_4$ appartiene $L(G_4)$ e w_5 appartiene $L(G_5)\}$

Es:

G_6: $S \rightarrow aSBc \mid abc$

$cB \rightarrow Bc$

$bB \rightarrow bb$

$S \Rightarrow a\underline{S}Bc$

$\Rightarrow a\ a\underline{S}Bc\ Bc$

$\Rightarrow aa\ abc\ B\underline{c\ B}c$

$\Rightarrow aaa\underline{b\ B}c\ cBc$

$\Rightarrow aaa\ bb\ c\underline{cB}c$

$\Rightarrow aaa\ bb\underline{cB}cc$

$\Rightarrow aaa\ b\underline{bB}ccc$

$\Rightarrow aaa\ bbb\ ccc$

$L(G_6) = \{a^n b^n c^n \mid a, b\ c > 0\}$

Es:

G_7: $S \rightarrow aB$

($\{S, B, a\}$, $\{A\}$, S, $\{S \rightarrow aB\}$) è una grammatica. Il problema, in questo caso, è che il linguaggio generato è il linguaggio vuoto.

Es:

G_8: $S \rightarrow \varepsilon$

E' una grammatica: ({S}, {}, S, {S → ε}). Il linguaggio generato è l'insieme che contiene ε, che è diverso dall'insieme vuoto ∅.
$L(G_8) = \{\varepsilon\}$

15/09/17

Es:
G_1: S → 0B | 1A
A → 0 | 0S | 1AA
B → 1 | 1S | 0BB
$L(G_1) = \{w \mid count(0, w) = count(1, w)\}$

Es:
Definire G_2 tale che $L(G_2) = \{a^k b^n \mid k, n > 0\}$

S → aS | aB	S → AB	S → ab | aS | Sb
B → bB | b	A → aA | a	
	B → bB | b	

Tutte quelle scritte sopra sono valide (ogni colonna è una grammatica). Ce ne sono anche altre.

Es:
Definire G_3 tale che $L(G_3) = \{a^k b^n c^{2n} \mid k, n > 0\}$
S → AB
A → aA | a
B → bBcc | bcc

Es:
Definire G_4 tale che $L(G_4) = \{a^k b^n d^{2k} \mid k, n > 0\}$
S → aSdd | aBdd
B → b | bB

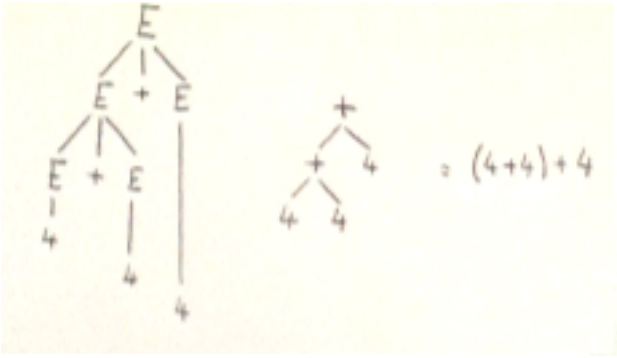
Alberi di Derivazione

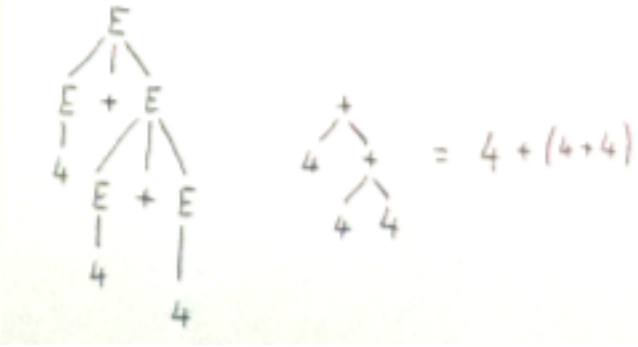
Le derivazioni che abbiamo fatto sono sempre state scritte su una riga con una freccia. Le grammatiche libere si prestano a descrivere le proprie derivazioni tramite un albero.

Questo è un esempio di albero finito di S → aSb | ab

Nel caso di grammatiche libere si definiscono le derivazioni canoniche destre e sinistre. Nel caso rightmost, si richiede che ad ogni passo di derivazione $\mu \Rightarrow \gamma$ venga rimpiazzato il terminale più a destra in μ. Nel caso leftmost, è ovviamente quello più a sinistra.
G è ambigua se $\exists$ w $\in$ L(G) tale che esistono per w due derivazioni canoniche distinte, entrambe destre oppure entrambe sinistre.

Es:
$E \rightarrow E + E \mid E * E \mid 4$





Sono entrambe derivazioni leftmost, quindi G è ambigua.

Finché si usa solamente il segno + non è un problema, ma se si inizia ad usare il * c'è il rischio che il risultato delle due derivazioni sia diverso (4 + (4 * 4) al posto di (4 + 4) * 4, ovvero 20 al posto di 32).

Es:
$S \rightarrow$ if b then S | if b then S else S | altro
Questa grammatica è ambigua perché si può avere "if b then if b then altro else altro", ma non sappiamo se l'else si riferisce al primo o al secondo if.

if(b) if(b)

```
        if(b)                                   if(b)
                altro                                   altro
else                                    else
        altro                                   altro
```

19/09/17

Linguaggi Liberi

Un linguaggio è libero se esiste una grammatica libera che lo genera.

Lemma 1: La classe dei linguaggi liberi è chiusa rispetto all'unione (se L_1 e L_2 sono liberi, allora $\{w \in L_1 \cup L_2\}$ è esso stesso un linguaggio libero)

L_1 libero $\Rightarrow \exists\ G_1 = (V_1, T_1, S_1, P_1)$ t.c. $L_1 = L(G_1)$

L_2 libero $\Rightarrow \exists\ G_2 = (V_2, T_2, S_2, P_2)$ t.c. $L_2 = L(G_2)$

$(V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, S, P_1 \cup P_2 \cup S \rightarrow S_1 \mid S_2\})$, con S nuovo rispetto a V_1 e V_2

Bisogna stare attenti a rinominare i non-terminali di G1 e di G2 in modo da non avere omonimie (clash).

Es: G_1: $S_1 \rightarrow aA_1$ $\quad$ G_2: $S_2 \rightarrow bA$

$A_1 \rightarrow a$ $\quad$ $A \rightarrow b$

Bisogna stare attenti alle omonimie di A e A, quindi la prima A si rinomina in A_1.

Lemma 2: La classe di linguaggi liberi è chiusa rispetto alla concatenazione (se L_1 e L_2 sono liberi, allora $\{w_1w_1 \mid w_1 \in L_1$ e $w_1 \in L_2\}$ è esso stesso un linguaggio libero)

$P_2 \cup P_2 \cup \{S \rightarrow S_1S_2\}$

Anche in questo caso bisogna stare attenti ad eliminare eventuali clash dei simboli non terminali di G2.

G_2' = (V_2', T_2, S_2', P_2') dove V_2', S_2', P_2' denotano possibili ridenominazioni dei non-terminali di G_2.

Sia S un simbolo nuovo tale che S non appartiene a $V_1 \cup V_2$'. Allora:

$G = (V_1 \cup V_2' \cup \{S\}, T_1 \cup T_2, S, P_1 \cup P_2' \cup \{S \rightarrow S_1S_2\})$

Es:

G: $S \rightarrow aSc \mid aTc \mid T$

$T \rightarrow bTa \mid ba$

G è ambigua?

Bisogna capire cosa genera. In questo caso: $\{a^nb^ma^mc^n \mid n >= 0, m > 0\}$

Si fa l'albero di derivazione e si vede che è ambigua:

Es:

G: $S \rightarrow CD$
$C \rightarrow aCA \mid bCB$
$AD \rightarrow aD$
$BD \rightarrow bD$
$Aa \rightarrow aA$
$Ab \rightarrow bA$
$Ba \rightarrow aB$
$Bb \rightarrow bB$
$C \rightarrow \varepsilon$
$D \rightarrow \varepsilon$

Se faccio $S \Rightarrow CD \Rightarrow C \Rightarrow aCA \Rightarrow aA$ non posso andare avanti, quindi da qui non posso derivare nessuna parola.
$S \Rightarrow \underline{C}D \Rightarrow aC\underline{AD} \Rightarrow aCaD \Rightarrow \Rightarrow aa$
$S \Rightarrow \underline{C}D \Rightarrow bC\underline{BD} \Rightarrow bCbD \Rightarrow \Rightarrow bb$
$S \Rightarrow \underline{C}D \Rightarrow a\underline{C}AD \Rightarrow abCB\underline{AD} \Rightarrow abC\underline{Ba}D \Rightarrow abCa\underline{BD} \Rightarrow abCabD \Rightarrow \Rightarrow abab$

$L = \{ww \mid w \text{ è una stringa sull'alfabeto } \{a, b\}^* \} \cup \{\varepsilon\}$

La grammatica di prima non era libera. Non può esistere una grammatica libera che genera questo linguaggio, perché quello non è un linguaggio libero. Per capirlo, si fa il procedimento descritto dopo.

20/09/17

Pumping Lemma per i linguaggi liberi

E' usato per determinare se un certo linguaggio dato NON è libero.

Sia L un linguaggio libero. Allora $\exists\ p \in N^+$ tale che $\forall\ z \in L : |z| > p.\ \exists\ u\ v\ w\ x\ y :$
$z = uvwxy$ e
$|vwx| <= p$ e
$|vx| > 0$ e
$\forall\ i \in N: uv^iwx^iy \in L$

In poche parole, per ogni parola esiste un "mapping" di tale parola (parola mappata su u v w x y) che rispetta le condizioni indicate sopra.

Le seguenti affermazioni servono per fare la dimostrazione del pumping lemma:
OSS 1:
Data una grammatica G, se ne può sempre creare un'altra G', modificata dalla prima, che genera lo stesso linguaggio.
Es:

S → aSb | ab

S → A
A → aAb | ab

OSS 2:

Una grammatica si può portare in forma normale (di Chomsky), togliendo eventuali ridondanze o produzioni che terminano per forza con ε (a meno che non debba fare parte del linguaggio, ma a quel punto si scrive come S → ε). Non ci serve saperle nel dettaglio, ma basta sapere che esistono.

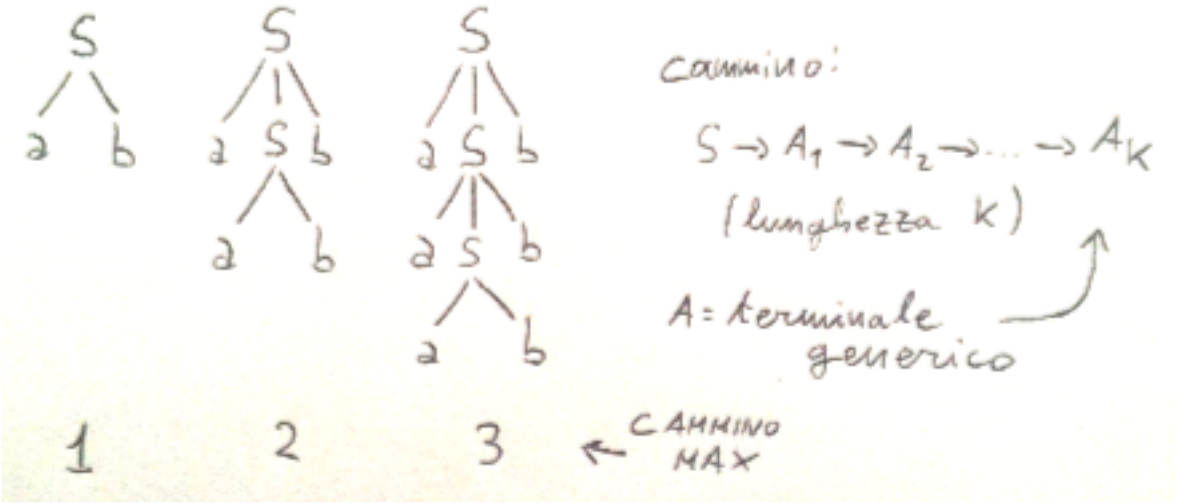
Dimostrazione del pumping lemma:

L è un linguaggio libero

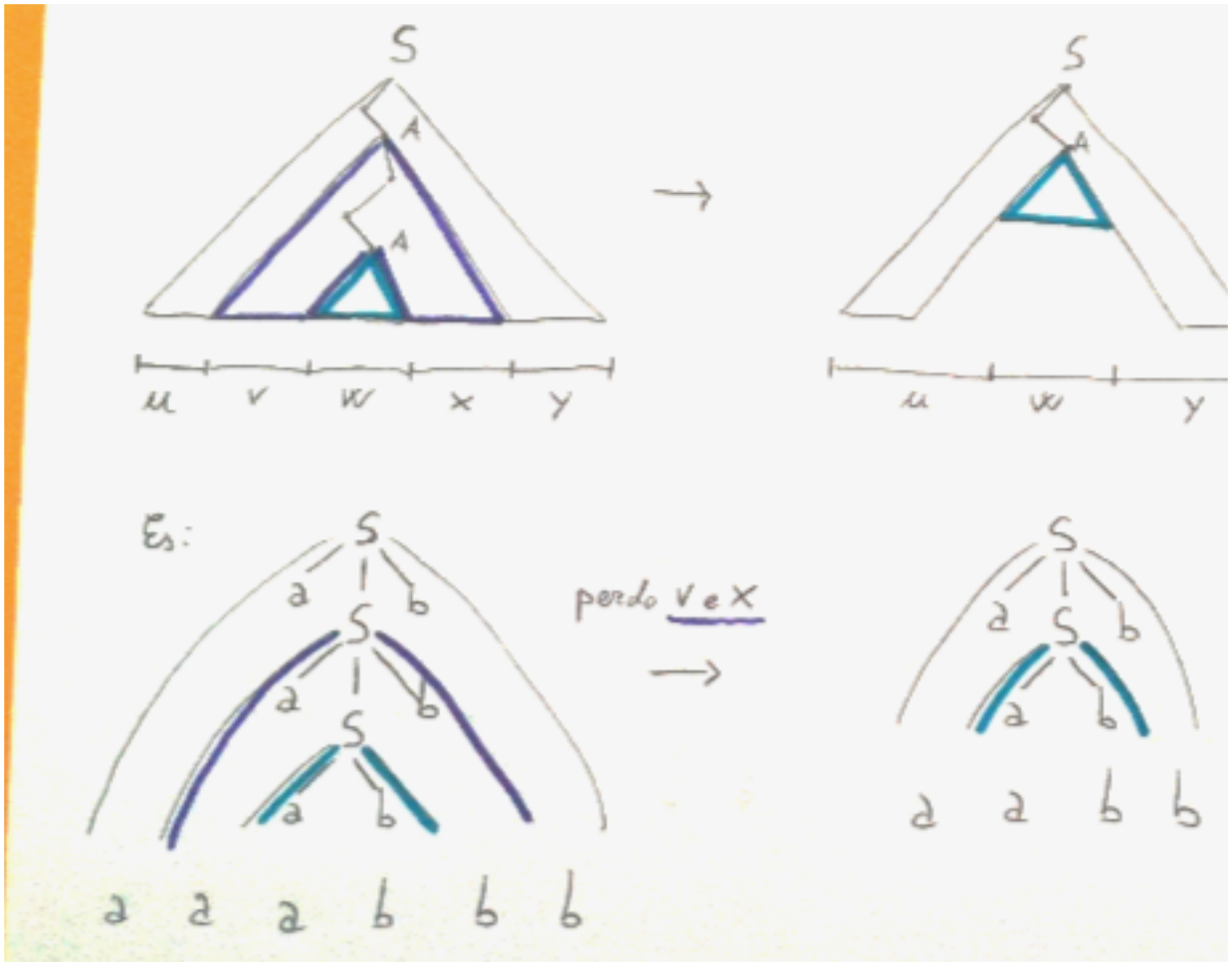
⇒ esiste una grammatica G in forma normale di Chomsky tale che L = L(G).

Definiamo p come la lunghezza della parola più lunga che può essere derivata usando un albero di derivazione i cui cammini dalla radice sono lunghi al più come il numero di simboli non terminali della grammatica (|V \ T|).

S → aSb | ab ha due non terminali e p = 2



se z ∈ L e |z| > p ⇒ nell'albero di derivazione di z esiste almeno un cammino la cui lunghezza è maggiore di |V \ T| ⇒ allora esiste almeno un non-terminale che occorre almeno due volte lungo quel cammino (A, nell'esempio sotto)



21/09/17

Come già detto, il pumping lemma è usato per dimostrare che un certo linguaggio dato non è libero, cioè che non esiste e non può esistere alcuna grammatica libera che lo genera.
Per farlo, si usa la tecnica per contraddizione (o per assurdo).

Dim: supponiamo che L_1 sia libero. Mostriamo che possiamo contraddire la tesi del lemma, ovvero che possiamo dimostrare la negazione della tesi.
NOT(tesi) = $\forall p \in N^+$ tale che $\exists\ z \in L : |z| > p.\ \forall\ u\ v\ w\ x\ y\ [$
$(z = uvwxy \text{ e } |vwx| <= p \text{ e } |vx| > 0) \rightarrow \exists\ i \in N$ tale che $uv^iwx^iy \notin L\ ]$

Es:
$L_1 = \{ ww \mid w \in \{a, b\}^* \}$ non è libero. Dimostrare.
Dim: Supponiamo che L_1 sia libero.
Sia p un numero naturale e positivo qualunque.
Sia $z = a^pb^pa^pb^p$ z = [a…….ab…….ba…….ab…….b], |z| = 4p
Allora $z \in L_1$, |z| > p
Siano u v w x y tali che z = uvwxy e |vwx| <= p e |vx| > 0
Distinguiamo varie possibilità:

1. vwx contiene solo ‘a’ di w_1
2. vwx contiene sia ‘a’ che ‘b’ in w_1
3. vwx contiene solo ‘b’ di w_1
4. vwx contiene ‘b’ di w_1 e ‘a’ di w_2
5. vwx contiene solo ‘a’ di w_2
6. vwx contiene sia ‘a’ che ‘b’ di w_2
7. vwx contiene solo ‘b’ di w_2

Nei casi 1, 3, 5, 7 considero la parola $z' = uv^0wx^0y$

1. $z' = a^kb^pa^pb^p$ con $k < p$ quindi $z' \notin L$
3. $z' = a^pb^ka^pb^p$ con $k < p$ quindi $z' \notin L$
5. $z' = a^pb^pa^kb^p$ con $k < p$ quindi $z' \notin L$
7. $z' = a^pb^pa^pb^k$ con $k < p$ quindi $z' \notin L$

Nei casi 2, 4, 6 considero ancora la parola $z' = uv^0wx^0y$

2. z’ ha una delle tre possibili forme:
   - $z' = a^kb^pa^pb^p$ con $k < p$
   - $z' = a^pb^ka^pb^p$ con $k < p$
   - $z' = a^jb^ka^pb^p$ con k < p, j e k possono essere sia uguali che diversi

4 e 6 sono simili, hanno la stessa struttura.

La cosa importante per questa dimostrazione è scegliere bene la z, altrimenti si potrebbe arrivare a non ottenere nulla di utile, come nel prossimo esempio.

Es (di cose da non fare):
$L_1 = \{ ww \mid w \in \{a, b\}^* \}$
Dim: Sia $z = a^pa^p$
Sia $z = (ab)^p(ab)^p$

Bisogna inoltre stare attenti a non scegliere una certa ‘p’ e fissarla, poiché quello che si vuole dimostrare deve essere vero per ogni p e non solo per una in particolare.

Esercizi:
$\{ a^nb^nc^j \;\; n, j >= 0 \}$. E’ libero?
$S \rightarrow A \mid C \mid AC \mid \varepsilon$
$A \rightarrow aAb \mid ab$
$C \rightarrow Cc \mid c$

22/09/17

Nota: La concatenazione di linguaggi liberi è un linguaggio libero

Esempio di dimostrazione:
$L_1 = \{ a^nb^nc^j \mid n,j >= 0 \}$ Libero
$L_2 = \{ a^jb^nc^n \mid j,n >= 0 \}$ Libero (concatenazione di $\{a^nb^n \mid n >= 0\}$ e $\{c^j \mid j >= 0\}$, entrambi liberi)
$L_3 = \{ a^nb^nc^n \mid n >= 0 \}$ non è libero
Dim: Supponiamo L_3 libero
Sia p appartiene N^+, sia < = $a^pb^pc^p$

Allora z appartiene L_3, |z| > p
z = a...ab...bc...c, |z| = 3p, A = a...a, B = b...b, C = c...c
Siano uvwxy tali che z = uvwxy e |vwx| <= p e |vx| > 0
Ci sono vari casi:

1. vwx è composto solo da a in A
2. vwx è composto da a in A e b in B
3. vwx è composto solo da b in B
4. vwx è composto da b in B e c in C
5. vwx è composto solo da c in C

Considero la parola z' = uv^0wx^0y

1. $z' = a^kb^pc^p$, k < p, $z' \notin L_3$
3. $z' = a^pb^kc^p$, k < p, $z' \notin L_3$
5. $z' = a^pb^pc^k$, k < p, $z' \notin L_3$

Considero ancora la parola z' = uv^0wx^0y

2. $z' = a^kb^jc^p$, k < p or j < p, $z' \notin L_3$
4. $z' = a^pb^kc^j$, k < p or j < p, $z' \notin L_3$

Quindi visto che la parola non appartiene mai ad L_3, il linguaggio non è libero.

Lemma: la classe dei linguaggi liberi NON è chiusa rispetto all'intersezione
Dim: L_1 è libero e L_2 è libero e $L_3 = L_1 \cap L_2$
Es: i 3 linguaggi degli esempi di prima

$L_4 = \{ a^nb^mc^{n+m} \mid n, m > 0 \}$ LIBERO
$S \rightarrow aSc \mid aBc$
$B \rightarrow bBc \mid bc$

$L_5 = \{ a^nb^mc^nd^m \mid n,m > 0 \}$ NON LIBERO
$L_6 = \{ wcw^R \mid w \in \{a, b\}^+ \}$ LIBERO
$S \rightarrow aSa \mid bSb \mid aca \mid bcb$

26/09/17

Automi a stati finiti

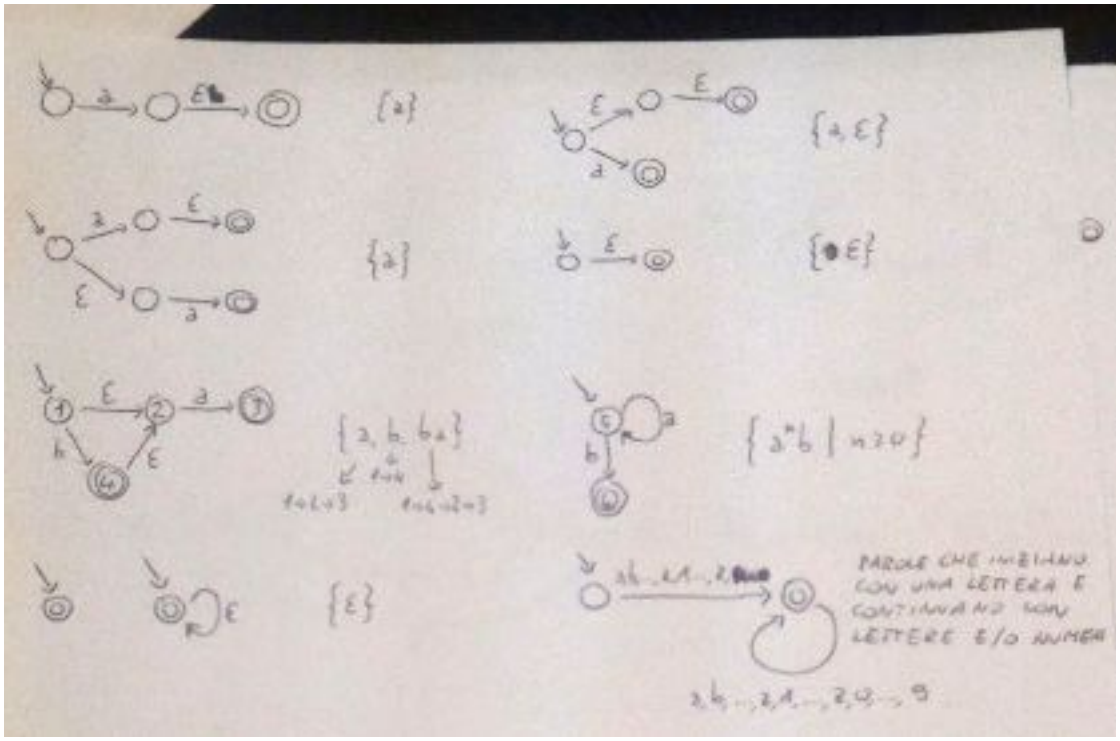
27/09/17

NFA riconosce/accetta un certo linguaggio.

Sia N uno NFA
Allora il linguaggio riconosciuto/accettato da N è l'insieme delle parole per le quali esiste almeno un cammino dallo stato iniziale di N ad uno stato finale di N.

OSS:

$\varepsilon\varepsilon = \varepsilon$



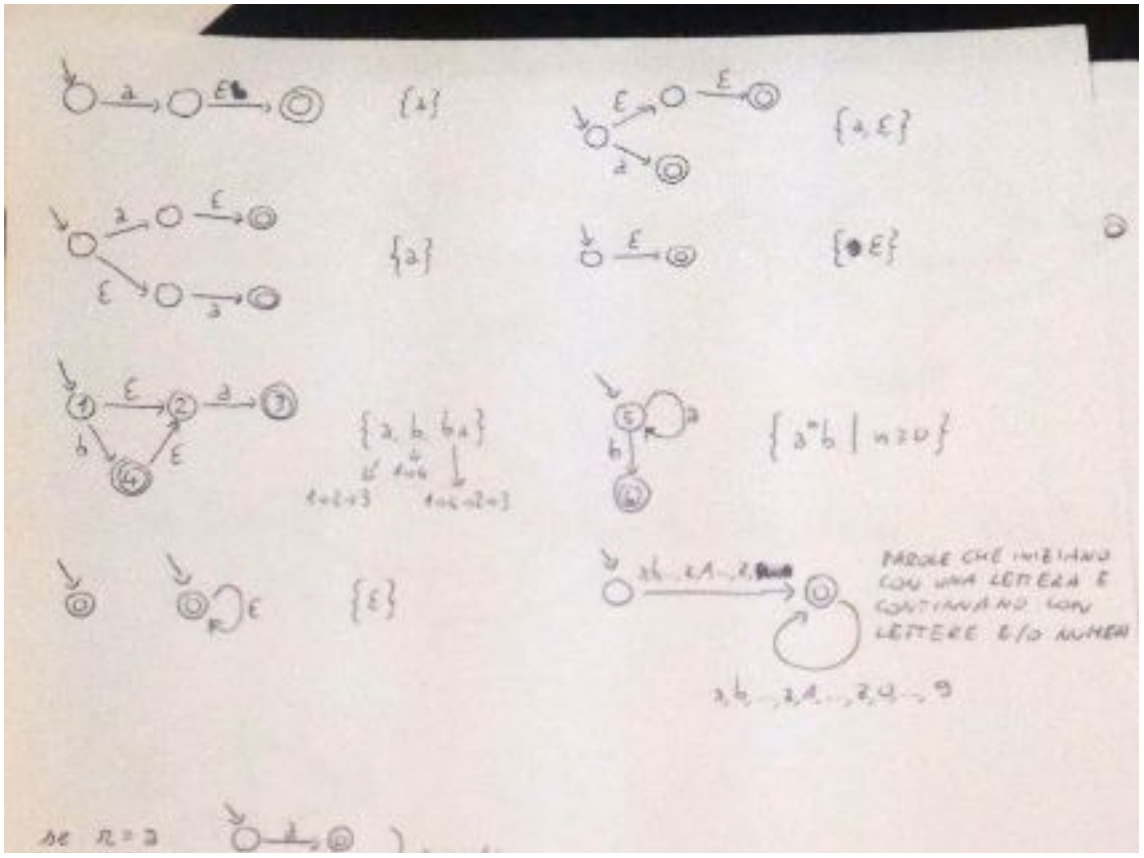
$\forall\ a \in A\ ,\ a\varepsilon = \varepsilon a = a$

Thompson Construction

input: regexp r
output: NFA N tale che L(N) = L(r)

Caratteristiche degli NFA utilizzati nei passi della costruzione:

- Ogni NFA ha esattamente uno stato finale
- Ogni NFA non ha archi entranti nello stato iniziale
- Ogni NFA non ha archi uscenti dallo stato finale

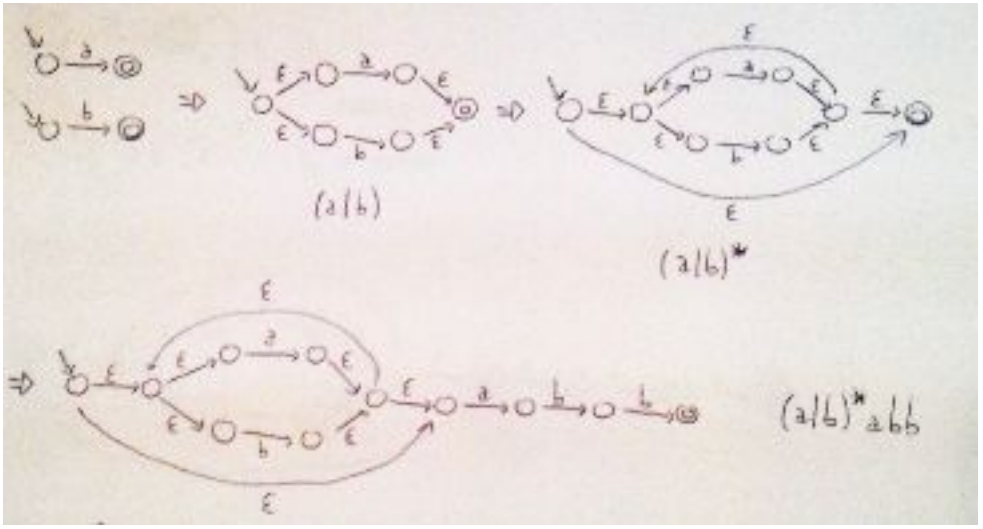


Lemma:

Lo NFA ottenuto dalle costruzioni di Thompson ha al massimo 2k stati e 4k archi per k lunghezza dell'espressione regolare di input.

OSS: Ogni passo della costruzione, sia base che induttivo, introduce al massimo 2 stati e 4 archi.

Es:



28/09/17

Simulare un NFA

Il backtracking consiste nel provare un percorso e, se non va bene, tornare indietro e provarne un altro, finché non li avremo provati tutti.

$N = (S, A, move_n, S_o, F)$
$t \in S, T \subseteq S$
ε-closure({t}) è l'insieme degli stati in S che sono raggiungibili tramite zero o più ε-transizioni.
Nota: $\forall\ t \in S, t \in$ ε-closure({t}).
ε-closure(T) = $\cup (t \in T)$ di ε-closure(t)

Questo ci permette di fare un algoritmo molto più efficiente del backtracking

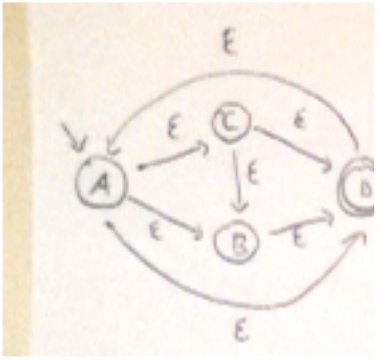
Algoritmo per la computazione

Strutture dati:

- pila
- array booleano di dimensione |S| chiamato "alreadyOn"
- array bidimensionale per la rappresentazione di $move_n$

Inizializzazione:

```
for(int i = 0; i < |S|, i++|)
      alreadyOn[i] = false;
closure(t, stack) {
      push t onto stack;
      alreadyOn[t] = true;
      foreach(i ∈ move_n(t, ε) ) {
            if(not alreadyOn[u])
                  closure(u, stack)
      }
}
```



Es:

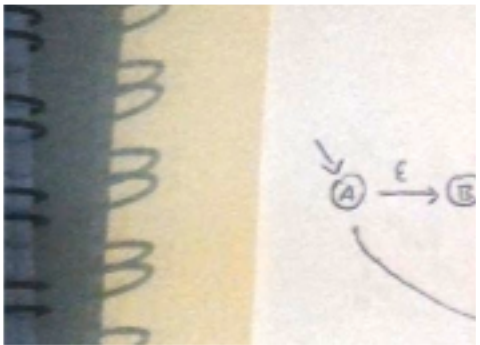
```
alreadyOn[F F F F]
closure(A, pila vuota)
      [A]   [T F F F]
            B non è ancora nella pila
            closure(B, [A])
                  [A, B]     [T T F F]
                  closure(D, [A, B])
                        [A, B, D]  [T T F T]
            closure(C, [A, B, D])
                  [A, B, C, D]     [T T T T]
```

Algoritmo per la simulazione di NFA

input: NFA N, w$
output: yes se w ∈ L(N), no se w ∉ L(N)

```
      N = (S, A, move_n, S_o, F)
1     states = ε-closure({S_0})
2     symbol = nextchar()
3     while(symbol != $) {
4           states = ε-closure(∪_{t ∈ states} di move_n(t, symbol))
5     symbol = nextchar()
      }
6     if(states ∩ F != ∅)
            return yes
      else
            return no
```

Es:



w = aaaaabb$

ho questi stati *poi setto il simbolo nuovo*
states = {A, B, C, D, H} symbol = a_1
//nella riga 4, $move_n$ contiene {E, I}
states = {I, E, G, H, B, C, D} symbol = a_2
//$move_n$ contiene {I, E} = {E, I}
states = {I, E, G, H, B, C, D} symbol = a_3
//$move_n$ contiene {I, E} = {E, I}
states = {I, E, G, H, B, C, D} symbol = a_4
//$move_n$ contiene {I, E} = {E, I}
states = {I, E, G, H, B, C, D} symbol = a_5
//$move_n$ contiene {I, E} = {E, I}
states = {I, E, G, H, B, C, D} symbol = b_1
//$move_n$ contiene {L, F}
states = {L, F, G, H, B, C, D} symbol = b_2
//$move_n$ contiene {M, F}
states = {M, F, G, H, B, C, D} symbol = $

yes, perché contiene M che è il simbolo contenuto in F (ovvero quelli con il doppio cerchio nel grafo)

Questo algoritmo ha complessità O(|w|(n+m)), mentre il calcolo dell'NFA partendo dalla regexp ha costo O(|r|).

29/09/17

DFA - Stati finiti deterministici

Sottoclasse degli NFA che rispettano le seguenti limitazioni:

Funzione di transizione totale:

- Non hanno ε-transizioni
- La funzione di transizione è tale che ∀ a ∈ A e ∀ s ∈ S, il target della transizione per (s, a) è <u>un unico</u> stato

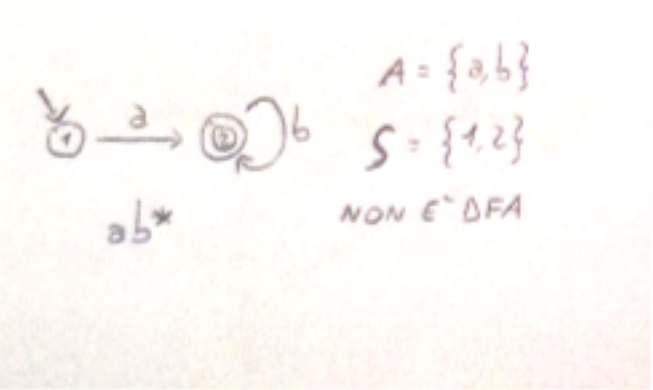
Funzione di transizione parziale:

- Non hanno ε-transizioni
- La funzione di transizione è tale che ∀ a ∈ A e ∀ s ∈ S, il target della transizione per (s, a) è <u>al più uno</u> stato

Es:

Questo è un DFA con funzione di transizione parziale perché dal nodo 1 manca una transizione b, e dal nodo 2 manca una transizione a.

Es:



Questo è DFA con funzione di transizione totale perché da ogni nodo c'è ogni transizione. Se da 1 uso la b transizione per andare in "sink", da li non riesco più ad andare ad un nodo terminale e di conseguenza, non genero una parola che inizia con b.

DFA: (S, A, $move_d$, s_0, F)
$move_d$: (S x A) → S

Dato il DFA D il linguaggio riconosciuto da D è denotato L(D) ed è definito come l'insieme delle parole $w=a_1...a_k$ tali che esiste un cammino in D dallo stato iniziale allo stato finale. Inoltre, ε ∈ L(D) se lo stato iniziale è anche finale.

Simulazione di un DFA con $move_d$ totale

input: w$, DFA D = (S, A, $move_d$, s_0, F)
output: yes se w ∈ L(D), no altrimenti

```
state = s0
while(symbol != $ && state != ⊥) {
      state = moved (state, symbol)
      symbol = nextchar()
}
if(state ∈ F)
      return yes
else
      return false
```

scriviamo $move_d(s, a) = \perp$ (bottom) se la funzione $move_d$ non è definita nel punto (s, a)

Es:

Costi:

simulazione NFA: O(|w|(n+m))
simulazione DFA: O(|w|)

Subset Construction

Conversione di NFA in DFA
input: NFA(S^n, A, $move_n$, S_0^n, F^n)
output: DFA(S^d, A, $move_d$, S_0^d, F^d)

```
S0^d = ε-closure({S0^n})
states = {s0^d}
tag S0^d come non marcato
while(∃ T ∈ states non marcato)
      marco T
      foreach (a ∈ A)
            T' = ε-closure(∪t ∈ T di moven(t, a))
            if(T' != ∅)
                  moved(T, a) = T'
                  if(T' ∉ states)
                        aggiungi T' a states come non marcato
foreach (T ∈ states)
      if( (T ∩ F^n) != 0)
            metti T' in F^d
```

03/10/17

Es:
Nota: L'esercizio fornisce il disegno a sinistra, quello a destra viene fatto dopo aver fatto la tabella, basandosi su di essa. Qui sono messi entrambi prima della tabella per motivi di spazio.

*in grassetto quelli già ε-chiusi

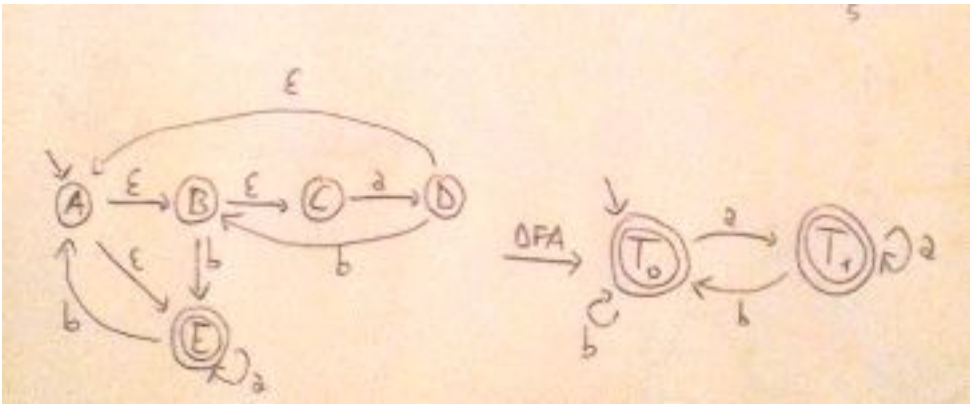
states	a	b
S_0^d = {**1** 2 4 }	T1	T2
T1 = { **3** }	T1	-
T2 = { **5**} oss	-	T2

Alla fine mi devo segnare che T0, T1 e T2 sono finali (poiché contengono almeno un nodo finale).

Es:

	epsilon	a	b
A	{B, E}	∅	∅
B	{C}	∅	{E}
C	∅	{D}	∅
D	{A}	∅	{B}
E	∅	{E}	{A}

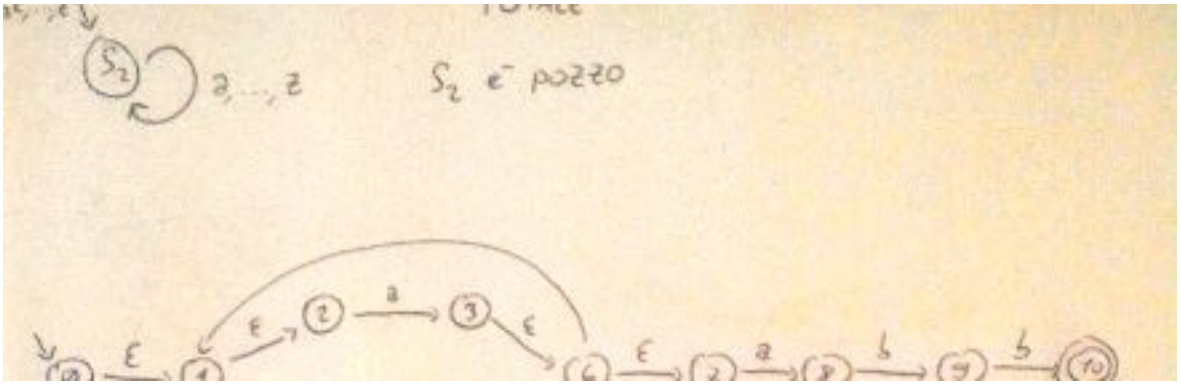
A stato iniziale, E unico stato finale. Fare il DFA.



in grassetto quelli già ε-chiusi

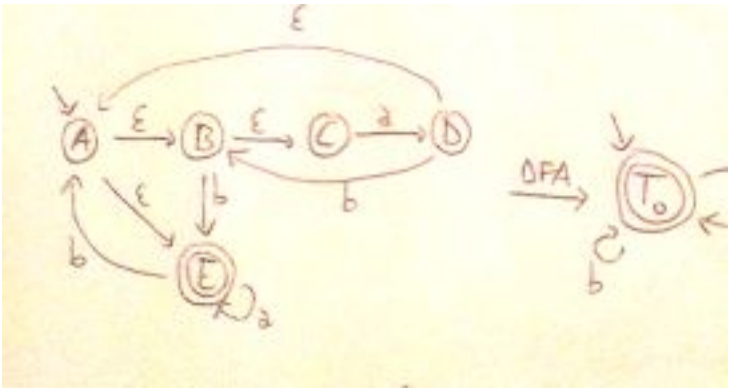
states	a	b
T0 = {**A** B C E}	T1	T2 = T0
T1 = {A B C **D E**}	T1	T0
T2 = {**A** B C **E**} = T0 (quindi T0 va in T0 tramite b)	come T0	come T0

Es:



*in grassetto quelli già ε-chiusi

states	a	b
S_0^d = {**0** 1 2 4 7 }	T1	T2
T1 = { 1 2 **3** 4 6 7 **8** }	T1	T3
T2 = { 1 2 4 **5** 6 7 }	T1	T2
T3 = { 1 2 4 **5** 6 7 9 }	T1	T4
T4 = { 1 2 4 5 6 7 10}	T1	T2



04/10/17

$Move_d$* e partizionabilità

Ipotesi di lavoro: DFA ha (in input) funzione di transizione totale.

Funzioni:

$move_d^*(s, \varepsilon) = s$

$move_d^*(s, wa) = move_d(move_d^*(s, w), a)$

Es:

$move_d^*(s, abba) \rightarrow w = abb, a = a$
$move_d^*(s, abb) \rightarrow w = ab, a = b$
$move_d^*(s, ab) \rightarrow w = a, a = b$
$move_d^*(s, a) \rightarrow w = \varepsilon, a = a$
$move_d^*(s, \varepsilon) = s$

Sia $D = (S, A, move_d, s_0, F)$ un DFA con funzione di transizione totale.
Allora $s_1, s_2 \in S$ sono equivalenti, scritto $s_1 \sim s_2$,
se e solo se $\forall\ w \in A^*$, $move_d^*(s_1, w) \in F$ se e solo se $move_d^*(s_2, w) \in F$

Quando si dice che un insieme B è partizionabile?
$D = (S, A, move_d, s_0, F)$
$move_d$ è totale
Siano B_i, B_j blocchi distinti di una partizione di S
Sia $a \in A$
Diciamo che B_i è partizionabile rispetto a (B_j, a) se $\exists\ s, t \in B_i$ tali che $move_d(s, a) \in B_j$ e $move_d(t, a) \notin B_j$

Inizialmente determino due blocchi di stati
B_1 contiene tutti gli stati in F
B_2 contiene tutti gli stati in $S \setminus F$
Controlliamo che:
1) ogni blocco contenga stati equivalenti
2) blocchi distinti non contengano stati equivalenti

Sostituiamo il blocco B_i con
$split(B_i, (B_j, a)) = \{s' \in B_i \mid move_d(s', a) \in B_j\}$ e $\{ t' \in B_i \mid move_d(t', a) \notin B_j \}$

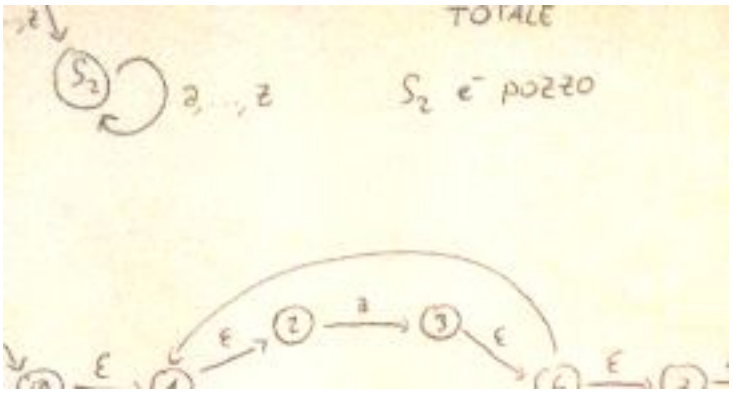
Algoritmo di partition refinement

input: DFA $D = \{S, A, move_d, s_0, F)$ con $move_d$ totale
output: partizione di S in blocchi di stati equivalenti

$B_1 = F$
$B_2 = S \setminus F$
$P = \{B_1, B_2\}$

```
while(∃ Bi, Bj ∈ P, ∃ a ∈ A, Bi è partizionabile rispetto a (Bj, a))
      sostituire Bi in P con split(Bi, (Bj, a))
```

Es:



Divido gli stati in due blocchi: quelli non terminali in un blocco e quelli terminali in un altro.

{A B C D} {E}

Considero le transizioni a. Da A, B, C e D vado in nodi dello stesso blocco. Non ho quindi motivo di splittare l'insieme.

{A B C D} {E}

Considero le transizioni b. Da A, B e C vado in nodi dello stesso blocco, ma con D vado in E, che è dell'altro blocco. Questo non va bene, quindi splitto l'insieme e tolgo la D. In maniera formale, faccio `split({A, B, C, D}, ({E}, b))`.

{A B C} {D} {E}

Considero le transizioni a. Da A, B e C vado in nodi dello stesso blocco. Non ho quindi motivo di splittare l'insieme.

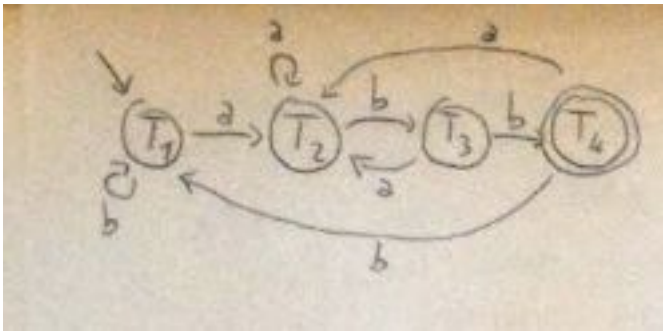
{A B C} {D} {E}

Considero le transizioni b. Da B vado in D con una transizione b, quindi come prima devo splittare l'insieme e togliere la B.

{A C} {B} {D} {E} (li chiamo t_1, t_2, t_3 e t_4)

Adesso controllo se devo splittare questi insiemi. Vanno tutti bene, quindi non bisogna dividerli ulteriormente.

Il grafo si può quindi disegnare così:



05/10/17

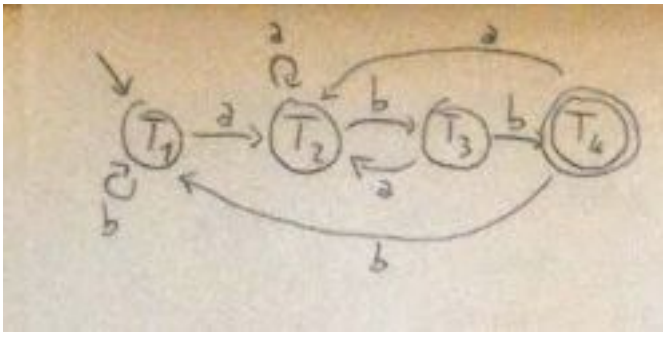
Algoritmo di minimizzazione di DFA

input: DFA D = {S, A, $move_d$, s_0, F) con $move_d$ totale
output: minimo DFA che riconosce lo stesso linguaggio del linguaggio riconosciuto dal DFA in input

- computare il partition refinement degli stati di D
  sia P = (B_1, …, B_k) l'output del raffinamento
- foreach (B_i in P) settiamo uno stato temporaneo t_i, se B_i contiene s_0 allora diciamo che t_i è iniziale per min(D)
- foreach (B_i in P tale che $B_i \subseteq F$) dichiariamo t_i stato finale di min(D)
- foreach ((B_i, a, B_j) tale che $\exists$ $s_i \in B_i$, $s_j \in B_j$ tali che $move_d(s_i, a) = s_j$) settiamo una transizione temporanea in min(D) da t_i a t_j secondo il simbolo 'a'
- foreach (dead state t_i)
  rimuovere t_i e tutte le transizioni da/verso t_i
- tutti i temporanei residui (sia stati che transizioni) sono gli stati e le transizioni di min(D)

Questo algoritmo ha complessità O(n*logn)

Continuo l'esempio di prima. Ero arrivato al punto in cui avevo fatto il partition refinement, ed ero nello stato

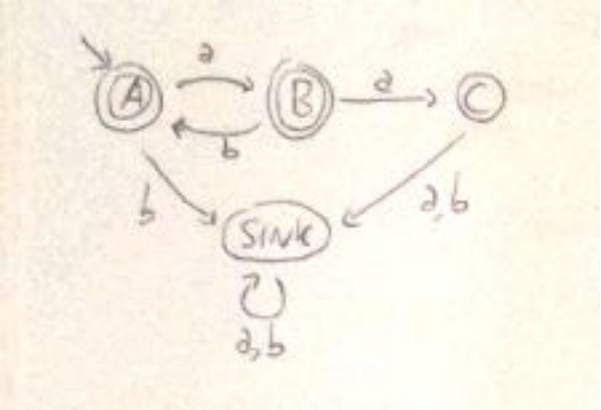


t_1 = {A C} t_2 = {B} t_3 = {D} t_4 = {E}
Posso ora applicare l'algoritmo di minimizzazione di DFA

06/10/17

Es:

Aggiungo il sink



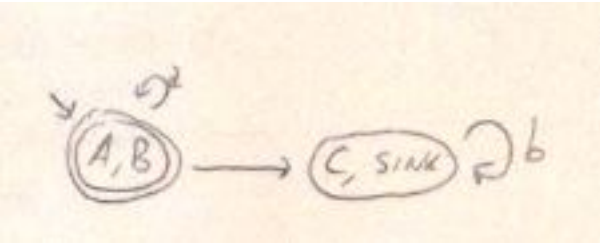
{A, B} {C, sink}

Considero le transizioni a. Da A e B vado in nodi dello stesso blocco. Non ho quindi motivo di splittare l'insieme.

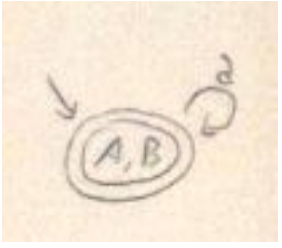
Considero le transizioni b. Da A e B vado nello stesso blocco, quindi non ho motivo di splittare l'insieme.

Considero le transizioni a del secondo blocco. Anche in questo caso vanno nello stesso blocco, così come le b transizioni. Non splitto nulla

Il grafo diventa quindi



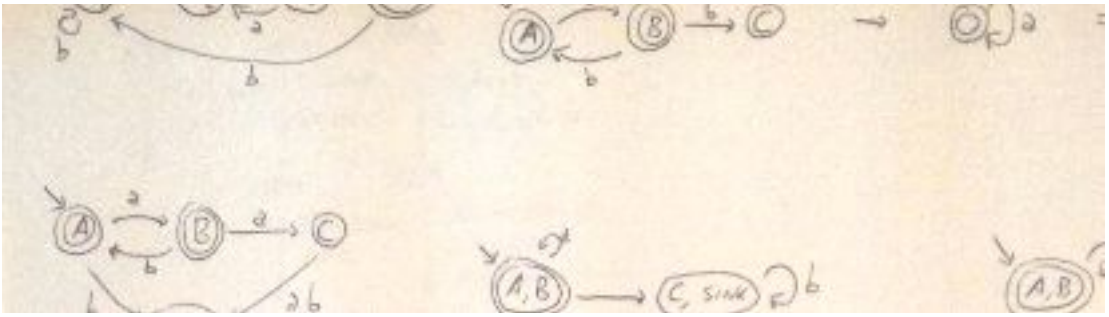
Visto che {C, sink} è un sink per questo grafo, posso togliere il nodo e tutte le transizioni da e verso di lui. Rimane quindi solo



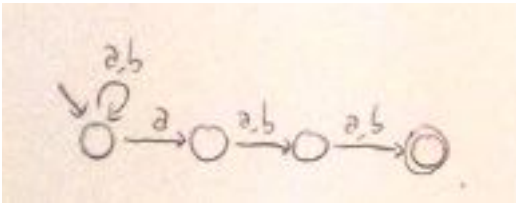
Es:

Sia r = (a|b)*a(a|b)(a|b)

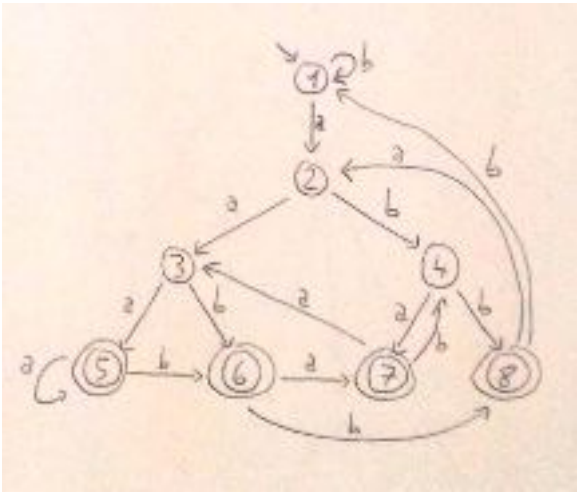
Determinare il minimo DFA per il riconoscimento di L(r)



Invece che fare la costruzione di thompson di sopra, usiamo un grafo comunque NFA ma più semplice (che ci si può inventare con la fantasia)



Il DFA totale è fatto così



10/10/17

Lemma: Per ogni $n \in N^+$, esiste un NFA con (n+1) stati il cui minimo DFA equivalente ha almeno 2^n stati.

Dim:

Consideriamo $L = L((a|b)^*a(a|b)^{n-1})$

Un NFA che riconosce L è

ed ha n+1 stati (INCLUSO L'ULTIMO)

Supponiamo per contraddizione che esista un DFA minimo per il riconoscimento di L con k stati per $k < 2^n$.
Osserviamo che esistono esattamente 2^n parole distinte su {a, b} di lunghezza n.
$\Rightarrow \exists\ w_1, w_2 : w_1 != w_2$ e $|w_1| = |w_2| = n$ e il loro riconoscimento conduce al medesimo stato del DFA
$\Rightarrow$ esiste almeno una posizione in cui w_1 e w_2 differiscono, consideriamo tra queste posizioni quella più a destra.
$\Rightarrow w_1 = w_1'ax \qquad w_2 = w_2'bx$ (iniziano in modi diversi ma finiscono uguali)
$\Rightarrow$ considero la parola $w_1'' = w_1'ab^{n-1} \qquad w_2'' = w_2'bb^{n-1}$
da cui possiamo raggiungere lo stato t che per definizione di L e di D è uno stato finale. La seconda parola, però, non appartiene al linguaggio, nonostante possa comunque raggiungere lo stato t. Per questo motivo contraddiciamo il fatto che lo stato t possa essere finale.

Per vedere se una certa parola fa parte di una regexp, abbiamo varie possibilità:

Complessità dei vari algoritmi:

Algoritmo	Complessità nello spazio	Complessità nel tempo
Thompson Construction	$O(\|r\|)$	$O(\|r\|)$ OR $O(n_n+m_n)$
Simulazione NFA	-	$O(\|w\|(n_n+m_n))$
Subset Construction	-	$O(n_d \|A\|(n_n+m_n))$

Minimizzazione DFA	-	$O(n_d \log n_d)$
Simulazione DFA	-	$O(\|w\|)$

I linguaggi (denotati da regexp / riconosciuti da NFA / riconosciuti da DFA) sono esattamente la stessa cosa. Un altro modo per chiamarli è “generati da grammatiche regolari”, ovvero le cui produzioni sono del tipo $A \rightarrow aB$ oppure $A \rightarrow \varepsilon$.

Es: Fornire una procedura che, dato un DFA D, restituisca una grammatica regolare G tale che L(G) = L(D)

> Se ho $A \rightarrow B$ con una a-transizione, diventa $A \rightarrow aB$. Segno quindi il nome del nodo che sto considerando prima della freccia e, dopo la freccia, la lettera sulla transizione seguita dal nodo di destinazione. Se ho un nodo terminale C, scriverò $C \rightarrow \varepsilon$.

11/10/17

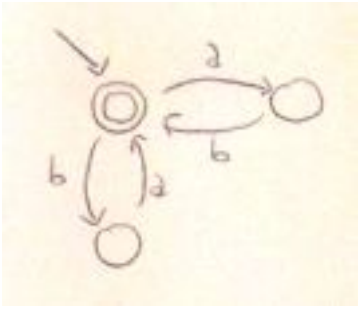
Es: Fornire una procedura che, data una grammatica regolare G, restituisca un automa a stati finiti D tale che L(G) = L(D)

> G = (V, T, S, P)
> Si fa il contrario rispetto alla procedura di prima. Per ogni $A \rightarrow aC$, faccio una a-transizione da A a C. Questo vale per ogni regola della grammatica.
> Se questo automa è non deterministico, applico la subset construction e lo faccio diventare deterministico (se richiesto).

Es: L = {w | w ∈ {a, b}* AND il numero di occorrenze di “a” in w è pari AND il numero di occorrenze di “b” in w è dispari}. Dire se L è un linguaggio regolare oppure no

> E’ regolare

Es: L = {w | w ∈ {a, b}* AND il numero di occorrenze di “a” in w è uguale all numero di occorrenze di “b” in w}



Ma non è regolare, e non può essere regolare per un motivo che tra un po' ci sarà ovvio.

Pumping lemma per linguaggi regolari

Sia L un linguaggio regolare.
Allora $\exists\ p \in N^+$ tale che $\forall\ z \in L : |z| > p$,
$\exists$ u, v, w tali che:

1) z = uvw AND
2) |uv| <= p AND
3) |v| > 0 AND
4) $\forall\ i \in N, uv^iw \in L$

Dim:

L è regolare, quindi può essere riconosciuto da un automa a stati finiti.
Sia $D = (S, A, move_d, s_0, F)$ il min DFA tale che L(D) = L
Sia p = |S|
Allora i cammini più lunghi che non passano più di una volta nel medesimo stato hanno al più lunghezza (p-1).
Allora se $z \in L$ con $|z| > p$, z è riconosciuta tramite un cammino che attraversa almeno due volte almeno uno stato.

12/10/17

Negazione della tesi del pumping lemma per i linguaggi regolari è:
$\forall\ p \in N^+ : \exists\ z \in L : |z| > p\ .\ \forall$ uvw
$(z = uvw \text{ AND } |uv| <= p \text{ AND } |v| > 0) \rightarrow (\ \exists\ i \in N : uv^iw \notin L\ )$

Lemma: $L = \{a^nb^n \mid n >= 0\}$ non è regolare
Dim:

Assumiamo che L sia regolare.
Sia p un qualunque numero naturale positivo.
Sia $z = a^pb^p$
Allora $\forall$ uvw tale che z = uvw AND |uv| <= p AND |v| > 0
(la stringa v contiene solo occorrenze di a, e ne contiene almeno una)

Allora uv^2w ha la forma $a^{p+k}b^p$ con $k > 0$
Allora $uv^2w \notin L$, il che contraddice il pumping lemma per i linguaggi regolari.

Es:

$L_1 = \{w \mid w \in \{a, b\}^* \text{ e contiene almeno una occorrenza di “aa”}\}$
$L_2 = \{ww \mid w \in \{a, b\}^*\}$
$L_3 = \{ww^R \mid w \in \{a, b\}^*\}$

L_1: $A \rightarrow aA \mid bA \mid aB$
$B \rightarrow aC$
$C \rightarrow aC \mid bC \mid \varepsilon$
è regolare e quindi anche libero

L_2: Già dimostrato che non è libero (pumping lemma), quindi non regolare

L_3: Non è regolare ma libero
$z = a^pb^pb^pa^p \in L_3$
visto che $uv < p$, uv è composto solo da a
$uv^iw = a^{p-k}b^pb^pa^p \notin L_3$, quindi non può essere regolare.

Esercizi tipo esame

Esercizio 1

Sia N_1 lo NFA con stato iniziale A, stato finale E e con la seguente funzione di transizione:

	ε	a	b
A	{B, E}	∅	∅
B	{C}	∅	{E}
C	∅	{D}	∅
D	{E}	∅	{B}
E	∅	{E}	{A}

1) Dire se $aa \in L(N_1)$
2) Chiamiamo D il DFA ottenuto da N_1 per subset construction, Q lo stato iniziale di D, $Q_{ab\_}$ lo stato di D che si raggiunge da Q tramite il cammino ab. Dire a quale sottoinsieme degli stati di N_1 corrisponde $Q_{ab\_}$

1) Si, facendo $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow E$
2) Facciamo la subset construction

	a	b
Q0 = {A, B, C, E}	Q1	Q0
Q1 = {D, E}	Q2	Q0
Q2 = {E}	Q2	Q0

13/10/17

Analisi Sintattica

Es:

S → cAd
A → ab | a

Parsing top-down

Parto dal simbolo iniziale ed espando le varie derivazioni espandendo per prima quella che si trova più a sinistra. SI cerca quindi di ricostruire una derivazione leftmost della stringa w data in input.

w$, $ ∉ V
w = cabd

Uso la prima produzione perché inizia con la c, proprio come la mia parola

S ⇒ cAd

Ora noto che devo trovare una a partendo da A, ma ho due modi per farlo (a oppure ab). Se scelgo quella sbagliata, mi diventa S ⇒ cAd ⇒ cad. Le prime due lettere mi vanno bene, ma la terza no, quindi questa stringa non mi è utile. Torno indietro di un passaggio e provo un'altra produzione di A.

cAd ⇒ cabd

In questo caso noto che tutte le lettere sono uguali alla mia stringa, quindi mi va bene così e ho finito.

Parsing top-down predittivo (o non ricorsivo)

Invece che considerare la grammatica indicata nell'esempio sopra, usiamo grammatiche del tipo S → cAd
A → aB
B → b | ε

La differenza è che faccio

S ⇒ cAd ⇒ caBd

A questo punto vedo che mi serve una b, quindi sceglierò per forza la produzione che inizia con la b. Non ho motivo di scegliere quella con ε. Evito quindi errori

caBd ⇒ cabd

Esiste un tipo di grammatica chiamata LL(1)
prima L: Leggiamo la input string da sinistra
seconda L: Ricostruiamo una leftmost derivazione
(1): Decidiamo quale operazione effettuare guardando un solo simbolo di input
Le grammatiche LL(1) sono un sottoinsieme delle grammatiche libere.

Algoritmi di parsing

Strutture dati:

- input buffer w$
- pila bottom [$] top
- parsing table con tante righe quanti non-terminali, e tante colonne quante sono i terminali (incluso $)
  In ogni cella della tabella metto un'eventuale trasformazione o "error"

Algoritmo di parsing non-ricorsivo

input: stringa w, tabella di parsing non ricorsivo T per G
output: derivazione leftmost di w se w ∈ L(G), error() nel caso contrario
init: w$ nell'input buffer
$S nella pila

```
let b il primo simbolo di w
let x il top dello stack
while (x != $)
      if (x == b)
            pop(x)
            let b il simbolo necessario di w
      else if (x è un terminale)
            error()
      else if (T[x,b] == error())
            error()
      else if (T[x,b] contiene X → Y1...Yn)
            restituisci in output X → Y1...Yn
            pop(x)
            push(Yn...Y1)
      let x il top dello stack
```

Esempio:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε

F → id

	id	+	*	$
E	E → TE’			
E’		E’ → TE’		E’ → ε
T	T → FT’			
T’		T’ → ε	T’ → *FT’	T’ → ε
F	F → id			

Quelli sottolineati sono la x (pila) e b (input) nell’algoritmo di prima

pila	input	output
$E	id+id*id$	E → TE’
$E’T		T → FT’
$E’T’F		F → id
$E’T’id		
$E’T’	+id*id$	T’ → ε
$E’		E’ → TE’
$E’T+	id*id$	
$E’T		
e avanti così		

17/10/17

Es:

S → aA | bB
A → c
B → d

w = ac$
Parsing: S ⇒ aA ⇒ ac
Nella tabella metto le produzioni della grammatica

	a	b	c	d	$
S	S → aA	S → bB			
A			A → c		
B				B → d	

Data una generica α ∈ V* per G = (V, T, S, P), first(α) è l'insieme dei simboli terminali b tali che α ⇒ bv.
Inoltre, se α ⇒ ε, allora ε ∈ first(α).
Es:

S → A | B
A → a | C
C → ε
Allora first(A) = { a, ε } (ε perché posso fare A ⇒ C ⇒ ε)

Es:

S → A | B
A → a | C
C → bB
Allora first(A) = { a, b } (b perché posso fare A ⇒ C ⇒ bB, ma B non esiste)

Es:

S → A | B
A → a | C
C → bB
B → c
Allora first(A) = { a, b } (A ⇒ C ⇒ bB ⇒ bc, ma tengo solo il primo simbolo (b))

Es:

A → a | C
C → bB | ε
B → c

Allora first(A) = { a, b, ε }

G = (V, T, S, P)
Sia X ∈ V. L'insieme first(X) viene calcolato come segue

1. Inizializzare first(X) = ∅ ∀ X ∈ V
2. Se X ∈ T allora first(X) = {X}
3. Se X → ε ∈ P allora aggiungere ε ai first(X)
4. Se X → $Y_1...Y_n$ ∈ P con n >= 1 allora uso la seguente procedura:

```
j = 1
while (j <= n)
      aggiungere ai first(X) ogni b tale che b ∈ first(Y_j)
      if (ε ∈ first(Y_j))
            j++
      else
            break
if (j == n + 1)
      aggiungere ε ai first(X)
```

Sia α = $Y_1...Y_n$ con n >= 1
Allora first(α) è calcolato applicando la seguente procedura (uguale ma con α al posto di X):

```
j = 1
while (j <= n)
      aggiungere ai first(α) ogni b tale che b ∈ first(Y_j)
      if (ε ∈ first(Y_j))
            j++
      else
            break
if (j == n + 1)
      aggiungere ε ai first(α)
```

Es:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

first:
E = { id, (} //ha gli stessi first di T (poiché è il primo non-terminale di E → TE')
E' = { +, ε }
T = { id, (} //ha gli stessi first di F (poiché è il primo non-terminale di T → FT')
T' = { *, ε }
F ={ id, (} //la parentesi tonda è un terminale

Per generare "id + id":

//tabella non completa: mancano le parentesi tra i terminali

	id	+	*	$
E	E → TE’			
E’		E’ → TE’		E’ → ε
T	T → FT’			
T’		T’ → ε	T’ → *FT’	T’ → ε
F	F → id			

Per ogni A ∈ V \ T, l’insieme follow(A) è calcolato applicando la seguente procedura:

```
inizializzare follow(A) ad ∅ ∀ A ∈ V \ T
aggiungere $ a follow(S)
repeat
      foreach (B → αAβ ∈ P)
            if(β == ε)
                  aggiungo follow(B) a follow(A)
            else
                  aggiungo first(β)\{ε} a follow(A)
                  if(ε ∈ first(β))
                        aggiungo follow(B) a follow(A)
until saturazione
```

18/10/17

Es:

S → aABb
A → Ac | d
B → CD
C → e | ε
D → f | ε

	first:	follow:
S =	{a}	{$}
A =	{d}	{e, f, b (da S → aABb), c (da A → Ac)}
B =	{e, f, ε}	{b (da S → aABb)}
C =	{a, ε}	{f (da B → CD)}
D =	{a, ε}	{}

Es:

S → aA | bBc
A → Bd | Cc

B → e | ε
C → f | ε

	first:	follow:
S =	{a, b}	{$}
A =	{e, d, f, c}	{$ (?)}
B =	{e, ε}	{c, d}
C =	{f, ε}	{c}

Es:

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | id

	first:	follow:
E =	{id, (}	{ $,) }
E’ =	{+, ε }	{} ed eredita i follow di E
T =	{id, (}	{ + } ed eredita i follow di E, E’
T’ =	{*, ε }	{} ed eredita i follow di T
F =	{id, (}	{ * } ed eredita i follow di T, T’

Una volta gestite le eredità, diventa

E =	{id, (}	{ $,) }
E’ =	{+, ε }	{ $,) }
T =	{id, (}	{ +, $,) }
T’ =	{*, ε }	{ +, $,) }
F =	{id, (}	{ *, +, $,) }

Parsing di “id+id*id$”
E ⇒ TE’ ⇒ FT’E’ ⇒ idT’E’ ⇒ …

19/10/17

Costruzione di tabelle di parsing predittivo top-down
input: G = (V, T, S, P)
output: una tabella T di parsing predittivo top-down se G è LL(1)

```
foreach ((A → α) ∈ P)
      ∀ b ∈ first(α), poniamo A → α in T[A, b]
      if (ε ∈ first(α))
            ∀ x ∈ follow(A) poniamo A → α in T[A, x]
poniamo error() in tutte le entry di T che sono rimaste vuote

if(la tabella non ha entry multiply-defined)
```

G è LL(1)

Es:

E → E+T | T
T → T*F | T
F → (E) | id

	first:	follow:	
E =	{ (id }	{ $ +) }	= { $ +) }
T =	{ (id }	{ * } ed eredita i follow di E	= { $ + *) }
F =	{ (id }	{} ed eredita i follow di T	= { $ + *) }

Guardo se è LL(1)

	id
E	E → E + T E → T

Ho fatto solo una parte della tabella, ma visto che ha 2 entry (quindi entry multiple), non è LL(1).

Una grammatica G esibisce LEFT RECURSION (o è ricorsiva a sinistra) se per qualche $A \in V \setminus T$ e per qualche $\alpha \in V^*$, $A \Rightarrow^* A\alpha$
La left recursion è immediata se G ha almeno una produzione del tipo $A \rightarrow A\alpha$ (ovvero se succede in un passo)
Nell'esempio di prima, c'era left recursion immediata nei primi due casi ($E \rightarrow E\alpha$ e $T \rightarrow T\alpha$).

Proposizione:
Ogni grammatica che esibisce left recursion non è LL(1)

Questo:
A → B
B → Aa
è un esempio di left recursion in più passi.

Eliminazione di left recursion immediata

Es:

$A \rightarrow A\alpha \mid \beta$ con β != A e α != ε
diventa
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

Più in generale:
$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_n$ $\beta_1, \ldots, \beta_n$!= $A\gamma$ $\alpha_1, \ldots, \alpha_n$!= ε

diventa

$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$ A' nuovo non-terminale in G

$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Es:

1. Eliminare la left recursion immediata da
   $E \rightarrow E+T \mid T$
   $T \rightarrow T*F \mid T$
   $F \rightarrow (E) \mid id$
2. Sia G la grammatica così ottenuta. Dire se G è LL(1) oppure no.

1. $E \rightarrow TE'$
   $E' \rightarrow +TE' \mid \varepsilon$
   $T \rightarrow FT'$
   $T' \rightarrow *FT' \mid \varepsilon$
   $F \rightarrow (E) \mid id$
2. Copiati da qualche esercizio fa:

	first:	follow:
E =	{id, (}	{ $,) }
E' =	{+, ε }	{ $,) }
T =	{id, (}	{ +, $,) }
T' =	{*, ε }	{ +, $,) }
F =	{id, (}	{ *, +, $,) }

Tabella di parsing:

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

I campi vuoti sono error. Non ci sono multiple entries, quindi è LL(1).

20/10/17

Es:

1. Eliminare la left recursion da
   $E \rightarrow E + E \mid E * E \mid (E) \mid id$
2. Dire se la grammatica ottenuta è LL(1)

1. La grammatica diventa:
   $E \rightarrow (E)E' \mid idE'$

E’ → +EE’ | *EE’ | ε

2. First e follow

	first:	follow:	
E =	{ (id }	{ $) + * } e i follow di E’	= { $) + * }
E’ =	{ + * E }	{ } e i follow di E	= { $) + * }

Parte di tabella (tanto non serve tutta):

	+	*
E		
E’	E’ → +EE’ E’ → ε	E’ → *EE’ E’ → ε

Visto che c’è almeno un’entry multipla, la grammatica non è LL(1)

L’eliminazione della left recursion dalla grammatica iniziale G ci ha dato la grammatica G_1, che non è LL(1), così come quella iniziale. Non è quindi detto che rimuovendo la left recursion si ottenga una grammatica LL(1). Nel nostro caso, G_1 è anche ambigua.

Lemma:

L’eliminazione della left recursion NON elimina l’ambiguità.

Left factoring

S → aSb | ab

	a	b	$
S	S → aSb S → ab		

La grammatica non è LL(1). Possiamo però fattorizzare le produzioni considerando una parte che è a sinistra ed è comune a più produzioni, per ottenere una grammatica LL(1) che genera lo stesso linguaggio.

S → aA’
A’ → Sb | b

DEF:

Una grammatica G può essere fattorizzata a sinistra quando esistono almeno due produzioni $A \to \alpha\beta_1$ e $A \to \alpha\beta_2$ per qualche $A \in V \setminus T$, $\alpha, \beta_1, \beta_2 \in V^*$ e α non comincia per A.

DEF:

G può essere fattorizzata a sinistra se

$A \rightarrow \alpha\beta_1$, $A \rightarrow \alpha\beta_2 \in P$ con
$\alpha, \beta_1, \beta_2 \in V^*$, α non ha A come primo simbolo, $A \in V \setminus T$

Lemma:
Se G può essere fattorizzata a sinistra allora G non è LL(1)

Algoritmo di fattorizzazione a sinistra

```
foreach (A ∈ V \ T)
      trovare il prefisso più lungo comune a due o più produzioni per A, chiamato
α
      if (α != ε)
            sostituire A → αβ1|...|αβn|γ1|...|γk
            con A → αA'|γ1|...|γk
            con A' → β1|...|βn e A' nuovo simbolo

            ---appunti sketo--
```

24/10/17

Bottom up

Ricostruire, se $w \in L(G)$, una rightmost derivation al contrario.

Es:
$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

w = abbcde
Visto che è rightmost, devo espandere per primo il simbolo più a destra (la B)
$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

La cosa sottolineata vuol dire “se arrivo in questo stato e sto leggendo come prossimo input una d o una b, posso fare la riduzione della b usando A. Lo stesso vale per le altre, ovviamente con i loro simboli. La roba tra parentesi graffe si chiama “look-ahead set”.

Nel grafo faccio quindi i seguenti passi (i numeri sono i nodi):

0	abbcde$
0 → 2 consumando ‘a’	a || bbcde$
2 → 4 consumando ‘b’	ab || bcde$
4 riduco A → b	aA || bcde$

A questo punto torno al nodo 2, ovvero il precedente. Vado quindi in 3, perché ho la A al posto della b che avevo prima.

torno a 2, vado in 3	
3 → 7 consumando ‘b’	aAb || cde$
7 → 8 consumando ‘c’	aAbc || de$
8 riduco A → Abc	aA || de$
torno a 7, torno in 3, vado in 9	
3 → 9 consumando ‘d’	aAd || e$
riduco B → d	aAB || e$
torno a 3, vado in 5, vado in 6	
5 → 6 consumando ‘e’	aABe || $
6 riduco S → aABe	S || $
torno a 0, vado in 1, ho finito	

Noi vogliamo avere grammatiche di tipo LALR(1). Grammatiche: SLR(1) ⊂ LALR(1) ⊂ LR(1).

Es:

S → aSb | ab
w = aaabbb$

0	aaabbb$
0 → 2	a || aabbb$
2 → 2	aa || abbb$

2 → 2	aaa || bbb$
2 → 4	aaab || bb$
4 riduco S → ab	aaS || bb$
torno a 2, vado in 3, vado in 5	
3 → 5	aaSb || b$
5 riduco S → aSb	aS || b$
torno a 3, vado in 5	
3 → 5	aSb || $
5 riduco S → aSb	S || $
torno a 0, vado in 1, ho finito	

Questa è una tabella:

	terminali ∪ $	V \ T
stati	shiftK: letti un simbolo di input e vai allo stato K; reduce A → b	gotoK: descrive le funzioni di transizioni identificate dai non-terminali (quando "consumi" roba)

Algoritmo di shift/reduce

(comune a SLR(1), LR(1), LALR(1))

input: w, tabella di parsing bottom-up M di tipo ♦, con ♦ scelto tra {SLR(1), LALR(1), LR(1)}, per la grammatica G.
output: derivazione rightmost di w se w ∈ L(G), altrimenti error()
init: s_0 sulla pila, w$ sull'input buffer

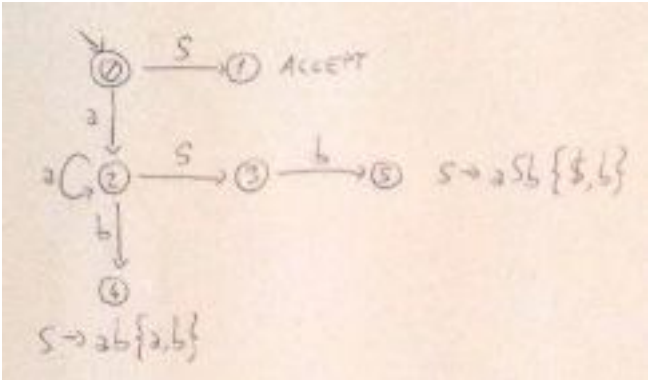
```
let b primo simbolo di w$
while true
      let s il top della pila
      if M[s, b] == shiftK
            push b sulla pila
            push k sulla pila
            let b il successivo simbolo nel buffer
      else if M[s, b] == "reduce A → β"
            pop di 2*|β| simboli dalla pila
            let j tale che M[m, A] = gj
            push A
            push j
            output "A → β"
      else if M[s, b] = accetta
            break
```

```
else
      error()
```

25/10/17

sketo
Ha detto robe sulle grammatiche LALR(1) e di come il bottom up non sia comodo per linguaggi tipo python a causa dell'indentazione. Inoltre, non ti prendono al tirocinio, quindi cazzo devo prenderti appunti io?

S → aSb | ab



Questo è uguale ma scritto in maniera diversa, per separare {$, b} in {$} e {b}. Nel caso di w = aaabbb$, un caso rappresenta il ramo più in alto, mentre l'altro il secondo ramo (più "interno").

- Automa caratteristico
- Lookahead Function

Coppie diverse di questi due insiemi ci danno tipi di grammatiche diverse
sketo
Non sono sicuro di questo elenco puntato, gianluca mi stava distraendo.

Gli automi che stiamo utilizzando devono essere in grado di ricordare abbastanza da essere in grado di tornare indietro fino al punto in cui abbiamo sostituito una certa sequenza di terminali/non terminali con un'altra.

G = (V, T, S, P)
Si aggiunge un simbolo S', e si aggiunge alla grammatica la produzione

$S' \rightarrow S$

G' = $(V \cup \{S'\}, T, S', P \cup \{S' \rightarrow S\})$

$A \rightarrow \alpha.\beta$
All'inizio ho .S, ovvero non ho ancora letto nulla, e devo leggere S.

All'inizio (il nodo iniziale), non ho ancora visto nulla. Visto che S può iniziare con aSb o ab, non sappiamo davanti a quale sviluppo ci troviamo. Il primo stato è quindi

$S' \rightarrow .S$
$S \rightarrow .aSb$
$S \rightarrow .ab$

Questo può essere visto come un nodo.
Da questo stato mi muovo verso un altro stato (con una a transizione, perché vedo che iniziano quasi tutte con a). In questo stato avrò

$S \rightarrow a.Sb$
$S \rightarrow a.b$

Adesso mi aspetto di vedere l'espansione di una S. Devo quindi aggiungere a questo nodo anche quelle produzioni, e diventa quindi

$S \rightarrow a.Sb$
$S \rightarrow a.b$
$S \rightarrow .aSb$
$S \rightarrow .ab$

Notare che le ultime due sono le stesse delle ultime due del nodo prima. Quella è la chiusura, mentre le due prima sono i generatori dello stato (kernel dello stato, kernel items). Gli stati che sono "terminali" (ovvero che nel disegno prima avevano le transizioni scritte vicino), sono del tipo $S \rightarrow ab.$, ovvero che hanno incontrato di tutto e di cui si può eseguire la riduzione Questi si chiamano "reducing items".
Dallo stato con 4 items che avevamo prima, si può fare una b transizione che va in uno di questi stati terminali, ovvero

$S \rightarrow ab.$

Questo perché la seconda produzione si aspetta b, che poi completa quello che viene generato da quella produzione. Sempre da quello stato con 4 produzioni partirà anche una a transizione e una S transizione. Per vedere che transizioni devo avere, devo vedere la prima lettera dopo il punto per ogni item di quel nodo.

Items

G = (V, T, S, P)

G’ = (V ∪ {S’}, T, S’, P ∪ {S’ → S}) con S’ ∉ V

Un LR(0)-item di G’ è una produzione di G con un punto in qualche posizione del body, ovvero A → α.β .
Alla produzione della forma A → ε corrisponde un solo LR(0)-item, ovvero A → .
L’item A → α.β è detto:

- INIZIALE se A = S’ && α = ε && β = S, cioè se l’item è S’ → .S
- ACCEPTING se A = S’ && α = S && β = ε, cioè se l’item è S’ → S.
- KERNEL se è o un iniziale o tale che α != ε
- CLOSURE se α = ε e non è iniziale
- REDUCING se non è accepting e β = ε (cioè se il punto è in fondo && !accepting)

26/10/17

Costruzione di automa caratteristico LR(0) o LR(1)

(data un’appropriata istanziazione di P_0 (stato iniziale) e closure function)

Prima di tutto facciamo la costruzione dell’automa caratteristico, usando l’algoritmo riportato sotto.
Istanziando P_0, closure otteniamo

- automi caratteristici LR(0)-automi ← parsing SLR(1)
- automi caratteristici LR(1)-automi ← parsing LR(1)

Lookahead function LA : (F x P) → p(T ∪ {$})
con F = insieme degli stati finali dell’automa caratteristico considerato. Si considerano stati finali gli stati che contengono almeno un reducing item.
L’automa caratteristico sarà necessario per la creazione della tabella di parsing.

```
Inizializzare la collezione Q di stati come {P0}
flag P0 come non marcato
while (c’è uno stato non marcato P in Q)
      marca P
      foreach (Y ∈ V) // Y simbolo di grammatica
            tmp = ∅
            foreach (A → α.Yβ ∈ P)
                  add A → αY.β a tmp
            if(tmp != ∅)
                  if(tmp == kernel(R) (per qualche R in Q))
                        τ(P, Y) = R
                  else
                        newstate = closure(Tmp)
                        τ(P, Y) = newstate
                        add newstate a Q come non marcato
```

sketo

Non so se la roba scritta da adesso serve davvero per eseguire la chiusura o no, non credo di averlo capito

Come eseguire la chiusura

Se ho la grammatica
S' → S
S → aSb | ab
$closure_0$({S' → .S}) è composta da
S → .aSb
S → .ab

Se invece ho una grammatica tipo
E' → E
E → E+T | T
T → T*F | F
F → (E) | id
Allora $closure_0$({E' → .E}) diventa
E → .E+T //quelle con il punto prima della E le o già aggiunte, non faccio nulla
E → .T //ora aggiungo quelle con il punto prima della T
T → .T*F //quelle con il punto prima della T le o già aggiunte, non faccio nulla
T → .F //ora aggiungo quelle con il punto prima della F
F → .(E) //terminale dopo il punto, non devo aggiungere nulla
F → .id //terminale dopo il punto, non devo aggiungere nulla
Si devono quindi fare in ordine per essere sicuri di non dimenticarsene alcune.

Algoritmo di $closure_0$(P)

```
tag ogni item in P come non marcato
while (c'è un item I non marcato in P)
      marca I
      if(I ha la forma A → α.Bβ)
            foreach ((B → γ) ∈ P)
                  if (B → .γ ∉ P)
                        add B → .γa
                        segna P come non marcato
return P
```

LR(0) automa si ricava utilizzando, nell'algoritmo di costruzione dell'automa caratteristico:
- P_0 = $closure_0$({S' → .S}) per la grammatica G arricchita con S' → S
- $closure_0$ per closure

Es:
S' → S
S → aSb | ab

P_0 = S' → .S
S → .aSb
S → .ab

P_1 = S' → S. // ci si arriva con una S-transizione

P_2 = S → a.Sb // ci si arriva con una a-transizione
S → a.b
S → .aSb
S → .ab

P_1 è finito, quindi guardo P_2

P_3 = S → aS.b // ci si arriva con una S-transizione da P_2

P_4 = S -> ab. // ci si arriva con una b-transizione da P_2

P_5 = S → .aSb // ci si arriva con una a-transizione da P_2 (stesse righe di P_2)
S → .ab // la freccia quindi va da P_2 a P_2, perché $P_5 \subset P_2$

Ora guardo P_3

P_6 = S → aSb. // ci si arriva con una b-transizione da P_3

Creazione della tabella di parsing

Sia τ la funzione di transizione dell'automa caratteristico considerato.
Sia LA la lookahead function considerata.
La tabella di parsing bottom-up per la coppia prescelta di automa e lookahead function è una matrice Q x (V ∪ {$}) dove Q è l'insieme degli stati dell'automa prescelto.
Ciascuna entry (P, Y) è compilata come segue:

- inserire "shift R" se Y ∈ T && τ(P, Y) = R
- inserire "reduce A → β" se A → β. ∈ P && Y ∈ LA(P, A → β)
- inserire "accept" se Y = $ && S' → S. ∈ P
- inserire "error()" se Y ∈ T ∪ {$} && non è ancora stato inserito nulla applicando i passi precedenti
- inserire "goto R" se Y ∈ V \ T && τ(P, Y) = R

27/10/17

SLR(1)
LR(0)-automa
LA(P, A → β) = follow(A) ∀ P ∈ Q (automa)

|NUMERO NODO| PARTE CHE PUÒ' AVERE CONFLITTI | GOTO PART |

	a	b	$	S
0	S2			g1
1			ACCEPT	
2	S2	S4		g3
3		S5		
4		R "S → ab"	R "S → ab"	
5		R "S → aSb"	R "S → aSb"	

S = shift
R = reduce
g = goto

Se ho la parola w = aabb$ faccio

0	aabb$	//sono in 0 e vedo a. Faccio [0, a] nella tabella
0a2	abb$	//sono in 2 e vedo a. Faccio [2, a] nella tabella
0a2a2	bb$	//sono in 2 e vedo b. Faccio [2, b] nella tabella
0a2a2b4	b$	//sono in 4 e vedo b. Faccio il reduce (cella 4b)
		//visto che crea 2S, e [2, S] = g3, aggiungo 3
0a2S3	b$	//sono in 3 e vedo b. Faccio [3, b] nella tabella
0a2S3b5	$	//sono in 5 e vedo $. Faccio il reduce (cella 5$)
		//visto che crea 0S, e [0, S] = g1, aggiungo 1
0S1	$	//ok perché [S, 1] è accept.

Il prossimo è un esempio con una grammatica ambigua.
Es:

E → E + E | E * E | id

automa LR(0)

//Se va a triangoli, non parte dal kernel, altrimenti si (?)

0 = E' → .E
E → .E + E
E → .E * E
E → .id

1 = E' → E.
E → E. + E
E → E. * E

2 = E → id.

3 = E → E +. E
E → .E + E
E → .E * E
E → .id

4 = E → E * .E
E → .E + E
E → .E * E
E → .id

5 = E → E + E.
E → E. + E
E → E. * E

6 = E → E * E.
E → E. + E
E → E. * E

	id	+	*	$
0	S2			
1		S3	S4	ACCEPT
2		R "E → id"	R "E → id"	R "E → id"
3	S2			
4	S2			
5		S3 R "E → E + E"	S4 R "E → E + E"	R "E → E + E"
6		S3 R "E → E * E"	S4 R "E → E * E"	R "E → E * E"

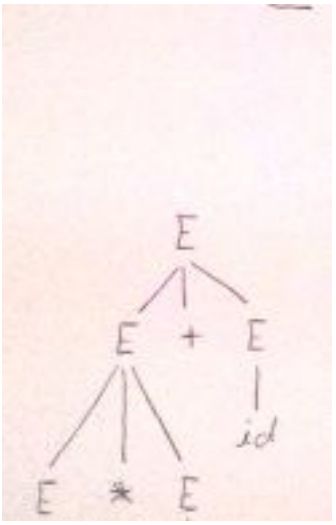
Visto che ci sono conflitti nella tabella, questa grammatica non è SLR. In questo caso i conflitti sono di tipo "Shift/Reduce", ma possono essere anche di tipo "Reduce/Reduce".

Tra le produzioni nelle celle colorate, quali sono quelle da "tenere" per avere una grammatica che associa a sinistra?

Nella cella [5, +] devo tenere R "E → E + E"
Nella cella [5, *] devo tenere S4
Nella cella [6, +] devo tenere R "E → E * E"
Nella cella [6, *] devo tenere R "E → E * E"

w = id*id+id$
0
0id2

0E1
0E1*4id2
0E1*4E6
0E1
0E1+3id2
0E1+3E5
0E1
ok



Es:

S → aAd | bBd | aBe | bAe
A → c
B → c

Questa grammatica produce 4 stringhe, ognuna in un modo, e quindi ovviamente non è ambigua.

0 = S' → .S 1 = S' → S. 2 = S → a.Ad

S → .aAd
S → .bBd
S → .aBe
S → .bAe

S → a.Be
A → .c
B → .c

3 = S → b.Bd
S → b.Ae
B → .c
A → .c

4 = S → aA.d

5 = S → aB.e

6 = A → c.
B → c.

	a	b	d	e	$
6		R “A → c” R “B → c”	R “A → c” R “B → c”		

Anche se non è una grammatica ambigua, ci sono multiple entries. Questo dipende dal modo in cui scegliamo i follow. Nel parsing canonico, infatti, non si usano gli item LR(0) ma gli item LR(1) perché ci portiamo dietro informazione direttamente dagli item. Questo riduce/rimuove i problemi di questo tipo.
Un item LR(1) canonico è del tipo $A \rightarrow \alpha.\beta,\Delta$ con $\Delta \subseteq T \cup \{\$\}$

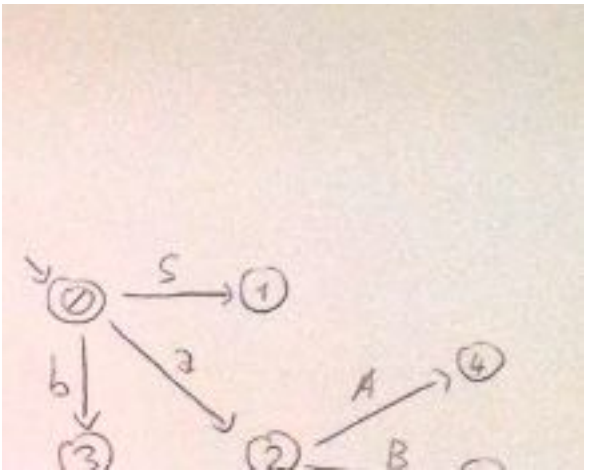
08/11/17

LR(1)

Negli automi caratteristici LR(1) si considera come stato iniziale lo stato che si ottiene facendo $closure_1(\{ [S' \rightarrow .S, \{\$\}] \})$

Es:

S → aAd | aBe | bBd | bAe
A → c
B → c



0 =	S' → .S, {$} S → .aAd, {$} S → .aBe, {$} S → .bBd, {$} S → .bAe, {$}	1 =	S' → S., {$}	2 =	S → a.Ad, {$} S → a.Be, {$} A → .c, {d} B → .c, {e}
3 =	S → b.Bd S → b.Ae B → .c A → .c	4 =	S → aA.d	5 =	S → aB.e
		6 =	A → c., {d} B → c., {e}		

Grigi = non li ha scritti alla lavagna, quindi vanno completati con i lookahead set.

Algoritmo di $closure_1(P)$

```
Segna ogni item in P come non marcato
while (c'è un item I non marcato in P)
      marcare I
      if (I ha la forma [A → α.Bβ,Δ])
            Δ1 = ∪ su (d ∈ Δ) di first(βd)
            foreach (B → γ ∈ P)
                  if (B → .γ ∉ proj(P))
                        add [B → .γ,Δ1] a P come item non marcato
                  else
                        if (B → .γ,Γ] ∈ P && Δ1 !⊆ Γ)
                              update [B → .γ,Γ] in [B → .γ,Γ ∪ Δ1]
                              segnare [B → .γ,Γ ∪ Δ1] come non marcato
return P
```

09/11/17

Es:

$S \rightarrow L = R \mid R$
$L \rightarrow *R \mid id$
$R \rightarrow L$

Fare la $closure_1(\{ [S' \rightarrow S, \{\$\}] \})$ (nodo 0) e disegnare l'automa LR(1). Capire poi se è SLR(1).

[Disegno 1]

0 =
S' → .S, {$}
S → .L = R, {$}
S → .R, {$}
L → .*R, {=, $}
L → .id, {=, $}
R → .L, {$}

1 =
S' → S., {$}

2 =
S → L. = R, {$}
R → L., {$}

3 =
S → R., {$}

4 =
L → *.R, {=, $}
R → .L, {=, $}
L → .id, {=, $}
L → .*R, {=, $}

5 =
L → id., {=, $}

6 =
S → L = .R, {$}
R → .L, {$}
L → .*R, {$}
L → .id, {$}

7 =
L → *R., {=, $}

8 =
R → L., {=, $}

9 =
S → L = R., {$}

10 =
R → L., {$}

11 =
L → *.R, {$}
R → .L, {$}
L → .*R, {$}
L → .id, {$}

12 =
L → id., {$}

13 =
L → *R., {$}

NOTA: nodi come 8 e 10 sembrano uguali, ma sono diversi poiché i lookahead set sono diversi. Anche nel grafo devono quindi essere distinti (mentre se avessimo fatto un automa LR(0) sarebbero stati lo stesso nodo, visto che non consideriamo i lookahead set in quel caso). Andare al nodo 8 o al 10 sarebbe esattamente la stessa cosa, e infatti sarebbero lo stesso nodo (come visto nel disegno sotto).

Stesso automa, ma LR(0):
[Disegno 2]

Questo automa ha (almeno) uno shift-reduce conflict, mentre quello LR(1) no.

Guardiamo se è SLR(1)

Nota: se scrivo solo il numero, vuol dire "GOTO numero". La "r" vuol dire "reduce". La S ovviamente vuol dire "shift".

	id	*	=	$	S	L	R

0	S5	S4			1	2	3
1				ACC			
2			S6	r R → L			
3				r S → R			
4	S5	S4				8	7
5			r L → id	r L → id			
6	S12	S11				10	9
7			r L→*R	r L→*R			
8			r R → L	r R → L			
9				r S→L=R			
10				r R → L			
11	S12	S11				10	13
12				r L → id			
13				r L→*R			

Visto che non ci sono conflitti, questa grammatica è LR(1).
Gli stati 4 e 11 hanno la stessa proiezione. Posso quindi fare:

"$P_1 \cup P_2$" $A \rightarrow \alpha.\beta, \Delta_1 \cup \Delta_2$

Questo posso farlo per i nodi 4 e 11, 5 e 12, 8 e 10, 7 e 13. Creo quindi i nodi 411, 512, 810 e 713.

Nella tabella posso quindi togliere quegli 8 nodi e aggiungere questi 4:

	id	*	=	$	S	L	R
411	S512	S411				810	713
512			r L → id	r L → id			
713			r L→*R	r L→*R			
810			r R → L	r R → L			

Dovrò anche aggiornare i vari shift e goto delle altre righe per farli andare ai nodi nuovi. uno S5, per esempio, diventerà S512. Anche uno S12 diventerà S512.

La grammatica così generata è LALR, perché nella tabella che risulta non ci sono conflitti.

Nota: SLR(1) ⊆ LALR(1) ⊆ LR(1)

Una grammatica SLR(1) è

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Una grammatica LALR(1) è

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

Una grammatica LR(1) è

$S \rightarrow aAd \mid aBe \mid bAe \mid bBd$

$A \rightarrow c$

$B \rightarrow c$

10/11/17

Algoritmo LALR(1) (inefficiente)

1. Costruzione dell'LR(1) automa A_1
2. Creo l'LR(1) Merged Automa A_2 (LRm(1)). Ogni stato di A_2 è ottenuto come unione di tutti gli item LR(1) che appartengono a stati di A_1 con la medesima proiezione. Se lo stato P di A_2 è ottenuto unendo gli stati $Q_1 \dots Q_k$ dell'automa A_1 e se Q_1 ha una Y-transizione a Q', allora P ha una Y-transizione allo stato P' tale che proj(P') = proj(Q').
3. $\forall$ stato finale P dell'automa &&
   $\forall [A \rightarrow \beta., \Delta_i] \in P$. $LA(P, A \rightarrow \beta) = \cup \Delta_j$ con $[A \rightarrow \beta., \Delta_j] \in P$

Es:

$S \rightarrow aAd \mid aBe \mid bBd \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

Faccio l'LR(1) automa

[Disegno 3]

0 =
$S' \rightarrow .S, \{\$\}$
$S \rightarrow .aAd, \{\$\}$
$S \rightarrow .aBe, \{\$\}$
$S \rightarrow .bBd, \{\$\}$
$S \rightarrow .bAe, \{\$\}$

1 =
$S' \rightarrow S., \{\$\}$

2 =
$S \rightarrow a.Ad, \{\$\}$
$S \rightarrow a.Be, \{\$\}$
$A \rightarrow .c, \{d\}$
$B \rightarrow .c, \{e\}$

3 =
$S \rightarrow b.Bd, \{\$\}$
$S \rightarrow b.Ae, \{\$\}$
$B \rightarrow .c, \{d\}$
$A \rightarrow .c, \{e\}$

4 =
$S \rightarrow aA.d, \{\$\}$

5 =
$S \rightarrow aB.e, \{\$\}$

6 =
$A \rightarrow c., \{d\}$
$B \rightarrow c., \{e\}$

7 = S → bB.d, {$}

8 = S → bA.e, {$}

9 = B → c., {d}
A → c., {e}

10 = S → aAd., {$}

11 = S → aBe., {$}

12 = S → bBd., {$}

13 = S → bAe., {$}

6 e 9 hanno la stessa proiezione. Creo quindi un nuovo stato

14 = A → c., {d}
B → c., {e}
A → c., {e}
B → c., {d}

E lo metto al posto del nodo 6. Tolgo anche il nodo 9, e faccio una transizione dal nodo 3 al nodo 14 (che prima andava invece in 9).

Non ci sono altri nodi con la stessa proiezione, quindi questo è il mio LRm(1) automa.

	a	b	c	d	e	$	S	A	B
0	S2	S3					g1		
1						ACC			
2			S14					g4	g5
3			S14					g8	g7
4				S10					
5					S11				
7				S12					
8					S13				
10						S→aAd			
11						S→aBe			
12						S→bBd			
13						S→bAe			
14				A → c B → c	B → c A → c				

Nella riga del nodo 14 ci sono due conflitti Reduce/Reduce. Questa grammatica non è quindi LALR.

Es:

S → Ma | bMc | dc | bda

M → d

Questa grammatica è LALR?
In teoria no, perché avresti M → d sia con lookahead set {a} che {d}, ma andrebbe fatto completamente per esserne certi.

Algoritmo LALR(1) (efficiente)

Es:

S → L = R | R
L → *R | id
R → L

[Disegno 4]

0 =
S' → .S, {x_0}
S → .L = R, {x_0}
S → .R, {x_0}
L → .*R, {=, x_0}
L → .id, {=, x_0}
R → .L, {x_0}

1 = S' → S., {x_1}

2 =
S → L. = R, {x_2}
R → L., {x_3}

3 = S → R., {x_4}

4 =
L → *.R, {x_5}
R → .L, {x_5}
L → .*R, {x_5}
L → .id, {x_5}

5 = L → id., {x_6}

6 =
S → L = .R, {x_7}
R → .L, {x_7}
L → .*R, {x_7}
L → .id, {x_7}

7 = L → *R., {x_8}

8 = R → L., {x_9}

9 = S → L = R., {x_{10}}

$x_0 = \{\$\}$ $x_1 = \{x_0\}$ $x_2 = \{x_0\}$ $x_3 = \{x_0\}$ $x_4 = \{x_0\}$
$x_5 = \{=, x_0\} \cup \{x_5\} \cup \{x_7\}$ $x_6 = \{=, x_0\} \cup \{x_5\} \cup \{x_7$ } $x_7 = \{x_2\}$
$x_8 = \{x_5\}$ $x_9 = \{x_5\} \cup \{x_7\}$ $x_{10} = \{x_7\}$

Ora devo risolvere il sistema

	class
$x_0 = \{\$\}$	x_0
x_1	x_0
x_2	x_0
x_3	x_0
x_4	x_0
$x_5 = \{=, x_0, x_5, x_7\}$	x_5
$x_6 = \{=, x_0, x_5, x_7\}$	x_6
x_7	x_0
x_8	x_5
$x_9 = \{x_5, x_7\}$	x_9
x_{10}	x_0

Mi rimangono quindi solo 4 variabili: x_0, x_5, x_6, x_9.
x_0 = {$}
x_5 = {=, x_0, x_5, x_7} = {=, x_0}
x_6 = {=, x_0, x_5, x_7} = {=, x_0, x_5}
x_9 = {x_5, x_7} = {x_5, x_0}

Creo un grafo mettendo sui nodi le variabili, con solamente i terminali di quella variabile (non quelli presi dalle altre. Solo quelli che non sono x_k insomma).
[Disegno 5]

Ottengo:
x_0 = {$}
x_5 = {=, $}
x_6 = {=, $}
x_9 = {=, $}

Questi sono i lookahead set che stavo cercando.

[Disegno 6]

Costruzione automa simbolico

```
x₀ = newVar()
Vars = {x₀}
P₀ = closure₁({[S' → .S, {x₀}])
inizializzo Queue con x₀ = {$}
States = {P₀}
porre P₀ come non marcato   //si aggiorna anche quello in states
while(c'è uno stato non marcato P in States)
      marcare P
      foreach Y ∈ V
            tmp = ∅
            foreach( [A → α.Yβ,Δ] ∈ P )
                  aggiungere [A → αY.β,Δ] a tmp
            se (tmp != ∅)
                  if(lo stato target non è ancora stato collezionato)
                        States.add(versione simbolica del target)
                        Queue.add(equazione per kernel item in tmp)
                  else
                        raffinare le equazioni delle variabili
                        associate ai kernel item del target
```

16/11/17

Grammatiche attribuite

SDD: Syntax Directed Definition
SDT: Syntax Directed Translation

Una grammatica attribuita è una grammatica a cui sono aggiunti degli attributi e delle regole.

Es:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Faccio 3 + 4 * 5

[Disegno 7]

Il valore assegnato in fondo (i numeri) viene "propagato" verso l'alto nell'albero e, quando viene eseguita un'operazione, viene scritto il risultato.

$E_1 \rightarrow E_2 + T$	{ E_1.val = E_2.val + T.val }	← Azione Semantica
$E \rightarrow T$	{ E.val = T.val }	
$T_1 \rightarrow T_2 * F$	{ T_1.val = T_2.val * F.val }	
$T \rightarrow F$	{ T.val = F.val }	
$F \rightarrow (E)$	{ F.val = E.val }	
$F \rightarrow id$	{ F.val = lexval(id) }	

L'abstract Syntax Tree di questa grammatica è
[Disegno 8]

Attributi Sintetizzati ed Ereditati

$A \rightarrow \alpha$ A.a definito come una funzione degli attributi dei terminali e non terminali in α. Gli attributi dei terminali sono informazioni ottenute durante l'analisi lessicale.

Es:

$D \rightarrow T\ L$	{ L.i = T.t }
$T \rightarrow int$	{ T.t = integer }
$T \rightarrow float$	{ T.t = float }
$L_1 \rightarrow L_2$, id	{ L_2.i = L_1.i ; addType(lexval(id) L_1.i) }
$L \rightarrow id$	{ addType(lexval(id), L.i) }

int pluto, pippo, paperino

[Disegno 9]

Facendo bottom-up, la prima cosa che faremo sarà T → int. A questo punto T conosce “int”, e lo dice alla L “più in alto”, ovvero quella subito sotto a D. Quella L lo dice alla L sotto di lui, e questo succede anche per il livello dopo. Adesso L va ad “id”, che diventa “pluto”, che L viene a conoscere (poiché viene passato in alto). Visto che alcuni attributi dipendono dai padri e dagli attributi dei fratelli, questi sono attributi ereditati.

Gli attributi ereditati sono funzione degli attributi di siblings e del padre.

Es:

S → N
N → o Digits
N → Digits
Digits → Digits d
Digits → d

Questo albero è simile a quello di prima: c’è solo “o” a sinistra e potenzialmente infinite Digits a destra, una sotto l’altra.

Come fare per essere in grado di distinguere se c’è la “o” oppure no?

Posso ribaltare l’albero, riscrivendo la grammatica come

S → Digits	{ Digits.qualcosa }
$Digits_1$ → $Digits_2$ d	{ $Digits_1$.val = $Digits_2$.val * $Digits_2$.base + “d” }
Digits → d	{ Digits.base = 8; Digits.val = “d” }
Digits → od	{ Digits.base = 10; Digits.val = “d” }

30/11/17

Laboratorio

```
bool ShiftReduce(char *w)
      bool Success = true, LexerTime = true;
      stack States, Symbols, Attributes
      int State = ∅, Toke
```