Appunti di Automi e Linguaggi Formali

Nicola Baesso

April 19, 2022

Contents

1 Introduzione **4**
- 1.1 Definizioni utili alla comprensione . . . 4
  - 1.1.1 Algoritmi e Problemi . . . 4
  - 1.1.2 Linguaggi formali . . . 4
  - 1.1.3 Automi . . . 4
  - 1.1.4 Alfabeto e Linguaggio . . . 5
  - 1.1.5 Operazioni sui linguaggi . . . 5

2 DFA, NFA, espressioni e Linguaggi (non) regolari **6**
- 2.1 DFA . . . 6
- 2.2 NFA . . . 8
  - 2.2.1 ε-NFA . . . 9
- 2.3 DFA to NFA (e viceversa) . . . 10
  - 2.3.1 Costruzione a Sottoinsiemi . . . 10
  - 2.3.2 ε-chiusure . . . 12
- 2.4 Espressioni Regolari . . . 13
- 2.5 Equivalenza tra FA e RE . . . 14
  - 2.5.1 Da RE a ε-NFA . . . 14
  - 2.5.2 Da DFA a RE . . . 15
- 2.6 Linguaggi non regolari . . . 16

3 Grammatiche di Linguaggi liberi da contesto e PDA **16**
- 3.1 Grammatiche Context-Free . . . 16
  - 3.1.1 Proprietà delle grammatiche context-free . . . 18
- 3.2 PDA . . . 19
- 3.3 Da Grammatiche Context-Free a PDA . . . 20
- 3.4 Da PDA a Grammatiche Context-Free . . . 20
- 3.5 Linguaggi non context-free . . . 21

Disclaimer

Questi appunti sono una raccolta parziale delle spiegazioni del prof. Bresolin, e sono state scritte con la gioia di poterle portare al primo parziale dell'anno. Si consiglia comunque di utilizzare solo come eventuale ripasso e non come testo di studio.

1 Introduzione

1.1 Definizioni utili alla comprensione

Per questo corso, ci si avvarrà di alcune definizioni riportate in seguito, utili per una migliore comprensione della materia.

1.1.1 Algoritmi e Problemi

Un problema è definito da 3 (tre) caratteristiche specifiche: l'insieme dei possibili input, l'insieme dei possibili output e la relazione che collega questi due insiemi.
Un algoritmo è una procedura meccanica che, eseguendo delle computazioni eseguibili da un calcolatore, risolve un determinato problema se per ogni input si ferma dopo un numero finito di passaggi e produce un output corretto.
Inoltre è composto da una complessità temporale, che indica il tempo di esecuzione, e una complessità spaziale, che indica la quantità di memoria utilizzata. Entrambe queste misure sono dipendenti dalla dimensione dell'input.

1.1.2 Linguaggi formali

Un linguaggio formale può essere definito come un'astrazione del concetto di problema.
Infatti, un problema può essere espresso sia come un insieme di stringhe (che da qui in avanti indicheremo con Linguaggio), con soluzioni che indicano se una stringa è presente nel Linguaggio o meno, oppure come una trasformazione tra vari Linguaggi, dove la soluzione trasforma una stringa di input in una stringa di output.
Quindi, ogni processo computazionale può essere ridotto ad una determinazione dell'appartenenza ad un insieme di stringhe, oppure essere ridotto ad una mappatura tra insiemi di stringhe.

1.1.3 Automi

Un automa è un dispositivo matematico (inteso in forma astratta) che può determinare l'appartenenza di una stringa ad un Linguaggio e può trasformare una stringa in una seconda stringa. Possiede ogni aspetto di un computer, poichè dato un input provvede a fornire un output, è dotato di memoria e ha la capacità di prendere delle decisioni.
La memoria per un automa è fondamentale. Sostanzialmente esistono automi a memoria finita e automi a memoria infinita, quest'ultimi con accesso limitato e non.
Chiaramente si hanno vari tipi di automi, ognuno di questi adatti ad una determinata classe di linguaggi, dove vengono differenziati per quantità di memoria e per il tipo di accesso ad essa.

1.1.4 Alfabeto e Linguaggio

Esattamente come nel caso del linguaggio naturale, un alfabeto è un insieme finito e non vuoto di simboli, ed è indicato con il simbolo Σ. Da esso si ha la stringa, ovvero una sequenza finita di simboli presi da un alfabeto. Inoltre, si definisce come stringa vuota la stringa senza alcun simbolo preso dall'alfabeto, e si indica con il simbolo ε. Infine la lunghezza di una stringa indica il numero di simboli presenti nella stringa, indicandola con |w|, con w una stringa qualsiasi .

Esempio Si consideri il codice binario, composto da 0 ed 1. Allora definiremo l'alfabeto come Σ={0,1}, una stringa valida come w=010110, e in questo particolare caso |w|=6.

La potenza di un alfabeto, espressa come Σ^k con k>0, esprime l'insieme delle stringhe composte da simboli dell'alfabeto di lunghezza k. L'espressione Σ^* indica l'insieme di tutte le stringhe sull'alfabeto.

Esempio Consideriamo nuovamente il codice binario. Per 0<k<2 (inclusi) si ha
$\Sigma^0=\varepsilon$ (la stringa vuota)
Σ^1={0,1}
Σ^2={00,01,10,11}
E così per ogni k positivo. Mentre $\Sigma^*=\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

Quindi la potenza di un alfabeto crea a sua volta un alfabeto, con stringhe di lunghezza k, e tale alfabeto è composto da 2^k stringhe.

Un linguaggio L è un sottoinsieme dell'insieme di tutte le stringhe dell'alfabeto. In simboli: $L \subseteq \Sigma^*$ per un certo Σ.

1.1.5 Operazioni sui linguaggi

Definiamo le operazioni che si possono fare sui linguaggi:

-Unione: $L \cup M=\{w : w \in L \text{ oppure } w \in M\}$
-Intersezione: $L \cap M=\{w : w \in L \text{ e } w \in M\}$
-Concatenazione: L.M={uv : u ∈ L e v ∈M}
-Complemento: $\overline{L}=\{w : w \notin L\}$
-Chiusura di Kleene (o Star): $L^*=\{w1,w2,...wk : k \geq 0 \text{ e ogni } wi \in L\}$

Tutte questa operazioni godono (e ci fanno godere) della proprietà di chiusura. In altre parole, se L e M sono linguaggi regolari, allora anche $L \cup M$, $L \cap M$, L.M, $\overline{L}$ e L^* sono linguaggi regolari.

Importante! Nelle prossime righe si parleranno di DFA, funzioni di transizioni e quant'altro. Se non si ha la più pallida idea di cosa significhino queste espres-

sioni, v'invito a passare per la sezione DFA del seguente file, e successivamente ritornare sull'esempio che segue.

Esempio Vogliamo (in realtà no, ma ci serve) dimostrare che la proprietà di chiusura vale per l'intersezione (dimostriamo la chiusura per intersezione).
Partiamo dal fatto (che vedremo in dettaglio più avanti) che, essendo L e M linguaggi regolari, esiste un DFA che accetta L e un DFA che accetta M. Quindi, essendo anche $L \cap M$ un linguaggio regolare, possiamo costruire un automa (più precisamente, un DFA) rappresentativo del linguaggio. In particolare, questo automa simula Al che Am in parallelo, e accetta una parola $\Leftrightarrow$ tale parola è accettata sia da Al che da Am.
La funzione di transizione dell'automa è formata da $(p,q) \rightarrow (s,t)$, con $p \rightarrow s$ funzione di transizione di Al, e $q \rightarrow t$ funzione di transizione di Am.
L'automa accetta una parola solo se Al e Am sono entrambi in uno stato finale. In altre parole, con p stato di Al e q stato di Am, l'automa accetta una parola solo se (p,q) è una coppia di stati finali. Formalmente:

$AL \cap M = (Ql \times Qm, \Sigma, \delta L \cap M, (ql,qm), Fl \times Fm)$
dove il simbolo $\times$ è il prodotto cartesiano.

L'esempio prende in considerazione il linguaggio $L \cap M$, ma il concetto della costruzione dell'automa è valido anche per dimostrare la proprietà di chiusura nelle altre operazioni.

2 DFA, NFA, espressioni e Linguaggi (non) regolari

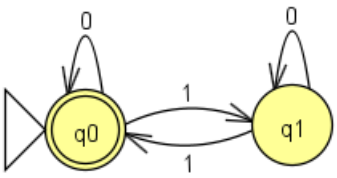
Dopo aver appreso le definizioni necessarie, andiamo a scoprire in questa sezione i primi automi (DFA ed NFA), cosa sono le espressioni regolari e cosa sono i Linguaggi non Regolari.

2.1 DFA

Gli Automi a Stati Finiti (in Inglese Deterministic Finite Automation, abbreviato in DFA) sono la forma più semplice di automa e dispongono di una quantità finita di memoria. Tali automi accettano una parola nel linguaggio unicamente se è in uno stato terminale.

Esempio Trovare ogni parola nel linguaggio binario che sia composta da un numero pari di 1.
Per creare questro DFA, abbiamo bisogno di due stati:
-Il primo sarà lo stato con il numero pari di 1. Tale stato sarà sia iniziale che terminale (e quindi accetterà la stringa).
-Il secondo sarà lo stato nel quale si ha un numero dispari di 1. In tale stato l'automa non accetterà la stringa.

Nel caso s'incontri uno 0, si deve rimanere nello stato attuale.
Segue il diagramma di transizione dell'automa, fatto con JFlap:



Si noti, nel esempio, che ogni stato esegue una transizione **per ogni simbolo nel linguaggio**. Inoltre, quello che fa l'automa è semplicemente "contare" quante cifre di 1 sono presenti, accettando la stringa solo quando questa cifra è pari.
In generale, possiamo dire che un DFA utilizza un numero di stati per "contare" ad un determinato scopo, ed ha una transizione per ogni simbolo del linguaggio che deve analizzare.

In maniera formale, un DFA si definisce con A=(Q,Σ,δ,q0,F), dove Q è un insieme finito di stati, Σ è un alfabeto finito (rappresenta i simboli dati in input al DFA), δ è una funzione di transizione (($q,a) \mapsto q'$), q0 è lo stato iniziale (e logicamente è nell'insieme di tutti gli stati dell'automa. In simboli: $q0 \in Q$), e F è un insieme di stati finali (ovvero gli stati dove l'automa accetta la stringa, in simboli $F \subseteq Q$).

Come già visto nell'esempio, un DFA può essere graficamente rappresentato con cerchi, ad indicare gli stati, e con frecce, ad indicarne le transizioni. Però un DFA può essere rappresentato anche tramite una tabella, chiamata tabella di transizione.

Esempio Riprendiamo l'esempio precedente.
Abbiamo già espresso l'automa che accetta ogni linguaggio con un numero pari di 1 come diagramma di transizione. Ora rappresentiamolo come tabella di transizione.
Per fare ciò, partiamo dallo stato iniziale, nel nostro caso q0, e ci segniamo le sue transizioni: se si ha uno 0, l'automa resta in q0, mentre se si ha un 1 l'automa va nello stato q1. Per q1 si fa la stessa cosa: se si ha uno 0, l'automa resta in q1, mentre con un 1 l'automa va (in questo caso, ritorna) in q0.
Di seguito, in forma tabellare, quel che ci siamo appena detti:

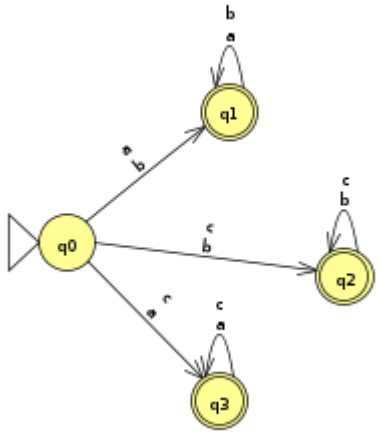
	0	1
$\rightarrow$ q0*	q0*	q1
q1	q1	q0*

Come si nota dall'esempio, nella tabella di transizione si segna con una freccia lo stato iniziale dell'automa, e con un asterisco lo stato finale dell'automa stesso (che nel nostro esempio coincide con lo stato iniziale, ma non è sempre così).
Importante! Un DFA accetta una parola w se la sua computazione accetta w, ovvero l'automa termina in uno stato finale. Inoltre, se il DFA accetta w, allora w appartiene ad un linguaggio regolare.
Più formalmente, si indica con L(A)={w $\in \Sigma^*$ |A accetta w} il linguaggio accettato, che è un linguaggio regolare.

2.2 NFA

Gli Automi a Stati Finiti non Deterministici (in inglese Nondeterministic Finite Automation, abbreviato in NFA) sono una forma di automi a stati finiti, che è utilizzato in contesti dove "contare" non è semplice, quindi si lascia spazio al non-determinismo.
Esempio Costruire un NFA che riconosca la parola che, sull'alfabeto {a,b,c}, non sia composto da tutti i simboli.
Per questo esempio, usare un DFA risulterebbe alquanto complesso, quindi ci avvarremo di un NFA per semplicità. Infatti, ci saranno sufficienti 4 stati e 6 transizioni in tutto. Di seguito troviamo sia tabella che diagramma di transizione per il seguente NFA:

	a	b	c
→ q0	{q1,q3}*	{q1,q2}*	{q2,q3}*
{q1,q2}*	q1*	{q1,q2}*	q2*
{q1,q3}*	{q1,q3}*	q1*	q3*
{q2,q3}*	q3*	q2*	{q2,q3}*
q1*	q1*	q1*	∅
q2*	∅	q2*	q2*
q3*	q3*	∅	q3*

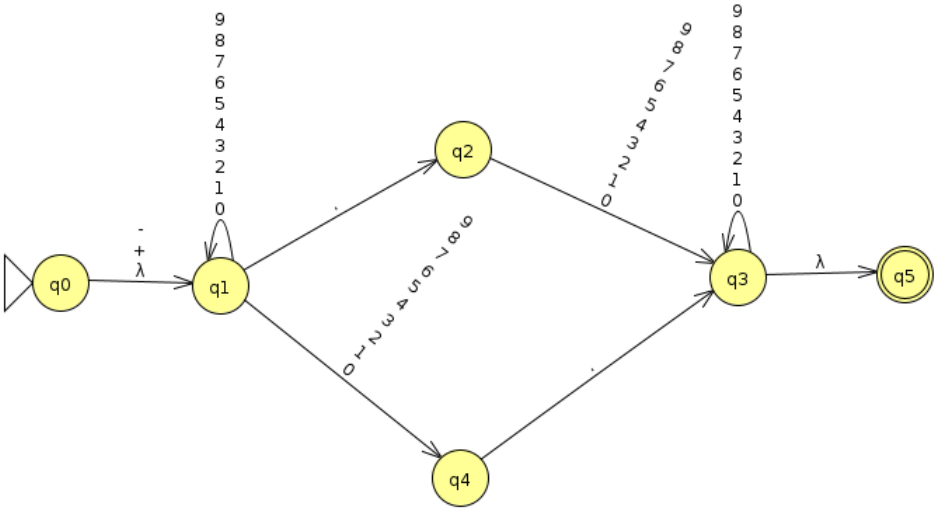


Esattamente come prima, nella tabella rappresentiamo lo stato iniziale con una freccia e con un asterisco rappresentiamo lo stato finale. Generalmente, può capitare che la tabella di transizione può essere "smagrita" ovvero che la tabella completa contiene transizioni a stati mai attraversati dall'automa (non è il caso dell'esempio), e quindi possono essere rimossi. Notiamo però che gli stati inclusi nella tabella non sono semplicemente solo gli stati semplici ma includo anche dei "gruppi" di stati, essendo le transizioni dirette in più stati (grazie al non-determinismo).
Formalizziamo: un NFA NA è descritto con NA=(Q,Σ,δ,q0,F), con l'unica differenza (rispetto ai DFA) che la funzione di transizione restituisce un sottoinsieme di Q.
Importante! Un NFA, a differenza di un DFA, può non arrivare alla fine della computazione ed interrompersi prima (come si nota bene dalle tabelle di transizioni), e quindi accetta una parola w se almeno una sua computazione accetta w, ovvero l'automa termina, in almeno una delle sue computazioni, in uno stato finale. Esattamente come nel DFA, se l'NFA accetta w, allora w appartiene ad un linguaggio regolare (la formalizzazione è la stessa).

2.2.1 ε-NFA

Gli ε-NFA sono NFA che utilizzano le ε-transizioni, ovvero utilizza transizioni che **non consumano il carattere**, quindi l'automa passa allo stato successivo mantenendo la stringa come prima.
Esempio Costruire un ε-NFA che accetti numeri decimali, riconoscendo il segno (opzionale),una stringa di decimali, un punto e un altra stringa di decimali, tenendo conto che una delle stringhe di decimali non può essere vuota.
In questo caso, usare un NFA semplice non è una buona idea, poichè non ci permette di dare opzionalità per quanto riguarda il segno. Inoltre non ci farebbe finire in uno stato finale al termine della lettura dei decimali. Di seguito troviamo sia tabella che diagramma di transizione per il seguente ε-NFA:

	ε	+,-	.	0,..,9
→ q0	q1	q1	$\emptyset$	$\emptyset$
q1	$\emptyset$	$\emptyset$	q2	{q1,q4}
q2	$\emptyset$	$\emptyset$	$\emptyset$	q3
q3	q5*	$\emptyset$	$\emptyset$	q3
q4	$\emptyset$	$\emptyset$	q3	$\emptyset$
q5*	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$



Essendo lo stato {q1,q4} non raggiungibile, si omette nella tabella.
Formalmente, un ε-NFA E è definito come E=(Q,Σ,δ,q0,F), dove l'unico parametro differente è la funzione di transizione, che prende in input uno stato in Q e un simbolo nell'alfabeto $\Sigma \cup \{\varepsilon\}$, restituendo un sottoinsieme di Q.
Essendo un particolare tipo di NFA, l'ε-NFA ha tutte le caratteristiche degli NFA.

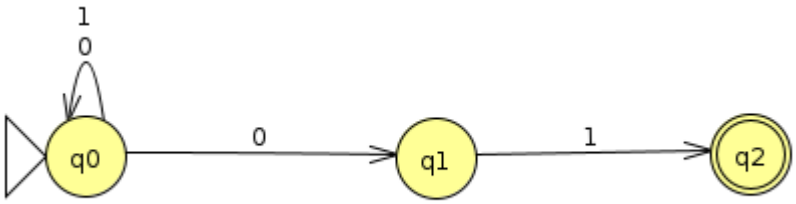
2.3 DFA to NFA (e viceversa)

Sebbene possa sembrare controintuitivo, sia NFA che DFA riconoscono gli stessi linguaggi cambiando unicamente il metodo con cui li riconoscono.
Infatti, $\forall$ NFA N $\exists$ un DFA D : L(D)=L(N) (e viceversa). Per far ciò, si usa un metodo chiamato Costruzione a Sottoinsiemi.

2.3.1 Costruzione a Sottoinsiemi

La Costruzione a Sottoinsiemi ci permette, dato un NFA, di ricavare un DFA che possa riconoscere il linguaggio del NFA. In particolare, ogni stato del DFA corrisponde ad un insieme di stati del NFA, il DFA inizia in {q0}, ovvero nell'insieme di stati con solo q0, nel DFA una stato finale corrisponde ad almeno uno stato finale del NFA, e la funzione di transizione del DFA si riferisce ad uno stato contenuto nel target della funzione di transizione di un NFA.

Esempio Prendiamo il seguente NFA:



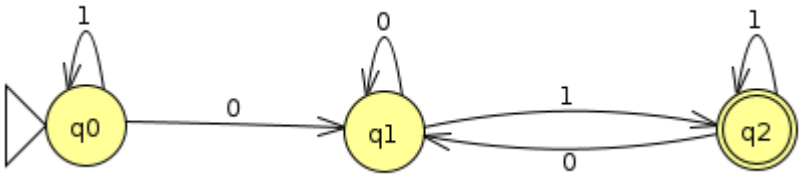
Per "tradurlo" in un DFA, dobbiamo fare una tabella di "conversione", come la seguente:

	0	1
→ {q0}	{q0,q1}	{q0}
{q1}	∅	{q2}*
{q2}*	∅	∅
{q0,q1}	{q0,q1}	{q0,q1,q2}*
{q0,q2}*	{q0,q1}	{q0,q1}
{q1,q2}*	∅	{q2}*
{q0,q1,q2}*	{q0,q1}	{q0,q1,q2}*

Ma se osserviamo, alcuni stati sono inutili, ovvero non contribuiscono in alcun modo alla creazione del DFA, e l'automa non passerà mai per quei stati. Se "puliamo" la tabella otteniamo:

	0	1
→ {q0}	{q0,q1}	{q0}
{q0,q1}	{q0,q1}	{q0,q1,q2}*
{q0,q1,q2}*	{q0,q1}	{q0,q1,q2}*

Che comprende gli stati che formano realmente il DFA. Se costruiamo il diagramma otteniamo:



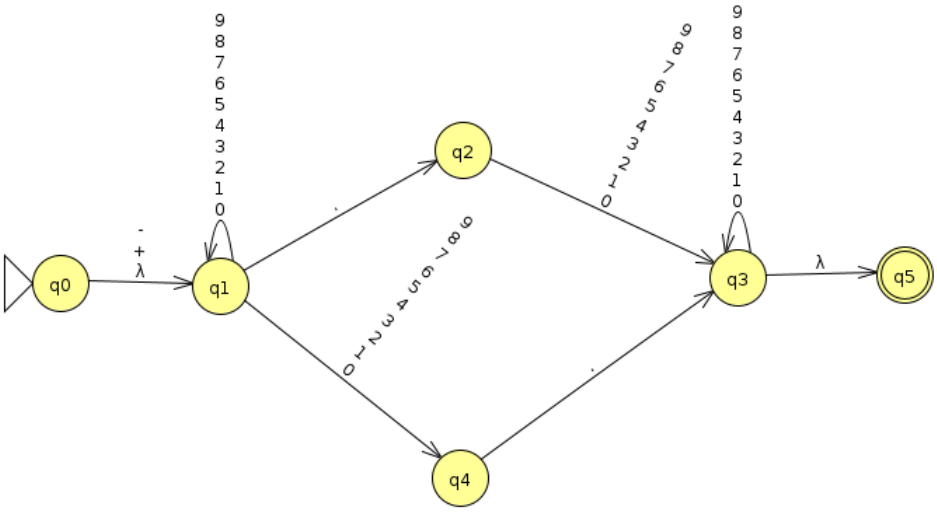
Che è effettivamente corrispondente agli stati e alle transizioni della seconda tabella.
Tale operazione è possibile grazie ad un teorema, che ora enunciamo.

Teorema di equivalenza
Un linguaggio L è accettato da un DFA ⇔ L è accettato da un NFA

2.3.2 ε-chiusure

Nel caso di ε-NFA, si utilizza una versione modificata della costruzione a sottoinsiemi, caratterizzata dalle ε-chiusure. Con ε-chiusura s'intende l'eliminazione delle ε-transizioni dagli stati che ne fanno uso.
A livello pratico. la costruzione a sottoinsiemi è uguale a quella usata partendo da un NFA, con la variante che lo stato iniziale del DFA è dato dalle ε-chiusure dello stato iniziale del ε-NFA.

Esempio Riprendiamo l'ε-NFA visto in precedenza:



Prima di effettuare la costruzione, dobbiamo effettuare le ε-chiusure:

ECLOSE(q0)={q0,q1}
ECLOSE(q1)={q1}
ECLOSE(q2)={q2}
ECLOSE(q3)={q3,q5}
ECLOSE(q4)={q4}
ECLOSE(q5)={q5}
Dopo questa operazione, possiamo applicare la costruzione. Riportiamo solo la tabella di transizione (per pigrizia):

	+,-	.	0,..,9
→ {q0,q1}	{q1}	{q2}	{q1,q4}
{q1}	∅	{q2}	{q1,q4}
{q2}	∅	∅	{q3,q5}*
{q1,q4}	∅	{q2,q3,q5}	{q1,q4}
{q2,q3,q5}*	∅	∅	{q2.q3,q5}
{q3,q5}*	∅	∅	{q3,q5}

Enunciamo anche il teorema per la conversione da ε-NFA a DFA:

Teorema
Un linguaggio L è accettato da un DFA $\Leftrightarrow$ L è accettato da un ε-NFA

2.4 Espressioni Regolari

Abbiamo visto come un linguaggio regolare possa essere espresso tramite un automa a stati finiti. Però non è l'unico modo per descriverlo. Infatti, possiamo esprimerlo anche tramite un'espressione regolare, che non è altro che un modo dichiarativo per descrivere un linguaggio regolare.
Gli ingredienti per costruire un'espressione regolare sono:
-un'insieme di costanti: ε indica la stringa vuota, $\emptyset$ indica il linguaggio vuoto e si usano i simboli presenti in un alfabeto Σ
-operatori: + viene usato per l'unione, . viene usato per la concatenazione e * viene usato per la chiusura di Kleene (* include anche ε)
-parentesi, per il raggruppamento

Con L(E) si indica il linguaggio creato dall'espressione regolare. Essendo una dichiarazione del linguaggio, possono esserci più espressioni corrette per un linguaggio.

Esempio Supponiamo di voler definire come espressione regolare il segeunte linguaggio:

L={w $\in$ {0,1}* : 0 e 1 alternati in w}

Possiamo descriverlo in due modi: o come unione delle possibilità, cioè unione di una alternanza 01, 10, 10101 o 01010, oppure come alternanza di 01, ma con la possibilità di avere un 1 davanti o uno 0.
Poniamolo come espressione, così che risulti più chiaro:

$$(01)^*+(10)^*+1(01)^*+0(10)^*$$
oppure
$$(\varepsilon+1)(01)^*(\varepsilon+0)$$

Si vede come, a livello pratico, non cambi nulla nel significato del linguaggio. Sta quindi a voi scegliere l'espressione che ritenete più opportuna.

Importante! Esattamente come nel linguaggio naturale, le espressioni regolari sono soggette alle precedenze. In particolare, in assenza di parentesi ogni operatore si lega al simbolo alla sua sinistra. Quindi, per fare un esempio, una espressione del tipo 01*+1 viene raggruppata come (0(1)*)+1, e quindi è differente da (01)*+1.

2.5 Equivalenza tra FA e RE

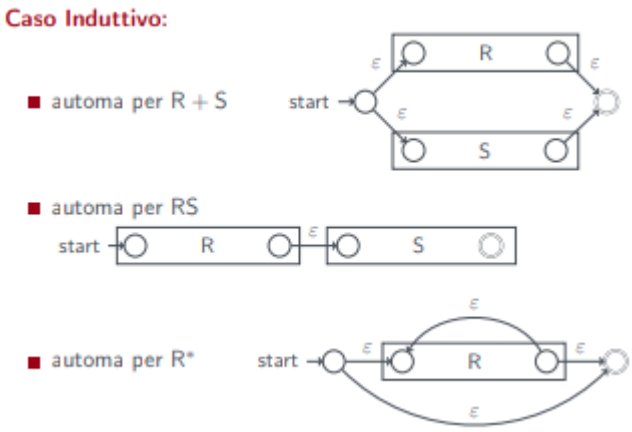
Abbiamo quindi visto finora gli automi a stati finiti (DFA, NFA, ε-NFA) e le espressioni regolari. Ma sappiamo anche che entrambi i modi rappresentano un linguaggio. Tramite questa conoscenza, possiamo passare da un automa (un DFA) ad una espressione regolare e viceversa (ma il quel caso si passerà ad un ε-NFA, come vediamo a breve).

2.5.1 Da RE a ε-NFA

Il primo modo è passare da una espressione regolare ad una ε-NFA. Usiamo il seguente teorema:

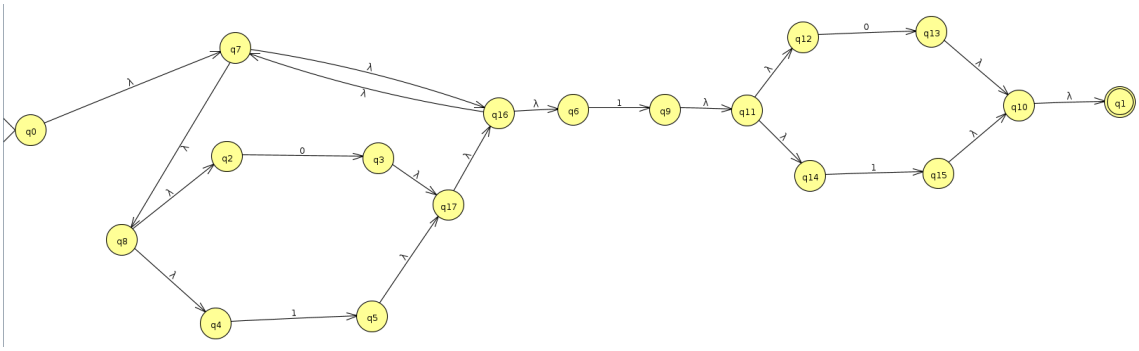
Teorema $\forall$ R espressione regolare possiamo costruire un ε-NFA A : L(A)=L(R)

Per far sì che il ε-NFA riconosca il linguaggio dell'espressione regolare, bisogna costruire un ε-NFA tale che abbia un solo stato finale, non abbia alcuna transizione entrante nello stato iniziale e non abbia nessuna transizione uscente dallo stato finale. Di seguito le regole per rappresentare unione, concatenazione e chiusura di Kleene.



Esempio Consideriamo l'espressione regolare (0+1)*1(0+1).
Notiamo che la prima unione è anche una chiusura di Kleene, quindi nel'automa per R*, il nostro R sarà l'automa che rappresenta l'unione 0+1. Seguirà una ε-transizione ad un automa che conti il primo carattere, il quale avrà un'altra

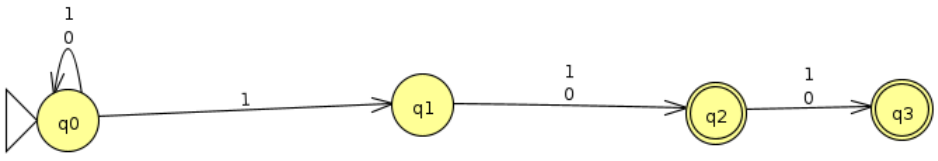
ε-transizione al secondo automa che rappresenta l'unione 0+1. Di seguito riportiamo il diagramma di transizione per l'ε-NFA appena descritto.



Notare che, fondamentalmente, si concatenano più pezzi di automi semplici tramite ε-transizioni.

2.5.2 Da DFA a RE

Se invece si ha un automa e si vuole passare ad una espressione regolare, si utilizza una tecnica chiamata **eliminazione di stati**.
In pratica, quando uno stato viene eliminato, vengono eliminati anche i cammini per quello stato. Al suo posto, si mettono nuove transizioni con espressioni regolari, che descrivono anche le transizioni perse. Si continua finchè non si ha un'unica transizione, contenente il linguaggio riconosciuto dall'automa.
Riassumendo, l'automa di partenza deve avere un unico stato finale (e in caso di più stati finali, se ne crea uno nuovo con ε-transizioni provenienti dai vecchi stati finali), s'iniziano a collassare le transizioni tra la stessa coppia di stati e si eliminano tutti gli stati tranne quello iniziale e quello finale. Se qr $\neq$ q0, ovvero stato finale e stato iniziale sono distinti, allora l'automa ha 4 transazioni: R(che rimane in q0),S(che da q0 va in qr),U(che rimane in qr) e T(che da qr ritorna in q0). L'espressione regolare che lo descrive è (R+SU*T)*SU*. Nel caso q0=qr, l'automa ha un unico stato R e la sua espressione regolare è R*.
Esempio Consideriamo il segeunte automa:



Guardando l'automa, possiamo già intuire che saremo nel caso in cui avremo q0 $\neq$ qr. La prima operazione è creare uno stato q4, che sarà il nostro stato

finale, essendoci 2 stati finali. Collassando le transizioni, otteniamo transizioni di unione, quindi nel nostro caso, in q0 per esempio, invece che una transizione per 0 ed una transizione per 1 otterremo un'unica transizione 0+1. Passando alla eliminazione degli stati, quando elimino q1 creo una transizione che va da q0 a q2, che in espressione regolare si esprime con 1(0+1), quando elimino q2 passo ad una transizione da q0 a q3 che vale 1(0+1)(0+1), mentre la transizione da q0 a q4, essendo 1(0+1) una transizione diretta, è 1(0+1)+1(0+1)(0+1).
Quindi l'espressione regolare relativa all'automa visto è (0+1)*(1(0+1)+1(0+1)(0+1)).

2.6 Linguaggi non regolari

Finora abbiamo visto tutti i possibili modi per mostrare un linguaggio regolare, dall'automa all'espressione regolare. Ma prendiamo in esempio il seguente linguaggio:

$$\text{L01}=\{0^n1^n : \text{n} \geq 0\}$$

Se un DFA con k stati legge 0^k, allora esiste un k+1 stato che è duplicato. Morale: se A legge 1^i e finisce in uno stato finale, accetta la parola 0^j1^i, che non è nel linguaggio, mentre se finisce in uno stato non finale, rifiuta la parola 0^i1^i, che è nel linguaggio, ingannando in ogni caso l'automa. Tali linguaggi sono definiti non regolari.
Il problema è che dire "Non riesco a costruirci un automa a stati finiti" non dice nulla, c'è bisogno di una prova oggettiva, che introduciamo con il teorema del Pumping Lemma.

Teorema Supponiamo che L sia un linguaggio regolare. Allora $\exists$ una lunghezza $\text{k} \geq 0$ tale che ogni parola $\text{w} \in \text{L}$ di lunghezza $|\text{w}|\geq\text{k}$ può essere spezzata in w=xyz, tale che $\text{y} \neq \varepsilon$, $|\text{xy}|\leq \text{k}$ e $\forall\ \text{i} \geq 0$, $\text{xy}^i\text{z} \in \text{L}$.

Quindi, per dimostrare che un linguaggio non è regolare, si usa il Pumping Lemma assumendo L regolare, portandosi in questo modo in una condizione di assurdità (non sempre funziona, ma nella maggior parte dei casi).

3 Grammatiche di Linguaggi liberi da contesto e PDA

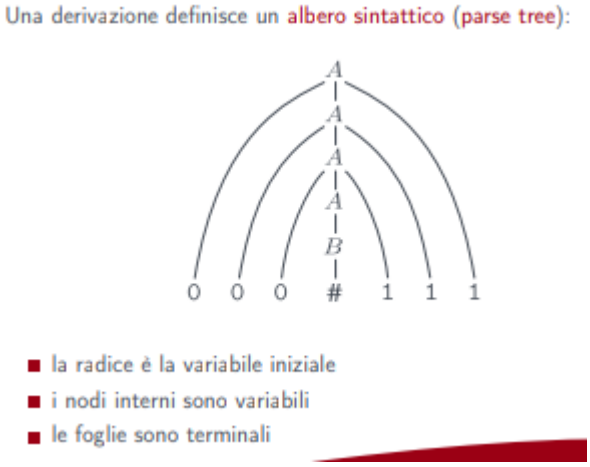
Passiamo ora ad una nuova parte, dove vedremo i inguaggi Context-Free.
Questi linguaggi, a differenza dei linguaggi regolari e non, non dipendono da un contesto, ma anche loro possono essere rappresentati in due modi, o come Grammatica Context-Free o come Automa a Pila, che vedremo successivamente.

3.1 Grammatiche Context-Free

Le Grammatiche Context-Free sono il primo modo per esprimere un linguaggio context-free. Esse utilizzano delle regole di sostituzione (generalmente hanno

forme tipo A ->B), e queste regole possono avere variabili (tra cui una iniziale) o terminali (ovvero simboli dell'alfabeto che si utilizza).
Esempio Vediamo la grammatica G1. Essa utilizza come alfabeto Σ={0,1,#} ed è composta da 3 regole:
-A->0A1
-A->B
-B->#
Un esempio di parola è 000#111.

Come si vede nell'esempio, si parte dalla variabile iniziale (nell'esempio, A), e si segue la regola, finchè non rimangono solo i terminali. Generalmente queste regole sono espresse anche come alberi sintattici, come il seguente:



Formalmente, una grammatica context-free è definita come (V,Σ,R,S), dove V è un insieme finito di variabili, Σ è un insieme finito di terminali disgiunto da V, R è un insieme di regole, S è la variabile iniziale (S $\in$ V).

Ma come si progettano le grammatiche context-free? Fondamentalmente non esistono processi meccanici, però ci sono delle tecniche che possono tornare utili. La prima riguarda l'unione di linguaggi più semplici, essendo molti linguaggi l'unione di linguaggi più semplici. L'idea è di costruire grammatiche separate per ogni componente, per poi unire le due grammatiche con una nuova regola iniziale.
La seconda, valida per i linguaggi regolari, consiste di prendere un DFA e trasformarlo in una grammatica context-free. L'idea sta nell'usare una variabile Ri per ogni stato qi, una regola ri ->aRj per ogni transizione e una regola Ri ->ε per ogni stato finale qi. Se q0 è lo stato iniziale, la variabile iniziale è R0.
La terza consiste nel vedere un linguaggio come formato da due sottostringhe collegate. L'idea in questo caso è usare regole nella forma R->uRv, che generano stringhe dove u corrisponde a v.

3.1.1 Proprietà delle grammatiche context-free

Diamo la definizione di albero sintattico: data una grammatica G=(V,Σ,R,S), un albero sintattico è un albero i cui nodi interni sono variabili di V, le foglie sono simboli terminali o ε e se un nodo interno è etichettato con A e i suoi figli sono, da sinistra a destra, X1,X2...Xk, allora A ->X1X2...Xk è una regola di G.
Attenzione che un albero può generare una stringa in due modi diversi! Infatti, una grammatica genera ambiguamente una stringa se esistono due alberi sintattici diversi per quella stringa. Quindi, generalmente si effettua la derivazione a sinistra per evitare ambiguità.
Diamo una definizione: una grammatica è definita ambigua se genera almeno una stringa ambiguamente.
Inoltre esistono i linguaggi inerentemente ambigui, ovvero linguaggi che sono generati solo da grammatiche ambigue.
Ora, una grammatica context-free è migliore se semplificata, ed una delle forme più semplici è la forma normale di Chomsky, che rende ogni regola della forma A->BC, A->a, dove a è un terminale, B e C non posssono essere la variabile iniziale.
Inoltre, può esserci la regola S ->ε per la variabile iniziale S. Per trasformare una grammatica context-free in forma normale di Chomsky, si aggiunge una nuova variabile, si eliminano le ε-regole, si eliminano le regole unitarie A->B e si trasformano le regole rimaste nella forma corretta appena vista.
Segue un esempio, che non riporto a mano:

Trasformiamo la grammatica G_6 in forma normale di Chomsky:

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

Per trasformare $G = (V, \Sigma, R, S)$ in Forma Normale di Chomsky

1. aggiungiamo una nuova variabile iniziale $S_0 \notin V$ e la regola

$$S_0 \to S$$

In questo modo garantiamo che la variabile iniziale non compare mai sul lato destro di una regola

2. Eliminiamo le ε-regole $A \to \varepsilon$:
   - se $A \to \varepsilon$ è una regola dove A non è la variabile iniziale
   - per ogni regola del tipo $R \to uAv$, aggiungiamo la regola

     $$R \to uv$$

   - attenzione: nel caso di più occorrenze di A, consideriamo tutti i casi: per le regole come $R \to uAvAw$, aggiungiamo

     $$R \to uvAw \mid uAvw \mid uvw$$

   - nel caso di regole $R \to A$ aggiungiamo $R \to \varepsilon$ solo se non abbiamo già eliminato $R \to \varepsilon$
   - Ripeti finché non hai eliminato tutte le ε-regole

3. Eliminiamo le regole unitarie $A \to B$:
   - se $A \to B$ è una regola unitaria
   - per ogni regola del tipo $B \to u$, aggiungiamo la regola

     $$A \to u$$

     a meno che $A \to u$ non sia una regola unitaria eliminata in precedenza
   - Ripeti finché non hai eliminato tutte le regole unitarie

4. Trasformiamo le regole rimaste nella forma corretta:
   - se $A \to u_1 u_2 \dots u_k$ è una regola tale che:
     - ogni u_i è una variabile o un terminale
     - $k \geq 3$
   - sostituisci la regola con la catena di regole

     $$A \to u_1 A_1, \quad A_1 \to u_2 A_2, \quad A_2 \to u_3 A_3, \quad \dots \quad A_{k-2} \to u_{k-1} u_k$$

   - rimpiazza ogni terminale u_i sul lato destro di una regola con una nuova variabile U_i, e aggiungi la regola

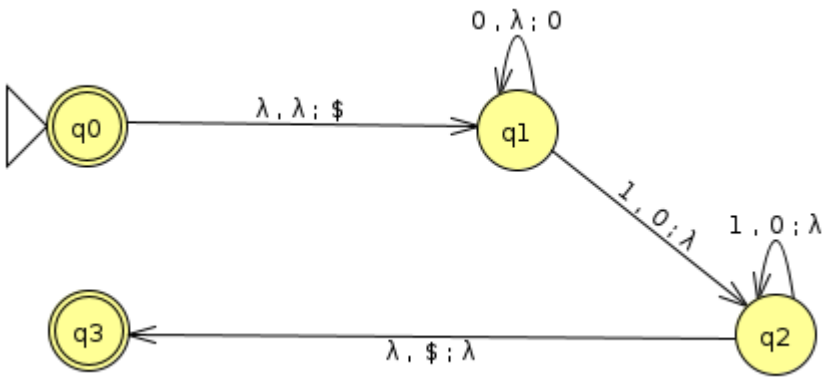
     $$U_i \to u_i$$

   - ripeti per ogni regola non corretta

3.2 PDA

Gli automi a pila (dall'Inglese Pushdown Automation, abbreviato in PDA) sono degli automi che, oltre ad utilizzare gli stati, utilizzano anche una pila. La funzione di transizione stabilisce quali possono essere gli stati successivi e i simboli da scrivere nella pila, dati stato corrente, simbolo in input e il simbolo in cima alla pila. Inoltre, essendo la pila un tipo di memoria LIFO, permette all'automa di avere memoria infinita ad accesso limitato.
Formalmente, un PDA viene definito come P=(Q,Σ,Γ,δ,q0,F), dove Q è un insieme finito di stati, Σ è l'alfabeto di input, Γ è l'alfabeto della pila, δ è la funzione di transizione, q0 è lo stato iniziale e F è l'insieme di stati accettanti.

Esempio Creiamo un PDA per il linguaggio L=$\{0^n1^n : n \geq 0\}$. Formalmente si definisce con P=({q0,q1,q2,q3},{0,1},{0,$},$\delta$,q0,{q0,q3}). Segue diagramma di transizione:



L'esempio appena visto accetta una parola per stato finale, ma un PDA può accettare anche per pila vuota. Infatti, un PDA accetta la parola w per pila vuota se $\exists$ una computazione che consuma tutto l'input e termina con la pila vuota. Inoltre, $\forall$ linguaggio accettato da un PDA per stato finale $\exists$ un PDA che accetta per pila vuota, e viceversa.

3.3 Da Grammatiche Context-Free a PDA

Enunciamo subito un teorema, che ci farà notare una somiglianza tra gli automi a stati finiti e i PDA:
Teorema Un linguaggio è context-free $\Leftrightarrow$ $\exists$ un PDA che lo riconosce
L'idea è che P simula i passi di derivazione in G, e P accetta w se esiste una derivazione di w in G.
La dimostrazione si fa creando un PDA P=({Qstart,Qloop,Qend},Σ,$\Sigma \cup V \cup$ {$},Qstart,Qend) e funzione di transizione ε,A->u per la regola A->u, a,a->ε per il terminale a (dove i vari stati sono presi dalla grammatica).

3.4 Da PDA a Grammatiche Context-Free

Per questo passaggio, andremo a fare una grammatica che fa un po' di più rispetto al PDA, ovvero generiamo una variabile Apq per ogni coppia di stati p,q di P, dove Apq genera ogni stringa che porta da p con pila vuota a q con pila vuota.
Occorre però che il PDA da trasformare rispetti 3 condizioni, ovvero ha un unico stato accettante qf, svuota la pila prima di accettare, e ogni transizione effettua il push o il pop, ma non entrambe le azioni. In particolare, quest'ultima azione prevede due casi, la prima riguarda l'inserimento all'inizio e la rimozione alla fine, la seconda vede un inserimento all'inizio ma una rimozione prima della fine

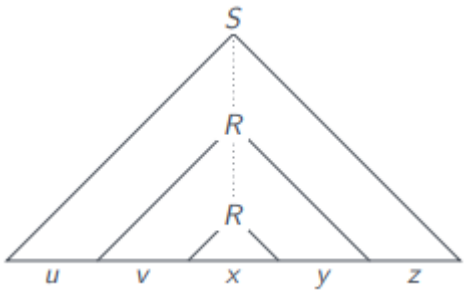
(in r, quindi invece che solo Apq, si avrà sia Apr che Arq).
Formalmente, dato un PDA P=(Q,Σ,Γ,δ,q0,{qf}), generiamo una grammatica G=(V,Σ,R,Aq0-qf) tale che
V={Apq : p,q ∈ Q},
∀ p,q,r,s ∈ Q, u ∈ Γ e a,b ∈ Σε, se δ(p,a,ε) contiene (r,u) e δ(s,b,u) contiene (q,ε), aggiungiamo la regola Apq->aArsb,
∀ p,q,r ∈ Q, aggiungiamo la regola Apq->AprArq,
∀ p ∈ Q, aggiungiamo la regola App->ε.

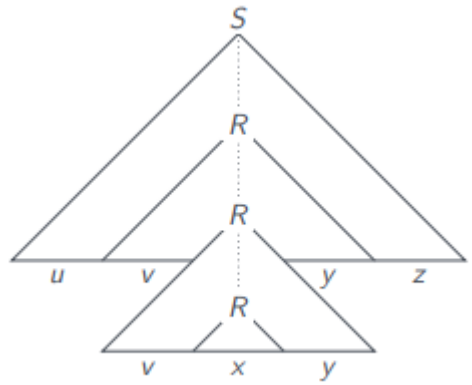
3.5 Linguaggi non context-free

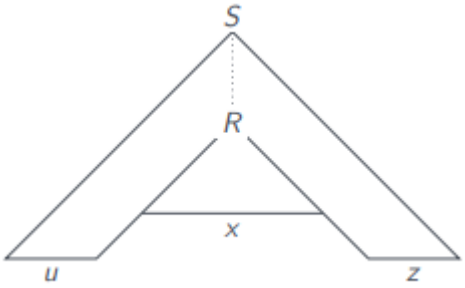
Abbiamo visto, per i linguaggi regolari, che il Pumping Lemma è il modo per dimostrarne la regolarità. Similmente, esiste il Pumping Lemma per linguaggi context-free, di cui ora diamo l'enunciato.

Teorema Sia L un linguaggio context-free. Allora ∃ una lunghezza k ≥ 0 tale che ∀ w ∈ L di lunghezza |w|≥ k può essere spezzata in w=uvxyz tale che |vy|>0, |vxy|≤ k, ∀ i ≥ 0, uv^ixy^iz ∈ L.

L'unica differenza, rispetto ai linguaggi regolari, è che in questo caso la parola viene divisa in 5 parti, e le parti "pompate" sono 2. Questo perché, per i linguaggi context-free, si replica il sottoalbero (ripetuto) di R rimanendo nel lingauggio (a parità di w "molto lunga" e albero sintattico relativo "molto alto"). Di seguito una dimostrazione grafica.





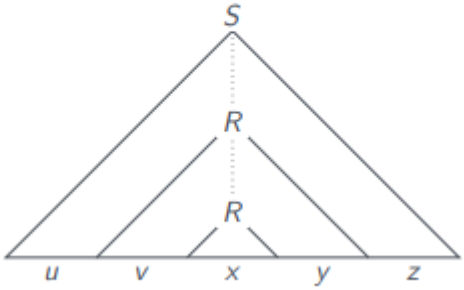


Includiamo inoltre proprietà degli alberi e dimostrazione, sempre in forma grafica.

- Sia G una grammatica per il linguaggio L
- Sia b il numero massimo di simboli nel lato destro delle regole
- In un albero sintattico, ogni nodo avrà al massimo b figli:
  - al più b foglie per un albero di altezza 1
  - al più b^2 foglie per un albero di altezza 2
  - al più b^h foglie per un albero di altezza h
- Un albero di altezza h genera una stringa di lunghezza minore o uguale a b^h
- Viceversa: una stringa di lunghezza maggiore o uguale a $b^h + 1$ richiede un albero sintattico di altezza maggiore di h

- Prendiamo come lunghezza del pumping $k = b^{|V|+1}$ ($|V|$ numero di variabili in G)
- Presa una stringa $w \in L$, se $|w| \geq k$ allora ogni albero sintattico per w deve avere altezza maggiore o uguale a $|V|+1$
- Prendiamo più piccolo albero sintattico τ per w
- Prendiamo il cammino più lungo in τ: sarà di lunghezza $\geq |V|+2$
- Quindi ci sono almeno $|V|+1$ variabili nel cammino
- Qualche variabile R si ripete nel cammino
- Scegliamo R in modo che si ripeta nei $|V|+1$ nodi più in basso nel cammino

- La variabile R ci dà la suddivisione $w = uvxyz$:
  - il sottoalbero più in alto genera vxy
  - il sottoalbero più in basso genera x



- possiamo sostituire i due sottoalberi tra loro, ottenendo di nuovo un albero sintattico corretto:
  - sostituire ripetutamente il più piccolo con il più grande ci dà uv^ixy^iz per ogni $i > 1$
  - sostituire il più grande con il più piccolo ci dà uxz
- in tutti i casi la nuova stringa appartiene a L
- rimane da dimostrare:
  - che $|vy| > 0$ (v e y non possono essere entrambi la parola vuota)
  - che $|vxy| \leq k$

- supponiamo che $v = \varepsilon$ e $y = \varepsilon$
- allora se sostituiamo il sottoalbero più grande con il più piccolo otteniamo di nuovo w:

$$w = uvxyz = u\varepsilon x\varepsilon z = uxz$$

- Assurdo: avevamo scelto τ come l'albero sintattico più piccolo!

- l'occorrenza più in alto di R genera vxy
- avevamo scelto le occorrenze tra i $|V| + 1$ nodi più in basso
- il sottoalbero che genera vxy è alto al massimo $|V| + 1$
- quindi può generare una stringa di lunghezza al più $b^{|V|+1} = k$

FINE!