# Riassunto LFA

Il linguaggio è un insieme di frasi per la comunicazione tra entità diverse, esistono i linguaggi naturali per comunicare tra le persone, i linguaggi di programmazione per la comunicazione tra uomo e macchina e il codice morse, per conoscere un linguaggio abbiamo bisogno dei vocaboli e la sintassi, cioè le regole per costruire frasi o programmi.

In questo corso con formali intendiamo la considerazione di questi concetti in maniera precisa utilizzando la matematica, il vantaggio è la possibilità di utilizzare dispositivi elettronici per automatizzare le operazioni, per esempio i compilatori e gli interpreti che però non sono in grado di capire il linguaggio naturale come l'italiano.

Per specificare un linguaggio ho bisogno di un sistema formale, ne esistono di due tipi:

- Sistemi generativi costituiti da vocaboli più regole che formano grammatiche
- Sistemi riconoscitivi che sono macchine a stati finiti dette automi

Possiamo pensare ad un linguaggio formale come un problema di decisione la cui risposta è si/no, questo problema è risolto da un programma

## Concetti centrali nella teoria dei linguaggi formali

**Alfabeto**: rappresenta un insieme finito di simboli $\Sigma = \{a_1, a_2, \dots, a_k\}$

**Parola**: una sequenza finita di simboli appartenenti ad un dato alfabeto

Esempio: $\Sigma = \{a\}$ parole = a, aa, aaa, … $aaa_k$ $\Sigma = \{0,1\}$ parole = 0, 1, 000, 01010, …

Esiste una parola non contenente alcun simbolo detta **parola vuota**, per convenzione viene definita con ε (epsilon)

**Lunghezza di una parola**: è definita dal numero di simboli, distinti per posizione, di cui è composta la parola e viene indicata con |parola| o l(parola), la lunghezza di ε è pari a zero.

Un alfabeto può includere la parola vuota o no, possiamo dichiarare l'inclusione/esclusione della parola vuota tramite i seguenti simboli dopo sigma (Σ):

- $\Sigma^*$ = insieme delle parole su Σ inclusa la parola vuota ε
- $\Sigma^+$ = insieme delle parole su Σ esclusa la parola vuota ε

**Prodotto di giustapposizione**: Il prodotto di giustapposizione è una operazione binaria su Σ* identificata con il simbolo "·", così definita:

$\forall x, y \in \Sigma^* \mid |x|, |y| \in N \qquad \exists z \mid z = x \cdot y$

Date due parole x,y, costruite sull'alfabeto Σ e le cui lunghezze sono due misure finite, possiamo asserire una terza parola z che sia il risultato del prodotto di giustapposizione tra x e y

**Le proprietà del prodotto**:

- Chiuso rispetto a Σ*
- È associativo $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Dotato di elemento neutro, $e \cdot x = x = x \cdot e$, solitamente "e" è la parola vuota ε

La lunghezza della parola risultato di un prodotto è costituita dalla somma delle lunghezze delle parole che costituiscono il prodotto, esempio: $|x \cdot y| = |x| + |y|$

## Monoide libero generato da Σ

Per via delle proprietà di cui gode l'operazione prodotto di giustapposizione possiamo definire il seguente monoide (Σ*, ·, ε) dove:

- Σ* = insieme infinito di parole costruibili su Σ
- · = operazione binaria associativa
- E = elemento neutro rispetto all'operazione

Il nostro monoide (Σ*, ·, ε) non gode della **proprietà di commutatività**

Esempio: x = la e y = go, x · y = lago mentre y · x = gola

## La scomposizione di parole

Prendendo in considerazione la parola "incatenare", possiamo identificare un prefisso (in), un suffisso (are) e un fattore "catena", il suffisso, prefisso e fattore non sono univoci.

**Definizione formale di prefisso**: la parola "x" è prefisso della parola "w" quando esiste w = x · y, dove y è una parola dell'alfabeto Σ*

**Definizione formale di suffisso**: la parola "x" è suffisso della parola "w" quando esiste w = y · x, dove y è una parola dell'alfabeto Σ*

**Definizione formale di fattore**: la parola "x" è fattore della parola "w" quando esiste w = y · x · z, dove y e z sono parole dell'alfabeto Σ*

Nel caso di prefisso e suffisso la parola "y" può essere la parola vuota ε, in questo caso tutta la parola è considerata prefisso/suffisso

Esiste due casi particolari in cui la parola è contemporaneamente prefisso, suffisso e fattore, cioè, data w appartenente a Σ* le parole "w" ed "ε" sono contemporaneamente prefisso, suffisso e fattore di w

**Definizione di un linguaggio formale**: un linguaggio L è un qualunque sottoinsieme di Σ*: L ⊆ Σ*, questo sottoinsieme può anche essere tutto Σ* in questo caso L = Σ*

Esistono quattro **tipologie di linguaggi**:

- **Linguaggio vuoto**: viene indicato con ∅, la sua cardinalità è zero ed è un linguaggio primo di qualsiasi elemento
- **Linguaggio della parola vuota**: è il linguaggio formato solamente da ε, la sua cardinalità è pari ad uno
- **Linguaggi finiti**: sono linguaggi composti da un numero finito di elementi, la sua cardinalità è finita
- **Linguaggi infiniti**: sono linguaggi composti da un numero di elementi innumerabile o infinito, la sua cardinalità è infinita

### Esempi di linguaggi finiti

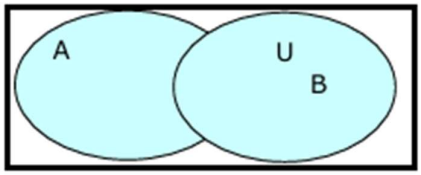
- L = ∅, la sua cardinalità è pari a zero, |L| = 0,
- L = {ε}, la sua cardinalità è pari ad uno, |L| = 1
- Σ = {a}, $L_n$ = {ε, a, aa, aaa, …, $a^n$}, la sua cardinalità è pari ad n+1, $|L_n|$ = n+1

### Esempi di linguaggi infiniti

- Σ = {a}, L = {ε, a, aa, aaa, …, $a^n$, …} = Σ* = {a}*
  {a}* = {$a^n$ | n ∈ N} dove $a^0$ è uguale a ε

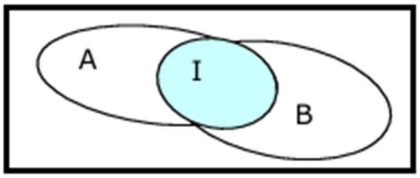
# Operazioni sui linguaggi

Dato che i linguaggi sono insiemi, in particolare sottoinsiemi di Σ*, possiamo definire delle operazioni insiemistiche che si possono applicare ai linguaggi.



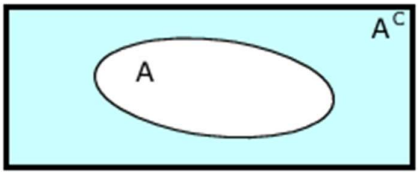

**L'operazione Unione**: presi due linguaggi A e B su un certo alfabeto Σ*, possiamo definire unione di A e B un terzo linguaggio che contiene le parole che appartengono almeno ad uno degli insiemi A e B

Formalmente viene definito così: $A \cup B = \{w \in \Sigma^* \mid w \in A \text{ oppure } w \in B\}$




**L'operazione Intersezione**: presi due linguaggi A e B su un certo alfabeto Σ*, possiamo definire intersezione di A e B un terzo linguaggio che contiene le parole che appartengono ad entrambi i linguaggi A e B

Formalmente viene definito così: $A \cap B = \{w \in \Sigma^* \mid w \in A \text{ e } w \in B\}$




**L'operazione Complemento**: preso un linguaggio A su un certo alfabeto Σ*, possiamo definire complemento di A un secondo linguaggio che contiene tutte le parole di Σ* che non appartengono all'alfabeto A, il complemento di un linguaggio finito è un linguaggio infinito

Formalmente viene definito così: $A^C = \{w \in \Sigma^* \mid w \notin A\}$

Possiamo notare che l'intersezione tra A e $A^C$ è formata da un linguaggio vuoto.

È possibile definire anche altre operazione tipiche sui linguaggi

**L'operazione Prodotto**: presi due linguaggi A e B su un certo alfabeto Σ*, definiamo prodotto un terzo linguaggio P che contiene le parole "w" tali che il prefisso di "w" è una parola del linguaggio di A e la restante parte di "w" è una parola del linguaggio di B. il prodotto tra due linguaggi finiti è un linguaggio finito.
il prodotto tra linguaggi non è commutativo $AB \neq BA$, con l'unica eccezione che se consideriamo un alfabeto unitario, cioè con cardinalità pari ad uno, e definiamo due linguaggi su di esso, allora il prodotto di questi due linguaggi è commutativo.

**L'operazione Potenza**: preso un linguaggio A su un certo alfabeto Σ*, la potenza k-esima di A consiste nell'eseguire il prodotto di A per sé stesso k volte, ossia $A^k = A \cdot A \cdot A \rightarrow$ k volte. Il risultato dell'operazione potenza sono le parole che hanno come lunghezza l'esponente indicato.

**La chiusura di Kleene**: questa operazione è un insieme dell'operazione di unione e l'operazione di potenza. preso un linguaggio L, la chiusura di Kleene si indica con $L^*$ o $L^+$ a seconda se:

**$L^*$** consiste nell'effettuare l'unione di tutti i linguaggi $L^i$ con i che va da zero a infinito ossia:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^k \cup \ldots \cup L^\infty = \bigcup_{i=0}^{\infty} L^i$$

**$L^+$** consiste nell'effettuare l'unione di tutti i linguaggi $L^i$ con i che va da uno a infinito, ossia escludendo $L^0$:

$$L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots \cup L^k \cup \ldots \cup L^\infty = \bigcup_{i=1}^{\infty} L^i$$

**Osservazione**: $L^* = L^+ \cup L^0$ $\quad L^+ = L^* \setminus L^0$ $\quad L^0 = \varepsilon$

Possiamo notare che **$L^+$ è diverso da $L^* \setminus \{\varepsilon\}$** in quanto in $L^+$ potrebbe esserci la parola vuota, derivante dal fatto che, se $L = \{\varepsilon, a\}$, ossia un linguaggio formato dalla parola vuota e "a", tutti gli $L^n$ conterranno ε, pertanto dire che $L^+ = L^* \setminus \{\varepsilon\}$ vuol dire rimuovere tutte le parole vuote e non escludere solo $L^0$

## Il Codice

**Codice**: Possiamo definire un linguaggio L un codice se ogni parola appartenente ad $L^+$ può essere decomposta in maniera univoca come prodotto di parole di L.
il codice L possiede dunque l'importante proprietà di univoca decifrabilità, infatti, data una parola $w \in L^+$, dove L è un codice, esiste uno e un solo modo di ottenere w come prodotto di parole di L. le parole in L sono l'alfabeto del codice.
$L^+$ costituisce l'insieme di tutti i messaggi che si possono scrivere con il codice.
i linguaggi codici non contengono ε. I linguaggi codice sono linguaggi finiti.

**Codici prefissi**: un linguaggio L in cui ogni parola di L non è prefisso di nessun'altra parola di L è chiaramente un codice, in cui ulteriormente ogni parola $w \in L^+$ può essere decodificata in linea leggendo da destra a sinistra, tali codici prendono il nome di codici prefissi o istantanei.

ESEMPIO DI CODICE PREFISSO

L = {aa, aba, baa}
Prendiamo una parola di $L^+$ e vediamo se è decomponibile in più modi: x = "baaabaaabaa", $x \in L^+$

decomposizione: baa · aba · aa · baa $\in L^+$

Non sono possibili altre decomposizioni di x in L, quindi L è un codice, altresì L è un "codice prefisso" in quanto ciascuna parola non è prefisso di nessun'altra parola di L, ossia

- "aa" non è prefisso ne di "aba" ne di "baa"
- "aba" non è prefisso ne di "aa" ne di "baa"
- "baa" non è prefisso ne di "aa" ne di "aba"

ESEMPIO DI CODICE NON PREFISSO

C = {0, 01}
Prendiamo una parola di $C^+$ e vediamo se è decomponibile in più modi: y = "00010101", $y \in C^+$

decomposizione: 0 · 0 · 01 · 01 · 01 $\in C^+$

Non sono possibili altre decomposizioni in C quindi è un codice.
la parola "0" $\in$ C è prefisso della parola "01" $\in$ C, quindi, nonostante sia un codice, non è un "codice prefisso"

ESEMPIO DI NON CODICE

L = {bab, aba, ab}
Prendiamo una parola di $L^+$ e vediamo se è decomponibile in più modi: k = "ababab", $k \in L^+$

decomposizione: ab · ab ·ab $\in L^+$

aba · bab $\in L^+$

sono possibili più decomposizioni quindi L non è un codice

## Definizione di linguaggi

Un linguaggio L è un insieme di parole su un dato alfabeto Σ.

Le modalità con cui possiamo definire un linguaggio sono due a seconda della sua cardinalità.

I linguaggi finiti li possiamo definire in modo **estensivo**, cioè tramite l'elencazione di tutte le parole che compongono il linguaggio, $L = \{w_1, w_2, \ldots, w_k\}$

I linguaggi infiniti, ma anche quelli finiti, li possiamo definire in modo **intensivo**, cioè tramite una proprietà che risulta vera su tutte e sole le parole del linguaggio, $L = \{w \in \Sigma^* \mid P(w) = true\}$

I linguaggi infiniti che ammettono una descrizione, non è detto che tutti i linguaggi ammettano una descrizione mediante un sistema, possono essere descritti con due sistemi diversi:

- **Sistemi riconoscitivi**: data una parola permettono di stabilire se essa appartiene o meno al linguaggio. I linguaggi che ammettono un sistema riconoscitivo sono detti linguaggi ricorsivi o decidibili, i più comuni sistemi riconoscitivi sono gli automi a stati finiti e automi a pila, capaci di

riconoscere due sottoclassi di linguaggi ricorsivi, detti rispettivamente linguaggi regolari e linguaggi liberi da contesto

- **Sistemi generativi**: permettono di costruire parole del linguaggio, i più comuni sistemi generativi sono le grammatiche.

## Teoria della calcolabilità – concetti base

**Procedura**: la procedura è una sequenza finita di istruzioni che possono essere eseguite automaticamente e che possono portare ad un risultato se il programma è progettato per fornirlo

**Algoritmo**: l'algoritmo è una procedura che termina sempre, qualsiasi sia l'input fornitogli in ingresso, quindi un algoritmo non prevede loop infiniti o break.

**Programma**: un programma è lo strumento mediante il quale vengono definite le procedure e gli algoritmi.

Possiamo descrivere il programma sotto due aspetti:

- **Aspetto sintattico**: un programma è una parola w $\in \{0,1\}^*$, ossia una sequenza di bit, questa connotazione fornisce la descrizione del programma w mediante la sua codifica ASCII, un programma viene quindi inteso sintatticamente come un insieme di parole su $\Sigma^*$, dove $\Sigma^*$ è l'alfabeto binario formato solo da "0" e "1".
- **Aspetto semantico**: un programma "w" è una procedura che, se opportunamente inizializzata genere una sequenza di passi di calcolo che possono terminare fornendo un risultato, indichiamo con $F_w$ la semantica del programma w. Se "w" sintatticamente è una parola binaria che corrisponde al codice ASCII di un programma e "x" è sintatticamente una parola binaria che viene passata come dato input al programma, indicheremo con $F_w(x)$ il risultato dell'esecuzione del programma w calcolato sull'input x.

**Osservazione**: l'input "x" è una parola binaria, il nostro programma "w" è una parola binaria, quindi anche quest'ultimo può essere passato in input, possiamo vedere un compilatore come esempio.

### Limitazione sui programmi durante la loro esecuzione

**Non terminano**: sono programmi che vanno in loop, il risultato dell'esecuzione di questa tipologia viene indicato con $F_w(x)\uparrow$, il quale indica che il programma w su input x non termina la sua esecuzione

**Terminano in 0/1**: sono programmi che terminano la loro esecuzione fornendo come output solo due tipologie di valori 0/1, il risultato dell'esecuzione di questa tipologia di programma viene indicato con $F_w(x)\downarrow$, il quale indica che il programma w su input x termina.
Se il risultato dell'esecuzione è pari a 0 scriveremo $F_w(x) = 0$, mentre se pari a 1 scriveremo $F_w(x) = 1$.

**La funzione caratteristica**: dato un linguaggio L è la funzione che ci premette di individuare se la parola fornita in ingresso appartiene o meno al linguaggio. Viene indicata con il simbolo $X_L(x)$, si legge "ki di L", prende in input un valore "x" e restituisce in output 0/1.

$$X_L(X) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{se } x \notin L \end{cases}$$

## La classificazione dei linguaggi

**Linguaggio ricorsivo**: un linguaggio L si dice ricorsivo se esiste un algoritmo implementato dal programma w tale che passandogli in input un dato x $\in \{0,1\}^*$ restituisce:

- $F_w(x) = 1$ se e solo se x appartiene ad L
- $F_w(x) = 0$ se e solo se x non appartiene ad L

Se L è un linguaggio ricorsivo allora $P_L$, il problema associato ad L, è detto decidibile e L ammette un sistema riconoscitivo.

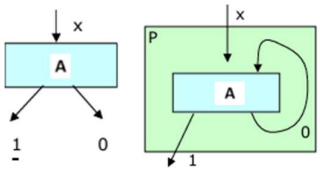
**Linguaggio ricorsivamente enumerabile**: un linguaggio L si dice ricorsivamente enumerabile se esiste una procedura implementata dal programma w tale che passandogli in input un dato x ∈ {0,1}* restituisce:

- $F_w(x) = 1$ se e solo se x appartiene ad L
- $F_w(x) = \uparrow$ se e solo se x non appartiene ad L (loop)

Se L è un linguaggio ricorsivamente enumerabile allora $P_L$, il problema associato ad L, è detto semidecidibile in quanto il programma non sempre termina e quindi non posso definire con certezza se x appartiene o non appartiene ad L, e L ammette un sistema generativo.


**Teorema**: Se L è un linguaggio ricorsivo, L è anche un linguaggio ricorsivamente enumerabile

**Dimostrazione:**




Per ipotesi visto che L è ricorsivo esiste un algoritmo A per L che restituisce 1 se x appartiene ad L o 0 se non appartiene, dobbiamo dimostrare l'esistenza di una procedura P che restituisce 1 se x appartiene ad L e va in loop se x non appartiene ad L.

Possiamo definire la nostra procedura nel seguente modo, assegniamo Y al nostro algoritmo A(x) che restituirà 0/1, se restituisce 1 diciamo alla nostra procedura di portare in uscita 1 altrimenti se Y = 0 rimaniamo in una situazione di loop.

```
Procedura P(x)
{  y = A(x);  // A esiste perché L è ricorsivo
   if (y=1) then return(1);
   loop;
}
```

**Dimostrazione di correttezza:**

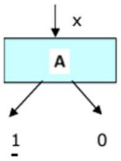
$x \in L \rightarrow A(x) = 1 \rightarrow Y = 1 \rightarrow P(x) = 1$

$x \notin L \rightarrow A(x) = 0 \rightarrow Y = 0 \rightarrow P(x)\uparrow$ (loop)

Da questa dimostrazione possiamo dire che L è ricorsivamente enumerabile.

Il contrario, cioè un linguaggio ricorsivamente enumerabile è ricorsivo, non è possibile in quanto non è sempre possibile trasformare una procedura in un algoritmo, in quanto una procedura ammette situazione di loop mentre un algoritmo termina sempre.

**La proprietà dei linguaggi ricorsivi**



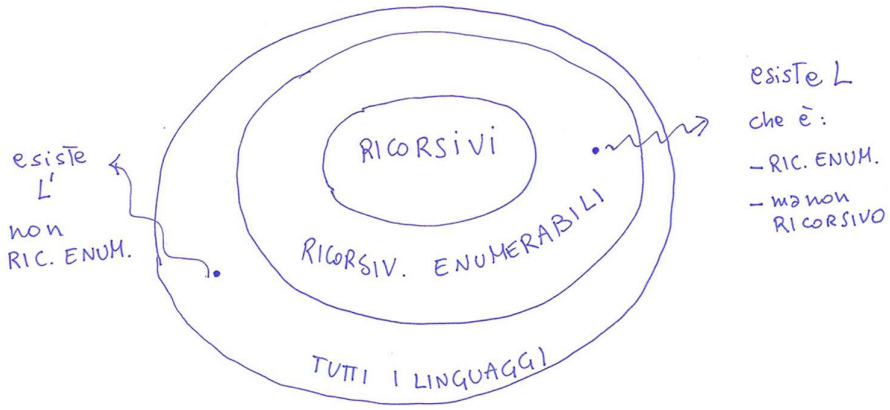

**Teorema**: Se L è un linguaggio ricorsivo allora possiamo dire che anche il suo $L^C$, linguaggio complementare, è ricorsivo

**Dimostrazione**: supponiamo di avere il seguente algoritmo A per un linguaggio L, che restituisce 1 se x appartiene e 0 se non appartiene

Dobbiamo costruire un algoritmo A' che preso un input x restituisce 1 se x non appartiene a L quindi appartiene a $L^C$ e restituisce 0 se x appartiene a L quindi non appartiene a $L^C$.

```
ALGORITMO A'(x)
{  Y = A(x);
   IF (Y=1) THEN RETURN (0);
   ELSE IF (Y=0) THEN RETURN (1);
}
```

## Risultati importanti della teoria della calcolabilità




Possiamo dire che esiste L che è ricorsivamente enumerabile ma non ricorsivo che quindi ammette una procedura e non è possibile fare meglio, ma esistono anche linguaggi che non sono ricorsivamente enumerabili che quindi non ammettono neanche una soluzione parziale.

**Conseguenze** (**Teorema di Rice**, 1950):

- Non è possibile verificare per via automatica la correttezza semantica dei programmi
- Non è possibile verificare per via automatica l'equivalente di due programmi, cioè:
  Dati "v" e "w" verificare che per ogni x ∈ {0,1}* esistono due programmi che prendendo x in input restituiscano lo stesso risultato $F_v(x) = F_w(x)$
- Non è possibile verificare per via automatica la terminazione dei programmi, cioè:
  Problema dell'arresto:
  Input: programma "w", dato "x"
  Output: verificare se $F_W(x)\downarrow$ va in loop
  Questo problema è indecidibile.
- Esistono teoremi matematici non dimostrabili – risultato di incompletezza di Goedel

### L'interprete

L'interprete viene indicato con "u" ed è un programma che prende in input un altro programma "w" ed un dato "x" e fornisce in output il risultato che ottiene simulando l'esecuzione del programma w sul dato x, cioè, fornisce in output $F_w(x)$ se w è un programma, in caso contrario il risultato è indeterminato.

$$F_u(w\$x) = \begin{cases} F_w(x) & \text{se } w \text{ è programma} \\ \perp & \text{altrimenti} \end{cases}$$

$w\$x \in \{0,1\}^*$

### Il linguaggio dell'arresto ristretto

**Definizione di D:**

$D = \{x \in \{0,1\}^* \mid F_u(x\$x)\downarrow\}$

Il linguaggio D è l'insieme di stringhe binarie "x" tali per cui l'interprete che prende in input, sia come dato che come programma, quindi il programma "x" eseguito dall'interprete con input "x" deve terminare.

**Definizione di $D^C$:**

$D^C = \{x \in \{0,1\}^* \mid F_u(x\$x)\uparrow\}$

Il linguaggio $D^C$ è l'insieme di stringhe binarie "x" tali per cui l'interprete che prende in input, sia come dato che come programma, quindi il programma "x" eseguito dall'interprete con input "x" va in loop.

**Teorema:**

Andiamo a definire l'esistenza di un linguaggio D così definito:

1) D è ricorsivamente enumerabile
2) D non è ricorsivo
3) $D^C$ non è ricorsivamente enumerabile

**La dimostrazione del punto 1 – D è ricorsivamente enumerabile:** $D = \{x \in \{0,1\}^* \mid F_u(x\$x)\downarrow\}$

Andiamo a dimostrare che D è un linguaggio ricorsivamente enumerabile, per effettuare questa dimostrazione definiamo una procedura chiamata RICNUM che restituisce 1 se x appartiene a D altrimenti non termina:

$$F_p(x) = \begin{cases} 1 & \text{se } x \in D \Rightarrow F_u(x\$x)\downarrow \\ \uparrow & \text{se } x \notin D \Rightarrow F_u(x\$x)\uparrow \end{cases}$$

```
RICNUM:
Richiede in input x ∈ {0, 1}*
Chiama l'interprete U e gli passa l'argomento (x $ x)
        U:
        Prende in input l'argomento (x $ x)
        Esegue Fu (x, x)
                Se x ∈ D → Fu (x, x) ↓, Fu (x, x) = 1 viene ritornato a RICNUM
                Se x ∉ D → Fu (x, x) ↑, la funzione non termina, non si torna a RICNUM
Output = 1 sse riceve da Fu (x, x) il valore 1
```

```
Proceolura RICNUM ( x ∈ {0,1}* )
{
    y = Fu (x $ x) ;
    return (1);
}
```

La procedura **RICNUM** prende in input la parola "x", richiama l'interprete "u" passandogli in input x$x e se l'interprete termina la sua esecuzione, quindi x appartiene a D, la procedura restituisce "1" mentre se l'interprete non termina rimaniamo in una situazione di loop.

**Dimostrazione di correttezza di RICNUM**

- $x \in D \rightarrow F_u(x\$x)\downarrow$ quindi eseguiamo l'assegnamento ad Y ed viene eseguito il return di 1, quindi la procedura RICNUM(x) restituisce 1
- $x \notin D \rightarrow F_u(x\$x)\uparrow$ quindi non terminando e andando il loop, non è possibile eseguire l'assegnamento, pertanto la procedura RICNUM rimane in loop → RICNUM(x)↑

Abbiamo trovato la procedura ricercata e quindi abbiamo dimostrato che D è ricorsivamente enumerabile

**La dimostrazione del punto 2 – D non è ricorsivo:** $D = \{x \in \{0,1\}^* \mid F_u(x\$x)\downarrow\}$

Per dimostrare che D non è ricorsivo dobbiamo utilizzare la tecnica dell'assurdo, andiamo a supporre che D sia ricorsivo e quindi possiamo costruire il seguente programma, il quale ci permette di ottenere un assurdo che ci dirà che l'ipotesi iniziale è errata, definiamo la procedura **ASSURDOA**:

```
ASSURDOA:
Richiede in input x ∈ {0, 1}*
Chiama la funzione APPARTENENZA_D e gli passa come argomento (x)
        APPARTENENZA_D:
        Prende in input l'argomento x ∈ {0, 1}*
        Esegue y = F_APPARTENENZAD (x)
                Se x ∈ D → F_APPARTENENZAD (x) ↓, y = 1
                Se x ∉ D → F_APPARTENENZAD (x) ↓, y= 0
        Output = y
If( y == 1)   then Output:  "1-Fu (x, x) "
Else                Output : " 0 "
```

Questo programma prende in input un argomento "x", se x appartiene a D restituisce 1 – $F_u(x\$x)$ altrimenti restituisce 0. Questa procedura è implementabile a patto che algoritmo D sia ricorsivo, da qui deriviamo il codice "e" che rappresenta la parola che codifica il programma ASSURDOA.

Ora passiamo alla procedura ASSURDOA il codice e quindi, ASSURDOA(e).

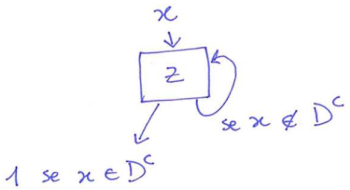
ASSURDOA(e) = $F_e(e) = F_u(e\$e)$, in quanto il programma ASSURDOA è codificato tramite la lettera "e", per cui ASSURDOA(e) equivale a dire un programma "e" che sta eseguendo se stesso cioè $F_e(e)$, per definizione dell'interprete possiamo dire che equivale a chiamare un interprete che esegue il programma "e" sull'input "e", cioè $F_u(e\$e)$.

Ora andiamo ad osservare entrambi i casi che il nostro programma ASSURDOA può generare:

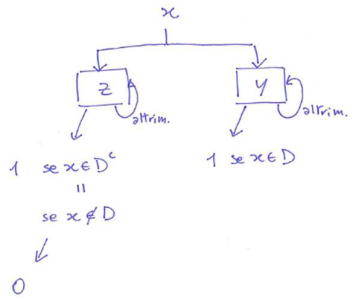
- caso $e \in D$
  Per definizione di D, l'esecuzione di $F_u(e\$e)\downarrow$ termina con output 0/1
  In quanto abbiamo detto che $F_u(e\$e)$ equivale ad ASSURDOA(e) abbiamo come risultato 1- $F_u(e\$e)$ per cui nel caso $F_u(e\$e) = 1$ otteniamo 1-1 = 0 e nel caso $F_u(e\$e) = 0$ otteniamo 1-0 = 1
  Questa è una contraddizione in quanto 0 non può essere un output di D.
- caso $e \notin D$
  Per definizione di D, l'esecuzione di $F_u(e\$e)\uparrow$ non termina
  $F_u(e\$e)$ equivale a dire ASSURDOA(e) che per sua definizione se e non appartiene a D restituisce 0, a questo punto si ottiene $\uparrow=0$ che è una contraddizione.

Date queste due contraddizione possiamo dire di aver costruito un programma "e" che non esiste dunque D non può essere ricorsivo.

**La dimostrazione del punto 3 – $D^C$ non è ricorsivamente enumerabile:** $D^C = \{x \in \{0,1\}^* \mid F_u(x\$x)\uparrow\}$




Per eseguire la dimostrazione del fatto che $D^c$ non è ricorsivamente enumerabile dobbiamo utilizzare la tecnica per assurdo, andiamo a supporre che $D^C$ sia ricorsivamente enumerabile quindi, per ipotesi, esiste una procedura Z tale che ricevendo input "x" termina e restituisce 1 se $x \in D^C$ e rimane in loop se $x \notin D^C$
Inoltre sappiamo che D è ricorsivamente enumerabile, dimostrazione del punto 1, quindi esiste una procedura Y tale che ricevendo input "x" termina e restituisce 1 se $x \in D$ e rimane in loop se $x \notin D$




Avendo questi due programmi tra di loro complementari possiamo creare un terzo programma unendo questi due come se si stessero eseguendo in parallelo su uno stesso input "x", quindi se "x" appartiene a D la procedura Y restituisce 1 mentre la procedura Z rimane in loop e se "x" appartiene a $D^C$ la procedura Z restituisce 1 mentre la procedura Y rimane in loop, quindi possiamo dedurre che se "x" appartiene a $D^C$ non appartiene a D quindi abbiamo trovato un algoritmo per D che restituisce 1 se $x \in D$ e 0 se $x \notin D$

Definiamo un programma **ASSURDOB**($x \in \{0,1\}^*$)

```
ASSURDOB:
Richiede in input x ∈ {0, 1}*
k = 0
h= z
      while ( z↑ and y↑)
      {
            if (h == z)   then    h = y
                          else    h = z
            esegui h per k passi
            k = k+1
      }
if (z↓) then   output: "1"
      else     output: "0"
```

k indica il numero di passi, in questa procedura eseguiamo un ciclo while inserendo come parametri Z(x) non termina in K passi and Y(x) non termina in K passi, se nessuna delle due termina incrementiamo K di uno, prima o poi si raggiunge il K passo in cui una delle due procedure termina e quindi usciamo dal ciclo.
da qui diciamo che se ha terminato Y ritorniamo 1 altrimenti ritorniamo 0.

### Funzionamento di ASSURDOB(x)

Possiamo definire che ASSURDOB(x) ha due casistiche:

- $x \in D$, se x appartiene a D vuol dire che esiste k passo dove la procedura Y(x) termina e quindi ASSURDOB(x) restituisce 1
- $x \notin D$, se x non appartiene a D vuol dire che esiste k passo dove la procedura Z(x) termina e quindi ASSURDOB(x) restituisce 0

Da questi due casi possiamo definire che ASSURDOB è un algoritmo per D che è una contraddizione in quanto D non è ricorsivo per dimostrazione del punto due del teorema, per cui la procedura Z, che abbiamo ipotizato, non esiste e quindi $D^C$ non è ricorsivamente enumerabile

### Problema dell'arresto

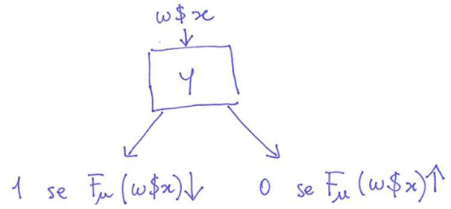
In input abbiamo un programma ed un dato, $w\$x \in \{0,1\}^*$ mentre in output ci chiediamo se tale programma w eseguito su x, cioè $F_w(x)$, termina, da questo possiamo definire il linguaggio dell'arresto

### Il Linguaggio dell'arresto

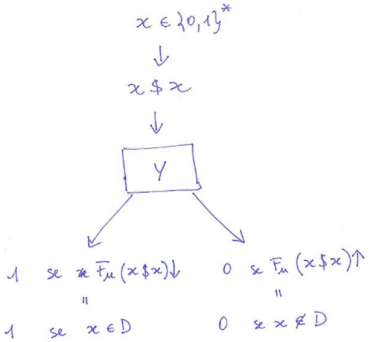
$A = \{w\$x \in \{0,1\}^* \mid F_u(w\$x)\downarrow\}$ $D = \{x \in \{0,1\}^* \mid F_u(x\$x)\downarrow\}$ → arresto ristretto

Teorema: il linguaggio A non è ricorsivo

Dimostrazione: usiamo il fatto che il linguaggio D non sia ricorsivo, dimostriamo che A non sia ricorsivo tramite una tecnica per assurdo, quindi supponiamo che il linguaggio A sia ricorsivo.




Ora sfruttiamo la procedura Y per dire che D sia ricorsivo, ma che D sia ricorsivo è un assurdo quindi tale procedura Y non esiste, quindi definiamo il seguente caso:




Siccome D è quel linguaggio che definisce che $F_u(x\$x)\downarrow$ termina, quindi possiamo dire che se Y restituisce 1 equivale a dire che x appartiene a D, mentre se restituisce 0 equivale a dire che x non appartiene a D.

Da questa procedura possiamo definire il Programma **ASSURDOC(x)**

ASSURDOC(x) va a restituire 1 se la parola x$x appartiene ad A, altrimenti restituisce 0

### Funzionamento di ASSURDOC(x)

Possiamo definire che ASSURDOC(x) ha due casistiche:

- $x \in D$, se x appartiene a D, vuol dire che $F_u(x\$x)\downarrow$ termina e quindi x$x appartiene ad A e il programma ASSURDOC(x) restituisce 1
- $x \notin D$, se x non appartiene a D, vuol dire che $F_u(x\$x)\uparrow$ non termina e quindi x$x non appartiene ad A e il programma ASSURDOC(x) restituisce 0

Tramite il seguente funzionamento abbiamo definito che ASSURDOC è un algoritmo per D, questa è una contraddizione in quanto D non è ricorsivo di conseguenza non possiamo dire che esiste un algoritmo Y per

A per cui il linguaggio A non è ricorsivo, quindi possiamo dire che il **problema dell'arresto è indecidibile**, cioe non dimostrabile.

## I sistemi generativi: la grammatica

Idea di grammatica sta nel generare le parole del linguaggio sotto certe regole sintattiche, in quanto il linguaggio italiano è molto complesso per mostrare un esempio di grammatica andiamo a lavorare su un sottolinguaggio quello degli annunci ferroviari in stazione

Partiamo le seguente regola per il messaggio di soppressione:
<soppressione> → il treno S<num> delle <orario> è soppresso

Le regole vengono indicate con la freccia "→" per esempio:

- <num> → 1, 2, …, 13
- <orario> → <ore> e <minuti>
- <ore> → 1, 2, …, 24
- <minuti> → 00, 01, …, 59

**Definizione dei simboli**:

- <…>: viene definito **metasimbolo** ed indica una variabile, un qualcosa che deve essere sostituito secondo alcuni parametri definiti
- Non <…>: viene definito simbolo **terminale** ed è un elemento fisso della grammatica
- <nome>: è il simbolo iniziale detto **assioma**, nel nostro esempio è <soppressione> ed indica il nome della frase che vogliamo generare
- … → …: indica una **regola di produzione**
- ⇒: viene definito **passo di derivazione** e va ad indicare l'applicazione di una regola
- ⇒*: viene definita **derivazione in zero** o più passi e va ad indicare l'applicazione di più regole

Esempio:

<soppressione> ⇒ Il treno S<num> delle <orario> è soppresso ⇒ Il treno S6 delle <orario> è soppresso ⇒ Il treno S6 delle <ore> e <minuti> è soppresso ⇒* Il Treno S6 delle 10 e 20 è soppresso

In questo esempio possiamo vedere una derivazione per la generazione di una frase degli annunci ferroviari, al primo passo viene usata una regola per sostituire <num>, successivamente usiamo la regola <orario> per dire che è formata da <ore> e <minuti>, da qui applichiamo due regole insieme per andare a definire l'ora e i minuti.
Ora abbiamo formato il nostro messaggio completo.

### Formalizziamo la definizione di Grammatica

**Definizione**: una grammatica è composta da una quadrupla: G = (Σ, M, S, P)

- **Σ**: indica l'insieme finito dei simboli terminali
- **M**: indica l'insieme finito dei metasimboli
  l'intersezione tra Σ e M è un insieme vuoto
- **S**: indica un simbolo unico in M, quindi S appartiene ad M, ed è l'assioma
- **P**: indica l'insieme finito delle regole di produzione

### Definizione di regola di produzione

Una regola di produzione è una copia "a" e "b" con intermezzo di una freccia: a → b, che sono parole composte da simboli terminali variabili che sono formati cosi: a ∈ $(\Sigma \cup M)^+$ e b ∈ $(\Sigma \cup M)^*$, "a" non può essere la parola vuota in quanto dal niente non è possibile generare qualcosa, ma è possibile da qualcosa cancellare e diventare la parola vuota

Esempio: S → ε è possibile mentre ε → S non è possibile

### Definizione di passo di derivazione

Un passo di derivazione è l'applicazione di una regola e si indica che "w" è derivabile da "z" in un passo e si scrive z ⇒ w, quando applico una regola, per esempio a → b ∈ P e le due parole z e w sono fattorizzate così: z = xay e w = xby, le quali sono parole a tre fattori che si differenziano solo da a e b che sono stati sostituito dalla regola applicata.

### Definizione di derivazione in zero o più passi

Una derivazione in zero o più passi consiste nell'applicazione di zero o più regole e si dice che "w" è derivabile da "z" in zero o più passi e si indica con z ⇒* w, possiamo scrivere le seguente formula quando esistono delle parole $w_1, w_2, \ldots, w_k$ ed una costante K appartenete ai numeri naturali escluso lo zero, per cui esistono singoli passi di derivazione che legano z con w, in questo caso stiamo applicando K regole, oppure

$z \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_K = w$

quando w = z non è necessario applicare nessuna regola di produzione.

Il **linguaggio generato dalla grammatica** è l'insieme di quelle parole w appartenenti a Σ* tale che dall'assioma in zero o più passi di derivazione otteniamo w, w è formata solamente da simboli terminali.

L(G) = {w ∈ Σ* | S ⇒* w}

Possiamo dire che in questo caso non è possibile applicare zero passi di derivazione in quanto w non può essere uguale ad S, (S ⇒* S), in quanto quest'ultimo non è un simbolo terminale ma un metasimbolo.

**Osservazioni**: un linguaggio ammette più grammatiche che lo possono generare e possiamo dire che due grammatiche G1 e G2 si dicono equivalenti se generano due linguaggi equivalenti L(G1) = L(G2)

**Esempio**: data la seguente grammatica ( {a}, {S}, S, {S → aS, S→a}) quale linguaggio genera L(G)?

La parola più corta che può generare è "a" dalla regola S → a in quanto una volta applicata questa regola non è più possibile procedere in quando non abbiamo più la variabile "S", mentre applicando la seconda regola possiamo continuare all'infinito fino all'applicazione della prima regola S → a.

Il linguaggio generato è l'insieme delle parole formate da a, L(G) = $a^+$

**Esempio di grammatiche equivalenti**: dato il seguente linguaggio L = {$a^{2n}$ | n >= 1}, riusciamo a definire due grammatiche equivalenti che con regole diverse generano il seguente linguaggio.

$S \to aa$

$S \to aaS$

$G_1$

$S \to aA$

$A \to aS$

$A \to a$

$G_2$

La regola di conclusione di generazione può essere definita regola tappo, in questo caso per G1 è S → aa e per G2 è A → a

Esempio di grammatica con regole dipendenti dal contesto: dato il seguente linguaggio {$a^n b^n c^n$ | n > 0}, possiamo definire le seguenti regole di generazione S → aSBC e S → aBC, in questo modo riusciamo a generare in diversi passi S ⇒* aaaBCBCBC, in modo disordinato rispetto al linguaggio, abbiamo bisogno di una regola per ordinare il linguaggio CB → BC, cosi otteniamo aaaBBBCCC, a questo punto definiamo le seguenti regole dipendenti dal contesto in quanto devono essere formate da un simbolo terminale e una

variabile per trasformare le variabili B e C in simboli terminali b e c, le regole sono le seguenti: aB → ab, bB → bb, bC → bc e cC → cc

Non tutti i linguaggi ammettono una grammatica che li genera, possiamo definire il seguente **teorema**:

Un linguaggio è generato da una Grammatica se e solo se è un linguaggio ricorsivamente enumerabile, vale anche il contrario un linguaggio ricorsivamente enumerabile ammette una grammatica

**Dimostrazione che da una grammatica deriva un linguaggio ric. enumerabile**:

Data una grammatica G per linguaggio L possiamo costruire una procedura "w" tale che: $F_w(x)$ restituisca 1 se $x \in L$ mentre se $x \notin L$ non termina, quindi il linguaggio L è ricorsivamente enumerabile

**Definizioni**:

**$F_i$** = è un insieme di parole di simboli terminali e variabili ottenute da S in "i" passi di derivazione

**$T_i$** = è un insieme di parole di soli simboli terminali ottenute da S in "i" passi di derivazione, quindi $T_i \subseteq F_i$

Vale il seguente fatto il L(G) è l'unione dei $T_i$ per ogni i

Andiamo a dimostrare che **L(G) è un sottoinsieme dell'unione dei $T_i$ e viceversa**

1) $x \in L(G) \Rightarrow$ ho $S \Rightarrow^* x$, quindi esiste un "i" tale che i è il numero di passi di derivazione in $S \Rightarrow^* x$, quindi possiamo dire che $x \in T_i$ e quindi $x \in$ l'unione di $T_i$ per ogni i
2) $x \in UT_i$, quindi esiste un "i" tale che $x \in T_i$, quindi abbiamo una derivazione $S \Rightarrow^* x$ in i passi di derivazione, quindi $x \in L(G)$

**Formalizzazione di $F_i$ e $T_i$**:

$$F_i = \{ \gamma \in (\Sigma \cup M)^* \mid \eta \Rightarrow \gamma \text{ e } \eta \in F_{i-1} \}$$

$$T_i = \{ x \in \Sigma^* \mid x \in F_i \}$$

$F_i$ è formalizzabile in modo ricorsivo dicendo che $F_i$ è l'insieme di quelle parole appartenteti a $(\Sigma \cup M)^*$, quindi costituite da simboli terminali e variabili, tale che esiste $n \in F_{i-1}$ a cui andremo ad applicare una regola di produzione della grammatica

$T_i$ è l'insieme di tutte le parole formate da simboli terminali appartenenti a $\Sigma^*$ tali che x appartiene a $F_i$

Dimostrazione che **L(G) sia ricorsivamente enumerabile** tramite la procedura **ELENCA**:

Sapendo che L(G) è l'unione di tutti gli i di $T_i$ siamo in grado di costruire una procedura ELENCA che è in grado di stampare tutte le parole di L(G) applicando regole di produzione

```
Procedura ELENCA(x ∈ Σ*){
G = (Σ, M, P, S); fisso G
F0 = {S}; inizializzo F0 con l'assioma
i = 1;
while (i >0) do
{       Fi = CostruisciF(Fi-1,G);
        Ti = CostruisciT(Fi, G);
(if) Output(Ti);
i = i + 1;
}}
```

```
CostruisciF(Fi-1, G){
Fi = 0;
foreach n ∈ Fi-1 do
foreach a → b ∈ P do
foreach x,y ∈ (ΣUM)*
tale che n = xay do
Fi = Fi ∪ {xby}
return (Fi);
}
```

```
CostruisciT(Fi, G){
Ti = 0;
foreach w ∈ Fi do
if (w ∈ Σ*) then
Ti = Ti ∪ {w};
return (Ti);
}
```

**Funzionamento di CostruisciF**: andiamo a verifica se la parola "n" appartenente ad $F_{i-1}$ e verifichiamo se ad "n" riusciamo ad applicare la regola a → b appartenente a P, con l'ultimo ciclo for andiamo a verificare in quali modi è possibile applicare la regola alla nostra parola e per ogni parola che contiene "xay" andiamo ad trasformarla in "xby", a questo punto eseguiamo l'unione di $F_i$ con "xby"

**Funzionamento di CostruisciT**: andiamo a verificare se "w" sia fatta solo da simboli terminali se si procediamo ad eseguire l'unione dell'insieme $T_i$ con "w"

Andiamo a trasformare la procedura ELENCA nella procedura W:

- prima modifica: passo $x \in \Sigma^*$ in input ad ELENCA
- seconda modifica: sostituiamo Output($T_i$) con if ($x \in T_i$) then return(1), ora se x appartiene a $T_i$ fermiamo la procedura e ritorniamo 1

**Correttezza di ELENCA modificata**: se $x \in L(G)$ deve restituire 1 altrimenti deve rimanere in loop

Possiamo vedere che se $x \in L(G)$ esiste un "i" tale che x appartenga a $T_i$ e quindi verrà eseguito return(1) per cui la funzione ELENCA(x) restituisce 1 se x appartiene a L(G)

Possiamo vedere che, se $x \notin L(G)$ non potrà esistere alcuna unione di i di $T_i$ a cui appartiene x, quindi per ogni i $x \notin T_i$ e quindi il ciclo while rimane in una situazione di loop, per cui ELENCA(x)↑

Da questa verifica di correttezza possiamo dire che L(G) è ricorsivamente enumerabile.

**Linguaggio delle espressioni booleane**: L su and e or

**Definizione**:

- $0,1 \in L$
- $x,y \in L$, quindi $(x \wedge y)$ e $(x \vee y)$ appartengono a L
- Nient'altro appartiene ad L

**Grammatica G**:

| | |
|---|---|
| $\Sigma = \{0, 1, (, ), \wedge, \vee\}$ | simboli terminali |
| $M = \{E\}$ | variabili |
| E | assioma |
| $P = \{E \to 0, E \to 1, E \to (E \wedge E), E \to (E \vee E)\}$ | regole di produzione |

Esempio di applicazione di G:

$\omega = (0 \wedge (1 \vee 0)) \in L$

$E \Rightarrow (E \wedge E) \Rightarrow (0 \wedge E) \Rightarrow$

$\Rightarrow (0 \wedge (E \vee E)) \overset{*}{\Rightarrow} (0 \wedge (1 \vee 0))$

**Albero di derivazione per w**:

Partiamo dalla nostra radice/assioma "E" a cui andiamo ad applicare la prima regola, le variabili le possiamo definire nodi interni mentre i simboli terminali li definiamo come foglie, la parola finale la otteniamo leggendo le foglie da sinistra verso destra.

Possiamo dire che è ammesso il sottoalbero solo se $A \to B_1 B_2 B_3 \dots B_k$ è una regola di G

**Osservazione**: possiamo avere alberi di derivazione, la grammatica G deve essere di tipo 2

**La classificazione di Chomsky – versione provvisoria**

La classificazione di Chomsky suddivide le grammatiche in quattro tipi in funzione delle regole che le definiscono, la grammatica è semplice se il tipo è alto invece è complessa se il tipo è basso in quanto genera un linguaggio difficile.

- **Tipo 0**: non richiede alcun vincolo sulle regole, quindi qualunque grammatica è possibile vederla come tipo 0, l'importante che vi siano regole valide, per esempio non è ammessa ε → S per il fatto che non sia accettata per definizione delle regole
- **Tipo 1**: ammette la seguente definizione di regola a → b con lunghezza di b maggiore o uguale alla lunghezza di a, definibile così |b| >= |a|, in questo caso la mia parola tenderà sempre a crescere o al massimo rimanere invariata
- **Tipo 2**: ammette la seguente definizione di regola A → b dove A appartiene ad M, a differenza del tipo 1 A può essere solamente una variabile e non simbolo terminale, mentre b appartiene all'insieme $(\Sigma \cup M)^+$ a differenza del tipo uno dove apparteneva a $(\Sigma \cup M)^*$, quindi non è possibile cancellare una variabile
- **Tipo 3**: ammette la seguente definizione di regola A → yB e A → x, dove A e B sono elementi dei metasimboli quindi variabili e appartenenti a M, mentre x e y sono simboli terminali, $y \in \Sigma^*$ e $x \in \Sigma^+$

Possiamo dire che una grammatica di tipo K è anche una grammatica di tipo K-1, quindi una grammatica di tipo 3 è anche di tipo 2,1 e 0

**Definizione**: un linguaggio si dice di tipo K se ammette una grammatica G di tipo K che lo genera, se non conosciamo la grammatica che lo genera è possibile dire che è di tipo 0

**Definizione**: possiamo suddividere i linguaggi a seconda delle grammatiche che lo generano, $R_k$ {L ⊆ Σ* | L è di tipo K}, per esempio:

- **$R_3$**: indica i linguaggi regolari, cioè i linguaggi generati da una grammatica di tipo 3
- **$R_2$**: indica i linguaggi liberi dal contesto, cioè i linguaggi generati da una grammatica di tipo 2
- **$R_1$**: indica i linguaggi dipendenti dal contesto, cioè i linguaggi generati da una grammatica di tipo 1
- **$R_0$**: indica tutti i linguaggi ricorsivamente enumerabili, cioè i linguaggi generati da una grammatica di tipo 0

**Esempio di tipo 3**: S → aaS e S → aa per $\{a^{2n} \mid n > 0\}$, questa grammatica soddisfa le seguenti condizioni A → yB e A → x

**Esempio di tipo 2**: S → aSb e S → ab per $\{a^n b^n \mid n > 0\}$, questa grammatica soddisfa la seguente condizione A → b, in quanto S appartiene a M e ab appartengono a $\Sigma^+$

**Esempio di tipo 1**: S → aSBC, S → aBC, CB → BC, aB → ab, bB → bb, bC → bc, cC → cc per $\{a^n b^n c^n \mid n > 0\}$ questa grammatica soddisfa la seguente condizione a → b con |b| >= |a|

## I documenti XML

I documenti XML sono composti da testo e tag, un singolo documento XML lo possiamo vedere come una parola dove i simboli terminali sono i caratteri del testo e i tag

I **tag** sono marcatori di testo che consentono di dare informazioni semantiche al testo

Esempio:
```
<rubrica>
    <nome> Bianchi </nome>
    <tel> 02874810 </tel>
    <tel> 349657831 </tel>
    <nome> Rossi </nome>
    <tel> 05321861 </tel>
</rubrica>
```

**Notazione**: tag aperto : <t> tag andiamo a segnarlo con "t", chiuso </t> andiamo a segnarlo con "t'"

**Condizioni**: Un documento XML deve soddisfare le seguenti condizioni:

1- Deve esistere un unico tag che contiene tutto il contenuto, nel nostro esempio è rubrica
2- Ogni tag aperto deve essere seguito dal suo reciproco tag chiuso
3- I tag devono essere innestati correttamente, vanno aperti e chiusi in ordine inverso, ultimo aperto primo chiuso, per esempio: <x> … <y> … </y> … </x>

**Definizione**: possiamo definire che un documento XML è corretto solamente se soddisfa queste tre condizioni

I documenti XML sono un linguaggio di tipo 2 e quindi ammettono una grammatica di tipo 2 che li generi, un documento XML oltre alla sua correttezza sintattica richiede che soddisfi un DTD, document type definition

Il **DTD** definisce come i tag possono essere innestati tra di loro

**Definizione**: un documento XML è valido secondo un certo DTD se è generato da quel DTD, che è una grammatica di tipo 2

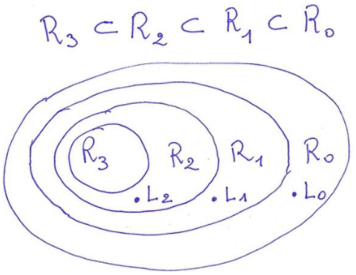
Forma delle regole per i DTD, la possiamo scrivere così A → a Ra a'

A = variabile del tag, Ra = espressione di variabili con +, ·, * ed Ra è trasformabile in regole di tipo 2

**Esempio**: S → s C* s'  C → c MN(EV)* c'  M → m m'  N → n n'  E → e e'  V → v v'

S = studenti, C = curriculum, M = matricola, N = nome, E = esame, V = voto

Questa grammatica verifica la correttezza dei tag in un documento XML




**Teorema di inclusione sugli $R_k$**

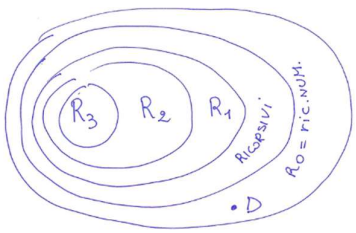
il teorema di inclusione dice che ogni classe di grado maggiore include le classi di grado minore, possono essere viste come insiemi e sottoinsiemi propri quindi le classi non possono essere uguali

**dimostrazione**:

L'inclusione tra gli $R_k$ segue dal fatto che abbiamo dato sui tipi di grammatica: Tipo K → Tipo K-1

Andiamo a dimostrare che l'inclusione è propria:

- esiste $L_2 \in R_2$ ma $L_2 \notin R_3$, $L_2 = \{a^n b^n \mid n > 0\}$ infatti $a^n b^n$ ammette una grammatica di tipo 2: S → aSb e S → ab e inoltre vedremo che $a^n b^n \notin R_3$ perché non ammette un automa a stati finiti che lo riconosce
- esiste $L_1 \in R_1$ ma $L_1 \notin R_2$, $L_1 = \{a^n b^n c^n \mid n > 0\}$ infatti $a^n b^n c^n$ ammette una grammatica di tipo 1 e inoltre vedremo che $a^n b^n c^n \notin R_2$ perché non soddisfa il pumping lemma per i linguaggi liberi da contesto
- esiste $L_0 \in R_0$ ma $L_0 \notin R_1$, $L_0 = D = \{x \in \{0,1\}^* \mid F_u(x\$x)\downarrow\}$




**dimostrazione**:

1° passo: $R_1$ è sottoinsieme dei linguaggi ricorsivi
2° passo: D non è ricorsivo quindi non appartiene a $R_1$, D è ricorsivamente enumerabile quindi appartiene a $R_0$

**Dimostrazione del primo passo**

Definizione di GR(x) con $x \in \Sigma^*$

GR = Grafo è una coppia di elementi, vertice (V) e archi (E) = $< V_x, E_x >$ dove $V_x = \{y \in (\Sigma \cup M)^* \mid |y| <= |x|\}$

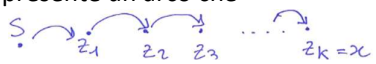
Gli **archi** sono definibili come una coppia di vertici, infatti $E_x = \{(y, y') \mid y \Rightarrow y')\}$, quando y' è derivabile da y con una solo regola di produzione della grammatica


Algoritmo w(x ∈ Σ*){
Costruisci GR(x); per costruire il grafo è necessaria G fissata all'interno dell'algoritmo
if (esiste un cammino S --------> x)
then return (1);
else return (0);}

**Osservazione**: il problema della generazione di x, quindi se x appartiene ad L(G) oppure no, si trasforma nel problema della ricerca di un cammino in un grafo, il tempo dell'algoritmo w è esponenziale in base alla lunghezza dell'input per via della funzione Costruisci GR(x) in quanto più x è lunga più sono le parole diverse generabili

**Correttezza dell'algoritmo w**, dobbiamo dimostrare che se gli viene passata una parola x appartenente ad L(G) l'algoritmo termina con 1 mentre se x non appartiene deve terminare con 0

**$x \in L(G)$** vuol dire che esiste una derivazione in G, G grammatica di tipo 1, che partendo da un S con k passi di derivazione mi permette di ottenere x, $S \Rightarrow Z_1 \Rightarrow \dots \Rightarrow Z_k = x$, tale che per ogni i $|Z_i| <= |Z_{i+1}|$ per cui per ogni i $|Z_i| <= |x|$ e quindi per ogni i $Z_i$ appartiene all'insieme dei vertici $V_x$ ed inoltre è presente un arco che connette $Z_i$ con $Z_{i+1}$ fino ad arrivare ad x, quindi possiamo vedere il grafo GR(x) cosi:




quindi data l'esistenza di un cammino per x l'algoritmo restituisce 1

**$x \notin L(G)$** vuol dire che non esiste alcuna derivazione in G che mi permetta di ottenere x partendo da S con k passi, andiamo a dimostrarla facendo la negazione quindi diciamo che, se $F_w(x) = 1$ indica $x \in L(G)$, allora se uguale a 0 vuol dire che non appartiene.

$F_w(x) = 1$ implica l'esistenza di un cammino che genera x, quindi in G esistono k passi che mi permettono di ottenere x da S e quindi $x \in L(G)$.

Data la classificazione di Chomsky provvisoria, se L è di tipo k, di che tipo è L unione {ε} in quanto S → ε è consentita solamente dal tipo 0, questo è poco ragionevole quindi apportiamo due modifiche alla classificazione

**1° modifica**: per le grammatiche di tipo K = 3, 2, 1 è ammessa la regola S → ε a patto che:

- S sia l'assioma
- S non compaia sulla destra di altre regole

Quindi se L è di tipo K allora anche L unione {ε} è di tipo K

**Dimostrazione**: ho una grammatica G per L di tipo K e devo costruire G' per L ∪ {ε} di tipo K che mantiene le stesse regole di G con l'introduzione di S' assioma per G' ed S' → S e S' → ε, rispettano i patti precedentemente elencati in quanto S' è assioma e non compare sulla destra di altre regole

**2° modifica**: per le grammatiche di tipo K = 3,2 ammettiamo le regole nella forma A → ε, dove A è una variabile arbitraria, quindi se ho G di tipo K con K = 3,2 con regole A → ε possono ottenere G' equivalente di tipo K che rispetta i patti precedentemente elencanti

**Esempio**: avendo una grammatica con le seguenti regole:

S → 0S0
S → 1S1
S → ε

possiamo dire che è di tipo 2 ma non rispetta i patti precedenti in quanto S compare alla destra di altre regole ed in aggiunta in quanto la lunghezza della parola generata può essere decrescente non è di tipo 2, in quanto non è ammessa la decrescenza a seguito di una regola applicata

Andiamo a modificare queste regole per renderla idonea al tipo e che rispetti i patti, quindi andiamo ad applicare S → ε ad altre regole ed a eliminare S → ε dalle nostre regole

S → 00 Queste due regole simulano il funzionamento di S → ε in modo di ottenere due nuove
S → 11 regole terminali in modo da avere una grammatica equivalente

Andiamo ad aggiungere anche S' come assioma con le seguenti regole S' → S ed S' → ε per mantenere la generazione della parola vuota nella nostra grammatica, in questo modo abbiamo una grammatica di tipo 2 che rispetta i patti precedenti

**La classificazione di Chomsky – versione definitiva**

**Tipo 0**: regole arbitrarie, ammette qualsiasi regola di produzione

**Tipo 1**: regole nella forma a → b con |b|>= |a|, nella quale è ammessa la regola S → ε se S è l'assioma ed S non compare sulla destra di altre regole

**Tipo 2**: regole nella forma A → b dove A ∈ M ed b ∈ (Σ ∪ M)*

**Tipo 3**: regole nella forma A → xB e A → y dove A,B ∈ M e x,y ∈ Σ*

**Teorema sugli $R_K$**: $R_3 \subset R_2 \subset R_1 \subset R_0$, questo teorema continua a valere grazie al fatto due

**Teorema**: rimane valida la seguente notazione $R_1 \subseteq L$ Ricorsivi in quanto la frase preserva la lunghezza non decrescente delle derivazione grazie ai nostri patti imposti

**Le forme equivalenti per il tipo 3**:

(i) $A \to \sigma B$, $A \to \sigma$, $A \to \varepsilon$
dove $A, B \in M$ e $\sigma \in \Sigma$

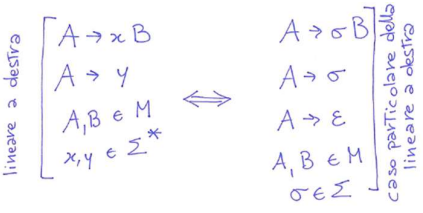
(ii) $A \to \sigma B$, $A \to \varepsilon$
dove $A, B \in M$ e $\sigma \in \Sigma$

(iii) $A \to \sigma B$, $A \to \sigma$ } solo se $\varepsilon \notin L$
dove $A, B \in M$ e $\sigma \in \Sigma$

**Trasformazioni delle forme equivalenti**

- Possiamo passare da (i) a (ii), dobbiamo eliminare le regole A → a
  Introduciamo una nuova variabile X e per ogni regola A → a introduco A → aX e X → ε
  Andiamo a cancellare le regole A → a
- Possiamo passare da (i) a (iii), dobbiamo eliminare le regole A → ε
  Per ogni regola A → aB aggiungo A → a se B → ε appartiene a P
  Cancelliamo le regole A → ε

Vale l'equivalenza tra la lineare a destra e la prima forma (i), la prima forma è un caso particolare della lineare a destra, il problema è derivare la lineare a destra dal caso particolare



**Trasformazione di una lineare a destra nel caso particolare** tramite un esempio:

A → abcB con a,b,c ∈ Σ cioè simboli terminali, per trasformarla nel caso particolare eliminiamo bcB con una variabile X e determiniamo le seguenti regole A → aX, X → bcB, a questo punto eliminiamo cB con una variabile Y e determiniamo le seguenti regole X → bY, Y → cB ottenendo le seguenti nuove regole: {A → aX, X → bY, Y → cB}, queste trasformazioni sono valide solo con |x|> 0

Infatti, se |x| = 0, le regole del tipo A → xB diventano regole unitarie A → B che possono essere trasformare in A → aB e A → a

## Teoria degli automi a stati finiti

Gli automi a stati finiti possono essere usati sia in ambito hardware che software, hardware per la progettazione di circuiti elettronici e modellare semplici sistemi, software per la costruzione di analizzatori lessicali, soluzione a problemi di ricerca nei testi e modellare pagine web

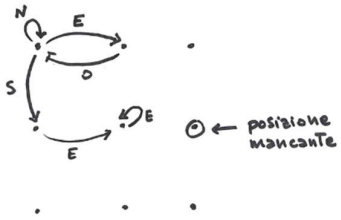
Esempio di un sistema da modellare con automi:

Abbiamo una griglia costituita da punti sulla quale si muove un robot, il robot è telecomandato da segnali che indicano la direzione, il comportamento del robot può essere modellato da un automa

Postazioni nella griglia → stati dell'automa

Segnali inviati al robot → simboli di input dell'automa: N, S, E, O

I puntini indicano gli stati dell'automa, mentre le frecce gli input direzionali ricevuti

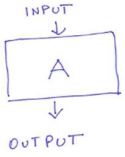



La posizione del robot dipende dalla posizione attuale e dal simbolo inviato in ingresso, il percorso creato dal robot diventa una parola

Grazie all'automa possiamo studiare i problemi legati al sistema

- Individuare una parola che mi porta da A a B
- Individuare una parola che mi porta da A a B attraverso C oppure lo evita
- Individuare tutte le parole prive di cicli che mi portano da A a B
- Individuare la parola più corta che mi porta da A a B

## Automi a stati finiti come riconoscitori di linguaggi formali




Visti dall'esterno gli automi vengono visti come una scatola che interagisce con l'esterno prendendo degli input e restituendo degli output, gli input possono essere una parola w ∈ Σ* e l'output consiste in una uscita booleana, 0 o 1, se w è rifiutata = 0 mentre se è accettata = 1

Il comportamento di A è dato dall'insieme delle parole che A accetta e il loro insieme rappresenta il linguaggio accettato, possiamo quindi dedurre che A è un riconoscitore di linguaggi, in particolare gli automi a stati finiti riconoscono i linguaggi di tipo 3

### Osservazioni sul funzionamento degli automi

- In ogni istante di tempo T l'automa si trova in un particolare stato, inizialmente A si trova in uno stato iniziale: $q_0$
- In funzione del simbolo letto e dello stato attuale, l'automa A cambia stato, la funzione di transizione viene indicata con il seguente simbolo "δ", δ(q,a) = stato prossimo di A essendo in q e leggendo a come input

- Una volta letta l'intera parola w, l'automa A raggiunge uno stato p e l'uscita dipende dallo stato raggiunto p:
  $\lambda(p) = 0$ o 1 ed è detta funzione di uscita

**Formalizziamo quanto detto sul funzionamento**:

**definizione**: un automa a stati finiti è una tupla dei seguenti elementi: $< \Sigma, Q, \delta, q_0, \lambda(F)>$

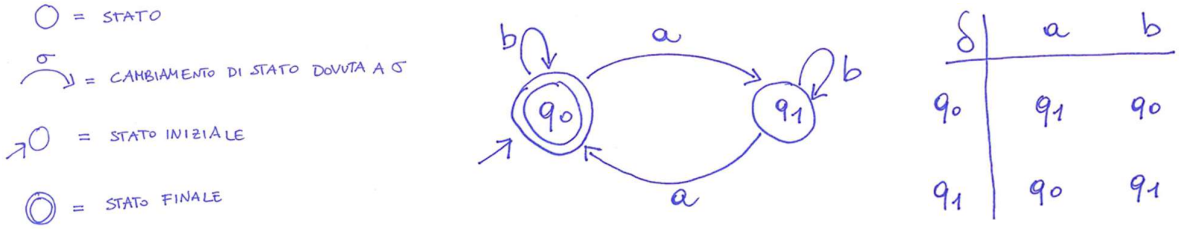
$\Sigma$ = alfabeto di input
Q = insieme degli stati, se Q è finito l'automa A è a stati finiti
$\delta$ = funzione di transizione, $\delta : Q \times \Sigma \rightarrow Q$
$q_0$ = stato iniziale, $q_0 \in Q$
$\lambda$ = funzione di uscita $\lambda$: $Q \rightarrow \{0,1\}$ oppure $F \subseteq Q$ che indica stati finali o accettanti, la macchina restituisce 1

**Osservazione**:

- data $\lambda$ possiamo avere F, $F = \{q \in Q \mid \lambda(q) = 1\}$
- data F possiamo avere $\lambda$, $\lambda(p) = 1$ se $p \in F$ altrimenti = 0 se $p \notin F$

**Rappresentazioni di $\delta$**:

- **Tabellare**: la funzione di transizione $\delta$ può essere rappresentata sotto forma di tabella che ha come campi sui nomi delle colonne gli input in ingresso e sui nomi delle righe gli stati possibili, sugli incroci degli stati e gli ingressi troviamo lo stato prossimo $\delta(q_i,a_j)$
- **Diagramma degli stati**: questa rappresentazione è grafica, formata dai seguenti simboli:




una parola è accettata se partendo da $q_0$, stato iniziale, induce un cammino che termina in uno stato finale

nel nostro esempio la parola bbb è accettata in quanto termina in $q_0$, la parola bab è rifiutata in quanto termina in $q_1$ e tutte le parole aaaa di lunghezza pari sono accettate

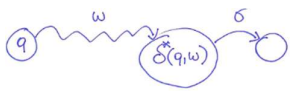
**Formalizziamo il concetto di linguaggio riconosciuto**

Introduciamo la $\delta^*$ che è la $\delta$ estesa alle parole invece che ai simboli, $\delta^*$: $Q \times \Sigma^* \rightarrow Q$, prende in input uno stato e una parola e da in output lo stato raggiunto da quella parola

**Definizione induttiva**:

$\delta^*(q, \varepsilon) = q$, la parola vuota è come non leggere alcun input infatti non cambia stato

$\delta^*(q, wa) = \delta(\delta^*(q,w),a)$ dove w parola appartenente a $\Sigma^*$ e a è un simbolo di input, in questa formula la parola viene analizzata simbolo per simbolo in modo ricorsivo partendo dall'ultimo



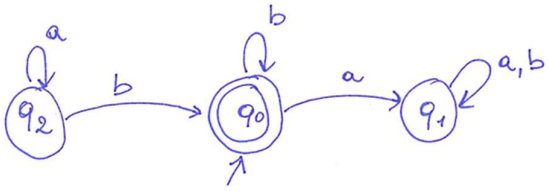

**L(A)** = linguaggio riconosciuto da A = $\{w \in \Sigma^* \mid \delta^*(q_0,w) \in F\} = \{w \in \Sigma^* \mid \lambda(\delta^*(q_0,w)) = 1\}$

Esempio: $L(A) = \{w \in \{a,b\}^* \mid |w|_a = 2n, n>0\}$ con $|w|_a$ andiamo ad indicare il numero di a in w

## Gli stati particolari

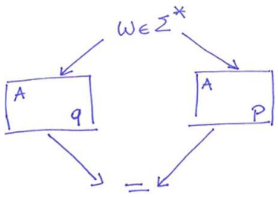
- **Stato trappola**: è uno stato nel quale una volta entrati non è più possibile uscirsi qualsiasi sia l'input e non ci da uno stato finale o di accettazione, possiamo definirlo cosi: se vale che per ogni a ∈ Σ, δ(q,a) = q e q ∉ F
- **Stato osservabile**: uno stato P si dice osservabile se esiste una parola che da $q_0$ mi porti a quel determinato stato, possiamo definirlo cosi: se vale ∃ w ∈ Σ* tale per cui δ*($q_0$,w) = P, altrimenti è detto stato non osservabile

**Esempio**:




$q_1$ è osservabile in quanto riceve "a" ed è anche trappola in quanto non è possibile uscirci e non è stato finale
$q_0$ è osservabile in quanto riceve "ε"
$q_2$ non è osservabile in quanto non ha un cammino da $q_0$ e si può eliminare

### Gli stati indistinguibili




La coppia di stati q e p, appartenenti alla stessa macchina, si dicono indistinguibili quando portano alle stesse uscite sulle stesse parole e cambia solamente lo stato iniziale, possiamo definirlo cosi: ∀ w ∈ Σ* si ha che λ(δ*(q, w)) = λ(δ*(p, w))
Lo stato finale possono essere anche due stati differenti che portano agli stessi risultati

### Gli stati distinguibili

La coppia di stati q e p, appartenenti alla stessa macchina, si dicono distinguibili quando esiste almeno una parola che data in ingresso ad entrambi gli stati ci porta a risultati differenti ed uno dei due non è stato finale, possiamo definirla cosi: ∃ w ∈ Σ* si ha che λ(δ*(q, w)) ≠ λ(δ*(p, w))

**Notazione**: stati indistinguibili q ≈ p e stati distinguibili q $\not\approx$ p

Possiamo dire che due stati, uno finale ed uno non finale possono essere sempre definiti distinguibili tramite la parola vuota in quanto fa tenere due comportamenti differenti agli stati, infatti, solitamente non si esegue questo confronto ma si confrontano solamente stati finali con stati finali e viceversa

## La riduzione degli stati

La relazione ≈ è una relazione di equivalenza ed è una relazione binaria su Q e gode delle seguenti proprietà:

- **Riflessiva**: ∀ q  q ≈ q
- **Simmetrica**: ∀ q,q'  q ≈ q' → q' ≈ q
- **Transitiva**: ∀ q, q', q''  q ≈ q' e q' ≈ q'' → q ≈ q''
- ∀ q, q'  q ≈ q' → ∀z ∈ Σ*  δ*(q,z) ≈ δ*(q',z)

## Partizione dell'insieme Q in classi di equivalenza

Una partizione di un insieme deve essere un sottoinsieme non vuoto, ∀i $C_i \neq \emptyset$, ciascuna partizione deve essere disgiunta dalle altre ∀i,j e i ≠ j l'intersezione tra $C_i \cap C_j = \emptyset$, l'unione di tutte le partizioni deve dare l'insieme di partenza $\cup_i C_i = Q$

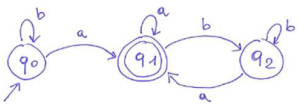
La classe di equivalenza viene cosi indicata [q]

**Notazione**: $[q]_{\approx}$ indica la classe di equivalenza che contiene l'elemento q

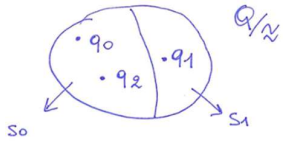
Dato un automa $A = (Q, \Sigma, q_0, \delta, F)$ posso costruire $A_{\approx}$ che ha come stati le classi di equivalenza $Q/\approx$

Formalmente: $A_{\approx} = (\Sigma, Q/\approx, [q_0]_{\approx}, \delta_{\approx}, F_{\approx})$ dove:

- $F_{\approx} = \{[p]_{\approx} \mid p \in F\}$ è l'insieme di tutte le classi di equivalenza che contengono stati finali (stati appartenenti all'insieme F dell'automa A), ossia quelle classi di equivalenza che, se raggiunte, mi consentono di ottenere uscita = 1;
- $\delta_{\approx} = Q/\approx \times \Sigma \rightarrow Q/\approx$ $\quad \delta_{\approx}([q]_{\approx}, a) = [\delta(q, a)]_{\approx}$




**Esempio**: Costruisco $A_{\approx}$ partendo da questo automa




Possiamo definire le seguenti classi di equivalenza, $q_0$ e $q_2$ in quanto indistinguibili formano una classe di equivalente mentre $q_1$ forma un'altra classe di equivalenza, quindi $Q/\approx$ ha i seguenti stati $\{s_0, s_1\}$

$A_{\approx}$ è cosi costituito: $(\Sigma = \{a,b\}, Q/\approx = \{s_0, s_1\}, [q_0]_{\approx} = S_0, \delta_{\approx}$ da costruire, $F_{\approx} = \{[p]_{\approx} \mid p \in F\} = \{[q_1]_{\approx}\} = \{s_1\}$

La tabella della funzione di transizione della nostra nuova macchina $A_{\approx}$:

| $\delta_{\approx}$ | a | b |
|---|---|---|
| $s_0$ | $s_1$ | $s_0$ |
| $s_1$ | $s_1$ | $s_0$ |

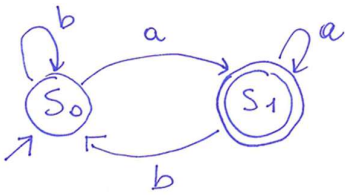
$\delta_{\approx}(s_0,a) = \delta_{\approx}([q_0]_{\approx},a) = [\delta(q_0,a)]_{\approx} = [q_1]_{\approx} = s_1$

$\delta_{\approx}(s_1,a) = \delta_{\approx}([q_1]_{\approx},a) = [\delta(q_1,a)]_{\approx} = [q_1]_{\approx} = s_1$

$\delta_{\approx}(s_0,b) = \delta_{\approx}([q_0]_{\approx},b) = [\delta(q_0,b)]_{\approx} = [q_0]_{\approx} = s_0$

$\delta_{\approx}(s_1,b) = \delta_{\approx}([q_1]_{\approx},b) = [\delta(q_1,b)]_{\approx} = [q_2]_{\approx} = s_0$

il suo grafico è il seguente:




**$L(A_{\approx}) = \{a,b\}^*a = L(A)$**

**Osservazione**: $A_{\approx}$ è equivalente ad A ma ha meno stati rispetto ad A, 3 stati di A contro i 2 stati di $A_{\approx}$

## Problema di sintesi di automi

Dato un linguaggio L in input dobbiamo trovare un automa A per L come output

In particolare, ci muoviamo nel seguente modo: andiamo a trovare un automa massimo per L e da questo automa cerchiamo di ottenere l'automa minimo per L, per **automa massimo** viene inteso un automa con il maggior numero di stati e quindi non ottimale, per **automa minimo** viene inteso un automa con il minor numero di stati possibili e solitamente è il nostro obbiettivo

## L'Automa massimo $G_L$

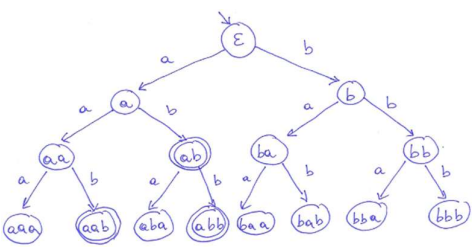
L'idea è quella che per ogni parola in input andiamo a raggiungere uno stato sempre diverso, ciò genera un automa infinito, per esempio $\forall w, w' \in \Sigma^* \rightarrow \delta(q_0,w) \neq \delta(q_0,w')$ da questo fatto possiamo dedurre che l'insieme degli stati Q equivale a $\neq \Sigma^*$, inoltre per non perdere stati, tutti gli elementi di Q devono essere osservabili quindi $\forall\, q \in Q$ esiste w tale che $q = \delta(q_0,w)$

**Definizione**: $G_L = (\Sigma^*, \Sigma, [\varepsilon], \delta_{GL}, F_{GL})$ dove $F_{GL} = \{[w] \in Q \mid w \in L\}$ e $\delta_{GL}([w],a) = [wa]$ $\quad Q = \Sigma^*$, $[w]$ = stato

Un automa massimo è equivalente per ogni linguaggio, cambiano solamente gli stati finali per via di $F_{GL}$, la funzione di transizione è indipendente dal linguaggio L

### Esempio di automa massimo per $L = \{a^n b^m \mid n,m > 0\}$

Questo esempio è un automa infinito in quanto possiamo valutare infiniti ingressi, gli stati finali, evidenziati con un doppio cerchio, sono gli stati finali validi per il nostro linguaggio L, alcuni rami non genereranno mai parole appartenenti al linguaggio, questi stati possono diventare trappola



## L'automa minimo $M_L$

Per ottenere un automa minimo abbiamo due tecniche:

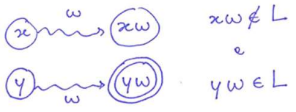
- Si usa l'automa massimo $G_L$
- Si usa un generico automa a stati finiti A per L

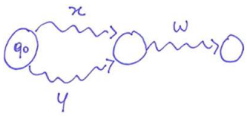
### Prima tecnica, automa massimo

**Teorema**: $G_L{\approx}$ è l'automa minimo per L, cioè $G_L{\approx} = M_L$

**Dimostrazione**: dobbiamo mostrare che, se A è un automa per L allora $|A| >= |G_L{\approx}|$
$|A|$ indica il numero di stati di A

**Fatto**: prendendo stati distinguibili in $G_L$ questi devono rimanere distinti in A, infatti: siamo [x] e [y] distinguibili in $G_L$ cioè $[x] \not\approx [y] \rightarrow \exists\, w \in \Sigma^*$ tale che partendo da x → xw non è stato finale quindi non accetto mentre con y → yw vado in uno stato finale




Allora in A: $\delta(q_0,x) \neq \delta(q_0,y)$ altrimenti partendo dallo stato iniziale e ricevendo x o y si raggiunge lo stesso stato e quindi ricevendo w si ottiene lo stesso stato, quindi se questo stato è finale quindi xw, yw ∈ L oppure non è finale e quindi xw, yw ∉ L

Supponiamo che esistano le seguenti classi di equivalenza $[x_1]_{\approx}$, $[x_2]_{\approx}$, $[x_3]_{\approx}$, …, come stati di $G_L{\approx}$, allora $[x_1]$, $[x_2]$, $[x_3]$, …, sono stati di $G_L$ tutti distinguibili tra di loro

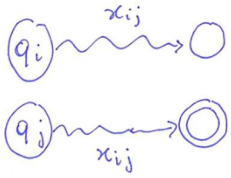
In virtù del fatto appena dimostrato $\delta(q_0,x_1)$, $\delta(q_0,x_2)$, $\delta(q_0,x_3)$, …, sono tutti stati distinti in A, dove A è un generico automa per L, quindi A ha almeno tanti stati quanti $G_L{\approx}$, cioè $|A| >= |G_L{\approx}| \rightarrow G_L{\approx}$ è minimo per L

### Seconda tecnica

**Teorema**: sia A un automa a stati finiti per L tale che gli stati di A siano tutti osservabili e siano tutti distinguibili, allora A è automa minimo per L

**Corollario**: $A_{\approx}$ è minimo per L se A ha tutti stati osservabili

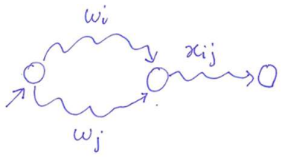
**Dimostrazione**: $\{q_1, q_2, q_3, \ldots, q_n\}$ sono gli stati di A con $q_1$ stato iniziale




- Sono osservabili, allora esistono parole $\{w_1, w_2, \ldots, w_n\}$ tali per cui ognuno degli stati è raggiunto da una di queste parole, $\forall i\ q_i = \delta(q_1,w_i)$
- Sono distinguibili, allora esistono parole $x_{i,j}$ che distinguono gli stati, quindi esiste una parola per stati diversi da un risultato diverso, uno finale e uno non, vedi immagine

Adesso sia A' un automa per L con meno stati di A quindi minore di "n", diamo in input ad A' le parole $\{w_1, w_2, \ldots, w_n\}$ si ha un assurdo in quanto A' non riconosce il linguaggio L, in quanto almeno due parole raggiungono lo stesso stato in quanto stiamo leggendo "n" parole ma abbiamo meno stati delle parole

Siano $w_i$ e $w_j$ queste due parole allora in A' otteniamo questo risultato:




così otteniamo che $w_i$ $x_{ij}$ e $w_j$ $x_{ij}$ o entrambe appartengono ad L oppure entrambe non appartengono a L, in quanto si ottiene solo uno stato finale o no, quindi siamo arrivati ad un assurdo e quindi A' non riconosce L e quindi non esiste un automa con meno stati di A

## Algoritmi di sintesi ottima di automi

Il problema della sintesi ottima di automi, supponiamo di avere un linguaggio L e di dover trovare l'automa minimo per L, ora abbiamo due tecniche per trovarlo:

- Se ho A per L, A ha stati finiti, posso eliminare gli stati non osservabili e costruire $A_{\approx}$
- Se non ho A per L dobbiamo costruire l'automa massimo $G_L$ e costruire $G_L{\approx}$ che equivale all'automa minimo $M_L$

Possiamo stabilire due algoritmi, per la prima tecnica è possibile confrontare gli stati seguendo un ordine casuale, mentre per la seconda tecnica devo applicare un algoritmo sui confronti ben preciso

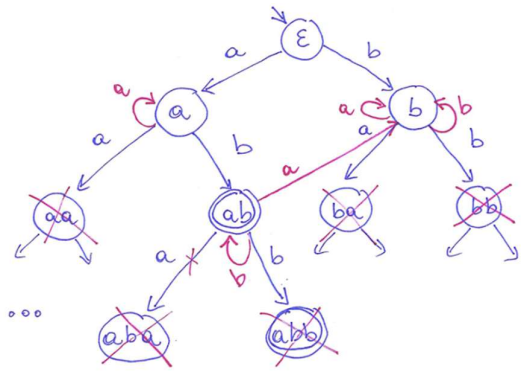
### Algoritmo per costruire l'automa minimo $G_L{\approx}$

Partiamo dalla radice e visitiamo i vari nodi, stati, in ampiezza, la visita consiste nel confrontare il nodo attuale con tutti i nodi precedenti visitati e verificando se i nodi sono indistinguibili o no

Supponendo che [wa], con $w \in \Sigma^*$ e $a \in \Sigma$, sia il nodo attuale, se accade che [wa] è indistinguibile dal nodo [x] allora cancello [wa] e il suo sottoalbero e resta pendente l'arco uscente da [w] etichettato con "a" e lo ridireziono verso [x] in quanto quello che poteva fare [wa] lo può fare il nodo [x] essendo equivalenti

**Osservazione**: lo stato iniziale resta [ε] e gli stati finali sono i sopravvissuti etichettati con parole di L

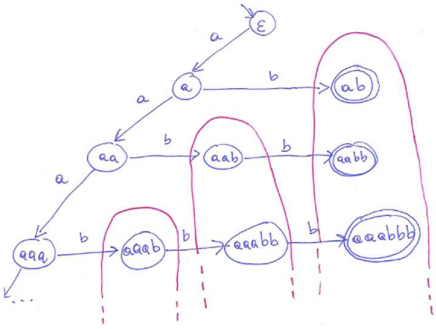
**Costruzione di $G_L{\approx}$** per $L = \{a^n b^m \mid n,m > 0\}$, applichiamo l'algoritmo ed effettuiamo i confronti



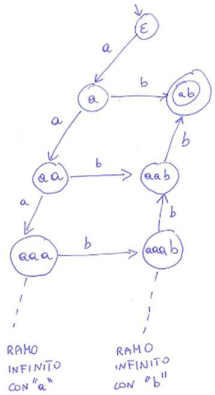

[ε] resta,
[a] $\not\approx$ [ε] a causa della parola "b"
[b] lo confronto con [ε] ed [a], [b] $\not\approx$ [ε] a causa della parola "ab" e [b] $\not\approx$ [a] a causa della parola "b"
[aa] lo confronto con [ε], [a] e [b], [aa] $\not\approx$ [ε] a causa della parola "b" che applicata ad [aa] porta ad uno stato finale, [aa] ≈ [a] è indistinguibile con la parola a
[ab] è uno stato finale
[ba] ≈ [b] e [bb] ≈ [b] sono indistinguibili da [b] in quanto portano a parole non accettate

[aba] ≈ [b] in quanto generano entrambi stati non accettanti e [abb] ≈ [ab] tengono lo stesso comportamento essendo entrambi stati finali, possiamo dire che [b] è uno stato trappola
Avendo trovato un automa minimo per il linguaggio L possiamo dire che è un linguaggio regolare

**Costruzione di $G_L{\approx}$** per $L = \{a^n b^n \mid n > 0\}$




In questo schema ci sono degli archi mancanti in quanto generano stati trappola, tipo [ε] che riceve b, lo stato [b] è trappola, evidenziamo solo gli stati che portano a stati finali, gli stati evidenziati in rosso nello stesso cerchio sono indistinguibili tra di loro, quindi possiamo ottenere questo automa ottimizzato dove tutti gli stati portano allo stato finale [ab], questo automa ha due rami infiniti quindi $G_L{\approx}$ ha infiniti stati e quindi $a^n b^n$ non è regolare




**Dimostrazione**: l'automa minimo $G_L{\approx}$ per $a^n b^n$ ha infiniti stati, allora $a^n b^n$ non ammette un automa a stati finiti e data l'equivalenza tra gli automi a stati finiti e grammatiche di tipo 3 possiamo dire che non è regolare

**Teorema equivalenza tra automi a stati finiti e linguaggi regolari**

L generato da una grammatica G di tipo 3 implica che L è riconosciuto da A a stati finiti e viceversa

Dimostrazione: da A andiamo a ricavare G, sia A = (Σ, Q, $q_0$, δ, F) posso definire G = (Σ, Q, $q_0$,
nella grammatica come simboli terminali andiamo a scegliere alfabeto dell'automa, come variabili l'insieme degli stati, come assioma lo stato iniziale e come regole q → ε se e solo se lo stato q è finale, q ∈ F e q → ap se e solo se δ(q,a) = p, a = simbolo terminale

**Correttezza della costruzione**:

andiamo a verificare che la grammatica di tipo 3 è in grado di generare l'automa a stati finiti

Proprietà: δ($q_0$,w) = p se e solo se $q_0$ ⇒* wp, la funzione di transizione è strettamente legata alla regola di derivazione in più passi

**Andiamo a dimostrarlo per induzione**:

caso base: δ($q_0$, ε) = $q_0$ questo vale se e solo se $q_0$ ⇒* ε · $q_0$ = $q_0$, il caso base è verificato

passo induttivo: suppongo vera la proprietà per una parola di lunghezza n e lo dimostro per una parola di lunghezza n+1, andiamo a considerare wa con |w| = n e a ∈ Σ e quindi |wa| = n+1

δ($q_0$,wa) = p, possiamo spezzarla nel seguente modo: δ(δ($q_0$,w),a) = p, come passaggio successivo andiamo a dare un nome allo stato δ($q_0$,w) = q e per costruzione, derivato dalle funzioni di prima, δ(q,a) = p, ora possiamo dire per ipotesi che nella grammatica arriviamo ad $q_0$ ⇒* wq e per costruzione diciamo q ⇒ ap, combiniamo queste due regole $q_0$ ⇒* wap

andiamo a verificare che la **grammatica e l'automa sono equivalenti L(A) = L(G)**

per dimostrare un uguaglianza tra due L devo partire da un elemento di un L e vedere se sta nell'altro insieme

w ∈ L(A) questo vuol dire che ∃ q ∈ F e δ($q_0$,w) = q, data questa transizione esistono le regole di produzione q ⇒ ε e $q_0$ ⇒* wq, sappiamo che q può sparire applicando la regola per ε e quindi arriviamo a $q_0$ ⇒* w e quindi possiamo dire che w ∈ L(G) e quindi la grammatica e l'automa sono equivalenti

**dimostrazione da G di tipo 3 ad A**: dobbiamo invertire la costruzione precedente ma G deve essere nel formato A →ε e A → aB

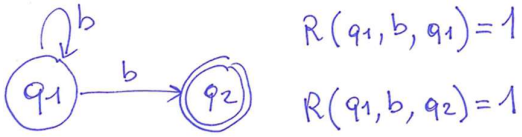
Data G = <T, V, S, P> definisco A = (Σ = T, Q = V, $q_0$ = assioma, δ: δ(A, a) = B se e solo se A → aB,
F = {A | A → ε ∈ p})

Possiamo notare che A non è un automa deterministico (DFA) ma è non deterministico (NFA) in quanto partendo da uno stato abbiamo due archi uguali che vanno in due stati differenti

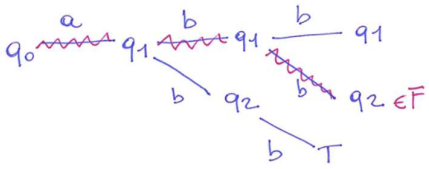
**Automi a stati finiti non deterministici (NFA)**

**Definizione**: un NFA è un sistema formato dalla seguente tupla A = (Σ, Q, $q_0$, R, F) dove R è detta relazione di transizione rappresentata da R = Q x Σ x q → {0,1}, per esempio R(q,σ,p) è uguale ad 1 se q tramuta in p grazie ad σ, altrimenti se manca l'arco è uguale a 0

L'automa A sceglie non deterministicamente di transitare in q1 o q2 all'ingresso di b



**Osservazione**: data una parola w in input ad A è possibile che induca più di un cammino possibile, per esempio in caso di abb possiamo vedere più cammini, in totale possiamo avere tre cammini



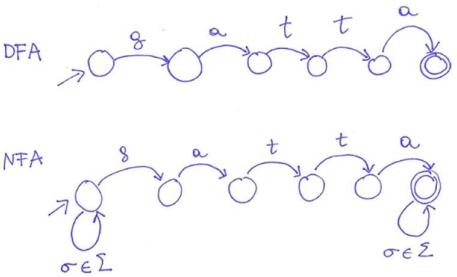

**Definizione di linguaggio riconosciuto da un NFA A**:

- w è accettata se ammette un cammino che porta in uno stato finale
- Il linguaggio riconosciuto è dato dalle parole che soddisfano il criterio di accettazione

**L(DFA) = L(NFA)**

**Semplici applicazione di questi automi**:

- Tramite un DFA è facile riconoscere una parola
- Tramite un NFA è facile riconoscere testi che contengono una certa parola




In questo esempio possiamo vedere come un DFA ha gli archi precisi per rilevare la parola gatta, mentre un NFA è molto simile solamente che allo stato di partenza e allo stato finale ha un arco rientrante nello stato attuale valido per ogni lettera appartenente all'alfabeto, per esempio "la gatta" è accettato mentre "il gatto" non è accettato perché non è presente l'arco che porta allo stato finale, per la frase "il gatto della gatta" viene accettata in quanto un simbolo rimane sempre nello stato iniziale, cosi possiamo ottenere un cammino differente che porta allo stato finale

**L(DFA)** = la classe dei linguaggi accettati da DFA

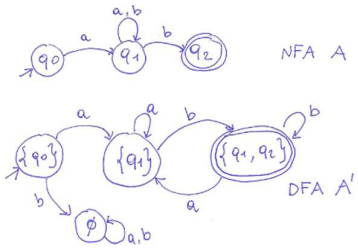
**L(NFA)** = la classe dei linguaggi accettati da NFA

Possiamo dire che un L(DFA) è un sottoinsieme di un L(NFA), in quanto i DFA sono un caso particolare dei cammini di un NFA, quindi possiamo dire il seguente **teorema**: per ogni L riconosciuto da un NFA esiste un DFA che lo riconosce, in quanto DFA e NFA sono equivalenti

**Corollario**: grammatica di tipo 3: $R_3$ = L(NFA) = L(DFA), tutti generano linguaggi regolari

**Dimostrazione del teorema**: dato A = (Σ, Q, $q_0$, R, F) NFA andiamo a costruire A' deterministico

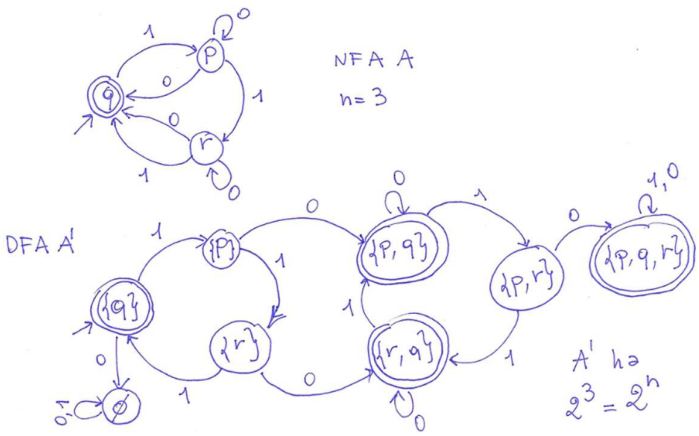
**A'** = $(\Sigma, 2^Q, \{q_0\}, \delta_R, F')$, $2^Q$ è l'insieme dei sottoinsiemi di Q, dove $F' = \{Y \in 2^Q \mid Y \cap F \neq \emptyset\}$, $\delta_R = 2^Q \times \Sigma \rightarrow 2^Q$ che soddisfa $\delta_R(X, a) = \cup$ dei $q \in X \{p \in Q \mid R(q, a, p) = 1\}$, quindi dato un elemento dell'alfabeto per il sottoinsieme di stati abbiamo un nuovo stato, graficamente lo possiamo vedere cosi:




in pratica abbiamo aggiunto uno stato trappola vuoto quando dallo stato iniziale si riceve "b" in quanto nel A NFA non è presente alcun ramo, poi lo stato $q_1$ è simile tra entrambi gli automi tranne per la "b" che in questo caso l'automa DFA passa allo stato $\{q_1,q_2\}$ dato che l'automa NFA ha due archi che rappresentano "b" in $q_1$, uno che porta a se stesso e altro allo stato $q_2$, che è stato finale, da $\{q_1,q_2\}$ abbiamo anche un arco "a" che torna indietro in quanto $q_1$ riceveva "a"

### Trasformazione di un NFA in DFA con incremento esponenziale degli stati

Quando si procede alla trasformazione di un NFA ad un DFA si ottiene un incremento esponenziale degli stati in quanto gli stati sono rappresentati da $2^Q$, dove Q è il numero di stati dell'automa NFA




in questo esempio il nostro automa NFA ha 3 stati, come possiamo notare trasformandolo in un automa DFA si avranno 8 stati che equivale a $2^3$

possiamo notare che data una parola w appartenente a Σ*, il percorso nel DFA indotto da w simula tutti i cammini possibili di w nel NFA

CAPIRE SLIDE 60 Cap 3

### Le espressioni regolari ER

**Definizione**: una espressione regolare su Σ è: Ø, ε e a ∈ Σ, queste sono dette espressioni regolari base, mentre se p e q sono espressioni regolari possiamo formare le seguenti espressioni regolari complesse p+q, p·q e p*

Le espressioni regolari denotano dei linguaggi per esempio:

| Espressione regolare | Linguaggio |
|---|---|
| Ø | Ø |
| ε | {ε} |
| σ | {σ} |
| p+q | $L_p \cup L_q$ |
| p·q | $L_p \cdot L_q$ |
| p* | $L_p$* |

Alcuni esempi di applicazione delle espressioni regolari:

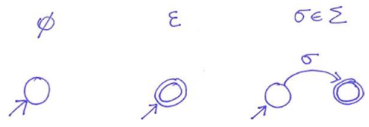
- (a+b)* · a denota {a,b}*·{a}
- Gooo*gle denota {google, gooogle, goooogle, …}, o* comprende la ε
- Identificatori di variabili (A+B+ …+ Z+ a+ …+ z) · (A+B+… + Z + a + … + z + 0 + … + 9)*
- Importi in euro (1+2+…+9) · (0+1+…+9)*,(0+1+…+9)$^2$

### Teorema di Kleene

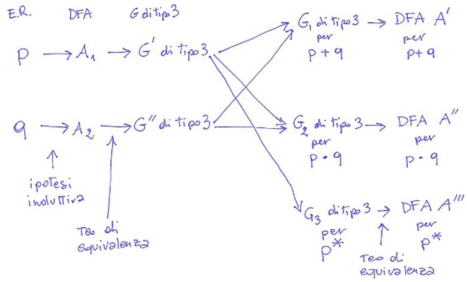
Il teorema di Kleene dice che, se L è denotato da una espressione regolare allora questo linguaggio L è riconosciuto da un automa DFA

**Dimostrazione**: L denotato da ER → L ammette un DFA, andiamo a dimostrarlo per induzione

Caso base: esistono DFA per le ER base: l'automa che riconosce Ø è uno stato trappola che non riconosce alcuna parola, l'automa che riconosce ε è uno stato trappola finale che riconosce e accetta solo ε, mentre l'ultimo ha uno stato finale raggiungibile da σ



Passo induttivo: se esistono DFA per ER p e q allora dimostriamo che esistono DFA per p+q, p·q e p*, in realtà useremo le grammatiche di tipo 3, essendo equivalenti agli automi

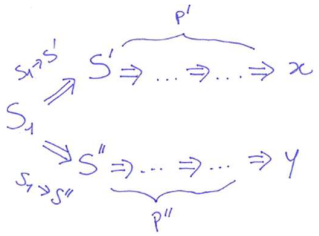



Supponendo per ipotesi induttiva che dati p e q esistono due automi $A_1$ e $A_2$ per p e q, per il teorema di equivalenza esistono due grammatiche di tipo 3, G' e G'', associati agli automi di p e q, da queste grammatiche andiamo a costruire $G_1$ di tipo 3 per p+q, $G_2$ di tipo 3 per p · q e $G_3$ di tipo 3 per p* e grazie al teorema di equivalenza ad ognuna di queste possiamo associare un automa DFA A

Abbiamo due grammatiche di tipo 3, con le seguenti regole A → σB e A → ε, dentro queste grammatiche non troviamo la regola A → σ, σ come simbolo terminale e B come variabile

Definiamo le seguenti grammatiche G' = (T', V', S', P') e G'' = (T'', V'', S'', P'') che generano i seguenti linguaggi L(G') = { x ∈ T'* | S' ⇒* x} e L(G'') = {y ∈ T''* | S'' ⇒* y} con T' = T'' = Σ, le parole di tipo x stanno nel linguaggio generato da L(G') e y stanno nel linguaggio generato da L(G'')

Andiamo a costruire $G_1$ che deve generare il linguaggio unione tra L(G') ∪ L(G'') cioè L(G') ∪ L(G'') = {w ∈ Σ* | w ∈ L(G') o w ∈ L(G'')}

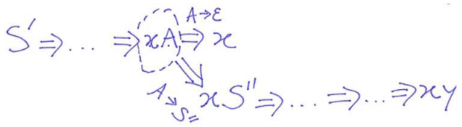



L'idea è che deve essere una grammatica in grado di generare sia parole di tipo x che di tipo y, per generare le parole di tipo x ho bisogno delle regole di produzione di G' e per le parole di tipo y ho bisogno delle regole di produzione G'', stabiliamo un nuovo assioma $S_1$ che è in grado di raggiungere i due assiomi delle due grammatiche aggiungendo le nuove regole $S_1$ → S' e $S_1$ → S'', la richiesta è che l'intersezione tra le variabili V' e V'' sia l'insieme vuoto e che quindi gli insiemi di variabili sia disgiunti in modo di non creare errori nella generazione di parole

Possiamo definire **$G_1$** così: $G_1$ = (Σ, V' ∪ V'' ∪ {$S_1$}, $S_1$, P' ∪ P'' ∪ {$S_1$ → S', $S_1$ → S''}

$G_1$ è di tipo 3 in quanto lineare a destra: A → xB e A → y con simboli terminali x,y ∈ Σ*

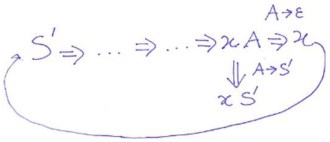
Andiamo a costruire $G_2$ che deve generare il linguaggio prodotto tra L(G') · L(G'') cioè L(G') · L(G'') = {xy | x ∈ L(G') e y ∈ L(G'')}
l'idea è quella di utilizzare S' per generare le parole di tipo x e tramite S'', generato da una regola che parte da S', andare a generare le parole di tipo y, non dobbiamo avere la regola che dà tipo x genera epsilon




Per esempio: A → aB, B → bC, C → cD e D → ε e quindi partendo da A ⇒ aB ⇒ abC ⇒ abcD ⇒ abc

Possiamo definire **$G_2$** cosi: $G_2$ = (Σ, V' ∪ V'', S', P' \ {a → ε | A ∈ V'} ∪ P'' ∪ {A → S'' | A → ε ∈ P'}) anche questa grammatica è di tipo 3

Andiamo a costruire $G_3$ che deve generare il linguaggio di L(G')*



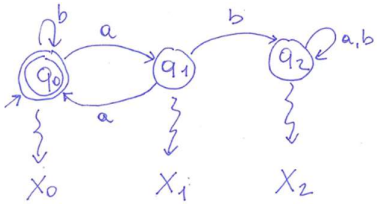

L(G')* = ⇒ U da i = 0 a infinito di $L(G')^i$ = {ε} ∪ $L(G')^+$, andiamo a creare un ciclo di generazione che data la parola finale di tipo xA tramite la regola A → S' otteniamo xS' dalla quale possiamo riniziare l'applicazione delle regole di S' e con A → ε andiamo a terminare la generazione

Possiamo definire **$G_3$** così: $G_3$ = (Σ, V', S', P' ∪ {A → S' | A → ε ∈ P'}) questa è una grammatica per $L(G')^+$ ma a noi serve includere anche ε quindi definiamo le seguenti regole se ε ∉ $L(G')^+$: $S_3$ →ε e $S_3$ → S', $S_3$ nuovo assioma al posto di S' in questo caso

Ora vediamo come possiamo ricavare un'espressione regolare partendo da un automa a stati finiti

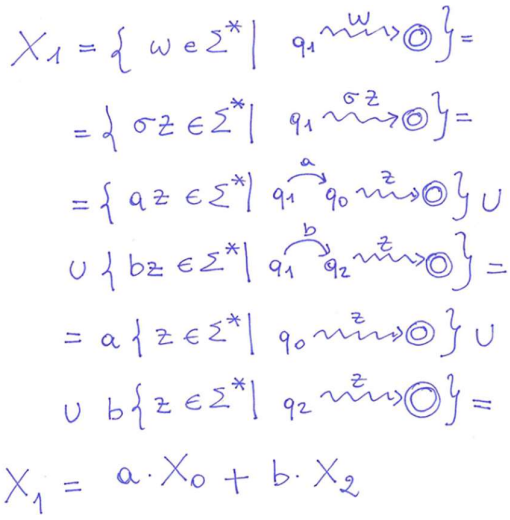
**Dimostrazione**: dato un automa DFA possiamo ricavare una espressione, dato il DFA $A = (\Sigma, Q, q_0, \delta, F)$ e sia $Q = \{q_0, q_1, q_2, \dots, q_k\}$ allora posso associare ad automa A i seguenti DFA per ogni stato iniziale:

$A_0 = (\Sigma, Q, q_0, \delta, F)$ $A_1 = (\Sigma, Q, q_1, \delta, F)$ … $A_k = (\Sigma, Q, q_k, \delta, F)$




Ognuno di questi DFA riconosce un linguaggio presumibilmente diverso, possiamo dire che $X_i$ sia il linguaggio riconosciuto da $A_i$ quindi $L(A) = L(A_0) = X_0$

Ora andiamo a cercare di esprimere ogni linguaggio in funzione degli altri, per esempio per ricavare l'espressione regolare di $X_0$ andiamo a cercare una espressione regolare per ogni $X_i$




possiamo definire $X_1$ come un linguaggio che data una w in ingresso questa lo porta ad uno stato finale, possiamo definire w come σz e dire che dato un σz dobbiamo arrivare ad uno stato finale, le due parole che σ può rappresentare sono "a" o "b", la prima manda in $q_0$ la seconda in $q_2$ e successivamente leggendo z arriviamo in uno stato finale, ora da entrambi possiamo tirare fuori l'elemento in comune, nel primo è "a" e quindi diciamo che data z passiamo da $q_0$ ad uno stato finale, questo è definibile come $X_0$, ci troviamo in $q_0$ dato che partiamo da "a" che da $q_1$ ci ha portato in $q_0$, nel secondo è "b" che dato z ci permette di passare da $q_2$ ad uno stato finale, questo è definibile $X_2$
quindi arriviamo alla conclusione che $X_1 = a \cdot X_0 + b \cdot X_2$
in questo esempio manca ε in quanto $q_1$ non è stato finale

in generale possiamo dire che $X_i = + \sigma X_j$, andiamo ad aggiungere ε solo se $q_i$ è uno stato finale
+ = sommatoria su tutti i sigma appartenenti all'alfabeto sigma tale che $\delta(q_i, \sigma) = q_j$

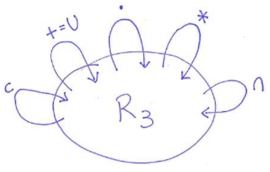
$X_0 = aX_1 + bX_0 + \varepsilon$
$X_1 = aX_0 + bX_2$
$X_2 = aX_2 + bX_2$

in conclusione, ricaviamo un sistema con k+1 equazioni, dato $q_0$, in k+1 incognite $X_i$
per andare a risolvere il seguente sistema ho bisogno di rifarmi ad un'equazione tipo $X = AX + B$ la cui soluzione è $X = A * B$, se $\varepsilon \notin A$ allora la soluzione è unica se $\varepsilon \in A$ allora $A^*B$ è la minima soluzione, A e B sono linguaggi/espressioni regolari e X è un incognita

RIVEDERE ESEMPIO DI APPLICAZIONE DELLA FORMULA

## La chiusura dei linguaggi regolari

Dato un linguaggio regolare se effettuiamo una delle operazioni indicate nello schema otteniamo nuovamente un linguaggio regolare, in caso di linguaggio complemento basta invertire gli stati finali con i non finali, in caso di intersezione abbiamo $((A \cap B)^c)^c = (A^c \cup B^c)^c$




## Linguaggi liberi dal contesto e grammatiche di tipo 2

### Il problema dell'ambiguità

**Definizione di G ambigua**: una grammatica è ambigua quando genera una parola ambigua, cioè quando la parola ammette due alberi di derivazione diversi

Nota: l'albero di derivazione dà significato alla parola

**Esempio di grammatica ambigua**:

$P \rightarrow i(x)\ C\ o(x)$

$C \rightarrow A \mid I \mid E$

$A \rightarrow x = -1 \cdot x$

$I \rightarrow \text{if } (T)\ C$

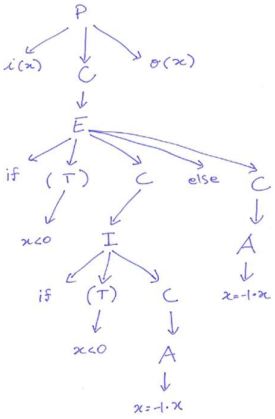
$E \rightarrow \text{if } (T)\ C \text{ else } C$

$T \rightarrow x < 0$

in questa grammatica P indica programma che viene espanso input(x) Comando C e output(x)
C è la lista di comandi dove A sta per assegnamento, I per "if" e E per "if else", con A possiamo assegnare x ad -1·x, quindi il cambio di segno, con I effettuiamo if (T) con il seguente comando C se soddisfatto, con E effettuiamo lo stesso if ma con l'aggiunta dell'else, T è uguale a x < 0

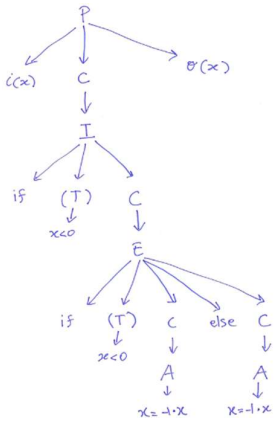
questa grammatica genera la seguente parola:
i(x) if(x<0) if(x<0) x = -1x else x = -1x σ(x), non è possibile stabilire se il primo if o se il secondo fa riferimento alla prima o seconda istruzione di assegnamento
la prima scelta è associare "a" al primo if con l'else e la seconda scelta di associare else al secondo if ad "b"

vediamo **l'albero di derivazione** di "a", else associato all'if più esterno




il programma si espande in input(x), comando e output(x), in questo caso il comando è un if else "E" che a sua volta si espande in if ed else, avremo cinque nuovi rami if, test(T), C, else e C, il primo test si espande in x < 0, il primo ramo C si espande in "I", cioè if, test e comando, dove il test si espande in x < 0, le ultime due istruzioni di comando si espandono in assegnamento (A) dove effettuano l'associamento x = -1 · x
vediamo cosa succede se passiamo una x positiva ad "a", l'if non viene verificato quindi eseguiamo il ramo else, dove viene eseguita l'associazione A, x = -1 · x, quindi la x diventa negativa, se viene passata una x negativa andiamo a finire nel ramo "if", ritesta nuovamente se la x è negativa e viene effettuata l'associazione A, x = -1 · x

vediamo l'**albero di derivazione** di "b", else associato all'if più interno




il programma si espande in input(x), C e output(x), il ramo C si espande in un "I", cioè, if creando i seguenti rami if, (T) e C, T esegue il test su x<0 mentre il comando C si espande in un else "E", dove genera i seguenti rami if, (T), C, else, C, il ramo T esegue il test su x < 0 mentre i due comandi C eseguono l'assegnazione A di x = -1 · x
Se passiamo una x positiva, viene eseguito il comando if che verifica se x è minore di 0, in questo caso no e non essendoci il ramo than viene eseguito direttamente output di x quindi x rimane inalterata, mentre se x è negativa l'if viene verificato e quindi viene eseguito il comando if-else, dove riviene testata se x < 0, in quanto vero viene effettuata l'assegnazione A x = -1 · x, quindi viene cambiato il segno ad x e diventa positiva

la funziona calcolata dai programmi è la seguente:

**$F_a(x)$** = -x if x >= 0 e -x if x < 0 quindi -x, in quanto effettua sempre il cambio di segno

**$F_b(x)$** = x if x >= 0 e x if x < 0 quindi |x|, viene sempre restituito il valore assoluto di x

Possiamo concludere che il programma è ambiguo, in quanto non si riesce a stabilire else a quale if è associato, e la grammatica G è ambigua

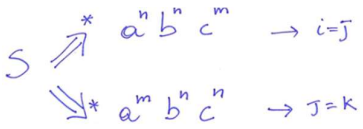
**Definizione**: una grammatica G non è ambigua se ogni parola generata non è ambigua

**Osservazione**: a volte è possibile disambiguare una grammatica, nel nostro caso si può usare una convenzione l'else è associato all'if più vicino, ma non sempre è possibile disambiguare una grammatica

**Definizione**: un linguaggio si dice inerentemente ambiguo quando ogni G che lo genera è ambigua

**Esempio**: il seguente linguaggio ambiguo $\{a^i b^j c^k \mid i = j \text{ oppure } j = k\}$, questo linguaggio è di tipo 2

**Idea**: partendo da S abbiamo una derivazione in più passi per le parole della forma $a^n b^n c^m$ in quanto i = j e una seconda derivazione in più passi per le parole della forma $a^m b^n c^n$ in quanto j = k, quindi possiamo dire che questo linguaggio per le parole $a^n b^n c^n$ ammetterà due alberi di derivazione diversi in quanto soddisfa le proprietà di entrambi i rami di derivazione




Possiamo dire che un linguaggio regolare non può essere inerentemente ambiguo, in quanto da un linguaggio regolare possiamo sempre ottenere una grammatica che per ogni parola ammetta solamente un ramo di derivazione

**Dimostrazione**: ad ogni linguaggio L regolare corrisponde un DFA, da tale DFA si ricava una grammatica G di tipo 3 che non è ambigua, in quanto in un DFA per ogni parola esiste solamente un percorso che l'accetta, quindi se c'è un unico percorso vuol dire che c'è solamente una derivazione

Dato un albero di derivazione io ho più derivazione associate in base alle espansioni effettuate per ogni livello, in base a questo ordine abbiamo diverse derivazioni, quando imponiamo la regola di espandere sempre la derivazione più a sinistra viene definita **derivazione leftmost**, ad esempio:
Data la parola abbAaBC, espandere prima il metasimbolo più a sinistra si espande A che se genera qualche altro metasimbolo si continua per quella strada, altrimenti si procede a quello più a destra, facendo cosi avremo una associazione univoca di derivazione leftmost

Una parola si dice ambigua se è generata da più di una derivazione leftmost

**Forme normali per G di tipo 2, senza ε**

Una grammatica di tipo 2 può essere sempre trasformata in una forma normale che può essere una forma normale di Chomsky, FNC, oppure una forma normale di Greibach, FNG, a seconda di come è fatto il beta

Nella **forma normale di Chomsky** il beta è fatto da due variabili, A → BC o un simbolo terminale A→ σ dove A, B, C ∈ V e σ ∈ T

Nella **forma normale di Greibach** il beta è formato da una variabile e un simbolo terminale, A → σW dove A ∈ V, W ∈ V* e σ ∈ T

Esempio: data la grammatica G = {{x, y, +, (, )}, {E}, P, E} dove P = {E → (E + E), E → x, E → y}

Andiamo a ricavare le due forme normali, FNC e FNG, da questa grammatica

FNG:
E → x  questa regola è ammessa in quanto già nella forma corretta
E → y  anche questa è già nella forma corretta
E → (E + E)  questa regola non è in forma normale di greibach in quanto troviamo un simbolo terminale e una variabile, seguita da un simbolo terminale, cosa non ammessa in forma di greibach, quindi dobbiamo scomporla nelle seguenti regole: E → (E P E D dove P → + e D → ), in questo modo soddisfiamo i requisiti della forma di greibach

FNC:
E → x  è ammessa in quanto porta ad un simbolo terminale
E → y  vale la stessa cosa della regola di prima
E → (E + E), non è ammessa in quanto porta ad un misto tra variabili e simboli terminali che non sono ammessi dalla forma normale di Chomsky, quindi dobbiamo scomporla cosi: E → AEPED, dove A → (, P → + e D → ), possiamo accoppiare le variabili AE con X e PE con Y, in questo ricaviamo X → CE, Y → PE ed E → XYD che non è ammessa in quanto costituita da tre variabili, a questo punto accoppiamo XY in Z e diciamo Z → XY e E → ZD, quest'ultima rispetta tutte le regole della forma di Chomsky

## I riconoscitori a pila per i linguaggi di tipo 2

Questi riconoscitori sono composti da due parti hardware, il primo è un nastro di input contenente la parola a lunghezza finita, simbolo per simbolo, che deve essere controllata e una testina che scandisce la parola da sinistra a destra, la seconda parte di hardware è costituita da una memoria, tendenzialmente una pila dove gli elementi sono salvati uno in testa all'altro, la pila costituisce un alfabeto, quando si accede ad una pila si accede all'elemento più in alto, la lettura di un elemento comporta la sua cancellazione

**Definizione di pila**: è una memoria ad accesso limitato con una politica di accesso LIFO, last in first out

**Notazione**: quando è possibile la definiamo tramite disegno, altrimenti possiamo scriverla ZYX dove il primo carattere sta per l'elemento in cima detto top e l'ultimo per quello in fondo

## Le operazioni di modifica della pila

**Push**: tramite l'operazione di push andiamo ad inserire un elemento in cima alla pila

**Pop**: tramite l'operazione di pop andiamo ad estrarre l'elemento in cima alla pila

**Formalizziamo**: PUSH(P,X) = XP e POP(P) = W se P = XW, indeterminato se P = ε

## Le operazioni di interrogazione della pila

**Top**: tramite l'operazione di top andiamo a leggere l'elemento in cima alla pila

**IsEmpty**: tramite l'operazione di IsEmpty possiamo chiedere se la pila è vuota

**Formalizziamo**: TOP(P) = X se P = XW o indeterminato se la pila è vuota
ISEMPTY(P) = restituisce 1 se la pila è vuota altrimenti restituisce 0

**Definizione di riconoscitore a pila**: un riconoscitore a pila è costituito da una tupla A = (Σ, K, S, δ) dove:
Σ = l'alfabeto di input, K = l'alfabeto della pila, l'intersezione tra i due alfabeti è l'insieme vuoto, S = simbolo iniziale della pila, δ = funzione di evoluzione della pila $\delta = K \times \Sigma \rightarrow 2^{k^*}$ con la scrittura $\delta(X,\sigma) = \{w_1, w_2, \ldots, w_s\}$ con $w_i \in K^*$, con questa scrittura vengono indicati i seguenti fatti: X è letto in cima alla pila, TOP, σ è letto sul nastro di input, X viene cancellato dalla pila, POP e viene scelto un $W_i$ non deterministicamente da inserire nella pila, PUSH

**Grafo di computazione** dovuto a $x = x_1, x_2, \ldots, x_n$ come input

Il grafo di computazione è un albero di pile, in cui si parte dalla radice, pila iniziale, contenente il simbolo S ed in base ai simboli di input abbiamo le evoluzioni della pila che modificano S, in base alla funzione δ, arrivati allo step finale possiamo dire che la parola produce le seguenti pile, per vedere se è accettata si usa il criterio di accettazione

**Il criterio di accettazione** dice che la parola x si dirà accettata se nel grafo di computazione di x esiste un cammino che partendo da S mi porta nella pila vuota, all'ultimo passo tramite la funzione IsEmpty dirà se la parola è accettata

**Formalizziamo**:

la **configurazione** è la fotografia della pila in un certo istante di tempo, la possiamo scrivere così: Pila · Input l'input è la parte ancora da leggere, per esempio S · x è la configurazione iniziale, ε · ε è la configurazione finale ed accettante

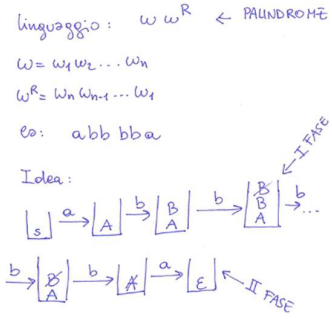
il **passo di computazione** viene definito tramite il seguente simbolo ⊢, per esempio configurazione ⊢ configurazione successiva, cioè dopo aver letto il simbolo di input e il simbolo in cima alla pila, quindi scriveremo: XW · σw ⊢ γW · w se e solo se γ ∈ δ(X, σ), cioè viene effettuata la lettura di σ e nel nastro rimane ancora da legge w

Zero o più passi di computazioni vengono indicati tramite il seguente simbolo: **⊢***

**il linguaggio riconosciuto da A**

L(A) = {x ∈ Σ* | S · x ⊢* ε · ε}, cioè il linguaggio riconosciuto da A indica che data una parola x in input possiamo arrivare alla forma accettate partendo da S · x

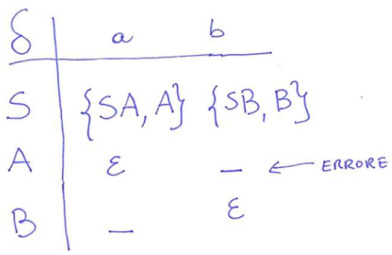
**Esempio**:




prendiamo come esempio il linguaggio delle parole palindrome pari formata da due parole w e w$^r$ che sono uno lo specchio dell'altro ad esempio abbbba la nostra idea di riconoscitore a pila che riconosce il linguaggio, andiamo a leggere la prima metà della parola, cioè w, e memorizziamo questi simboli nella nostra pila, dopo aver letto la prima metà avremo BBA partendo dal top, ora andremo a leggere la pila e li confrontiamo con la seconda metà se sono uguali li cancello e se alla fine raggiungiamo la pila vuota la parola viene accettata altrimenti rifiutiamo, per andare ad individuare il centro della parola andiamo ad utilizzare il non determinismo cioè ad ogni passo della prima fase possiamo dire continua la prima fase oppure inizia la seconda fase

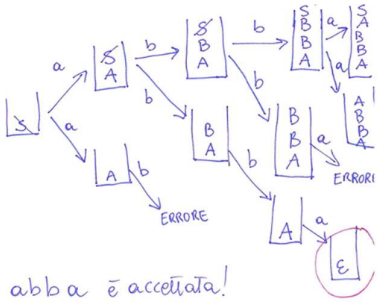
utilizziamo il simbolo S per dire che siamo nella prima fase, ovvero l'inserimento dei simboli nella pila e se manca S e abbiamo solamente i simboli A,B siamo nella seconda fase, ovvero il confronto dei simboli di input con il contenuto della pila

la funzione delta dell'esempio:




abbiamo sul nastro di input i simboli a e b, mentre come simboli del contenuto della pila abbiamo S, A e B, all'incrocio abbiamo i passi di computazione quando si legge il simbolo in testa alla pila e sotto la testina di input, per esempio se siamo in S e riceviamo a possiamo andare in SA se siamo nella prima fase oppure in A, cancellando S, se siamo nella seconda, nei restanti passi andiamo a cancellare in quanto siamo nella seconda fase

**Grafo di computazione di "abba"**




Partiamo da S e andiamo a leggere a, segniamo le due evoluzioni, cioè SA o A, ora andiamo a leggere b che nel primo ramo da SA porta a SBA oppure a BA, mentre nel secondo ramo A porta ad un errore, se andiamo a leggere la b successiva, dal ramo BA andiamo a cancellare B e otteniamo A che alla lettura successiva, cioè a, diventa ε quindi viene accettata, gli altri rami eseguono evoluzioni ma che non sono inerenti alla risoluzione di questa parola

## La classe dei linguaggi riconosciuti dai riconoscitori a pila

I riconoscitori a pila riescono a riconoscere i linguaggi di tipo 2, ovvero i linguaggi liberi da contesto

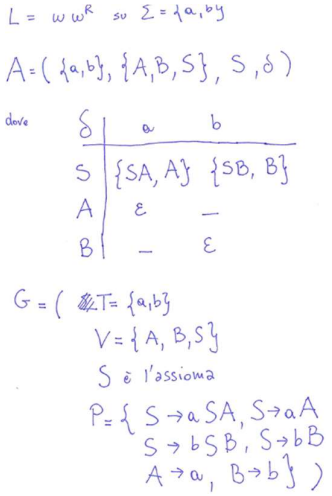
Il modello dei riconoscitori a pila deve essere non deterministico, $S : K \times \Sigma \rightarrow 2^{K^*}$, ovvero associa alla coppia un sottoinsieme finito di parole prese da $2^{K^*}$

**Teorema**: L è generato da una grammatica G di tipo 2 se e solo se L è riconosciuto da un riconoscitore a pila

**Dimostrazione**: vediamo il verso da A riconoscitore a pila a G grammatica di tipo 2

Sia $A = (\Sigma, K, S_A, \delta)$ posso costruire la seguente grammatica di tipo 2:
$G = (T = \Sigma, V = K, S = S_A, P: X \rightarrow \sigma W$ se e solo se $W \in \delta(X, \sigma))$

Esempio:




Andiamo a vedere il linguaggio $ww^r$, parole specchiate, con $\Sigma = (a,b)$
$A = (\{a,b\}, \{A,B,S\}, S, \delta)$, la tabella indica l'evoluzione della funzione in base agli input
Possiamo derivare la seguente grammatica:
simboli terminali $T = \{a,b\}$
variabili $V = \{A, B, S\}$
assioma = S
$P = \{ S \rightarrow aSA, S \rightarrow aA, S \rightarrow bSB, S \rightarrow bB, A \rightarrow a, B \rightarrow b\}$

La grammatica ottenuta è in forma normale di greibach di:
$S \rightarrow aSa, S \rightarrow aa, S \rightarrow bSb, s \rightarrow bb$

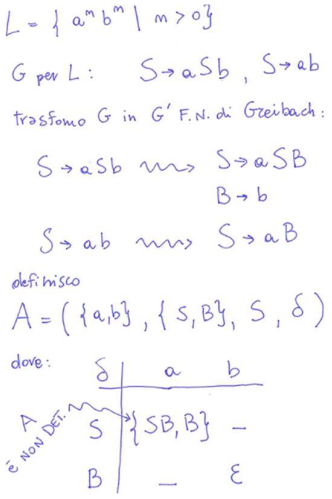
**Dimostrazione**: vediamo il verso da G grammatica di tipo 2 ad A riconoscitore a pila

Abbiamo una grammatica di tipo 2 e per prima cosa la trasformiamo in FN di greibach

$G \rightarrow G' = (T, V, S_{G'}, P)$ e costruiamo il nostro riconoscitore A:

$A = (\Sigma = T, K = V, S = S_{G'}, \delta: \delta(X, \sigma) = \{W \mid X \rightarrow \sigma W \in P\})$

Esempio:




dato il linguaggio $L = \{a^n b^n \mid n > 0\}$ con la seguente grammatica G:
$S \rightarrow aSb, S \rightarrow ab$, andiamo a trasformarla in FN di greibach:
$S \rightarrow aSb$ diventa $S \rightarrow aSB$ e $S \rightarrow ab$ diventa $S \rightarrow aB$ ed aggiungiamo la regola $B \rightarrow b$
definiamo il nostro riconoscitore A: $(\{a,b\}, \{S,B\}, S, \delta)$ ed otteniamo la tabella in figura

A non è deterministico per via della casella (S,a)

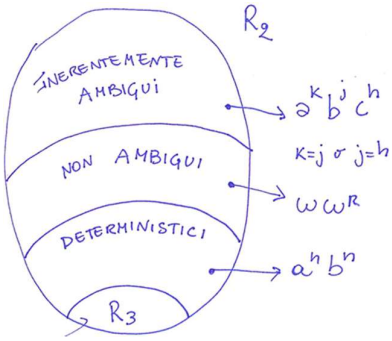
Per questo linguaggio non è il miglior riconoscitore esistente, infatti ne esiste uno deterministico:

A' = ({a,b}, {S, A, B}, S, δ) dove ricaviamo la seguente tabella di evoluzione degli input:

| δ | a | b |
|---|---|---|
| S | A | – |
| A | AB | ε |
| B | – | ε |

note:
A in cima, sono nella prima meta e inserisco nella pila una B
B in cima, sono nella seconda meta e cancello le B della pila

Riassumendo possiamo dire che questa è la classe dei linguaggi di tipo 2, dove sono presenti i linguaggi

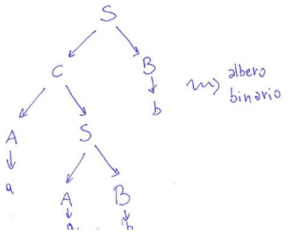



deterministici che ammettono riconoscitori a pila deterministici, per esempio $a^nb^n$, i deterministici sono un sotto insieme dei linguaggi non ambigui, per esempio $ww^r$, questi ultimi sono sotto insiemi dei linguaggi inerentemente ambigui, per esempio $a^kb^jc^h$ con k = j o j = h
all'interno dei linguaggi deterministici abbiamo i linguaggi regolari, in quanto un linguaggio regolare ammette un riconoscitore a pila deterministico

**Dimostrazione**: $L \in R_3$ dice che esiste un automa A DFA che è simulabile tramite un riconoscitore a pila, metto lo stato del DFA nella pila, la funzione isEmpty viene sostituita con IsAFinalState

## Alberi binari

Data una grammatica di tipo 2 in forma normale di Chomsky, cioè A → BC e A → σ, si hanno alberi di derivazione binari, cioè ad ogni nodo partono due figli escludendo i nodi finali
ad esempio avendo S → AB, S → CB, C → AS, A → a e B → b, che definisce la grammatica $a^nb^n$, abbiamo il seguente albero




In un albero binario con m andiamo ad indicare il numero di foglie e con n la sua altezza, in un albero bilanciato il numero di foglie m è uguale a $2^n$, mentre se abbiamo un albero generico abbiano $m <= 2^h$, in generale possiamo dire che un albero binario è sempre formato da $m <= 2^h$ da cui deriviamo $\log n <= h$

## Il Pumping Lemma

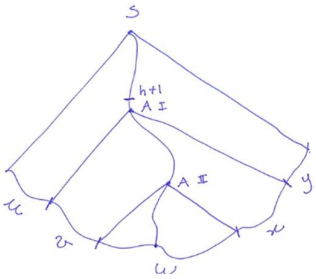
Il pumping lemma esprime una condizione necessaria per i linguaggi di tipo 2, se L non soddisfa il pumping lemma vuol dire che L non è di tipo 2 e se L soddisfa il pumping lemma può essere che L sia di tipo 2

**Enunciato**: per ogni L di tipo 2 esiste una costante H > 0 tale che per ogni $Z \in L$ con $|Z| > H$ esiste una scomposizione in uvwxy = Z, cioè, spezzata in fattori, che soddisfa i seguenti punti:

1- $|vx| >= 1$
2- $|vwx| <= H$
3- $\forall K >= 0\ uv^kwx^ky \in L$

**Dimostrazione**: per dimostrare l'enunciato del pumping lemma andiamo ad analizzare la sua frase pezzo per pezzo

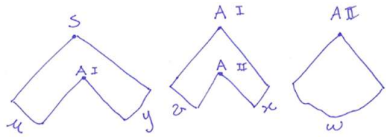
- "Per ogni L di tipo 2 esiste H > 0", dice che, se L è di tipo 2 allora ammette una grammatica G in FN di Chomsky, andiamo a fissare H = $2^{h+1}$ dove h è il numero di variabili in G
- "Per ogni Z ∈ L tale che |Z| > H", dato che Z appartiene ad L vuol dire che esiste un albero di derivazione binario che ha sulle foglie la composizione di Z, il fatto che |Z| sia maggiore di H ci dice che altezza dell'albero >= log|Z| > logH = h+1
- "Esiste una scomposizione di Z in uvwxy", la possiamo ricavare dall'albero di derivazione nei seguenti modi:
  consideriamo il ramo più lungo dell'albero dalla radice alla foglia, dal ramo più lungo risaliamo di h+1 nodi, per il principio della piccionaia due nodi sono etichettati con la stessa variabile A, la scomposizione si ricava da questi due nodi A1 e A2
- "la scomposizione soddisfa 1,2,3"




Condizione 1: |vx| >= 1 ci dice che v e x non possono essere contemporaneamente ε in quanto A1 e A2 sono distinti, se sono vuote vuol dire che A1 e A2 sono lo stesso nodo cosa non possibile

Condizione 2: |vwx| <= H ci dice che |vwx| <= $2^{h'}$ <= $2^{h+1}$ = H, dove h' è l'altezza dell'albero con radice A1

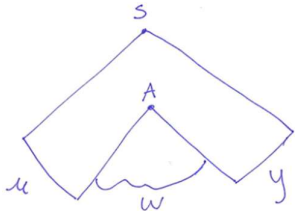
Condizione 3: ∀ K >= 0 $uv^kwx^ky$ ∈ L, per vedere questa condizione andiamo a ritagliare l'albero nei seguenti modi:



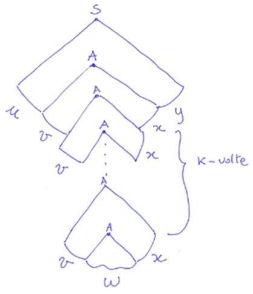

da questi sottoalberi possiamo definire queste regole di derivazione, il primo S ⇒* uAy, il secondo A ⇒* vAx, il terzo A ⇒* w

nel caso **K = 0**, con K = 0 v e x spariscono e otteniamo la parola uwy e dobbiamo mostrare che uwy ∈ L

possiamo prendere il primo sottoalbero e il terzo e li possiamo unire tramite la radice ottenendo il seguente sottoalbero che ci dice che uwy appartiene ad L, possiamo vederla nel seguente modo S ⇒* uAy ⇒* uwy




nel caso **K > 0**, le parole v e x vengono ripetute k volte e dobbiamo dimostrare che appartengano ad L, andiamo a prendere il primo sottoalbero per generare u e y, andiamo a prendere il secondo sottoalbero facendolo combaciare con la radice e lo andiamo a ripetere K volte, alla K-esima volta andiamo a fare combaciare il terzo sottoalbero per generare w, quindi visto che ammette un albero di derivazione possiamo dire che appartiene ad L, possiamo vederla anche nel seguente modo: S ⇒* uAy ⇒* uvAxy ⇒* … ⇒* $uv^kAx^ky$ ⇒* $uv^kwx^ky$




**L'applicazione del pumping lemma**, sappiamo che $a^nb^nc^n$ non è di tipo 2

**Dimostrazione**: consideriamo la parola z = $a^Hb^Hc^H$ dove |z| = 3H > H

Facciamo vedere che qualunque sua scomposizione in uvwxy non soddisfa contemporaneamente le tre condizioni necessarie del pumping lemma

Z = aa…a bb…b cc…c, la scomposizione in cinque fattori deve avvenire su questi simboli

per vwx abbiamo i seguenti casi:

1- vxw contiene sia a,b,c
2- oppure vxw contiene a,b o b,c
3- oppure contiene a o b o c

andiamo a studiare ogni caso per vedere se soddisfano le condizioni:

1- nel primo caso vwx = abb…bc, dove b è ripetuta H volte, quindi |vwx| > H ciò significa che la seconda condizione è falsa
2- nel secondo caso possiamo dire che vwx non contiene le "c"
3- consideriamo $uv^Kwx^Ky$ appartien ad L, cioè supponiamo che vale la condizione tre allora $|uv^Kwx^Ky| = 3H = |uvwxy|$, è uguale perché il numero di "c" è H da cui si ottiene che non vale la prima condizione