

LINGUAGGI FORMALI E COMPILATORI

Appunti Lezioni - Anno 2020/2021



GAIA SERAVELLI

Sommario

| | | | |
|---|----------------|----------------|----|
| Fasi della Compilazione | 23.02.21..... | 4 | |
| Sistema di Compilazione | | 4 | |
| Da Linguaggio ad Alto Livello a Istruzioni Macchina | | 5 | |
| Fase di Analisi | | 5 | |
| Lessico, Sintassi e Semantica | | 5 | |
| 1. Analisi Lessicale..... | | 5 | |
| 2. Analisi Sintattica..... | | 6 | |
| 3. Analisi Semantica Statica | | 6 | |
| Fase Sintetica..... | | 7 | |
| 1. Generazione del Codice Intermedio | | 7 | |
| 2. Ottimizzazione | | 7 | |
| 3. Generazione del Codice Oggetto | | 7 | |
| 4. Ottimizzazione Peep-Hole..... | | 7 | |
| Organizzazione del Compilatore | | 7 | |
| Teoria dei Linguaggi Formali | 24.02.21..... | 8 | |
| Teorema di Cantor | | 8 | |
| Grammatica a struttura di frase | | 02.03.21..... | 9 |
| Gerarchia di Chomsky | | 10 | |
| Linguaggi di tipo 0 | | 10 | |
| Linguaggi di tipo 1 (Grammatiche Contestuali) | | 10 | |
| Grammatiche Monotòne (sempre di tipo 1)..... | | 10 | |
| Linguaggi di tipo 2 (Grammatiche non contestuali) | | 11 | |
| Linguaggi di tipo 3 (Grammatiche Regolari)..... | | 11 | |
| Teoria degli Automi a Stati Finiti | 03.03.21 | 12 | |
| Atomi a Stati Finiti Deterministici | | 9.03.21 | 13 |
| Automi a Stati Finiti non Deterministici | | 13 | |
| La costruzione | | 15 | |
| Algoritmo di Determinizzazione | | 15 | |
| Automa a Stati Finiti non Deterministico con ϵ Transizioni | | 10.03.21 | 16 |
| La costruzione | | 17 | |
| Espressioni Regolari | | 18 | |
| Teorema di Kleene | | 19 | |
| Dall'Espressione Regolare all'automa (sintesi) | | 19 | |
| Dall'Automa all'espressione regolare (analisi) | | 17.03.21 | 21 |
| Operazioni Booleane..... | | 23 | |
| Automa Minimo | 23.03.21 | 24 | |

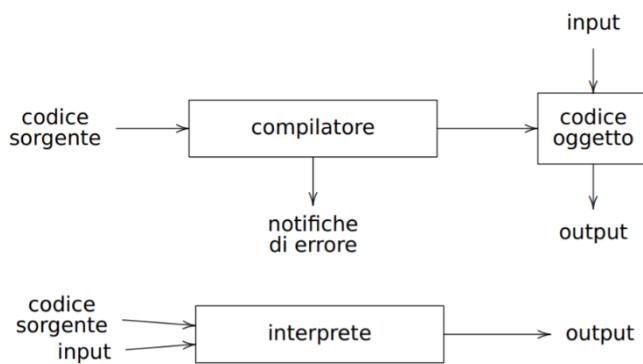
| | |
|--|----|
| Equivalenza di Nerode..... | 24 |
| Grammatiche Regolari 24.03.21 | 28 |
| Dalla grammatica all'automa | 28 |
| Dall'Automa alla Grammatica | 29 |
| Grammatiche Lineari..... | 30 |
| Lemma di Iterazione 30.03.21 | 30 |
| Analisi Lessicale..... | 31 |
| Come si svolge l'analisi lessicale | 32 |
| Linguaggi non contestuali 31.03.21 | 33 |
| Linguaggio Dick..... | 33 |
| Alberi di derivazione..... | 33 |
| Semplificazioni nelle grammatiche non contestuali 20.04.21 | 35 |
| ε -produzioni e produzioni unarie..... | 35 |
| Eliminazione delle ε -produzioni..... | 35 |
| Eliminazione delle produzioni unarie | 37 |
| Variabili improduttive e inaccessibili | 37 |
| Procedura di Riduzione | 38 |
| Forma Normale di Chomsky 21.04.21 | 39 |
| Forma Normale di Greibach..... | 40 |
| Lemma di iterazione per i linguaggi non contestuali..... | 40 |
| Algoritmo di Cocke-Kasami-Younger (CYK) 27.04.21 | 41 |
| Parsing Top-Down 28.04.21..... | 43 |
| Endmaker | 43 |
| Parsing Predittivo..... | 44 |
| Come Realizzare un Parser | 45 |
| Automi a Pila 04.05.21..... | 46 |
| Funzionamento | 46 |
| Come descrivere l'Automa..... | 47 |
| Metodi di Accettazione..... | 47 |
| Automa a pila deterministico..... | 48 |
| Metodi di Accettazione..... | 48 |
| Il caso deterministico | 49 |
| Teorema di caratterizzazione dei linguaggi non contestuali 05.05.21 | 50 |
| Dalla grammatica all'automa | 50 |
| Dall'automa alla grammatica | 51 |
| Proprietà di chiusura dei linguaggi non contestuali 12.05.21..... | 52 |
| Unione..... | 52 |

| | |
|--------------------------|----|
| Concatenazione..... | 52 |
| Chiusura di Kleene | 52 |
| Intersezione | 52 |
| Morfismi | 53 |
| Sostituzioni | 54 |

Quando vogliamo scrivere un programma, usiamo un compilatore e un linguaggio simile a quello umano (es.: C, Pascal, Java, ecc...), un linguaggio ad alto livello.

Il Compilatore esegue la traduzione dal **Codice Sorgente** (codice in C, Pascal, Java,...) al **Codice Oggetto** (sequenza di istruzioni macchina), prima dell'esecuzione.

Non per forza la traduzione deve avvenire tra Codice Sorgente e Codice Oggetto, è infatti possibile anche avere una traduzione tra due linguaggi di programmazione (es.: da Pascal a C).



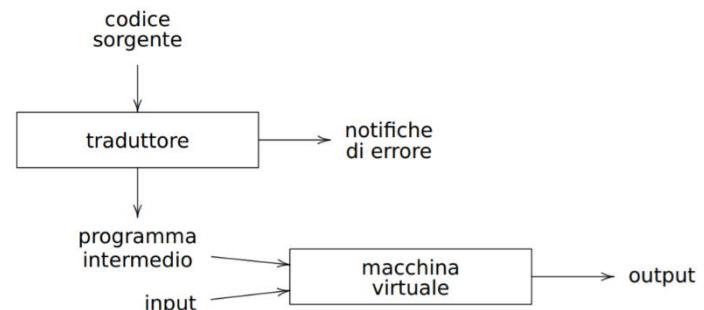
Il compilatore, inoltre, notifica anche la presenza di eventuali errori.

Il Codice Oggetto, o codice in altro linguaggio di programmazione, è un programma che a sua volta può prendere dei dati in input e emettere dati in output.

Oltre ai compilatori abbiamo gli **Interpreti**, che svolgono un lavoro diverso: prendono in ingresso sia il Codice Sorgente che l'eventuale input e poi, istruzione per istruzione, traducono il codice sorgente e lo eseguono.

Una terza categoria è costituita dai **Compilatori Ibridi**.

Java ne è un esempio: il codice Java viene dato in pasto a un traduttore che trasforma il codice sorgente in Java Byte Code, cioè un nuovo linguaggio di programmazione chiamato **programma intermedio**, e poi questo programma intermedio viene interpretato da una **macchina virtuale** (Java Virtual Machine), che prende anche eventuali dati in input, e poi restituisce un output.



I programmi interpretati sono molto più lenti dei programmi compilati perché la traduzione istruzione per istruzione, non è mai efficiente.

Nel caso dei Compilatori Ibridi, tuttavia, l'interpretazione è più efficiente rispetto ai normali interpreti perché il programma intermedio è in Java Byte Code, che può essere interpretato velocemente.

Il secondo vantaggio della macchina virtuale è che essa esegue tutti i programmi in uno **spazio protetto**, quindi controlla passo per passo che non ci siano operazioni malevole nel programma intermedio.

Sistema di Compilazione

La compilazione richiede una serie di operazioni:

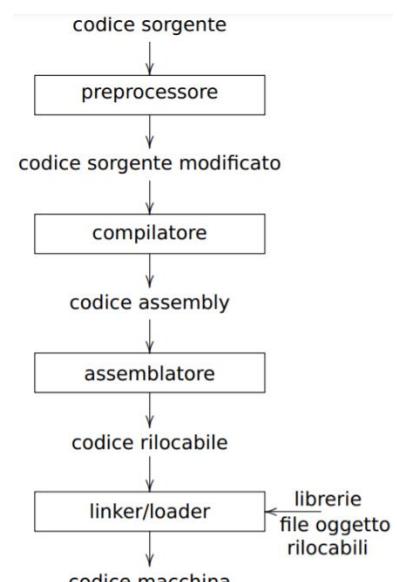
1) **Processing**: il codice sorgente viene dato in pasto a un preprocessore che produce un codice sorgente modificato.

2) **Compilazione**: il compilatore produce tipicamente un codice Assembly, non un codice macchina.

3) **Assembly**: il codice assembly viene poi assemblato e diventa sequenza di istruzioni macchina, il cosiddetto **Codice Rilocabile**.

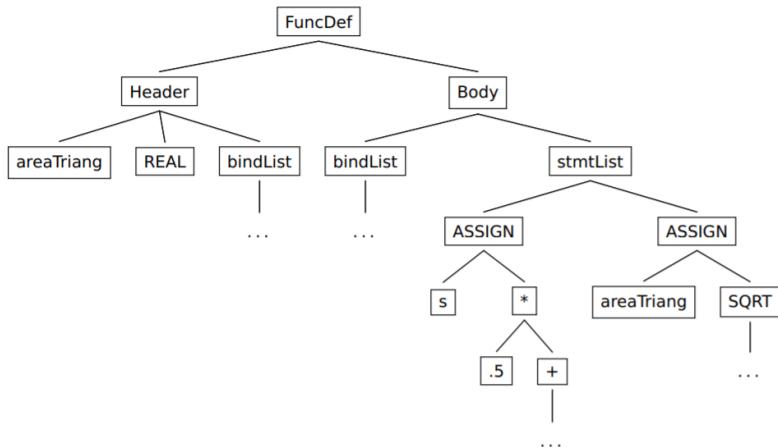
4) A questo codice poi andranno aggiunte tutti **moduli delle librerie e file oggetto**. Questa operazione è svolta dal **linker o loader**, e poi avremo un **codice macchina** pronto per essere eseguito.

Quindi in realtà ci sono ben 4 operazioni per ottenere il nostro codice eseguibile.



Da Linguaggio ad Alto Livello a Istruzioni Macchina

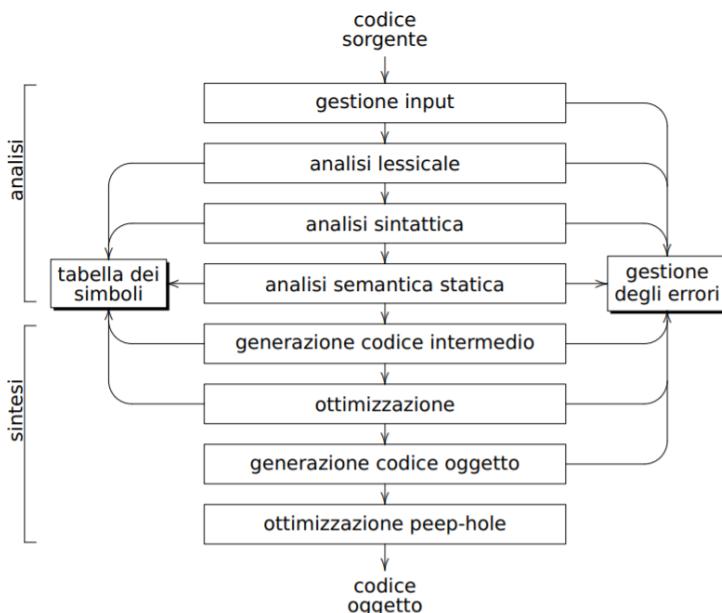
```
double areaTriang(double a, double b, double c) {
    double s = (a+b+c)*0.5;
    return Math.sqrt(s*(s-a)*(s-b)*(s-c));
}
```



(bindList), il puntatore alla variabile successiva s (stmtList), e successivamente una serie di istruzioni. Il primo nodo ASSIGN, ad esempio, assegna ad s il prodotto di 0.5 + una certa somma (a+b+c).

La seconda parte della compilazione, fase di **Sintesi**, consiste nel trasformare questo albero in una serie di istruzioni in linguaggio macchina.

Sia l'Analisi che la Sintesi, si compongono di più passi durante i quali vengono usate: una struttura dati, la tabella dei simboli, e una funzione importante di gestione degli errori.



significato.

1. Analisi Lessicale

Costituisce la prima fase di compilazione, viene eseguita da un modulo del compilatore detto **Scanner**, che ha il compito di raggruppare i caratteri in sequenze elementari detti **lessemi**. Il file del nostro programma, infatti, non è altro che una sequenza di lettere.

Ad ogni lessema verrà associata una **classe lessicale (Token)**, e un **attributo** che viene registrato nella tabella dei simboli.

JAVA Come passare da un linguaggio di alto livello come Java, al Codice Macchina?

La prima parte della compilazione consiste nel creare un **Albero Sintattico Astratto**, ovvero una struttura dati che viene creata automaticamente. Questa fase è detta di **Analisi**.

La **radice** rappresenta la definizione della funzione, che è costituita da due parti: un header e un corpo.

Header contiene il nome della funzione, il tipo del valore da restituire e una lista di variabili.

Corpo contiene un'altra lista delle variabili locali

(bindList), il puntatore alla variabile successiva s (stmtList), e successivamente una serie di istruzioni. Il

Fase di Analisi

Lessico, Sintassi e Semantica

Lessico: "To be or not to be" frase composta da parole che non fanno parte del lessico italiano. Termini che costituiscono un determinato linguaggio.

Sintassi: "Popolo del è sovranità la" le parole fanno tutte parte del lessico italiano ma non rispettano le regole sintattiche.

Regole con cui i termini lessicali devono essere organizzati all'interno di un programma.
(ad es.: ad ogni parentesi aperta deve corrispondere una parentesi chiusa)

Semantica: "Il mattone mangia una canzone verde" qui è rispettato sia il lessico che la sintassi, ma non ha significato.

Esempio

position = initial + rate * 60

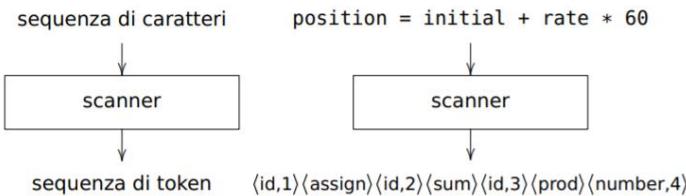
| Lessema | Token | Attributo | Tabella dei simboli |
|----------|--------|-----------|---------------------|
| position | id | 1 | 1 position |
| = | assign | - | 2 initial |
| initial | id | 2 | 3 rate |
| + | sum | - | 4 60 |
| rate | id | 3 | |
| * | prod | - | |
| 60 | number | 4 | |

Nell'esempio sopra, l'analizzatore lessicale (scanner) dovrà capire che `position` è un lessema e quindi va guardato tutto insieme, e che costituisce il nome di una variabile, quindi un identificatore. Poi crea un puntatore nella tabella dei simboli a cui questo token è associato.

Il simbolo uguale non ha bisogno di attributo perché è l'unico lessema del token assegnazione (`assign`), stessa cosa per il `+` e `*`. Tuttavia, il progettista del compilatore potrebbe anche fare una scelta differente e decidere che tutti gli operatori facciano parte di una stessa classe lessicale (`op`), in tal caso andrà aggiunto l'attributo che mi chiarisce il tipo di operazione che quell'operatore identifica.

Per fare tutto ciò si usa una teoria matematica chiamata **Teoria degli Automi a Stati Finiti**.

| Lessema | Token | attributo | tabella dei simboli |
|-----------------------|---------------------|-----------|---------------------|
| <code>position</code> | <code>id</code> | 1 | 1 position |
| <code>=</code> | <code>op</code> | 2 | 2 assign |
| <code>initial</code> | <code>id</code> | 3 | 3 initial |
| <code>+</code> | <code>op</code> | 4 | 4 sum |
| <code>rate</code> | <code>id</code> | 5 | 5 rate |
| <code>*</code> | <code>op</code> | 6 | 6 prod |
| 60 | <code>number</code> | 7 | 7 60 |



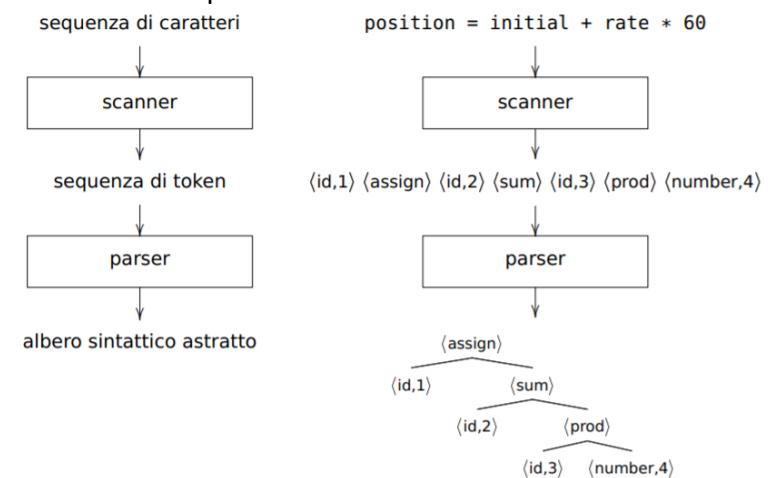
2. Analisi Sintattica

L'analizzatore sintattico (**parser**) ha il compito di organizzare i **token** prodotti dallo scanner in un albero (**Albero Sintattico Astratto**).

In questo albero i nodi interni rappresentano operazioni, e i figli del nodo rappresentano gli argomenti dell'operazione.

Ad esempio per la radice: la prima operazione è un'assegnazione, quindi la radice sarà `assign`.

L'assegnazione ha due operandi, un identificatore (`id,1`) a cui va assegnato il valore, e l'altro è un dato.

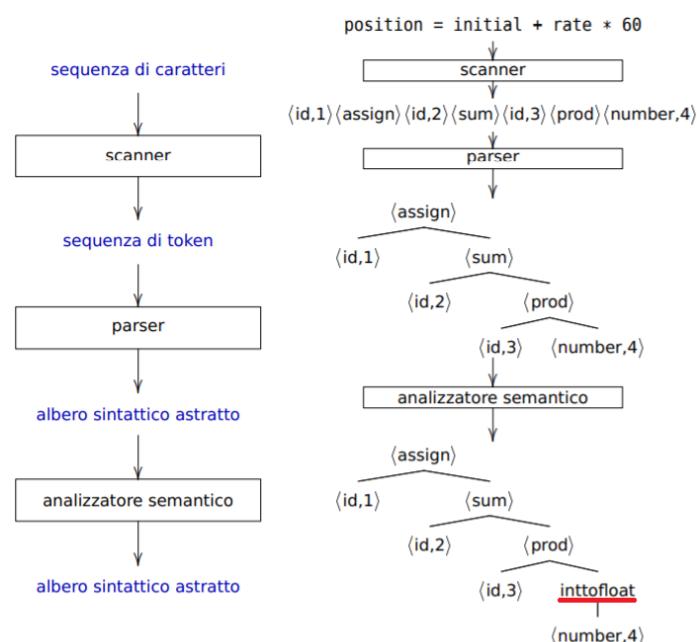


3. Analisi Semantica Statica

Utilizza l'albero sintattico astratto e la tabella dei simboli per verificare che il programma sorgente sia **semanticamente coerente** con la definizione del linguaggio.

Attraverso il **Type Checking**, ad esempio, verifica che ogni operatore abbia tutti gli operandi del tipo corretto. In alcuni linguaggi (tipo lo C), è possibile in caso eseguire la coercizione: se ho una variabile intera, la sua rappresentazione in macchina sarà diversa da quella di una variabile reale, quindi se voglio sommare una variabile reale ad un intero, devo prima tradurre l'intero in reale.

L'analisi semantica è **statica** perché rileva tutti quegli errori che si possono verificare senza eseguire il programma. Esistono anche errori che non possono essere rilevati da questa analisi perché si verificano a runtime. Con ciò, finisce la fase di Analisi.



Fase Sintetica

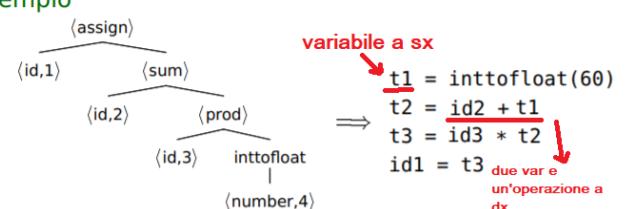
1. Generazione del Codice Intermedio

Dall'albero astratto e dalla tabella dei simboli, si ottiene un codice in un linguaggio che non è ancora un linguaggio macchina ma che è composto da **istruzioni elementari e facili da tradurre** in codice macchina (tipo l'Assembler).

Questo codice deve essere **indipendente dall'Architettura**, quindi non deve specificare gli indirizzi in memoria né deve dire i dati in quali registri devono essere caricati, ecc...

Nell'esempio viene creato un **codice a tre indirizzi**, ovvero un codice dove in tutte le istruzioni c'è una variabile a sinistra e, al massimo, un'operazione e due variabili a destra.

Esempio



2. Ottimizzazione

L'ottimizzazione cerca di **ridurre il tempo o lo spazio** necessario all'esecuzione del codice intermedio, così da ottenere un codice che sia il più efficiente possibile.

Esempio

```

t1 = inttofloat(60)
t2 = id3 * t1      => t2 = id3 * 60.0
t3 = id2 + t2      => id1 = id2 + t2
id1 = t3
  
```

Nell'esempio andiamo a risparmiare su due istruzioni perché, invece di fare un'operazione apposita per convertire 60 in float, assegniamo direttamente a $t2 = id3 * 60.0$.

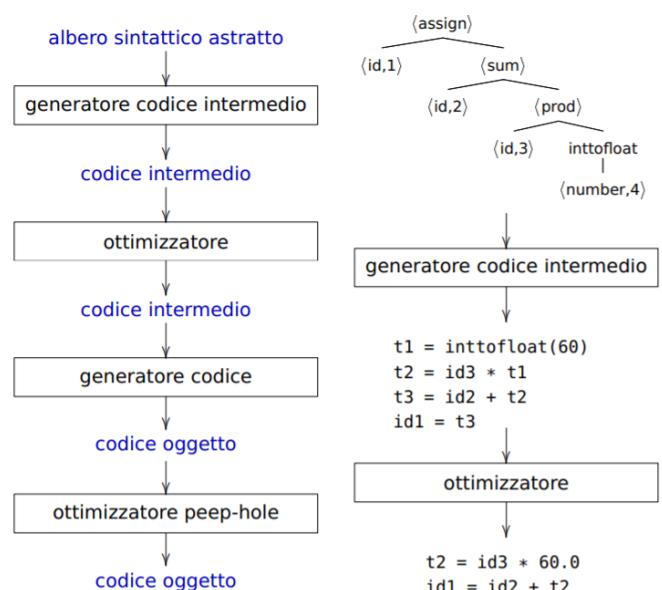
Nello stesso modo a $id1$, andremo ad assegnare direttamente $id2+t2$ senza il bisogno dell'istruzione intermedia che l'assegna a $t3$.

Tutto ciò che è stato fatto fino ad ora va bene indipendentemente da quale sia il computer sul quale eseguiamo la traduzione, e indipendentemente dal sistema operativo che usiamo sul nostro computer. È tutto portatile. Dalla prossima fase in poi, invece, dipendono dall'Architettura su cui stiamo lavorando.

3. Generazione del Codice Oggetto

Dobbiamo ora tradurre il codice ottenuto in codice macchina (codice oggetto), per fare questo è necessario: fissare le locazioni di memoria dei dati, generare il codice per accedere a tali dati, selezionare i registri per i calcoli intermedi, ecc...

Questa operazione è strettamente legata alla piattaforma e va quindi ripetuta per tutte le possibili architetture che abbiamo a disposizione. Rappresenta, di conseguenza, la parte più complessa della compilazione, tuttavia è riutilizzabile perché è possibile usarla sia per tradurre da C al linguaggio macchina, da Pascal a linguaggio macchina, ecc... È infatti indipendente dal linguaggio di alto livello da cui si parte.



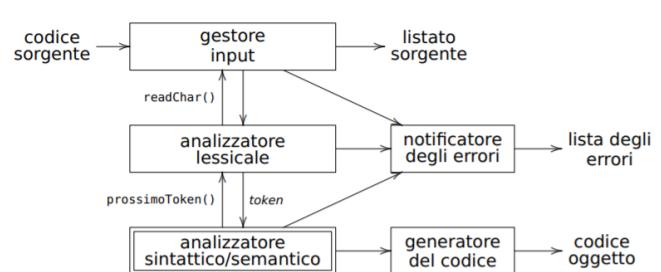
4. Ottimizzazione Peep-Hole

Infine è possibile un'ulteriore ottimizzazione, legata alle specificità dell'Architettura.

Organizzazione del Compilatore

Tutte queste fasi possono essere eseguite separatamente ma, generalmente, il nucleo del compilatore è l'analizzatore sintattico semantico che realizza anche l'analisi semantica statica. Questo nucleo invoca la funzione `prossimoToken` implementata nell'analizzatore lessicale, e poi l'analizzatore sintattico attiva la generazione del codice mediante la chiamata di funzioni opportune.

È quindi tutto un unico programma che invoca diverse funzioni.



La Teoria dei linguaggi formali tratta gli **alfabeti**: cioè insiemi finiti, non vuoti, di simboli detti **lettere**.

Esempi

$$\Sigma_0 = \{a, b\}, \quad \Sigma_1 = \{0, 1\}, \quad \Sigma_2 = \{a, b, c\},$$

$$\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}.$$

a, abb, ababbabb sono parole sull'alfabeto Σ_0 ,

01001010, 0110, 0000 sono parole sull'alfabeto Σ_1 .

2020, 5E7, CD078B sono parole sull'alfabeto Σ_2 .

Dopodiché potremo prendere le sequenze finite di lettere di questo alfabeto: le **parole**.

L'insieme delle parole sull'alfabeto Σ (sigma maiuscolo) è denotato con Σ^* .

Fra le tante parole ce n'è una che non contiene nessuna parola, che viene chiamata parola vuota, ed è denotata con ϵ epsilon (oppure Λ lambda maiuscolo).

La **lunghezza** di una parola u si denota con $|u|$ (è lunghezza di u , non valore assoluto), o anche $\ell(u)$.

Quindi: $|a| = 1; |abb| = 3; |\epsilon| = 0$.

La **concatenazione** è un'operazione binaria (prende due parole in input) e totale (avendo le due parole la concatenazione è definita). È un'operazione associativa, ha l'elemento neutro ϵ , e ha la proprietà della cancellatività a dx e a sx.

La concatenazione di due parole dà come risultato la somma della sequenza delle lettere delle due parole:
 $u = a, b, b; \quad v = a, a, a, b; \quad uv = a, b, b, a, a, b$.

Un **fattore** è una sequenza di lettere consecutive all'interno della parola. Nel caso in cui la prima lettera della parola è vuota allora il fattore è detto prefisso, nel caso sia l'ultima lettera della parola ad essere vuota, allora è un suffisso. Quando il fattore è diverso dalla parola stessa, p detto fattore proprio.

Definizione

Diremo che una parola v è un **fattore** di una parola w se risulta $w = xvy$ per opportune parole x, y . Nel caso in cui $x = \epsilon$ (risp., $y = \epsilon$) il fattore v si dice **prefisso** (risp., **suffisso**) di w . Diremo che v è un fattore **proprio** se $v \neq w$.

Un **linguaggio formale** è ogni sottoinsieme Σ^* sull'alfabeto Σ . Un linguaggio è quindi un insieme di parole. Se è finito, posso elencarne gli elementi, quindi è facile da utilizzare.

Mentre se è infinito, non è possibile elencarne gli elementi, ed è quindi difficile da descrivere e capire se un elemento ne faccia parte o meno.

Per fortuna è possibile definire/descrivere alcuni linguaggi infiniti, anche se non posso elencarne gli elementi. Tutto ciò che è possibile descrivere, è descrivibile grazie ad una sequenza di simboli presi da un linguaggio finito. È quindi possibile scrivere una parola, una volta definito il linguaggio. Risulta, invece, difficile scrivere insiemi infiniti di parole (linguaggi).

Fanno parte dei linguaggi infiniti, tutti quei linguaggi che se dati in pasto ad un compilatore non mi generano errore. Per capire se una parola fa parte di un determinato linguaggio infinito, devo darla in pasto al compilatore.

Teorema di Cantor

Abbiamo detto che è possibile definire alcuni linguaggi infiniti.

Il Teorema di Cantor ci dice che, però, non è possibile definire *tutti* i linguaggi infiniti.

Se io prendo gli l'insieme dei linguaggi su un dato alfabeto e poi prendo le parole sul medesimo alfabeto, non posso trovare una funzione iniettiva dal primo insieme nel secondo. Quindi non c'è una funzione che a ogni linguaggio sull'alfabeto Σ , mi associa una parola sull'alfabeto Σ .

Una grammatica a struttura di frase è una quadrupla costituita da: un alfabeto finito detto **vocabolario totale**, un sottoinsieme di questo alfabeto costituito dai **simboli terminali**, un altro sottoinsieme costituito dai simboli non terminali (**variabili**) tra i quali selezioniamo una particolare variabile chiamata **simbolo iniziale** o assioma della grammatica, e dalle **produzioni**.

Le produzioni sono essenzialmente coppie di parole con una freccetta tra loro.

Una parola β è una **conseguenza diretta** di α , se si ottiene andando a individuare in α un fattore che è il lato sx di una produzione e lo sostituiamo con il lato dx. Se β quindi si ottiene in questo modo, sostituendo all'interno di α un lato sx di una produzione per un lato dx, diremo che β è una conseguenza diretta.

β è una **conseguenza** di α se esiste una sequenza di parole, ognuna conseguenza diretta della precedente, in cui la prima è proprio α e l'ultima è β .

Le conseguenze del simbolo iniziale si dicono **forme sentenziali**.

Le forme sentenziali che non contengono variabili, sono le parole che costituiscono il **linguaggio generato** dalla grammatica.

Due **grammatiche** sono **equivalenti** se generano lo stesso linguaggio.

Quando andiamo a segnare una grammatica, abbiamo un meccanismo per definire un linguaggio in maniera effettiva, perché eseguendo una sequenza di derivazioni, in uno dei tanti modi possibili, otteniamo una parola del linguaggio. Per ogni parola c'è una derivazione in questa grammatica, quindi se le trovassimo tutte, otterremmo tutte le parole del linguaggio.

Tuttavia non è possibile provarle tutte per sapere se una parola fa parte o meno di un linguaggio, perché diventa troppo complesso.

Capire se una parola appartiene a un linguaggio (ricognizione) è molto importante perché ci permette di capire se il nostro codice è sintatticamente corretto. Equivale a verificare se il nostro codice come sequenza di token, è una delle parole appartenenti al linguaggio e alla grammatica.

Il problema di **Ricognizione** prende quindi in input una grammatica e una parola sull'alfabeto determinato dalla mia grammatica, e avrà come output SI se la parola è generata dalla grammatica oppure NO.

Il problema di **Parsing** prende come input una grammatica, una parola del linguaggio generato, e vuole trovare una derivazione della parola, cioè una sequenza di forme sentenziali ognuna conseguenza diretta della precedente, in maniera tale che partendo dal simbolo iniziale si ottenga la nostra parola.

Non esiste un algoritmo che risolva questi problemi nel caso generale, occorre quindi restringersi a classi particolari di grammatiche bilanciando l'efficienza degli algoritmi (non solo vorremmo algoritmi per questi problemi, ma vorremmo che fossero anche veloci) e l'espressività delle grammatiche (non possiamo mettere troppe restrizioni sennò le nostre grammatiche finiranno per generare dei linguaggi troppo banali per essere utili).

La strategia è quindi quella di **classificare** (gerarchia di Chomsky) le nostre grammatiche in base alla forma delle produzioni: produzioni con una forma particolarmente semplice avranno algoritmi di ricognizione e parsing, semplici e veloci, però ovviamente non saranno in grado di generare linguaggi molto espressivi. Altri tipi di grammatiche con produzioni di forma più generale avranno la capacità di produrre linguaggi molto espressivi, ma avranno algoritmi di ricognizione e di parsing molto inefficienti o addirittura inesistenti.

Gerarchia di Chomsky

La gerarchia di Chomsky divide le grammatiche e i linguaggi generati da esse, in 4 tipi, andando dalle forme più complesse a quelle più semplici.

Linguaggi di tipo 0

Fanno parte di questo livello le **grammatiche a struttura di frase**. I linguaggi generati da queste grammatiche si diranno **linguaggi di tipo 0** o anche **linguaggi ricorsivamente enumerabili**.

Esempio

La grammatica con le produzioni

$$\begin{array}{lll} S \rightarrow N & A \rightarrow N & N \rightarrow a \\ S \rightarrow N, AF & A \rightarrow N, A & N \rightarrow b \\ & , NF \rightarrow \&N & N \rightarrow c \end{array}$$

genera, per es., le parole $a \ a, b\&c \ a, c, a\&b$

È una grammatica di tipo 0.

Queste grammatiche hanno la **massima espressività possibile**: se io ho una procedura effettiva infinita che mi genera una lista di parole, quella lista di parole è un linguaggio di tipo 0.

Linguaggi di tipo 1 (Grammatiche Contestuali)

Le produzioni di queste grammatiche hanno il lato sx che contiene una variabile, mentre il lato dx si ottiene sostituendo quella variabile con una parola qualsiasi purché **non vuota**.

Quindi mentre prima potevamo sostituire una parola con una parola, qui possiamo sostituire solo una lettera all'interno di una parola con una parola, che per di più non deve essere vuota.

Esempio

La grammatica con le produzioni

$$\begin{array}{llll} S \rightarrow NVS & VN \rightarrow, N & N \rightarrow a & U \rightarrow a \\ S \rightarrow U & VU \rightarrow \&U & N \rightarrow b & U \rightarrow b \\ & & N \rightarrow c & U \rightarrow c \end{array}$$

è una grammatica di tipo 1, equivalente a quella dell'esempio precedente.

I linguaggi generati da grammatiche di **tipo 1** si dicono linguaggi di tipo 1 o anche **sensibili al contesto**.

Ogni grammatica di tipo 1, è automaticamente una grammatica di tipo 0.

A differenza dei linguaggi di tipo 0, per questo tipo di linguaggi esistono algoritmi che risolvono i problemi di parsing e riconoscione, ma sono molto costosi.

Si dicono contestuali perché **dipendono dal contesto**, ad esempio: guardando la foto dell'esempio, la seconda coppia di sostituzioni VN e VU, sono possibili solo se V è seguita da N, e solo se V è seguita da U.

Grammatiche Monotòne (sempre di tipo 1)

Grammatica in cui tutte le produzioni hanno il lato dx che ha **lunghezza maggiore o uguale** a quella del lato sx. Quindi non c'è più l'obbligo di sostituire singole variabili, ma c'è la sola regola che il lato dx non può essere più corto di quello sx.

Come possiamo vedere le grammatiche sensibili al contesto sono particolari grammatiche monotòne,

Esempio

La grammatica con le produzioni

$$S \rightarrow aSBc, \quad S \rightarrow abc, \quad cB \rightarrow Bc, \quad bB \rightarrow bb.$$

è monotona ma non è sensibile al contesto. Il linguaggio generato $L = \{a^n b^n c^n \mid n > 0\}$ è un linguaggio di tipo 1.

perché in esse si sostituisce, all'interno del contesto, una variabile con qualcosa che non è vuoto, quindi quello che otterremo sarà qualcosa di più lungo o al massimo della stessa lunghezza di quello che già abbiamo.

Al contrario, non tutte le grammatiche monotone sono grammatiche sensibili al contesto. Nonostante le grammatiche monotone siano una classe di grammatiche molto più ampia rispetto a quelle sensibili al contesto, di fatto non sono più potenti di esse. Infatti, si può dimostrare, che per ogni grammatica monotona si può costruire una grammatica sensibile al contesto equivalente che genera lo stesso linguaggio.

Linguaggi di tipo 2 (Grammatiche non contestuali)

Una grammatica a struttura di frase si dice non contestuale se le produzioni hanno a sx una singola variabile.

I linguaggi generati da queste grammatiche si dicono di tipo 2 o non contestuali.

Sono non contestuali perché la variabile può essere sostituita **indipendentemente dal contesto** in cui appare.

Esempio

La grammatica con le produzioni

$$\begin{array}{lll} S \rightarrow N & M \rightarrow N \& N & N \rightarrow a \\ S \rightarrow M & M \rightarrow N, M & N \rightarrow b \\ & & N \rightarrow c \end{array}$$

è una grammatica di tipo 2, equivalente a quella di un esempio precedente.

I linguaggi di tipo 2 costituiscono una **sottoclasse** dei linguaggi di tipo 1.

Gli algoritmi di riconoscione e parsing per le grammatiche di tipo 2 si comportano abbastanza bene, alcuni sono **moderatamente efficienti** (Complessità di n elevato a qualche esponente, quindi vanno bene in molti casi, ma non vanno bene per i nostri compilatori).

La maggior parte dei linguaggi di programmazione (i codici sintatticamente corretti) sono generati da una grammatica non contestuale.

Linguaggi di tipo 3 (Grammatiche Regolari)

Le produzioni hanno una forma ancora più semplice: a sx sempre una variabile come nelle grammatiche di tipo 2, ma a dx ho un terminale da solo o un terminale seguito da una variabile (terminale a sx e variabile a dx).

I linguaggi generati da grammatiche di tipo 3 si dicono linguaggi di tipo 3 o anche regolari.

I linguaggi di tipo 3 costituiscono una sottoclasse dei linguaggi di tipo 2 perché

le produzioni a sx hanno una singola variabile. Quindi un linguaggio generato da una grammatica di tipo 3, è anche un linguaggio generato da una grammatica di tipo 2.

Anche in questo caso, però, ci sono linguaggi di tipo 2 che non sono linguaggi di tipo 3.

Esempio

La grammatica con le produzioni

$$\begin{array}{lllll} S \rightarrow a & S \rightarrow aR & R \rightarrow \& N & M \rightarrow aR & N \rightarrow a \\ S \rightarrow b & S \rightarrow bR & R \rightarrow, M & M \rightarrow bR & N \rightarrow b \\ S \rightarrow c & S \rightarrow cR & & M \rightarrow cR & N \rightarrow c \end{array}$$

è una grammatica di tipo 3, equivalente a quella dell'esempio precedente.

Gli algoritmi di riconoscione per i linguaggi di tipo 3 esistono e sono **estremamente efficienti**, sono molto importanti per l'**analisi lessicale** ovvero l'analisi di token che costituiscono il nostro codice. Questo si ottiene attraverso la **Teoria degli Automi a Stati Finiti**.

Teoria degli Automi a Stati Finiti

03.03.21

Immaginiamo di avere un dispositivo che può raggiungere un certo numero finito di configurazioni interne. Questo dispositivo ha un nastro di input diviso in celle sul quale si può scrivere per l'appunto input: a ogni passo il dispositivo legge il contenuto della cella, passa in un nuovo stato dipendente esclusivamente dallo stato precedente e dal contenuto della cella letta, e infine passa a esaminare la cella successiva.

Terminata la lettura del nastro, in base allo stato raggiunto, o si accetta o si rifiuta la parola letta. Le parole accettate costituiscono un linguaggio di tipo 3 che, a sua volta, è sempre accettato da un automa a stati finiti.

Questo algoritmo è lineare e in tempo reale, perché ad ogni cella letta so se tutto ciò che è venuto prima è stato accettato o meno.

Formalmente un Automa a stati finiti deterministico (quanto abbiamo descritto sopra) è una quintupla:

- insieme degli stati: tutte le configurazioni che il dispositivo può assumere. Q
- alfabeto di input. Σ
- funzione di transizione: legge con cui i nostri stati cambiano in base ad uno stato precedente e alla lettera scritta sul nastro. È una funzione che quindi prende come input uno stato e una lettera e restituisce un nuovo stato. $\delta : Q \times \Sigma \rightarrow Q$ (funz. delta δ è uguale a Stato x Lettera = NuovoStato)
- stato iniziale. $q_0 \in Q$
- insieme degli stati finali: se al termine della computazione, il dispositivo si trova in uno stato che è compreso nell'insieme degli stati finali, la parola viene accettata. $F \subseteq Q$ (insieme F degli stati finali è sottoinsieme dell'insieme degli stati)

cappellino per far capire che la funzione delta è stata estesa
La funzione δ si estende a $\widehat{\delta} : Q \times \Sigma^* \rightarrow Q$ ponendo

$$\begin{aligned}\widehat{\delta}(q, \varepsilon) &= q, && \text{per ogni } q \in Q, \\ \widehat{\delta}(q, va) &= \delta(\widehat{\delta}(q, v), a), && \text{per ogni } q \in Q, v \in \Sigma^*, a \in \Sigma.\end{aligned}$$

|
parola 'va'
|
ultima lettera della parola

E' praticamente una funzione ricorsiva.

Come si può estendere la funzione di transizione in modo che non lavori solo con le coppie stato-lettera ma anche con le coppie stato-parola?

Quindi una parola w è accettata dall'automa se $(cappello)\delta(q_0, w) \in F$.

Esempio

Calcoliamo $\widehat{\delta}(q_0, aabb)$ per l'automa dell'esempio precedente:

$$\begin{aligned}\widehat{\delta}(q_0, \varepsilon) &= q_0, \\ \widehat{\delta}(q_0, a) &= \delta(\widehat{\delta}(q_0, \varepsilon), a) = \delta(q_0, a) = q_0, \\ \widehat{\delta}(q_0, aa) &= \delta(\widehat{\delta}(q_0, a), a) = \delta(q_0, a) = q_0, \\ \widehat{\delta}(q_0, aab) &= \delta(\widehat{\delta}(q_0, aa), b) = \delta(q_0, b) = q_1, \\ \widehat{\delta}(q_0, aabb) &= \delta(\widehat{\delta}(q_0, aab), b) = \delta(q_1, b) = q_1.\end{aligned}$$

| $Q = \{q_0, q_1, q_2\}$ | δ | a | b |
|-------------------------|----------|-------|-------|
| $\Sigma = \{a, b\}$ | q_0 | q_0 | q_1 |
| $F = \{q_0, q_1\}$ | q_1 | q_2 | q_1 |
| | q_2 | q_2 | q_2 |

L'insieme delle parole accettate dall'automa (A) si dice linguaggio riconosciuto/accettato, e si denota con $L(A)$.

Dal punto di vista della simulazione, le definizioni che abbiamo dato sono sufficienti, ma per capire meglio il

comportamento globale di un automa è consigliabile usare una rappresentazione completamente diversa che si basa sull'uso di un **grafo diretto con frecce etichettate**: grafo dove ci sono dei **vertici** che corrispondono agli stati dell'automa, e delle **frecce** che sono delle triple costituite da stato di partenza, stato di destinazione ed etichetta.

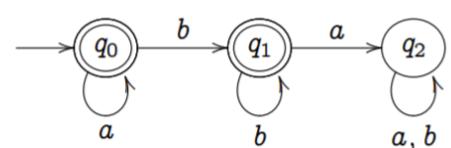
In questo grafo gli stati iniziali sono denotati da una freccetta entrante, mentre gli stati finali sono denotati da un doppio bordo.

Parola aabb = accettata.

Parola ba = rifiutata (perché finisco in q_2)

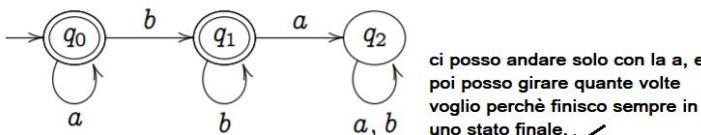
Esempio

| δ | a | b | F |
|----------|-------|-------|-----|
| q_0 | q_0 | q_1 | x |
| q_1 | q_2 | q_1 | x |
| q_2 | q_2 | q_2 | |



Per ogni stato q e ogni lettera a , c'è esattamente una freccia con etichetta a che esce da q .
 Esiste un cammino da uno stato p a uno stato q , con etichetta w , se e solo se $q=(\text{cappello}) \delta(p, w)$.
 Le parole accettate dall'automa sono esattamente le etichette dei cammini da uno stato iniziale (q_0) a uno stato finale.

Esempio



- le etichette dei cammini da q_0 a q_0 sono le parole a^n , con $n \geq 0$;
- le etichette dei cammini da q_0 a q_1 sono le parole $a^n b^m$, con $n \geq 0, m > 0$;

stessa cosa qui, ma la potenza di b deve essere almeno 1 per poter arrivare in q_1 .

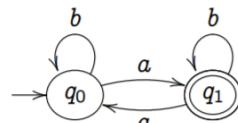
ci posso andare solo con la a , e poi posso girare quante volte voglio perché finisco sempre in uno stato finale.

Quindi nel caso dell'esempio, il linguaggio accettato dall'automa sarà:

$$L(A) = \{a^n b^m \mid n, m \geq 0\}$$

Quali parole accetta il seguente automa a stati finiti?

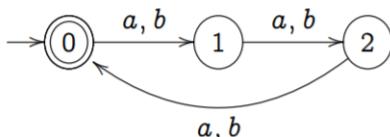
Tutte le parole sull'alfabeto $\{a, b\}$ che contengono un numero dispari di a .



Realizziamo un automa che riconosca il linguaggio

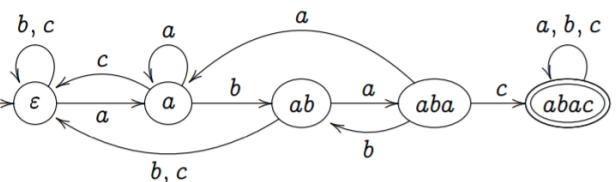
$$L = \{w \in \Sigma^* \mid |w| \text{ è multiplo di } 3\} \text{ sull'alfabeto } \Sigma = \{a, b\}.$$

Un automa a 3 stati 0, 1, 2 tale che $\widehat{\delta}(0, w) = |w| \bmod 3$, $w \in \Sigma^*$.



Sia $\Sigma = \{a, b, c\}$. Realizziamo un automa che riconosca

$$L = \{w \in \Sigma^* \mid abac \text{ è fattore di } w\}.$$



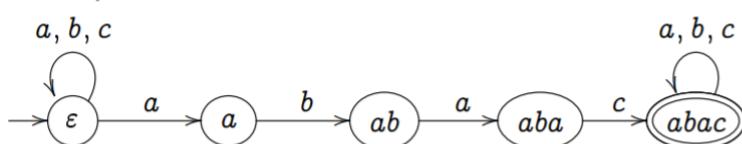
Atomi a Stati Finiti Deterministici

9.03.21

Per automi a stati finiti deterministici si intendono gli automi che abbiamo descritto precedentemente.

Se si eliminasse la regola che "per ogni stato q e ogni lettera a , c'è esattamente una freccia con etichetta a che esce da q " avremmo i pro e i contro degli automi a stati finiti non-deterministici: come pro avremmo la **libertà della progettazione** di automi che possono essere progettati con più facilità, come contro avremo la **complessità della computazione**, diventa infatti difficile verificare se una parola è accettata o meno.

Esempio L'esempio precedente diventerebbe:



programma informatico) al modello deterministico.

Per trovare una soluzione che ci consenta di avere i pro del modello non deterministico, senza avere i contro, si potrebbe fare il progetto con modello non-deterministico e poi la conversione automatica (con un

Atomi a Stati Finiti non Deterministici

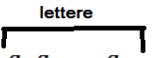
Possiamo definire Automi a Stati finiti non Deterministici, dei meccanismi che non hanno la restrizione descritta sopra, dei modelli deterministici.

Un meccanismo che può assumere un numero predefinito di configurazioni interne, che ha un nastro di input, ma che ad ogni passo non esegue più una sola transizione ben determinata ma dispone di un **ventaglio di possibilità** e può quindi eseguire varie transizioni (o nessuna).

L'input viene accettato se tra le tante computazioni possibili c'è n'è almeno una che termina in uno stato finale. Quindi una parola viene rifiutata se tutte le computazioni terminano in uno stato non finale o terminano prima che il nastro sia esaurito.

Formalmente un **Automa a Stati Finiti non Deterministico** è di nuovo una quintupla:

- insieme degli stati: Q
- alfabeto di input. Σ
- funzione di transizione: $\delta : Q \times \Sigma \rightarrow \wp(Q)$ (funz. delta δ è uguale a Stato x Lettera = insieme di Stati)
- stato iniziale. $q_0 \in Q$
- insieme degli stati finali $F \subseteq Q$ (insieme F degli stati finali è sottoinsieme dell'insieme degli stati)

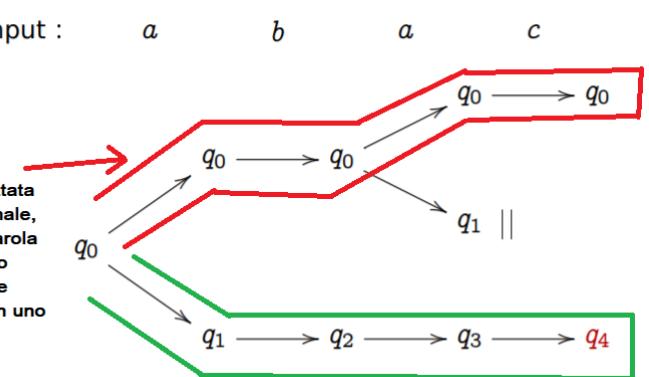

Una parola $w = a_1 a_2 \dots a_n$, ($a_1, a_2, \dots, a_n \in \Sigma$, $n \geq 0$), è accettata da A se esistono stati $q_1, q_2, \dots, q_n \in Q$ tali che

$$\begin{array}{ll} q_i = \text{da } q_1 \text{ a } q_n \\ a_i = \text{da } a_1 \text{ ad } a_n \end{array} \quad q_i \in \delta(q_{i-1}, a_i), \quad 1 \leq i \leq n, \quad q_n \in F.$$

- una parola w può essere etichetta di molti cammini uscenti dallo stato iniziale, o anche nessuno;
- la parola w è accettata se e solo se c'è un cammino dallo stato iniziale a uno stato finale con etichetta w ;
- questo non esclude però che vi possano essere altri cammini con etichetta w che partono dallo stato iniziale e terminano in uno stato non finale.

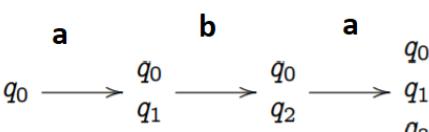
| lettere | | | | |
|----------|----------------|-------------|-------------|----------|
| δ | a | b | c | F |
| q_0 | $\{q_0, q_1\}$ | $\{q_0\}$ | $\{q_0\}$ | |
| q_1 | \emptyset | $\{q_2\}$ | \emptyset | |
| q_2 | $\{q_3\}$ | \emptyset | \emptyset | |
| q_3 | \emptyset | \emptyset | $\{q_4\}$ | |
| q_4 | $\{q_4\}$ | $\{q_4\}$ | $\{q_4\}$ | \times |

questa computazione non è accettata perché non finisce in uno stato finale, ma questo non vuol dire che la parola non sia accettata, perché possono esserci altre computazioni (come quella in verde) che terminano in uno stato finale.



Apparentemente questo tipo di progettazione non ci da nessun vantaggio, perché le computazioni da fare diventerebbero troppe. Supponiamo infatti di avere 2 cammini possibili per ogni scelta, e dovessimo leggere una parola di lunghezza 10, questo vorrebbe dire che dovremmo fare 2^{10} computazioni.

Proviamo però a ragionare come segue: se mi trovo in q_0 e il mio input è a , posso andare o in q_0 o in q_1 , ecc... Se mi trovo in q_1 e l'input è b , andrò in q_2 ...


Quindi ad ogni passo, in base alla colonna che avevo al passo precedente, e alla lettera di input, produco una colonna nuova. Questo procedimento è praticamente uguale a quello di un automa a stati deterministico, perché in base alla configurazione del mio dispositivo e alla lettera di input, determina la configurazione successiva. In questo caso la differenza è che abbiamo delle colonnine di stati.

Detto ciò, questo vuol dire che: Sia A un automa a stati finiti non deterministico, esiste **effettivamente** un **equivalente automa** a stati finiti deterministico A' tale che $L(A) = L(A')$, cioè esiste l'automa deterministico e l'algoritmo per costruire questo automa a partire dall'automa non deterministico.

Quindi, prima andremo a disegnare l'automa a stati non deterministico (prima immagine con più percorsi in verde e rosso), cosa che risulta molto più facile rispetto al crearne uno deterministico, e poi sapremo che esiste l'algoritmo in grado di trasformare quel grafo nel grafo di un automa a stati deterministico (seconda immagine con colonnine).

In sostanza vuol dire che nel grafo dove non c'è più il limite al numero di frecce che escono da uno stato, c'è un cammino che ha come etichetta w , che parte da uno stato iniziale e arriva in uno stato finale.

La costruzione

Sia $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ l'automa non deterministico. Parto dall'automa non deterministico

Definisco l'automa deterministico $\mathcal{A}' = \langle Q', \Sigma, \delta', s_0, F' \rangle$ come segue:

- l'insieme degli stati è l'insieme $Q' = \wp(Q)$ costituito dai sottoinsiemi di Q ,
- lo stato iniziale è $s_0 = \{q_0\}$, singleton (insieme con 1 stato) costituito dallo stato iniziale dell'A non-deterministico.
- gli stati finali sono tutti i sottoinsiemi di Q che contengono almeno un elemento di F , cioè

quindi almeno uno stato finale dell'A di partenza $F' = \{s \in \wp(Q) \mid s \cap F \neq \emptyset\}$,

• la funzione di transizione $\delta' : Q' \times \Sigma \rightarrow Q'$ è definita da

prende in input uno stato dell'A', cioè un insieme degli stati dell'automa non-deterministico, e una lettera a. $\delta'(s, a) = \bigcup_{q \in s} \delta(q, a), \quad s \in \wp(Q), \quad a \in \Sigma.$
eseguo la funzione per tutti gli stati q contenuti nell'insieme di stati s.

Come abbiamo accennato precedentemente, però, sarebbero presenti troppi stati da controllare (se $Q=n$, allora $Q'=2^n$). Per fortuna non tutti questi stati sono necessari: possiamo definire uno **stato necessario/accessibile** come un cammino nel grafo dell'automa che, dallo stato iniziale, mi porta a q.

Tutti gli altri stati si dicono

inaccessibili.

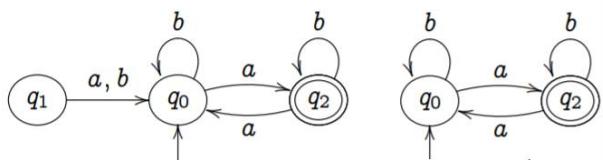
Definizione

Sia $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ un automa a stati finiti deterministico. Uno stato $q \in Q$ si dice **accessibile** se $q = \widehat{\delta}(q_0, w)$ per qualche $w \in \Sigma^*$.

Se io elimino tutti questi stati

inaccessibili e restringo la funzione di transizione (delta cappello) solo agli stati accessibili, ottengo un automa con un numero di stati più piccolo, equivalente a quello di partenza, che accetta lo stesso linguaggio di quello di partenza.

Esempio



Nell'esempio vediamo come lo stato q_1 è stato eliminato perché, iniziando da q_0 (stato di partenza), non c'è nessun cammino che mi porta in q_1 .

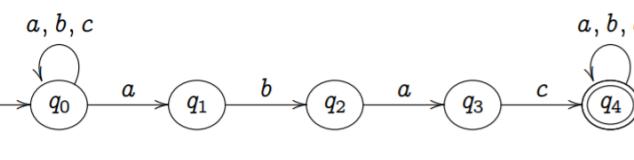
Attraverso un **Algoritmo di Determinizzazione** andremo a progettare un automa che prevede solo gli stati accessibili, e di conseguenza avrà un numero di stati molto inferiore di 2^n .

Algoritmo di Determinizzazione

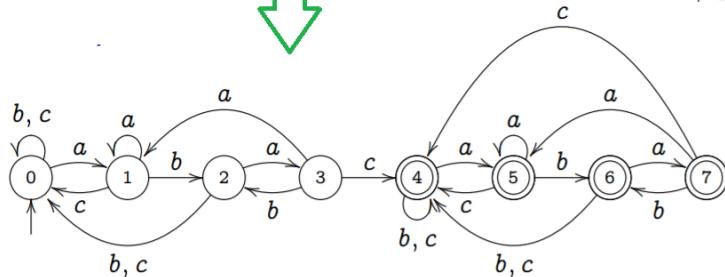
1. Prima di tutto inserisco nella lista degli stati, lo stato iniziale dell'automa deterministico, che come abbiamo visto prima corrisponde all'insieme costituito dal solo stato iniziale dell'automa non deterministico di partenza: $s_0 = \{q_0\}$
2. Per ogni stato r della lista (che per ora contiene il solo elemento q_0 , singleton) e ogni lettera $a \in \Sigma$
 - a. Calcolo il valore della funzione di transizione sulla coppia stato-lettera: $s = \delta'(r, a)$; Una volta calcolato ho due possibilità: o lo stato è già nella lista degli stati, oppure no.
 - b. Se s non è nella lista degli stati allora lo appendo alla lista degli stati.
 - c. Se s contiene uno stato finale di A, aggiungo s alla lista degli stati finali.

Partendo da questo automa a stati non deterministico, applicando l'algoritmo, riusciremo ad arrivare all'automa a stati deterministico:

Esempio



| i | s_i | $\delta'(i, a)$ | $\delta'(i, b)$ | $\delta'(i, c)$ | $i \in F$ |
|-----|----------------------|-----------------|-----------------|-----------------|-----------|
| 0 | q_0 | 1 | 0 | 0 | |
| 1 | q_0, q_1 | 1 | 2 | 0 | |
| 2 | q_0, q_2 | 3 | 0 | 0 | |
| 3 | q_0, q_1, q_3 | 1 | 2 | 4 | |
| 4 | q_0, q_4 | 5 | 4 | 4 | \times |
| 5 | q_0, q_1, q_4 | 5 | 6 | 4 | \times |
| 6 | q_0, q_2, q_4 | 7 | 4 | 4 | \times |
| 7 | q_0, q_1, q_3, q_4 | 5 | 6 | 4 | \times |



Come possiamo notare, l'automa deterministico ha più stati rispetto a quelli che avevamo trovato noi manualmente nel primo esempio *abac* (pag.11), successivamente vedremo infatti un altro algoritmo in grado di trovare partendo da un automa non deterministico, un automa deterministico con il minimo numero di stati possibile (**automa minimo**).

Automa a Stati Finiti non Deterministico con ϵ Transizioni

10.03.21

Un Automa a stati finiti non deterministico con ϵ transizioni è una quintupla dove Q , Σ , q_0 e F , sono come nell'Automa deterministico, mentre la funzione $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \wp(Q)$ è la funzione di transizione, che ora non è definita solo sulle coppie stato-lettera, ma anche sulle coppie **stato-parolaVuota**, e restituisce un insieme di stati (che potrebbe essere anche vuoto).

Una parola w è accettata dall'Automa se esiste un cammino dallo stato iniziale ad uno finale nel grafo con etichetta w , e se w può essere scritta come una sequenza di lettere e parole vuote.

Una parola $w \in \Sigma^*$ è **accettata** da A se esistono $n \geq 0$, $a_1, \dots, a_n \in \Sigma \cup \{\epsilon\}$ e $q_1, \dots, q_n \in Q$ tali che

$$w = a_1 a_2 \cdots a_n, \quad q_i \in \delta(q_{i-1}, a_i), \quad 1 \leq i \leq n, \quad q_n \in F.$$

- Una parola w può essere etichetta di molti cammini uscenti dallo stato iniziale, o anche nessuno;
- la parola w è accettata se e solo se c'è un cammino dallo stato iniziale a uno stato finale con etichetta w ;
- Questo non esclude però che vi possano essere altri cammini con etichetta w che partono dallo stato iniziale e terminano in uno stato non finale.

Anche qui, come prima, per ogni automa A a stati finiti non deterministici con ϵ -transizioni, esiste *effettivamente* un automa a stati finiti deterministico A' tale che: $L(A) = L(A')$.

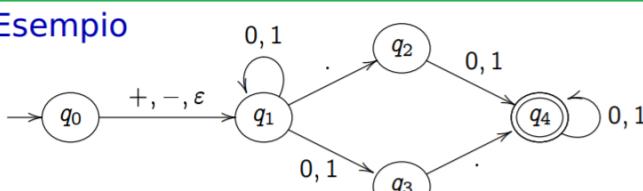
Se però con l'automa a stati finiti non deterministico le nostre computazioni si limitavano alla potenza-numero-di-lettere della parola, ora teoricamente

posso avere anche **infinite ϵ -transizioni** (la concatenazione di un miliardo di ϵ , è sempre una parola vuota).

Quindi bisogna utilizzare una tecnica per trattare queste ϵ -transizioni: **la ϵ -chiusura di uno stato**.

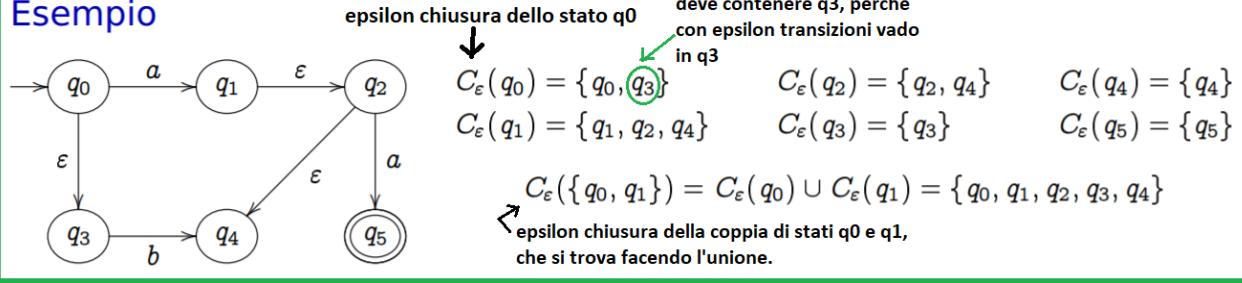
La ϵ -chiusura di uno stato è l'**insieme degli stati** che da questo si possono raggiungere utilizzando solo ϵ -transizioni.

Esempio



| δ | ϵ | + | - | . | 0 | 1 | F |
|----------|-------------|-------------|-------------|-------------|----------------|----------------|----------|
| q_0 | $\{q_1\}$ | $\{q_1\}$ | $\{q_1\}$ | \emptyset | \emptyset | \emptyset | |
| q_1 | \emptyset | \emptyset | \emptyset | $\{q_2\}$ | $\{q_1, q_3\}$ | $\{q_1, q_3\}$ | |
| q_2 | \emptyset | \emptyset | \emptyset | \emptyset | $\{q_4\}$ | $\{q_4\}$ | |
| q_3 | \emptyset | \emptyset | \emptyset | $\{q_4\}$ | \emptyset | \emptyset | |
| q_4 | \emptyset | \emptyset | \emptyset | \emptyset | $\{q_4\}$ | $\{q_4\}$ | \times |

Esempio



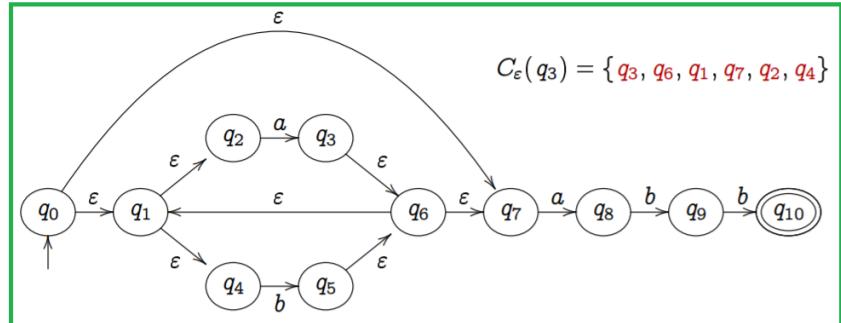
L'esempio appena visto ci dà un algoritmo per calcolare una ϵ -chiusura di uno stato:

1. Inizialmente la lista contiene solo lo stato q : $R \leftarrow (q)$
2. Poi utilizzammo un puntatore che punta al primo elemento della lista R : $p \leftarrow$
3. **While $p \neq \text{NIL}$** (nil=null). Prendiamo tutti gli elementi che si ottengono partendo dal primo elemento della lista e seguendo tutte le ϵ transizioni e le aggiungiamo alla lista, avendo l'accortezza di aggiungere solo quelle nuove, e non quelle che già appaiono in essa, quindi non creiamo ripetizioni.)
 - a. Appendi a R tutti gli elementi di $\delta(p, \epsilon) \setminus R$;
 - b. $p \leftarrow \text{successivo}(p)$ (passiamo all'elemento successivo)
4. **return R**

Seguendo questo algoritmo otterremo →

La costruzione

Dato un automa non deterministico con ϵ -transizioni, è sempre possibile costruire un automa deterministico che accetta il medesimo linguaggio, ed esiste inoltre un algoritmo che mi permette di trovare questo automa deterministico equivalente.



Sia $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ l'automa non deterministico con ϵ -transizioni.

Definisco l'automa deterministico $\mathcal{A}' = \langle Q', \Sigma, \delta', s_0, F' \rangle$ come segue:

- l'insieme degli stati è l'insieme $Q' = \wp(Q)$ costituito dai sottoinsiemi di Q ,
- lo stato iniziale è $s_0 = C_\epsilon(q_0)$,
- gli stati finali sono tutti i sottoinsiemi di Q che contengono almeno un elemento di F , cioè

$$F' = \{s \in \wp(Q) \mid s \cap F \neq \emptyset\},$$

L'insieme degli stati è costituito dalle parti dell'insieme degli stati dell'automa di partenza. Cioè uno stato del nuovo A deterministico è un insieme di stati dell'A non det.

Lo stato iniziale è la Epsilon chiusura dello stato iniziale dell'A di partenza. Quindi non è più un singleton come abbiamo visto in precedenza, ma è un insieme di stati anche il primo stato.

gli stati finali sono tutti quei sottoinsiemi che contengono almeno uno stato finale dell'A precedente.

Calcolo delta' di s,a dove

s=stato del mio nuovo A det., cioè un'insieme di stati dell'A di partenza. a= una qualunque lettera

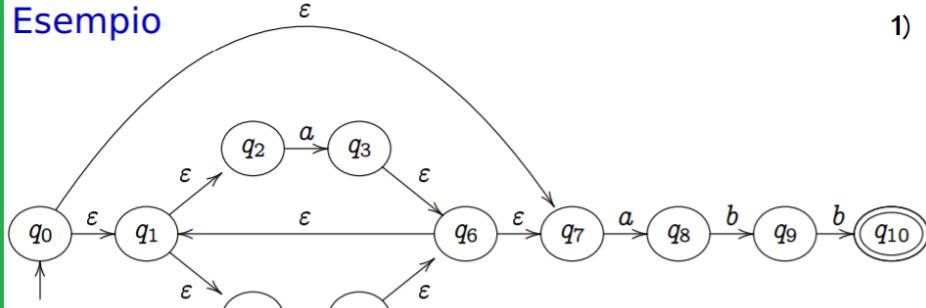
- la funzione di transizione $\delta': Q' \times \Sigma \rightarrow Q'$ è definita da

$$\delta'(s, a) = C_\epsilon \left(\bigcup_{p \in s} \delta(p, a) \right), \quad s \in \wp(Q), a \in \Sigma.$$

Per ogni stato del vecchio A che sta dentro s, vado a calcolare $\delta(p, a)$, faccio l'unione di tutti questi insiemi di stati che ottengo e poi faccio la Epsilon chiusura.

In sostanza, $\delta'(s, a)$ sarà l'insieme di tutti quegli stati che posso raggiungere nel vecchio A partendo da uno qualunque degli stati di s e poi seguendo la freccia con etichetta a, e poi tutte le frecce che voglio con etichetta Epsilon.

Esempio



2)

| j | s_j | $\delta'(j, a)$ | $\delta'(j, b)$ | $j \in F$ |
|-----|--|-----------------|-----------------|-----------|
| 0 | $\{q_0, q_1, q_2, q_4, q_7\}$ | 1 | 2 | |
| 1 | $\{q_1, q_2, q_3, q_4, q_6, q_7, q_8\}$ | 1 | 3 | |
| 2 | $\{q_1, q_2, q_4, q_5, q_6, q_7\}$ | 1 | 2 | |
| 3 | $\{q_1, q_2, q_4, q_5, q_6, q_7, q_9\}$ | 1 | 4 | |
| 4 | $\{q_1, q_2, q_4, q_5, q_6, q_7, q_{10}\}$ | 1 | 2 | ✗ |

stato chiusura di q_3 e q_8 , che corrisponde a j_1 .
 q_3 e q_8 perchè con 'a' partendo da j_0 , posso arrivare a questi due stati.

| i | $C_\epsilon(q_i)$ |
|-----|------------------------------------|
| 0 | $\{q_0, q_1, q_2, q_4, q_7\}$ |
| 1 | $\{q_1, q_2, q_4\}$ |
| 2 | $\{q_2\}$ |
| 3 | $\{q_1, q_2, q_3, q_4, q_6, q_7\}$ |
| 4 | $\{q_4\}$ |
| 5 | $\{q_1, q_2, q_4, q_5, q_6, q_7\}$ |
| 6 | $\{q_1, q_2, q_4, q_6, q_7\}$ |
| 7 | $\{q_7\}$ |
| 8 | $\{q_8\}$ |
| 9 | $\{q_9\}$ |
| 10 | $\{q_{10}\}$ |

Devo seguire da ciascuno di questi 5 stati le frecce etichettate 'a'. Gli unici stati che hanno frecce uscenti 'a', sono q_2 e q_7 . Da q_2 vado in q_3 , e da q_7 vado in q_8 . Ora si fa la Epsilon chiusura di queste copie di stati. q_8 non ha frecce uscenti etichettate Epsilon, quindi la sua chiusura si ferma a q_8 . q_3 invece contiene $q_3, q_6, q_7, q_1, q_2, q_4$. Dobbiamo quindi aggiungere alla nostra lista di stati, quelli nuovi che abbiamo trovato: q_3, q_6, q_8 . Infine controllo se lo stato è finale, ma dato che non contiene q_{10} , non lo è.

Il grafo dell'esempio sarà quindi →

Espressioni Regolari

Abbiamo detto che la teoria dei linguaggi formali può essere vista come l'arte di rappresentare degli oggetti infiniti con una descrizione finita.

Per costruire nuovi linguaggi partendo da linguaggi noti, possiamo applicare delle operazioni:

- **unione**,
- intersezione,
- **potenza**: concatenazione di n copie di un linguaggio con sé stesso. Ad es. L^2 sarà L concatenato L.
- **chiusura di Kleene**: prendo tutte le potenze di un linguaggio e ne faccio l'unione. A differenza delle prime 3 operazioni che se applicate a insiemi finiti mi ridanno insiemi finiti, la chiusura di Kleene mi dà un insieme infinito, a meno che l'insieme di partenza non sia: o l'insieme vuoto, o l'insieme che contiene la sola parola vuota. Conterrà quindi tutte le parole che posso ottenere concatenando un numero arbitrario di parole di L. $L^* = \bigcup_{n \geq 0} L^n = \{u_1 u_2 \cdots u_n \mid n \geq 0, u_1, \dots, u_n \in L\}$

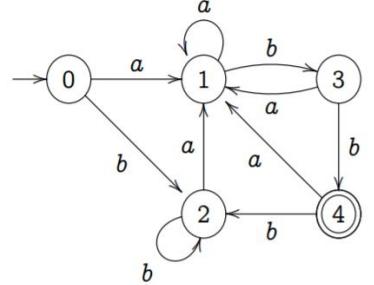
Unione, concatenazione e chiusura di Kleene sono dette **operazioni regolari**.

Quindi, i linguaggi finiti li sappiamo definire facendo una semplice lista delle parole; con questi linguaggi finiti posso poi fare le operazioni di unione e concatenazione ed ottenere nuovi linguaggi finiti, e con la chiusura di Kleene ottengo nuovi **linguaggi infiniti**.

Con questi linguaggi infiniti posso fare altre operazioni di unione e concatenazione, ottenendo altri linguaggi sempre più complessi.

Questo vuol dire che posso scrivere la definizione dei linguaggi infiniti semplicemente indicando le operazioni che eseguo per ottenerli: ovvero le **espressioni regolari**.

Le espressioni sono parole che costruisco partendo dalle lettere dell'alfabeto e dal simbolo \emptyset , seguendo le regole. Rappresentano quindi una sequenza di operazioni fatte a partire dall'alfabeto iniziale.



Definizione

ampliato
nuovo alfabeto in cui dispongo di questi simboli
che servono a denotare delle operazioni.

+ unione
* Kleene

Sia $\widehat{\Sigma}$ l'alfabeto ottenuto aggiungendo a Σ le lettere \emptyset , $+$, $*$, $($, $)$.

Si dicono **espressioni regolari** sull'alfabeto Σ le parole sull'alfabeto $\widehat{\Sigma}$ che si ottengono applicando un numero finito di volte le regole seguenti: **regole**

- (i) Ogni lettera $a \in \Sigma$ è un'espressione regolare,
 \emptyset è un'espressione regolare, unione concatenazione Kleene
- (ii) Se E e F sono espressioni regolari, allora $(E + F)$, (EF) e E^* sono espressioni regolari.

A ogni espressione regolare è associato un linguaggio, detto **linguaggio denotato dall'espressione regolare** e definito dalle regole seguenti:

- (i) per ogni $a \in \Sigma$, l'espressione regolare a denota il linguaggio $\{a\}$; l'espressione regolare \emptyset denota il linguaggio vuoto.
- (ii) Detti L_E e L_F i linguaggi denotati dalle espressioni regolari E ed F , i linguaggi denotati dalle espressioni regolari $(E + F)$, (EF) , E^* sono, rispettivamente, $L_E \cup L_F$, $L_E L_F$, L_E^* .

La chiusura di Kleene sarà denotata dall'espressione regolare del linguaggio con la $*$ vicino).

Se si omettono le parentesi, la **priorità delle operazioni** sono: chiusura di Kleene, concatenazione e poi somma.

Teorema di Kleene

Il teorema di Kleene ci dice che: un linguaggio è regolare se e solo se è riconosciuto da un automa a stati finiti.

Questa corrispondenza è effettiva, cioè:

- esiste un algoritmo che a partire da un'espressione regolare produce un automa a stati finiti, che accetta il linguaggio denotato da tale espressione (**sintesi**).
- Dato un automa a stati finiti, questo produce un'espressione regolare che denota il linguaggio accettato da tale automa. Quindi dalla macchina si ottiene la descrizione del comportamento (**analisi**).

ESEMPI

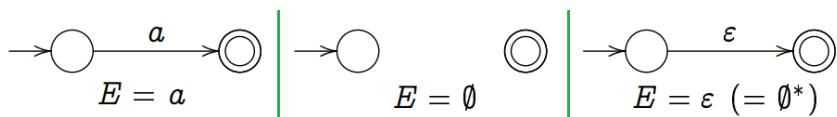
- 1 $a + b$ Espressione regolare che denota il linguaggio composto dalle due parole: a , b .
- 2 $(a + b)^*$ Chiusura di Kleene: tutte le concatenazioni di parole che appartengono all'insieme a , b . Cioè tutte le concatenazioni di lettere a , b . In sostanza è l'insieme delle parole sull'alfabeto $\{a,b\}$.
- 3 \emptyset^* Se concateno 0 parole dell'insieme vuoto: linguaggio che contiene solo la parola vuota.
- 4 $((a + b)(a + b)(a + b))^*$ ottengo tutte le parole di lunghezza 3, e poi ne faccio la chiusura di Kleene: e ottengo tutte le parole che hanno lunghezza multipla di 3.
- 5 $(a + ab)^*$
- 6 $(a + b)^* abb$ otteniamo tutte le parole che finiscono con abb .
- 7 $(b^* ab^* ab^*)^* b^* ab^*$
- 8 $(a + b + c)^* abac(a + b + c)^*$
- 9 $((a + b + c),)^*(a + b + c) & (a + b + c) + a + b + c$

Dall'Espressione Regolare all'automa (sintesi)

Costruiremo un automa a stati finiti non deterministico con ϵ -transizioni con un unico stato finale.

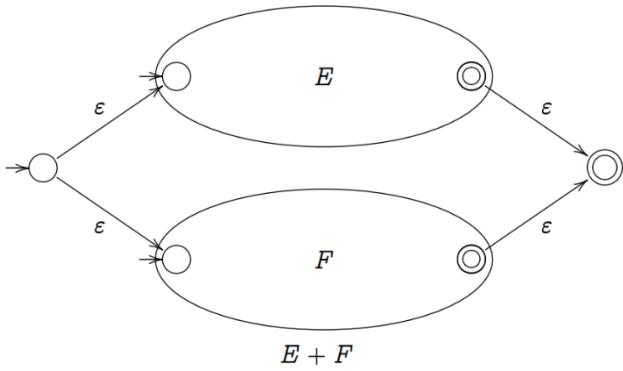
Un'espressione regolare, dove E e F sono espressioni regolari, può essere:

- O una **lettera**, un **insieme vuoto**, o una **parola vuota** (che corrisponde alla chiusura di Kleene dell'insieme vuoto).
- oppure è $(E+F)$, cioè l'**unione**.
- (EF) , cioè la **concatenazione**.
- E^* , cioè la chiusura di **Kleene**.

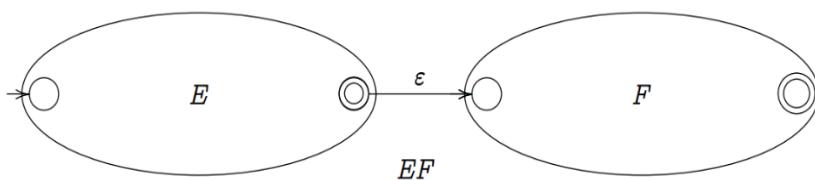


I linguaggi denotati da espressioni regolari si dicono **linguaggi regolari**. (I linguaggi di **tipo 0** sono detti anche linguaggi regolari)

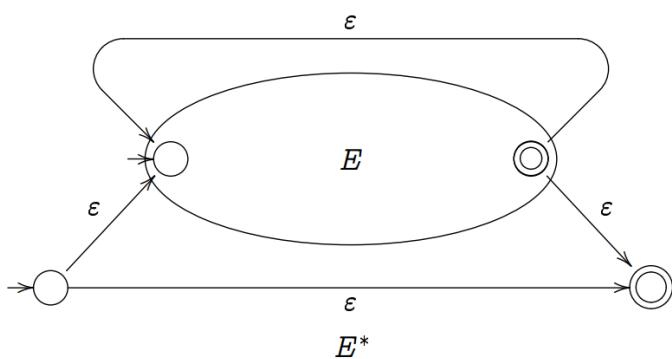
La classe dei linguaggi regolari è la più piccola famiglia di linguaggi finiti. È chiusa per le operazioni regolari (perché se due linguaggi sono denotati da espressioni regolari, la loro unione sarà denotata anche lei da un'espressione regolare. Lo stesso per la concatenazione).



Unione: avendo già costruito gli automi di E e F (gli ovali nella figura), si aggiunge un nuovo stato iniziale e un nuovo stato finale, e due ϵ transizioni che mi portano dallo stato iniziale nuovo agli stati iniziali vecchi, e due ϵ transizioni che mi portano dagli stati finali vecchi allo stato finale nuovo. Questo nuovo automa accetta l'unione dei linguaggi, accettati dai due automi E e F. Quindi tutti i cammini riusciti dallo stato iniziale a quello finale, avranno come etichetta (l'ovale) una parola accettata dall'automa E o F.



Concatenazione: si connette con una ϵ transizione lo stato finale del primo automa con lo stato iniziale del secondo automa. Lo stato iniziale del nuovo automa sarà quello del primo automa, mentre il suo stato finale sarà quello del secondo automa.



Kleene: abbiamo l'automa E che riconosce una certa espressione, e poi il nuovo automa che riconosce la chiusura di Kleene del linguaggio accettato da questo, ovvero che accetti tutte le parole che si ottengono concatenando 0 o più parole accettate dall'automa di partenza.

Aggiungo una ϵ transizione che dallo stato finale mi riporta a quello iniziale: così da accettare tutte le parole che si ottengono concatenando una o più parole

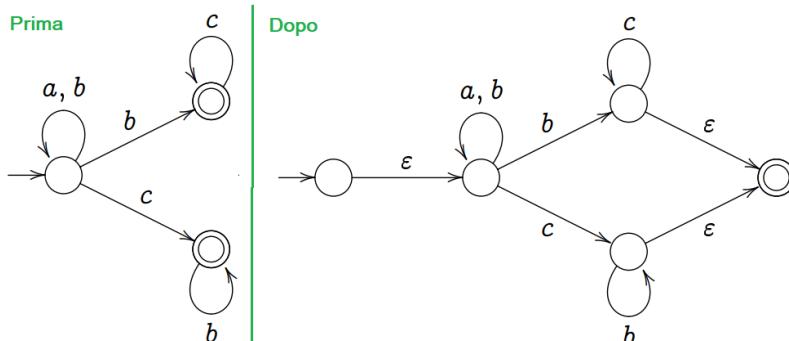
dell'automa di partenza. Mi manca però il riconoscimento della parola vuota, quindi aggiungo un nuovo stato iniziale e un nuovo stato finale e poi una ϵ transizione dallo stato iniziale nuovo a quello vecchio, una ϵ transizione dallo stato finale vecchio a quello nuovo, e una ϵ transizione dallo stato iniziale nuovo allo stato finale nuovo.

Proposizione Data un'espressione regolare G , si può effettivamente costruire un automa a stati finiti che riconosce il linguaggio denotato da G .

← In breve, l'algoritmo di sintesi.

- ➊ se G è un'espressione regolare di base (lettera o insieme vuoto), allora restituisco l'automa corrispondente;
- ➋ se $G = (E + F)$, allora calcolo gli automi A_1 e A_2 corrispondenti a E e F ; costruisco l'automa dell'espressione $(E + F)$;
- ➌ se $G = (EF)$, allora calcolo gli automi A_1 e A_2 corrispondenti a E e F ; costruisco l'automa dell'espressione (EF) ;
- ➍ se $G = E^*$, allora calcolo l'automa A_1 corrispondente a E ; costruisco l'automa dell'espressione E^* ;

Avendo un automa A, vogliamo che: abbia un unico stato finale e, per semplicità, non abbia frecce che entrano nello stato iniziale né frecce che escono dallo stato finale. Questo implica che un percorso uscito dallo stato iniziale non potrà più rientrarci (perché non ci sono frecce entranti in esso) così come non potrà passare per lo stato finale se non all'ultimo (perché non ci sono frecce uscenti da esso).



Aggiungo quindi delle ϵ transizioni che mi portano dal nuovo stato iniziale al vecchio stato iniziale, e dai vecchi stati finali al nuovo stato finale.

Questo nuovo automa accetterà le stesse parole dell'automa vecchio.

Inoltre, supponiamo anche di dare dei nomi ai vari stati del nostro automa: q_1, q_2, \dots, q_n , dove lo stato iniziale corrisponde al q_{n-1} (quindi al penultimo stato) e lo stato finale a q_n .

Detto ciò, vediamo ora come possiamo costruire un'espressione regolare partendo da tale automa.

Prendiamo dei linguaggi che in qualche modo approssimano il linguaggio che vogliamo cercare e cerchiamo delle espressioni regolari per questi linguaggi.

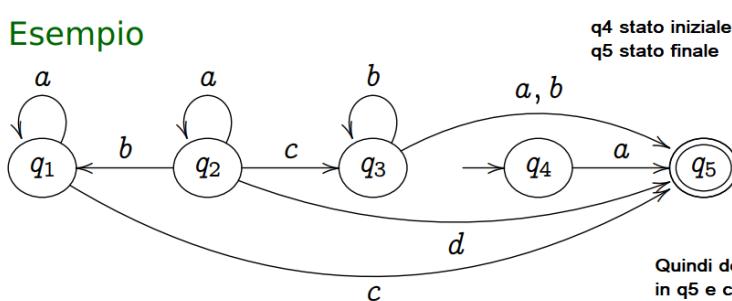
Per ogni i, j, k con $0 \leq k < i, j \leq n$ consideriamo l'insieme L_{ijk} contenente le seguenti parole:

indici i e j compresi tra 1 e n, mentre k compreso tra 0 e n.

le etichette dei cammini nel grafo di A che iniziano in q_i , terminano in q_j , e attraversano solo stati di indice $\leq k$.

L'insieme che contiene tutte le etichette dei cammini che iniziano in q_i , terminano in q_j , e attraversano solo stati di indice $\leq k$ (che li attraversi solo, non che parta o arrivi in uno di questi stati).

Esempio



Quindi devo prendere tutti i cammini che partono da q_2 , che arrivano in q_5 e che attraversano solo q_1 .

Nel calcolo di L_{251} devo considerare i cammini:

ma non:

$$\begin{aligned} &q_2 \xrightarrow{d} q_5, \\ &q_2 \xrightarrow{b} q_1 \xrightarrow{c} q_5 \\ &q_2 \xrightarrow{b} q_1 \xrightarrow{a} q_1 \xrightarrow{a} \dots \xrightarrow{a} q_1 \xrightarrow{c} q_5 \end{aligned}$$

$$\begin{aligned} &q_2 \xrightarrow{a} q_2 \xrightarrow{d} q_5 \\ &q_2 \xrightarrow{c} q_3 \xrightarrow{a} q_5 \\ &q_2 \xrightarrow{c} q_3 \xrightarrow{b} q_3 \xrightarrow{a} q_5 \end{aligned}$$



Il linguaggio accettato sarà quindi: $L(A) = L_{n-1 \ n \ n-2}$

Se riesco a trovare un'espressione regolare per tutti gli L_{ijk} , avrò un'espressione regolare per $L_{n-1 \ n \ n-2}$, che è proprio il linguaggio accettato dall'automa. Ho quindi ottenuto l'espressione regolare corrispondente al linguaggio.

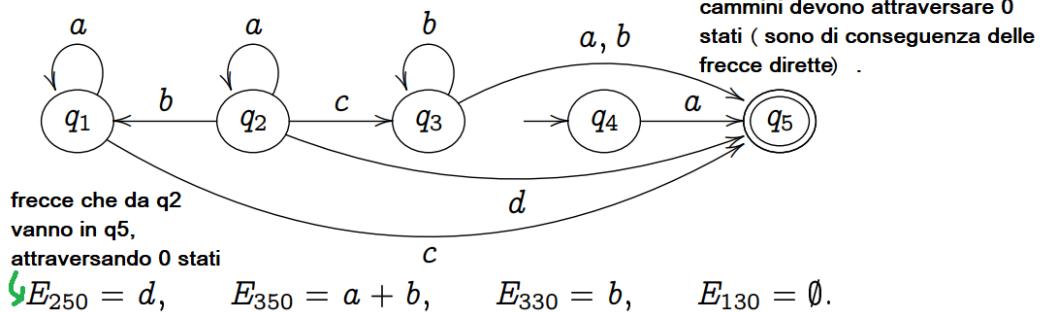
Quello che dobbiamo mostrare è un meccanismo che ci permetta di determinare delle espressioni regolari che denotano questi linguaggi L_{ijk} .

Per ottenere tutti questi L_{ijk} , si procede per induzione su k .

Il linguaggio L_{ij0} è denotato dall'espressione regolare

$$E_{ij0} = \begin{cases} \text{somma delle etichette delle frecce da } q_i \text{ a } q_j, \text{ se ne esistono} & \leftarrow \text{Supponendo che k=0.} \\ \emptyset, & \text{altrimenti} \end{cases}$$

Esempio

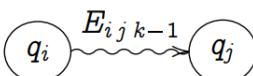


Supponiamo $k \geq 1$.

I cammini nel grafo di \mathcal{A} con origine in q_i , termine in q_j e che attraversano esclusivamente stati di indice $\leq k$ si possono ripartire in due categorie:

Passo induttivo: come faccio per ogni k , a calcolare E_{ijk} , a partire da E_{ijk-1} . Dobbiamo prendere:

- 1 ● cammini da q_i a q_j che attraversano solo stati di indice $\leq k - 1$, Questi cammini per q_k potrebbero non passare proprio, oppure possono passare per q_1, q_2, \dots, q_{k-1} . Per questi l'espressione regolare già l'abbiamo.



- 2 ● i cammini costituiti dalla concatenazione di

- un segmento da q_i a q_k ,
- zero o più cammini con origine e termine in q_k ,
- un segmento finale con origine da q_k a q_j

tutti che attraversano solo stati di indice $\leq k - 1$.

I cammini che passano una o più volte per q_k .

Quindi la concatenazione di:
 1. Un segmento dallo stato iniziale a q_k .
 2. dei cammini che da q_k arrivano al secondo passaggio con q_k , e così via, fino all'ultimo passaggio con q_k .
 3. Un ultimo segmento che da q_k arriva allo stato finale.



Ora dobbiamo trovare l'espressione E_{ijk} , cioè l'espressione per il linguaggio costituito dalle etichette di tutti i cammini delle forme precedenti. Innanzitutto facciamo l'unione dei cammini del primo punto, con i cammini del secondo punto. I cammini del secondo gruppo saranno a loro volta riportati come la concatenazione del primo segmento, con i segmenti intermedi (concatenazione di 0 o più copie di un'espressione, utilizzando la chiusura di Kleene), e l'ultimo segmento.

$$E_{ijk} = \underline{E_{ijk-1}} + \underline{(E_{ikk-1})(E_{kkk-1})^*(E_{kjkk-1})}, \quad 1 \leq k < i, j \leq n.$$

In questo modo è possibile calcolare tutte le espressioni regolari, compresa $E_{n-1}n^{n-2}$, che denota il linguaggio accettato dall'automa.

Metodo di Eliminazione degli Stati

Per fare tutto ciò utilizzeremo il

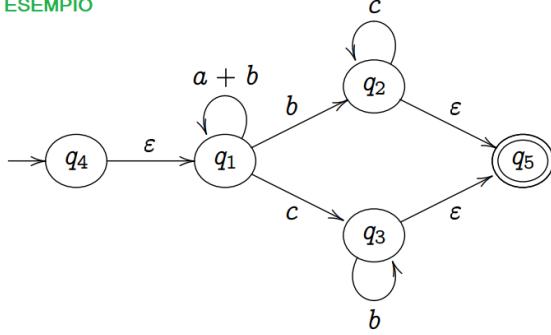
Metodo di eliminazione degli stati.

Avendo il nostro automa che corrisponde ad un grafo, sulle cui etichette delle frecce ci sono espressioni particolari che possono essere o lettere, o ϵ o somma di lettere.

Metodo di eliminazione degli stati

- 1 selezioniamo uno stato q_k che non sia né iniziale né finale;
- 2 per ogni freccia $q_i \xrightarrow{E_{ik}} q_k$ che entra nello stato q_k e ogni freccia $q_k \xrightarrow{E_{kj}} q_j$ che esce da tale stato ($i, j \neq k$)
 - 1 rimpiazziamo l'etichetta della freccia $q_i \xrightarrow{E_{ij}} q_j$ con $E_{ij} + E_{ik}E_{kk}^*E_{kj}$ (o $E_{ij} + E_{ik}E_{kj}$ qualora manchi la freccia da q_k a q_k);
 - 2 se tale freccia è assente, la creiamo (con etichetta $E_{ik}E_{kk}^*E_{kj}$, ovvero $E_{ik}E_{kj}$);
- 3 rimuoviamo lo stato q_k e tutte le frecce che entrano o escono da esso;
- 4 ripetiamo la procedura fino a ottenere un automa con due stati e una sola freccia;
- 5 l'etichetta di tale freccia è l'espressione regolare cercata.

ESEMPIO



1.

Prima devo trovare: E_{ij0} , $0 < i, j < 5$ cioè tutte le frecce che da q_1 arrivano in q_j

Es.: E₂₅₀, cerco la freccia che va da q_2 a q_5 , e trovo scritto Epsilon.

2.

Ora applichiamo la formula ricorsiva di $E_{ijk} = \dots$ e troviamo:

$$E_{421} = E_{420} + E_{410}(E_{110})^*E_{120} = \emptyset + \epsilon(a+b)^*b = (a+b)^*b,$$

$$E_{431} = E_{430} + E_{410}(E_{110})^*E_{130} = \emptyset + \epsilon(a+b)^*c = (a+b)^*c,$$

$$E_{ij1} = E_{ij0}, \text{ nei casi rimanenti}$$

Perchè sommare qualcosa all'insieme vuoto, lascia la cosa così com'è, e stessa cosa se gli viene sommata la parola vuota.

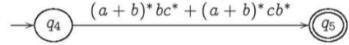
Andando avanti:

E_{453}

$$E_{453} = E_{452} + E_{432}(E_{332})^*E_{352} = (a+b)^*bc^* + (a+b)^*cb^* = (a+b)^*bc^* + (a+b)^*cb^*,$$

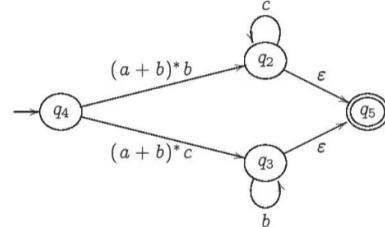
$$E_{ij3} = E_{ij2} = \emptyset, \text{ nei casi rimanenti}$$

Otterremo l'automa:



Che corrisponde finalmente all'espressione regolare del nostro linguaggio.

Applicando tutto ciò otterremo il seguente automa:



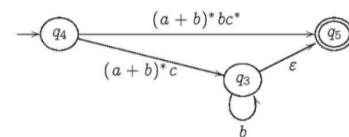
che ha come etichette delle frecce, delle espressioni regolari.

Facendo lo stesso procedimento di prima su questo nuovo automa:

$$E_{ij2}, \quad 2 < i, j < 5$$

$$E_{452} = E_{451} + E_{421}(E_{221})^*E_{251} = \emptyset + (a+b)^*bc^*\epsilon = (a+b)^*bc^*, \\ E_{ij2} = E_{ij1}, \text{ nei casi rimanenti}$$

Otterremo un altro automa:



N.B.: Questo metodo si chiama "di eliminazione degli stati" perché man mano stiamo togliendo stati all'automa originario.

Ogni volta che calcoliamo le espressioni regolari al secondo passo, andiamo ad aggiungerle alle espressioni già presenti nell'etichetta della freccia corrispondente; oppure, se non esiste ancora una freccia, la creiamo assegnandogli quell'espressione regolare come etichetta.

Operazioni Booleane

Altra conseguenza del teorema di Kleene: la classe dei linguaggi regolari è un'**algebra booleana**.

Ci sono delle operazioni che sono naturali sulle espressioni regolari e altre operazioni che sono naturali sugli automi a stati finiti.

Come abbiamo detto in precedenza, infatti, la famiglia dei linguaggi regolari è chiusa per le operazioni di unione, concatenazione e chiusura di Kleene. Ed è anche chiusa per complemento ed intersezione. Quindi per tutte le operazioni booleane.

- **Complemento:** per il Teorema di Kleene, un linguaggio regolare è accettato da un automa a stati finiti deterministico. Le parole che prima erano accettate, ora con il complemento non devono essere accettate, e viceversa. Ciò si ottiene scambiando stati finali con stati non finali e viceversa. Quindi, per il Teorema di Kleene, è regolare.
- **Intersezione:** l'intersezione si esprime per mezzo di unione e complemento. Quindi se faccio l'intersezione di due linguaggi regolari, otterrò un linguaggio regolare:

$$L \cap M = \Sigma^* - ((\Sigma^* - L) \cup (\Sigma^* - M))$$

- **Differenza Insiemistica:** anch'essa si esprime per mezzo di unione e complemento. Quindi se faccio la differenza insiemistica di due linguaggi regolari, otterrò un linguaggio regolare.

$$L - M = \Sigma^* - ((\Sigma^* - L) \cup M)$$

Automa deterministico con il minor numero di stati.

L'**automa di Nerode** di un linguaggio regolare L è un automa deterministico che accetta L col minimo numero di stati possibile. Quindi definisco prima la congruenza di Nerode, così da definire un'equivalenza cioè una partizione dell'insieme Σ^* in un numero finito di classi di equivalenza. Costruisco in seguito un automa minimo partendo da queste classi di equivalenza, che accetta il linguaggio L .

Equivalenza: insieme sul quale è definita una relazione **riflessiva** (cioè ogni elemento è equivalente a se stesso), **simmetrica** (se x è equivalente a y , allora y è equivalente a x) e **transitiva** (se abbiamo tre elementi e il primo è equivalente al secondo, il secondo è equivalente al terzo, allora il primo è equivalente al terzo).

C'è una corrispondenza tra la nozione di equivalenza, quindi un insieme di coppie che soddisfa le proprietà citate sopra, e la **nozione di partizione** su un insieme, cioè la divisione di un insieme in sottoinsiemi che sono a due a due disgiunti (privi di elementi in comune), e che uniti mi danno tutto l'insieme.

Quindi se ho un'equivalenza su un insieme, le classi di equivalenza (cioè l'insieme di tutti gli elementi che sono equivalenti l'uno all'altro), costituiscono una partizione sull'insieme.

Allo stesso modo se ho una partizione, gli insiemi che la costituiscono sono classi di un'opportuna equivalenza.

Congruenza destra: qui non abbiamo solo un insieme, ma un insieme che ha un'operazione definita su di esso. Ogni volta che si va a concatenare a dx di due parole equivalenti, una lettera, ottengo due parole equivalenti (rispettivamente per la congruenza sx). Una relazione che è contemporaneamente una congruenza dx e sx, si dice congruenza. Una relazione di congruenza si denota con \sim .

Esempio

Si considerino le relazioni \sim su Σ^* definite da

prop. riflessiva

Equivalenza e congruenza
(è sia dx che sx)

- 1 $u \sim v$ se u e v iniziano con la stessa lettera (o $u = v = \epsilon$)
- 2 $u \sim v$ se u e v terminano con la stessa lettera (o $u = v = \epsilon$)
- 3 $u \sim v$ se u e v hanno la stessa lunghezza
- 4 $u \sim v$ se u e v le prime due lettere di u sono una permutazione delle prime due lettere di v (o $u = v$)

Equivalenza (rispetta tutte le prop. di una equivalenza), e congruenza

E' un equivalenza e una congruenza dx, ma non sx.

Per es. abaa \sim bab, aabb \sim aaabaa, ma aabb $\not\sim$ baaabaa.

Equivalenza di Nerode

Tra le congruenze c'è una famiglia particolarmente interessante: l'equivalenza di Nerode.

Partendo da un linguaggio L sull'alfabeto Σ , l'equivalenza di Nerode è un'equivalenza sull'insieme Σ^* .

Prendiamo due parole su Σ^* e diciamo che sono equivalenti se, per ogni parola y , si ha $uy \in L$ se e soltanto se $vy \in L$.

Quindi diciamo che due parole sono equivalenti se le parole che mi permettono di completare la prima (u , a dx) dentro una parola di L , sono esattamente le stesse che mi permettono di completare v in una parola di L .

Questo teorema ci dà una caratterizzazione vera e propria dei linguaggi regolari sottoforma algebrica: un linguaggio è regolare se lo posso ottenere come unione di classi di una congruenza dx di indice finito.

Le **proprietà** dell'Equivalenza di Nerode sono:

- **La relazione N_L è una congruenza dx.**

Ogni parola è in un'equivalenza N_L con sé stessa (prop. Riflessiva) perché i suoi completamenti in L sono gli stessi suoi completamenti in L . Se u e v hanno gli stessi completamenti, allora v e u avranno gli

stessi completamenti (prop. Simmetrica). Se u e v hanno gli stessi completamenti, e v e w hanno gli stessi completamenti, allora u e w hanno gli stessi completamenti (prop. Transitiva).

- **L è unione di classi di equivalenza di N_L .**

Il linguaggio L si può decomporre in classi di equivalenza N_L . In altre parole, N_L è un'equivalenza quindi ci determina quella partizione in classi di equivalenza dell'insieme Σ^* . Ci sono queste famiglie di parole che sono tutte equivalenti tra loro, ogni parola sta in una classe di equivalenza e classi equivalenza diverse non hanno elementi comuni.

- ➊ N_L è una congruenza destra,

Dimostrazione

N_L è un'equivalenza.

Dobbiamo verificare che per ogni $u, v \in \Sigma^*$, $a \in \Sigma$, se $u N_L v$, allora $ua N_L va$.

Invero sia $u N_L v$ e $a \in \Sigma$.

Dalla definizione di N_L si ha $uy \in L \iff vy \in L$ per ogni $y \in \Sigma^*$

Sostituendo y con ay , ottengo $uay \in L \iff vay \in L$

Quindi si ha anche $ua N_L va$.

- ➋ L è unione di classi di equivalenza di N_L ,

Dimostrazione

Sia $u N_L v$. Allora $uy \in L \iff vy \in L$ per ogni $y \in \Sigma^*$.

Prendendo $y = \epsilon$ ottengo $u \in L \iff v \in L$.

Quindi ogni classe di equivalenza o sta tutta dentro L o sta tutta nel complemento.

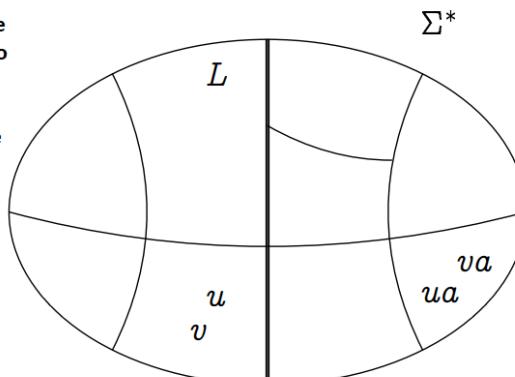
Teorema (Myhill-Nerode, 1958) Sia L un linguaggio sull'alfabeto Σ .

Le seguenti proposizioni sono equivalenti:

- L è regolare,
- L è unione di classi di una congruenza destra su Σ^* di indice finito,
- N_L ha indice finito. (indice = numero di classi di equivalenza della equivalenza stessa.)

L'ovale rappresenta l'insieme di tutte le parole sull'alfabeto sigma.

Viene partizionato in 2 parti:
 1) Linguaggio L , cioè le parole che stanno nel linguaggio L .
 2) Le parole che stanno fuori dal linguaggio L e che non appartengono ad esso.



Posso vedere queste classi come gli stati di un Automa, in cui la transizione con la lettera 'a' mi porta da una classe che contiene le parole u,v a una classe che contiene le parole ua,va.

Sopra quest'ovale diviso in un numero finito di pezzi, ci costruiremo sopra l'Automa del linguaggio L .

Dobbiamo ora **dimostrare** che le 3 condizioni della slide sopra (i, ii, iii) sono equivalenti, quindi se è vera una di esse, sono vere tutte e tre. Per fare ciò utilizzeremo una dimostrazione circolare: se è vera i è vera ii, se è vera ii è vera iii, e se è vera iii è vera i.

i → ii :

Se **L regolare** è accettato da un automa deterministico (e posso supporre che tutti gli stati siano accessibili).

Per ogni stato di questo automa, vado a considerare l'insieme L_q , cioè l'insieme delle parole che mi portano dallo stato iniziale allo stato q .

L_q costituisce una partizione di Σ^* . Per dire che è una partizione devo vedere che gli insiemi siano a

due a due disgiunti ed è chiaro che ogni parola dell'automa deterministico mi porta in un solo stato, quindi ogni parola apparterrà ad uno solo degli insiemi L_q che sto definendo. E ogni parola sta dentro ad uno di questi stati.

Quindi L_q è effettivamente una partizione ovvero una famiglia di sottoinsiemi di Σ^* tali che gli insiemi sono disgiunti a due a due, non hanno elementi comuni e la loro unione è tutto Σ^* .

L è l'unione delle classi L_q con q che è stato finale.

Ora dobbiamo verificare che sia una congruenza \sim .

- supponiamo $u \sim v$, $u, v \in \Sigma^*$; u e v equivalenti, cioè fanno parte della stessa classe di equivalenza L_q .
- allora $u, v \in L_q$ per qualche stato q ;
- quindi $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v) = q$;
- pertanto, per ogni lettera a , $\hat{\delta}(q_0, ua) = \hat{\delta}(q_0, va) = \delta(q, a) = q'$;
- cioè, $ua, va \in L_{q'}$, per cui $ua \sim va$.

ii → iii :

L unione di classi di una congruenza destra di indice finito $\Rightarrow \mathcal{N}_L$ ha indice finito, (in particolare dobbiamo dimostrare che la congruenza di Nerode ha indice finito)

- L sia unione di classi di una congruenza destra \sim su Σ^* di indice finito;
- basta mostrare che ogni classe di \mathcal{N}_L è unione di classi di \sim ;
- cioè che se $u \sim v$ allora $u \mathcal{N}_L v$, $u, v \in \Sigma^*$;
 - supponiamo $u \sim v$, $u, v \in \Sigma^*$;
 - dato che \sim è una congruenza destra, si ha $ua \sim va$ per ogni $a \in \Sigma$;
 - più in generale, si ha $uy \sim vy$ per ogni $y \in \Sigma^*$;
 - visto che L è unione di classi di \sim , ne segue che $uy \in L \iff vy \in L$ e quindi $u \mathcal{N}_L v$.
- abbiamo mostrato che ogni classe di \mathcal{N}_L è unione di classi di \sim e quindi contiene almeno una classe di \sim ;
- ne concludiamo che l'indice di \mathcal{N}_L è minore o uguale a quello di \sim e, pertanto, è finito.

iii → i :

\mathcal{N}_L ha indice finito $\Rightarrow L$ è regolare,

- Per il teorema di Kleene, basta costruire un automa che accetta L ;
- sia \mathcal{A} l'automa ottenuto nel modo seguente (automa di Nerode di L):
 - gli stati sono le classi dell'equivalenza \mathcal{N}_L ,
 - la funzione di transizione δ è definita nel modo seguente:
Siano $q \in Q$ e $a \in \Sigma$. Poiché q è una classe dell'equivalenza \mathcal{N}_L , tutte le parole ua con $u \in q$ appartengono a un'unica classe p dell'equivalenza \mathcal{N}_L . Poniamo allora $\delta(q, a) = p$.
 - lo stato iniziale q_0 è la classe d'equivalenza della parola vuota,
 - gli stati finali sono le classi d'equivalenza incluse in L .
- per ogni $w \in \Sigma^*$, $\hat{\delta}(q_0, w)$ è la classe di equivalenza della parola w ;
- poiché L è unione di classi dell'equivalenza \mathcal{N}_L , tale stato è finale se e solo se $w \in L$.
- se ne conclude che il linguaggio accettato da \mathcal{A} è proprio L .

Ritornando all'**automa minimo**, vediamone la dimostrazione:

Proposizione Sia L un linguaggio regolare. L'automa di Nerode di L è un automa deterministico che accetta L col minimo numero di stati possibile.

Dimostrazione

- Sia \mathcal{A} l'automa deterministico che accetta L col minimo numero di stati possibile e sia n tale numero;
- nella dimostrazione (i) \Rightarrow (ii) si è visto che L è unione di classi di una congruenza destra \sim di indice n ;
- nella dimostrazione (ii) \Rightarrow (iii) si è visto che l'indice di \mathcal{N}_L è minore o uguale di quello di \sim ;
- quindi l'indice di \mathcal{N}_L è minore o uguale a n ;
- nella dimostrazione (iii) \Rightarrow (i) si è costruito un automa che accetta L con un numero di stati pari all'indice di \mathcal{N}_L ;
- quindi l'indice di \mathcal{N}_L è maggiore o uguale a n ;
- ne segue l'asserto.

Problema della minimizzazione

Dato un automa a stati finiti deterministico $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$, costruire un automa deterministico equivalente ad \mathcal{A} col minimo numero di stati.

([la costruzione dell'automa minimo](#))

←Costruzione
dell'**Automa di Nerode**.

- Possiamo ridurci al caso in cui tutti gli stati siano accessibili;
- gli stati dell'automa di Nerode sono le classi di \mathcal{N}_L ...
- ma anche unione di insiemi L_q con $q \in Q$ (cf. dimostrazione del Teorema di Nerode, (ii) \Rightarrow (iii));
- insomma, la partizione di Σ^* nelle classi di \mathcal{N}_L induce una partizione di Q ; Ogni classe di \mathcal{N}_L contiene delle classi L_q e quindi corrisponde ad un insieme di stati q .
- tale partizione gode delle seguenti tre proprietà:
 - 1 l'insieme F è unione di classi,
 - 2 se p e q sono nella medesima classe, allora per ogni lettera $a \in \Sigma$, $\delta(p, a)$ e $\delta(q, a)$ cadono in una stessa classe,
 - 3 è la meno fine tra le partizioni che soddisfano le due condizioni precedenti;
- il problema della minimizzazione si ridurrà pertanto alla ricerca della partizione di Q che soddisfa tali condizioni

Algoritmo di minimizzazione

La partizione di Q che soddisfa le precedenti Condizioni 1–3 si ottiene per approssimazioni successive: costruiamo la sequenza di partizioni Π_n di Q definite come segue:

stati finali separati da quelli non finali.

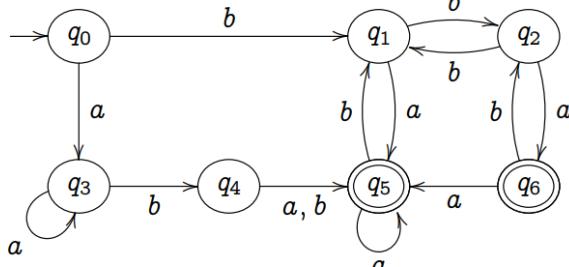
- La partizione Π_0 ha le sole due classi F e $Q \setminus F$;
- per ogni $n \geq 0$, le classi di Π_{n+1} si ottengono spezzando quelle di Π_n in modo da 'separare' le coppie di stati che non soddisfano la Condizione 2; spezzo una classe in modo da separare le coppie di stati che non soddisfano la condizione 2 vista prima, cioè che hanno l'immagine che va in due classi differenti.
- ci arrestiamo non appena $\Pi_{n+1} = \Pi_n$. Continuo a spezzare classi finché non posso spezzare più nulla e mi fermo. Creando così il minor numero di classi possibile.

L'automa minimo di L si ottiene nel modo seguente

- gli stati sono le classi della partizione Π ,
- per ogni classe C e ogni lettera $a \in \Sigma$, $\delta'(C, a)$ è la classe che contiene tutti gli stati $\delta(q, a)$ con $q \in C$,
- lo stato iniziale è la classe C_0 che contiene lo stato iniziale q_0 di \mathcal{A} ,
- gli stati finali sono le classi contenute in F .

-
1. Q e F devono essere un'unione di classi.
 2. Se due elementi stanno nella stessa classe, la loro immagine deve stare nella stessa classe (e il complementare).
 3. Minor numero di classi possibile.

ESEMPIO Voglio minimizzare il seguente automa deterministico.



Innanzitutto divido l'insieme degli stati in due classi: stati finali e stati non finali.

$$\Pi_0 = (Q \setminus F, F) = (\{q_0, q_1, q_2, q_3, q_4\}, \{q_5, q_6\}).$$

Dopodichè vado a vedere le lettere (a,b) su ogni stato, in quale classe mi riportano.

| | 0 | | | | | 1 | | Lo stato q_0 , con la lettera 'a' mi porta in q_3 , che fa parte della classe 0 (la prima classe composta da q_0, q_1, q_2, q_3 e q_4). Quindi scrivo 0. |
|---|-------|-------|-------|-------|-------|-------|-------|---|
| | q_0 | q_1 | q_2 | q_3 | q_4 | q_5 | q_6 | |
| a | 0 | 1 | 1 | 0 | 1 | 1 | 1 | |
| b | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |

Ora vado a vedere in ognuna delle singole classi le colonnine corrispondenti ad ogni stato. (Es.: q_0 ha la colonnina 0,0; q_1 1,0; ecc...).

Gli stati che hanno colonnine uguali e che si trovano nella stessa classe, devono rimanere insieme: quindi q_0 e q_3 hanno entrambi colonnina 0,0 e devono restare insieme.

$$\Pi_1 = (\{q_0, q_3\}, \{q_1, q_2\}, \{q_4\}, \{q_5, q_6\}).$$

Anche se q_2 e q_5 hanno la stessa colonnina rimangono separati perché si trovano già in classi diverse.

Ora ci ritroviamo con 4 classi:

| | | | |
|----------|----------|----------|----------|
| classe 0 | classe 1 | classe 2 | classe 3 |
|----------|----------|----------|----------|

$$\Pi_1 = (\{q_0, q_3\}, \{q_1, q_2\}, \{q_4\}, \{q_5, q_6\}).$$

Ripeto lo stesso identico meccanismo di prima e mi ritroverò con 5 classi.

| | 0 | | 1 | | 2 | | stati finali |
|---|-------|-------|-------|-------|-------|-------|--------------|
| | q_0 | q_3 | q_1 | q_2 | q_4 | q_5 | |
| a | 0 | 0 | 3 | 3 | 3 | 3 | |
| b | 1 | 2 | 1 | 1 | 3 | 1 | |

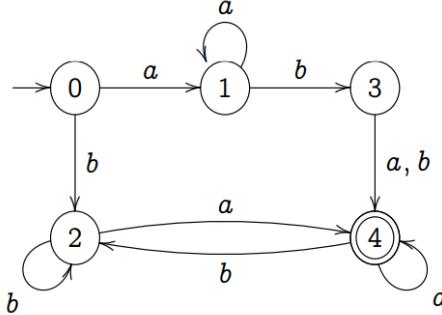
$$\Pi_2 = (\{q_0\}, \{q_3\}, \{q_1, q_2\}, \{q_4\}, \{q_5, q_6\}).$$

Rifaccio la tabellina con queste nuove classi, e mi rendo conto che non c'è più nulla da spezzare. Il partizionamento è finito. Gli stati dell'automa saranno le classi di equivalenza con indice 0, 1, 2, 3, 4.

| | 0 | | 1 | | 2 | | 3 | | 4 | |
|---|-------|-------|-------|-------|-------|-------|-------|-------|-------|--|
| | q_0 | q_3 | q_1 | q_2 | q_4 | q_5 | q_6 | q_5 | q_6 | |
| a | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | |
| b | 2 | 3 | 2 | 2 | 4 | 2 | 2 | 2 | 2 | |

$$\Pi_3 = \Pi_2.$$

Il nuovo automa minimizzato è il seguente, e avrà 5 stati invece dei 7 iniziali.



Grammatiche Regolari

24.03.21

Una grammatica a struttura di frase $G = \langle V, \Sigma, P, S \rangle$ è di tipo 3 se tutte le produzioni hanno le forme $X \rightarrow aY$, $X \rightarrow a$, con $X, Y \in N$ (x e y sono variabili), $a \in \Sigma$ (a è un terminale).

Nel caso in cui S (simbolo iniziale) non compaia nei lati destri delle produzioni è ammessa anche la produzione $S \rightarrow \epsilon$. Fondamentale per far sì che il linguaggio possa produrre la parola vuota.

In ogni derivazione ho S all'inizio, ma dopo non ce l'avrà mai più perché si trova solo a sx. Quindi questa produzione $S \rightarrow \epsilon$ serve esclusivamente per generare la parola vuota perché, o la uso al primo passo e ho finito (perché ho derivato solo la parola vuota), oppure non la uso proprio.

Un **linguaggio è regolare** se e solo se è generato da una **grammatica di tipo 3**.

Data G , si può effettivamente costruire un automa a stati finiti che accetta il linguaggio generato da G . Viceversa, dato un automa a stati finiti \mathcal{A} , si può effettivamente costruire una grammatica di tipo 3 che genera il linguaggio accettato da \mathcal{A} .

Dalla grammatica all'automa

- Sia $G = \langle V, \Sigma, P, S \rangle$ una grammatica di tipo 3,
- per semplicità, supponiamo che non abbia la produzione $S \rightarrow \epsilon$,
- costruiamo l'automa non deterministico \mathcal{A} nel modo seguente:
 - gli stati sono le variabili della grammatica G e la parola vuota,
 - per ogni produzione $X \rightarrow aY$, nel grafo di \mathcal{A} c'è la freccia $X \xrightarrow{a} Y$,
 - per ogni produzione $X \rightarrow a$, nel grafo di \mathcal{A} c'è la freccia $X \xrightarrow{a} \epsilon$,
 - lo stato iniziale è S ,
 - l'unico stato finale è ϵ .
- Si verifica che $L(G) = L(\mathcal{A})$.

Dobbiamo verificare che l'automa costruito accetta lo stesso linguaggio generato da G .

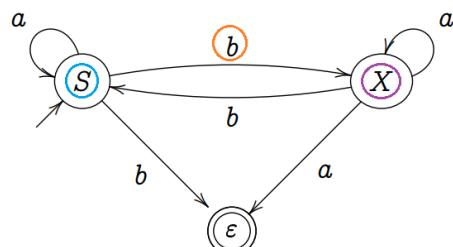
Se $S \rightarrow \varepsilon$ è una produzione di G (e quindi S non compare nei lati destri delle produzioni), allora occorrerà aggiungere S all'insieme degli stati finali di \mathcal{A} .

ESEMPIO: DALLA GRAMMATICA ALL'AUTOMA

Se G ha le produzioni

$$S \rightarrow aS, \quad S \rightarrow b, \quad X \rightarrow aX, \quad X \rightarrow bS, \quad X \rightarrow a,$$

allora $L(G)$ è accettato dall'automa



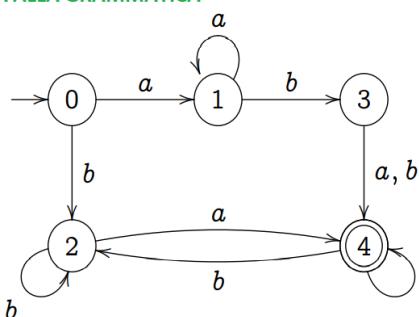
Dall'Automa alla Grammatica

- Sia $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ un automa a stati finiti, privo di ε -transizioni,
- per semplicità, supponiamo $\varepsilon \notin L(\mathcal{A})$, supponiamo che non accetti la parola vuota.
- costruiamo la grammatica G nel modo seguente:
 - le variabili della grammatica G sono gli stati dell'automa \mathcal{A} ,
 - per ogni freccia $X \xrightarrow{a} Y$ nel grafo di \mathcal{A} , aggiungiamo a G la produzione $X \rightarrow aY$,
 - se Y è uno stato finale, aggiungiamo a G anche la produzione $X \rightarrow a$,
 - il simbolo iniziale è lo stato iniziale $S = q_0$.
- Si verifica che $L(G) = L(\mathcal{A})$ (dimostrazione analoga al caso precedente).

Se $\varepsilon \in L(\mathcal{A})$ (se il linguaggio contiene la parola vuota), sarà necessario modificare la grammatica G in modo da fargli generare anche la parola vuota. Se però S compare nel lato destro di qualche produzione, non è possibile aggiungere la produzione $S \rightarrow \varepsilon$. Dovremo pertanto prima introdurre una nuova variabile S' le cui produzioni avranno gli stessi lati destri delle produzioni di S , prendere S' come simbolo iniziale al posto di S e, infine, aggiungere la produzione $S' \rightarrow \varepsilon$.

ESEMPIO: DALL'AUTOMA ALLA GRAMMATICA

Dall'automa



si ottiene la grammatica con simbolo iniziale X_0 e produzioni:

$$\begin{array}{lllll} X_0 \rightarrow aX_1 & X_1 \rightarrow bX_3 & X_2 \rightarrow bX_2 & X_3 \rightarrow b & X_4 \rightarrow a \\ X_0 \rightarrow bX_2 & X_2 \rightarrow a & X_3 \rightarrow a & X_3 \rightarrow bX_4 & X_4 \rightarrow aX_4 \\ X_1 \rightarrow aX_1 & X_2 \rightarrow aX_4 & X_3 \rightarrow aX_4 & X_4 \rightarrow bX_2 & \end{array}$$

Il linguaggio generato da tale grammatica coincide con quello accettato dall'automa.

Grammatiche Lineari

Una grammatica si dice lineare se tutte le produzioni hanno la forma

$$X \rightarrow uYv \text{ o } X \rightarrow u, \text{ con } X, Y \in N \text{ e } u, v \in \Sigma^*$$

Una grammatica si dice lineare destra se tutte le produzioni hanno la forma

$$X \rightarrow uY \text{ o } X \rightarrow u, \text{ con } X, Y \in N \text{ e } u \in \Sigma^*$$

In altre parole, una grammatica è lineare se le produzioni hanno a sx una variabile, e a dx hanno al più una variabile; è lineare destra se tali variabili compaiono solo alla fine di tale termine (alla fine della parola).

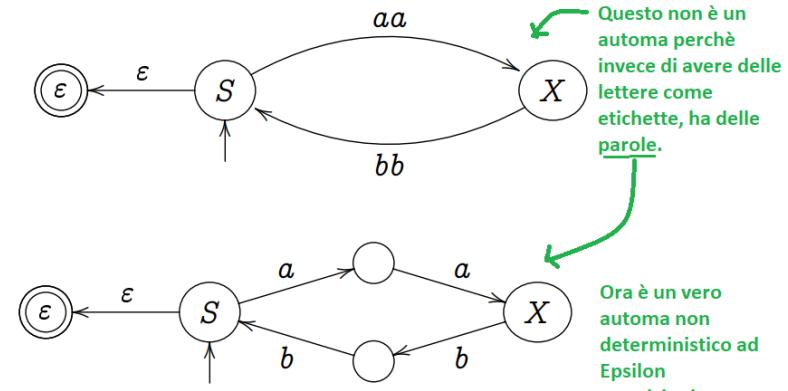
Le grammatiche di tipo 3 considerate finora sono evidentemente lineari destre. Viceversa, si può dimostrare che ogni grammatica lineare destra genera un linguaggio di tipo 3.

ESEMPIO:

Sia G la grammatica con le produzioni

$$S \rightarrow aaX, \quad S \rightarrow \epsilon, \quad X \rightarrow bbS.$$

L'automa che accetta $L(G)$ si ottiene nel modo seguente:



Lemma di Iterazione

30.03.21

Una **proprietà** dei linguaggi regolari è il **Lemma di Iterazione**.

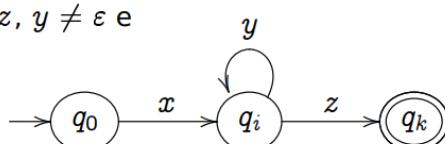
Se prendiamo un linguaggio regolare, possiamo trovare un **intero n** per cui tutte le parole del linguaggio che siano più lunghe di questo intero n, possono essere scomposte in **3 pezzi**: xyz.

Dove la parte centrale y non è la parola vuota, ma un **fattore** che può essere **ripetuto n volte** senza che si esca dal linguaggio.

Un linguaggio che non supporta questa proprietà di iterazione (**pumping**), non può essere regolare, anche se esistono linguaggi che hanno la suddetta proprietà ma non sono regolari.

Dimostrazione

- Per il Teorema di Kleene, L è accettato da un automa a stati finiti \mathcal{A} ,
- sia n il numero degli stati,
- sia $w \in L$ e $|w| \geq n$. Scriviamo $w = a_1 a_2 \dots a_k$, con **prendiamo una parola che abbia lunghezza $\geq n$**
- **nel grafo di \mathcal{A} c'è un cammino** $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_k} q_k \in F$ → **c'è un cammino da uno stato iniziale ad uno finale che ha w come etichetta.**
- dato che $k \geq n$, troveremo uno stato ripetuto $q_i = q_j$, $0 \leq i < j \leq k$,
- quindi, posto $x = a_1 \dots a_i$, $y = a_{i+1} \dots a_j$, $z = a_{j+1} \dots a_k$



Dato che $k \geq n$, ci saranno almeno $n+1$ stati. Quindi questi stati sono più del numero di stati totale dell'automa, ci saranno di conseguenza delle ripetizioni.

- pertanto \mathcal{A} accetta tutte le parole $xy^n z$ con $n \geq 0$,
- cioè $xy^n z \in L$ per ogni $n \geq 0$.

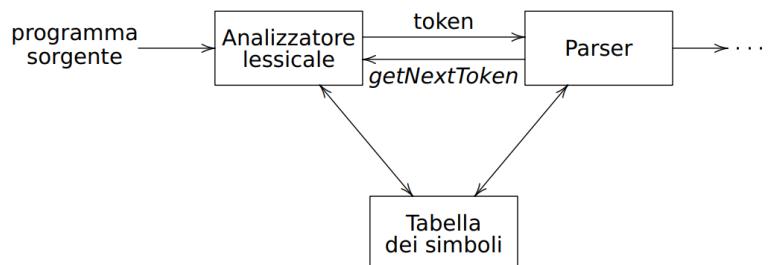
Analisi Lessicale

Come abbiamo visto inizialmente, l'analisi è la prima fase nella compilazione di un codice sorgente.

Dobbiamo identificare gli elementi atomici che costituiscono il nostro codice e individuare a quale categoria lessicale fanno riferimento.

Il programma sorgente viene dato in pasto all'Analizzatore lessicale, che restituirà un token al Parser ogni volta che questo deve procedere con la sua analisi sintattica.

Contemporaneamente all'invio del token, l'analizzatore scriverà nella tabella dei simboli gli attributi del token, che poi potranno essere recuperati dal parser.



E' importante separare l'analisi lessicale da quella sintattica perché si ha un'enorme semplificazione nella progettazione del compilatore, una maggiore efficienza e portabilità (perché cambiando sistema, cambia la gestione dell'input) rispetto a che se fossero svolte insieme.

- **Token:** è un simbolo astratto che rappresenta un'unità lessicale (per es., una parola chiave, un identificatore, una costante numerica, ecc.) I simboli processati dal parser sono i token. A ogni token viene associato un pattern.
- **Lessema:** è una sequenza di caratteri di un codice sorgente associata a un token. L'analizzatore lessicale identifica i lessemi come istanze del token a cui sono associati.
- **Pattern:** è una descrizione della forma che i lessemi possono avere per poter essere associati ad un determinato token. In pratica, si tratta di un'espressione regolare.

Se più di un lessema è associato al medesimo pattern, è necessario tenere traccia di ulteriori informazioni sul particolare lessema letto dall'analizzatore lessicale.

Tali informazioni sono registrate nella **Tabella dei simboli**.

| token | descrizione informale | esempi |
|-------------------|------------------------------------|----------------------|
| if | if | if |
| else | else | else |
| comparazione | <, >, <=, >=, == e != | <, >, <=, ... |
| id | lettera seguita da lettere e cifre | valore, a, c1, ... |
| numero | costanti numeriche | 3.14159, 0, 6.02e23 |
| stringa letterale | qualunque cosa tra due " | "hello", "Toni", ... |

←In questo esempio le **classi di token** sono:

1- un token per ogni parola chiave.
2- token per gli operatori (uno per ciascuno o raggruppati).

- 3- un unico token per gli identificatori.
- 4- uno o più token per le costanti.
- 5- un token per ogni segno di punteggiatura.

Tuttavia nessuno ci obbliga ad utilizzare tali classi.

Esempio: consideriamo l'espressione $E = M * C^{**} 2$. Questa produrrà la seguente serie di token e attributi.

1. < id , puntatore alla riga di E nella tabella dei simboli >
2. < op assegnazione >
3. < id , puntatore alla riga di M nella tabella dei simboli >
4. < op prodotto >
5. < id , puntatore alla riga di C nella tabella dei simboli >
6. < op potenza >
7. < numero , valore intero 2 >

Come si svolge l'analisi lessicale

I token sono identificati da espressioni regolari (pattern) e hanno un ordine di priorità (ad esempio: if potrebbe essere sia un id di una variabile chiamata if, sia il costrutto if che conosciamo tutti). L'analizzatore lessicale quindi va ad identificare il pezzo più lungo del testo da analizzare che possa essere considerato un lessema, dopodiché restituisce il primo token che corrisponde a questo lessema cioè il primo token il cui pattern matcha il lessema. Questo token sarà quello di massima priorità, ai cui pattern posso associare il lessema.

ESEMPIO:

```
int interesse = 5 ;
```

- il più lungo lessema iniziale è int, che corrisponde ai pattern dei token int e identificatore
- l'analizzatore restituisce int, perchè ha priorità maggiore,
- alla successiva attivazione analizza interesse = 5 ;
- anche questo inizia con il lessema int, ma non è il più lungo,
- il più lungo lessema iniziale è interesse, che corrisponde al pattern del token identificatore,
- l'analizzatore restituisce identificatore,
- alla successiva attivazione analizza = 5 ;

Il problema dell'analisi lessicale ha come **dati**: una sequenza finita di pattern, e una parola w (il testo da analizzare).

Avendo questi dati devo **determinare**: il più lungo prefisso u di w che appartenga al linguaggio denotato dall'espressione regolare, e il più piccolo indice j tale che u appartiene al linguaggio

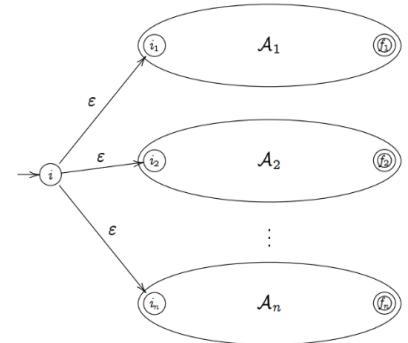
denotato (quindi devo capire qual è l'espressione regolare con cui questa parola fa match e, se ce n'è più di una, scegliere quella con priorità maggiore).

Automa per l'analisi lessicale

- per ognuna delle espressioni $E_j, j = 1, \dots, n$ costruiamo un automa a stati finiti (**espressioni regolari = pattern**)

$$A_j = \langle Q_j, \Sigma, \delta_j, \{i_j\}, \{f_j\} \rangle$$

- che accetta il linguaggio denotato da E_j ,
- possiamo supporre che tali automi non abbiano stati comuni,
- costruiamo l'automa \mathcal{A} che accetta il linguaggio denotato da $F = E_1 + \dots + E_n$, con:
 - gli stati, le transizioni e gli stati finali di A_1, A_2, \dots, A_n e inoltre
 - un nuovo stato iniziale i_0 ,
 - delle ϵ -transizioni da i_0 agli stati iniziali originali degli automi A_1, A_2, \dots, A_n , a differenza della costruzione vista nel teorema di Kleene, qui non aggiungiamo un nuovo stato finale perchè vogliamo vedere da quali dei miei n automi la parola viene accettata (a quale espressione regolare corrisponde).
- con la costruzione dei sottoinsiemi, otteniamo un automa deterministico $\mathcal{A}_D = \langle P, \Sigma, \delta, q_0, F' \rangle$ che accetta il linguaggio denotato da F ,
- osserviamo che
 - gli stati di \mathcal{A}_D sono sottoinsiemi di $Q = \{i_0\} \cup \bigcup_{j=1}^n Q_j$
 - si ha che w sta nel linguaggio denotato da E_j se e soltanto se $f_j \in \hat{\delta}(q_0, u)$.



Soluzione del nostro problema

Troviamo il più lungo prefisso u di w accettato da \mathcal{A}_D e il più piccolo indice j tale che $f_j \in \hat{\delta}(q_0, u)$,

Quando fermare la ricerca?

Gli stati di \mathcal{A}_D sono sottoinsiemi di Q , tra i quali c'è \emptyset che funge da 'pozzo'. L'ultimo transito da uno stato finale prima dell'ingresso nel pozzo individua il più lungo prefisso di w accettato da \mathcal{A}_D .

Sono linguaggi di tipo 2, e lo strumento necessario che ci permette di eseguire l'analisi sintattica.

Una grammatica si dice non contestuale o di tipo 2 (CFG Contest-Free Grammar) se tutte le produzioni sono della forma:

$$X \rightarrow \alpha, \text{ con } X \in N, \alpha \in V^*$$

Cioè se hanno a sx una singola variabile.

Un linguaggio si dice non contestuale o di tipo 2 (CFL) se è generato da una grammatica non contestuale.

Ogni grammatica lineare destra, e in generale le grammatiche lineari, sono non contestuali, perché a sx delle produzioni c'è sempre una variabile da sola.

Quindi, dato che sono generati da grammatiche regolari dx, i linguaggi regolari sono non contestuali. Essi costituiscono un sotto-insieme dei linguaggi non contestuali.

Non vale l'inverso.

Fra tutti i linguaggi non contestuali ce n'è uno che rappresenta l'archetipo di questo tipo di linguaggi: il **linguaggio Dick**.

Si usa raggruppare le produzioni con lo stesso lato sinistro, separando i lati destri con una barra verticale.
Per esempio: $S \rightarrow a | b | \emptyset | (S + S) | (SS) | S^*$ (cioè S produce a, S produce b, ecc...)

Linguaggio Dick

$$\Sigma_n = \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\}.$$

Sia $G_n = \langle V, \Sigma_n, P, S \rangle$ la grammatica con unica variabile S e produzioni

$$\begin{aligned} S &\rightarrow a_i S b_i, \quad i = 1, 2, \dots, n, \\ S &\rightarrow S S, \\ S &\rightarrow \epsilon. \end{aligned}$$

Il linguaggio D_n generato da G_n è detto il linguaggio di **Dick**.

Esempio

Supponiamo $n = 2$ e poniamo $a_1 = ($, $b_1 =)$, $a_2 = [$, $b_2 =]$.

Allora le parole

$$(), [], [()]([]), ((())[([()])]), \text{ sequenza di parentesi ben bilanciate.}$$

sono elementi di D_2 , mentre $)()$ non lo è.

- sequenze di parentesi ben bilanciate

Alberi di derivazione

Rappresentazione di una derivazione di una parola in una grammatica non contestuale.

Un albero è un insieme di nodi, tra i quali ce n'è uno chiamato radice, e poi c'è una relazione tra i nodi per cui a ogni nodo è associata una sequenza di figli. Ogni nodo può avere tanti figli, ma ogni figlio ha un solo genitore; e la radice non ha genitore. I figli di ciascun nodo hanno un ordinamento (il nodo a sx, il nodo a dx, il secondo nodo da sx, ecc...), e i nodi privi di figli sono detti foglie, anch'esse ordinate.

Quando guardiamo i primi due figli del nodo radice, sappiamo che i nodi discendenti del primo hanno priorità sui nodi discendenti dal secondo.

Tutti i nodi hanno delle etichette, nel nostro caso (negli alberi di derivazione) la radice deve avere come etichetta il simbolo iniziale della grammatica.

Definizione

T è un albero di derivazione della grammatica G se verifica le seguenti condizioni:

- 1 l'etichetta della radice è S , (variabili)
- 2 le etichette dei nodi interni sono elementi di N ,
- 3 le etichette delle foglie sono elementi di $\Sigma \cup \{\epsilon\}$, (simboli terminali o parola vuota. Nel secondo caso la foglia non deve avere fratelli e deve essere l'unico figlio del suo genitore.)
- 4 le foglie con etichetta ϵ sono 'figli unici', (l'unico figlio del suo genitore.)
- 5 se un nodo con etichetta X ha figli etichettati nell'ordine $\alpha_1, \dots, \alpha_k$, allora $X \rightarrow \alpha_1 \dots \alpha_k$ è una produzione di G . (nella grammatica deve esserci la produzione X produce alfa1, ..., alfak)

La parola che si ottiene leggendo, nell'ordine, le etichette delle foglie è la **parola associata** a T .

Una parola appartiene ad un linguaggio generato dalla grammatica se e solo se esiste un albero di derivazione della grammatica associato alla suddetta parola.

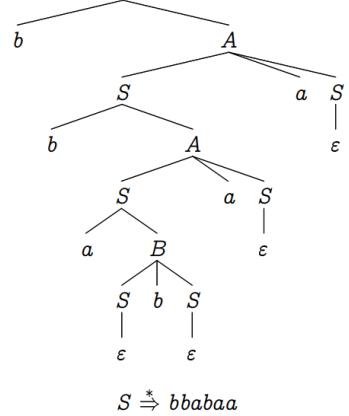
Una grammatica si dice **non ambigua** se a ogni parola del linguaggio generato corrisponde un'unico albero di derivazione.

Al contrario, le grammatiche **ambigue** sono quelle grammatiche dove ad ogni parola del linguaggio generato corrispondono più alberi di derivazione.

Non è sempre possibile aggiustare grammatiche ambigue, di conseguenza esistono i **linguaggi internamente ambigui**.

Un linguaggio non contestuale si dice **inerentemente ambiguo** se non è generato da nessuna grammatica non contestuale non ambigua.

$$S \rightarrow aB \mid bA \mid \epsilon, \quad A \rightarrow SaS, \quad B \rightarrow SbS.$$



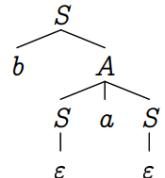
Esempio

$$S \rightarrow aB \mid bA \mid \epsilon, \quad A \rightarrow SaS, \quad B \rightarrow SbS$$

la parola ba ha due derivazioni:

$$\begin{aligned} S &\Rightarrow bA \Rightarrow bSaS \Rightarrow baS \Rightarrow ba, \\ S &\Rightarrow bA \Rightarrow bSaS \Rightarrow bSa \Rightarrow ba, \end{aligned}$$

ma un solo albero:



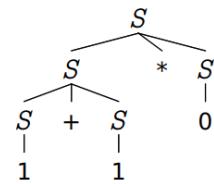
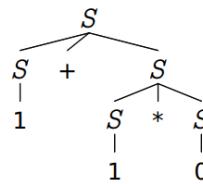
Esempio

Sia G la grammatica con $\Sigma = \{0, 1, *, +\}$, $N = \{S\}$ e produzioni

$$S \rightarrow S + S \mid S * S \mid 0 \mid 1$$

Allora $L(G) = (\{0, 1\} \{+, *\})^* \{0, 1\}$.

La parola $1 + 1 * 0$ ha due alberi di derivazione:



Dal punto di vista 'semantico' il primo corrisponde alla 'computazione' di $1 + (1 * 0)$ e il secondo alla 'computazione' di $(1 + 1) * 0$.

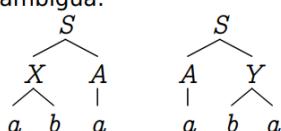
Esempio

Sia G la grammatica con le produzioni

$$S \rightarrow XA \mid AY, \quad X \rightarrow aXb \mid ab, \quad A \rightarrow aA \mid a, \quad Y \rightarrow bYa \mid ba.$$

Si ha $L = \{a^m b^m a^n \mid m, n > 0\} \cup \{a^m b^n a^n \mid m, n > 0\}$

G è ambigua:



L è inerentemente ambiguo.

Quando abbiamo a che fare con le grammatiche, i due problemi principali sono quello della **ricognizione** (stabilire se una parola fa parte del linguaggio generato) e del **parsing** (data una parola che è nel linguaggio, trovare una derivazione).

Sono problemi complessi soprattutto perché ci sono delle produzioni che danno particolarmente fastidio:

- $X \rightarrow \epsilon$ (ϵ -produzioni).
- $X \rightarrow Y$ (variabile produce variabile, produzioni 1-arie).

Quando abbiamo queste produzioni, quindi, potremmo cercare di creare delle **grammatiche equivalenti** che non hanno queste produzioni fastidiose.

Un'altra cosa fastidiosa sono le variabili che non compaiono nelle forme sentenziali, e da cui non si derivano parole prive di variabili (variabili che non producono nulla, non producono forme sentenziali).

ϵ -produzioni e produzioni unarie

Sia G una grammatica non contestuale.

Le produzioni della forma $X \rightarrow \epsilon$ si dicono ϵ -produzioni.

Le produzioni della forma $X \rightarrow Y$, con X, Y variabili, si dicono produzioni 1-arie.

L'ideale sarebbe avere a sx sempre una variabile, e a dx avere o un terminale, o una sequenza di almeno due lettere variabile-terminale.

In assenza di ϵ -produzioni e produzioni 1-arie, se $\alpha \Rightarrow \beta$ (α primo termine, β secondo termine) allora:

- $|\beta| > |\alpha|$ oppure
- $|\beta| = |\alpha|$ ma β contiene un terminale in più di α .

Pertanto, una parola di lunghezza n può essere generata in **2n – 1 passi** al più (perché ad ogni passo o la mia forma sentenziale si allunga, oppure una variabile viene sostituita da un terminale).

Ogni linguaggio non contestuale è possibile generarlo da una grammatica non contestuale priva di ϵ -produzioni, tranne, la produzione $S \rightarrow \epsilon$ (se presente), ove S è il simbolo iniziale. Inoltre, si può assumere che il simbolo iniziale non compaia nei lati destri delle produzioni.

Visto che le parole vuote ci servono nella generazione dei nostri linguaggi, dobbiamo mantenere la possibilità di effettuare una ϵ produzione. La possiamo mettere al primo passo $S \rightarrow \epsilon$, e solo se S non compare nei lati dx delle produzioni. Quindi questa produzione $S \rightarrow \epsilon$ c'è, ma è come se non ci fosse perché viene utilizzata solo ed unicamente per generare la parola vuota. Se non la uso al primo passo, non la posso utilizzare più.

Eliminazione delle ϵ -produzioni

Una variabile si dice **annullabile** se genera la parola vuota. Questo vale non solo per le produzioni dirette come $X \rightarrow \epsilon$, ma anche per produzioni concatenate come: $X \rightarrow Y \rightarrow W \rightarrow \epsilon$.

Quando andiamo a costruire una grammatica possiamo avere diversi **casi**:

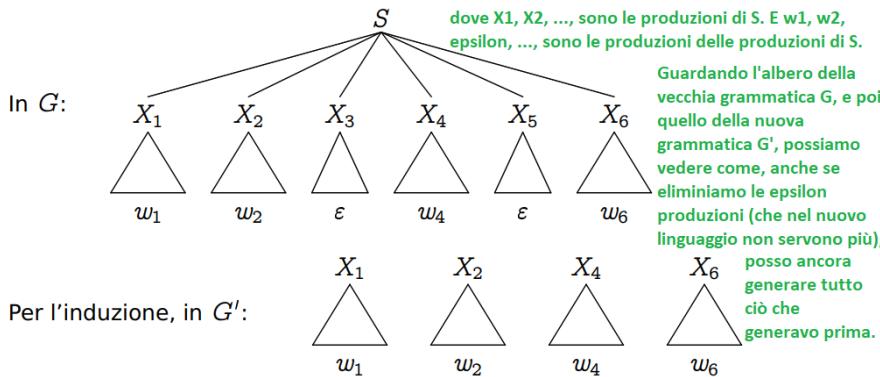
1. **S non è annullabile**, quindi non è possibile generare la parola vuota con il linguaggio.

Partendo da una grammatica G non semplificata, costruiamo una nuova grammatica G' nel modo seguente:

- aggiungiamo a G' tutte le produzioni che si ottengono cancellando nei lati dx delle produzioni di G , una o più occorrenze di variabili annullabili.

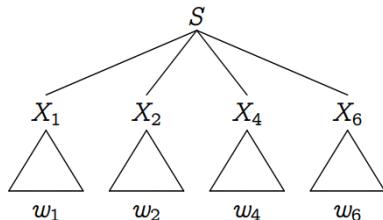
- cancelliamo le ε -produzioni.

In questo modo G' sarà equivalente a G anche se ho tolto tutte le ε -produzioni.



Inoltre $S \rightarrow X_1 X_2 X_3 X_4 X_5 X_6$ in G e X_3 e X_5 sono annullabili.

Pertanto in G' c'è la produzione $S \rightarrow X_1 X_2 X_4 X_6$ e l'albero di derivazione



2. S è annullabile.

A partire da G costruiamo una nuova grammatica G' nel modo seguente:

- Procedendo come nel caso precedente si ottiene una grammatica G' che genera le stesse parole di G , eccetto ε ;
- se S non compare nei lati destri, basta aggiungere la produzione $S \rightarrow \varepsilon$;
- se S compare nei lati destri, si aggiungono una nuova variabile S' , che sarà il nuovo simbolo iniziale, e le produzioni $S' \rightarrow \varepsilon$ e $S' \rightarrow S$. In questo modo ho fatto scomparire il simbolo iniziale dai lati dx delle produzioni, e posso aggiungere ε .

Ricerca delle variabili annullabili

Per fare ciò, però, dobbiamo sapere quali sono le variabili annullabili del linguaggio.

Costruiamo quindi insiemi sempre più grandi di variabili annullabili:

1. all'inizio prendo l'insieme di tutte le variabili che producono ε .
2. Poi costruisco un altro insieme in cui oltre a queste variabili, aggiungo tutte quelle produzioni che hanno un lato dx fatto interamente da variabili annullabili (che conosco già grazie al passo precedente).
3. Ripeto il secondo passo più volte fino a quando gli insiemi non cresceranno più e che ci indicheranno che tutte le variabili annullabili sono state trovate.

ESEMPIO

Sia G la grammatica con $N = \{S, O, P, E\}$, $\Sigma = \{a, b, x\}$ e con le produzioni

$$S \rightarrow aOb, \quad O \rightarrow P \quad | \quad aOb \quad | \quad OO, \quad P \rightarrow x \quad | \quad E, \quad E \rightarrow \varepsilon.$$

O ha a dx una variabile annullabile.

| n | W_n | $N - W_n$ |
|-----|----------------|-----------|
| 1 | <u>E</u> | S, O, P |
| 2 | <u>E, P</u> | S, O |
| 3 | <u>E, P, O</u> | S |
| 4 | <u>E, P, O</u> | S |

derivazione diretta di epsilon.

Le variabili annullabili sono: E, P, O .

Quindi G è equivalente alla grammatica con produzioni

$$S \rightarrow aOb \quad | \quad ab, \quad O \rightarrow P \quad | \quad aOb \quad | \quad OO \quad | \quad ab \quad | \quad O, \quad P \rightarrow x \quad | \quad E.$$

Eliminazione delle produzioni unarie

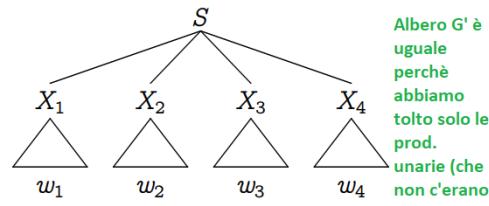
A partire da G costruiamo una nuova grammatica G' nel modo seguente:

1. si eliminano le ε -produzioni con la costruzione precedente,
2. per ogni coppia di variabili A, B tali che $A \Rightarrow^* B$ (A deriva B in uno o più passi), si aggiungono i lati destri delle produzioni di B a quelli delle produzioni di A (tutte le cose che produce B le aggiungo a quelle che produce A),
3. si cancellano le produzioni unarie.

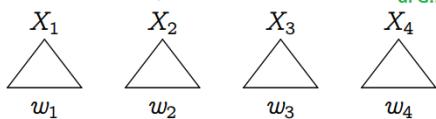
Anche qui G' sarà equivalente a G , perché non ho aggiunto nulla di nuovo, e posso generare tutto ciò che generavo con G .

Anche qui abbiamo diversi casi:

1. S ha più di un figlio.

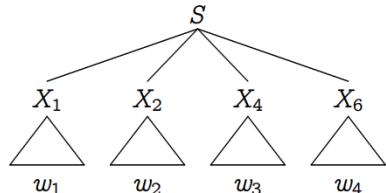


Per l'induzione, in G' :



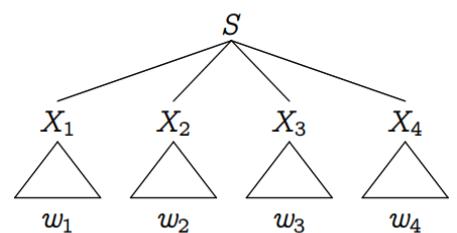
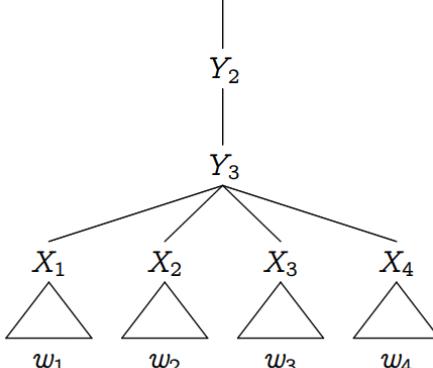
Inoltre $S \rightarrow X_1 X_2 X_3 X_4$ in G e anche in G' .

Pertanto in G' c'è l'albero di derivazione



2. S ha un unico figlio.

Quindi in G' c'è la produzione $S \rightarrow X_1 X_2 X_3 X_4$ e, tenendo conto dell'induzione, c'è l'albero di derivazione



Abbiamo tolto tutte le produzioni unarie ($S \rightarrow Y_1, S \rightarrow Y_2$, e $S \rightarrow Y_3$) e abbiamo aggiunto tutto a dx di S . Quindi ora S le deriva direttamente.

Ricerca delle derivazioni unarie

Costruiamo un insieme (T_n) di coppie di variabili con tutte le produzioni unarie dirette.

Ad ogni passo, vado ad aggiungere all'insieme, tutte le coppie X, Y per cui ci sono già X, Z e Z, Y nell'insieme. Perché so che da $X \rightarrow Z$ e da $Z \rightarrow Y$, allora da $X \rightarrow Y$.

Procediamo così fin quando posso aggiungere coppie all'insieme, altrimenti vorrebbe dire che abbiamo trovato tutte le derivazioni unarie.

T_1 è l'insieme delle coppie X, Y per cui Y si deriva da X in un passo. T_2 in al massimo due passi. T_3 in al massimo tre passi. T_n in al massimo 2^{n-1} passi. Quindi il procedimento è abbastanza veloce perché la lunghezza massima sarà il numero delle variabili.

Variabili improduttive e inaccessibili

Una variabile X si dice **produttiva** se esiste $w \in \Sigma^*$ tale che $X \Rightarrow^* w$, **improduttiva** altrimenti.

Quindi X è produttiva se da essa si può derivare una parola costituita interamente da simboli terminali, altrimenti è improduttiva.

Quando ho una variabile improduttiva in una derivazione, so già che quella derivazione non andrà a buon fine, perché non giungerò mai a una parola terminale. In questo caso posso cancellare tale variabile e tutte le sue produzioni ottenendo una grammatica equivalente, perché quella variabile improduttiva non portava comunque a nulla.

Una variabile X si dice **accessibile** se compare in qualche forma sentenziale, **inaccessibile** altrimenti.

Ricerca delle variabili produttive

Creiamo un insieme di variabili (W_n) e gli aggiungiamo tutte le variabili che sono direttamente produttive, cioè le variabili che producono subito tutti simboli terminali (comprese le ε -produzioni).

Dopodiché gli aggiungo tutte quelle variabili che hanno un lato dx costituito da terminali e variabili che già so essere produttive.

Ripeto questo procedimento finché l'insieme non cresce più.

Tutte le variabili che rimangono fuori da questo insieme sono improduttive e quindi possono essere buttate via.

Ricerca delle variabili accessibili

Creiamo un insieme di variabili (K_n) e gli aggiungiamo S che sappiamo essere accessibile.

Poi aggiungeremo tutte le variabili che compaiono nei lati dx delle produzioni di S.

In seguito aggiungeremo tutte le variabili che hanno nel lato sx delle variabili che sappiamo già essere accessibili.

Ripetiamo il procedimento finché l'insieme non smette di crescere.

Procedura di Riduzione

1. Determinare le variabili produttive
2. Eliminare le variabili improduttive e le produzioni che contengono tali variabili (potrebbe generare nuove variabili inaccessibili, ecco perché si fa prima del 3° passo),
3. determinare le variabili accessibili della grammatica ottenuta,
4. eliminare le variabili inaccessibili e le produzioni che contengono tali variabili.

In questo caso la nostra nuova grammatica sarà ridotta.

ESEMPIO

Sia G la grammatica con $N = \{E, F, T, R\}$, $\Sigma = \{+, *, -, a, (,)\}$, simbolo iniziale E e produzioni

$$E \rightarrow E + E \mid T \mid F, \quad F \rightarrow E * E \mid (T) \mid a,$$

$$T \rightarrow E - T, \quad R \rightarrow E - F.$$

| Variabili produttive | | | Variabili accessibili | | |
|----------------------|--------------|-----------|-----------------------|---------------|-----------|
| Produttive | Improduttive | | Accessibili | Inaccessibili | |
| n | W_n | $N - W_n$ | n | K_n | $N - K_n$ |
| 1 | F | E, T, R | 0 | E | F, R |
| 2 | F, E | T, R | 1 | F, E | R |
| 3 | F, E, R | T | 2 | F, E | R |
| 4 | F, E, R | T | | | |

Perchè deriva il simbolo terminale 'a'.
T non è presente perchè è già stata eliminata dato che è improduttiva.

$E \rightarrow E + E \mid F,$
 $F \rightarrow E * E \mid a,$
 $R \rightarrow E - F.$

Forma Normale di Chomsky

21.04.21

Una grammatica non contestuale si dice in forma normale di **Chomsky** se ha solo produzioni dei tipi:

- $X \rightarrow YZ$, con $X, Y, Z \in N$, quindi abbiamo una variabile a dx e una coppia di variabili a sx.
- $X \rightarrow a$, con $X \in N$, $a \in \Sigma$, abbiamo una variabile a dx e un simbolo terminale a sx.
- $S \rightarrow \varepsilon$, ma solo a condizione che S non compaia nei lati destri delle produzioni.

Ad esempio la grammatica con le seguenti produzioni è in forma normale Chomsky:

$S \rightarrow AB \mid \varepsilon, A \rightarrow AA \mid a, B \rightarrow BB \mid b$

Negli alberi di derivazione di una grammatica in forma normale di Chomsky le foglie sono figli unici, e i nodi interni costituiscono un albero binario completo, perché ogni variabile che non ha il simbolo terminale avrà 2 variabili.

Ogni linguaggio non contestuale è effettivamente generato da una grammatica non contestuale in forma normale di Chomsky.

Questa procedura funziona come segue:

Passo 1: Eliminare le ε -produzioni e produzioni unarie;

restano solo produzioni 'variabile produce terminale' $X \rightarrow a$ con $X \in N$ e $a \in \Sigma$ e produzioni 'variabile produce lato dx che contiene almeno due lettere terminale-variabile' $X \rightarrow \gamma$ con $|\gamma| \geq 2$;

Passo 2: Ridursi al caso $\gamma \in N^*$ (caso in cui γ gamma contiene solo variabili).

Per ogni terminale 'a' che compare, non da solo, in qualche lato destro:

1. Introduco una nuova variabile A e una nuova produzione $A \rightarrow a$,

2. sostituisco tutte le occorrenze

di 'a' nei lati destri delle produzioni con A (salvo che a non sia l'intero lato destro).

ESEMPIO

Sia G la grammatica con produzioni

$S \rightarrow aOb \mid ab, O \rightarrow aOb \mid OO \mid ab \mid x.$

Con la procedura indicata, otteniamo

$S \rightarrow AOB \mid AB, O \rightarrow AOB \mid OO \mid AB \mid x, A \rightarrow a, B \rightarrow b.$

In questo modo riesco a creare una grammatica che nei lati dx ha solo ed almeno due variabili.
creo due nuove variabili

Passo 3: tuttavia vogliamo che tutte le variabili a dx siano solo 2.

Quindi vogliamo ridurre al caso $k = 2$.

Ad esempio, avendo una produzione $X \rightarrow X_1 \dots X_k$ con $k \geq 3$, (con quindi k diverso da 2):

1. introduco una nuova variabile Z e sostituisco la produzione $X \rightarrow X_1 \dots X_k$ con la coppia di produzioni

$X \rightarrow X_1 \dots X_{k-2}Z, Z \rightarrow X_{k-1}X_k$;

2. ripeto il passo precedente finché necessario.

ESEMPIO

Sia G la grammatica con produzioni

$S \rightarrow AOB \mid AB, O \rightarrow AOB \mid OO \mid AB \mid x, A \rightarrow a, B \rightarrow b.$

Con la procedura indicata, otteniamo

$S \rightarrow A\textcolor{green}{Z} \mid AB, O \rightarrow A\textcolor{green}{Z} \mid OO \mid AB \mid x, \textcolor{green}{Z} \rightarrow OB, A \rightarrow a, B \rightarrow b.$

La grammatica ottenuta, equivalente a G , è in forma normale di Chomsky.

Forma Normale di Greibach

Una grammatica non contestuale G si dice in forma normale di **Greibach** se ha solo produzioni dei tipi:

- $X \rightarrow a\gamma$, con $a \in \Sigma$ e $\gamma \in N^*$, quindi abbiamo a sx una variabile e a dx un terminale e una sequenza di variabili.
- $S \rightarrow \varepsilon$, ma solo a condizione che S non compaia nei lati destri delle produzioni.

Ad esempio, la grammatica con produzioni $S \rightarrow aSB \mid aB, B \rightarrow b$, è in forma normale di Greibach.

In una grammatica in forma normale di Greibach, ogni derivazione di una parola $w \in L(G)$ di lunghezza n richiede esattamente n passi.

Ogni grammatica può essere messa in forma normale di Greibach.

Ogni linguaggio non contestuale è effettivamente generato da una grammatica non contestuale in forma normale di Greibach.

Lemma di iterazione per i linguaggi non contestuali

Come abbiamo visto precedentemente, il **lemma per i linguaggi regolari** consisteva in:

Sia L un linguaggio regolare.

Esiste un intero n tale che ogni parola $w \in L$ di lunghezza $|w| \geq n$

si può fattorizzare $w = xyz$ con $y \neq \varepsilon$ e $xy^n z \in L$ per ogni $n \geq 0$.

Esempio

$L = \{a^m b^m \mid m > 0\}$ non è regolare.

Invero, se $xyz \in L$ con $y \neq \varepsilon$, allora $xyyz \notin L$.

Però per $x = y = z = \varepsilon$, $u = a^m$, $v = b^m$, si ha

$$a^m b^m = xuyvz, \quad xu^n yv^n z = a^{mn} b^{mn} \in L.$$

Si possono iterare due blocchi parallelamente.

Perchè y corrisponderebbe ad un blocco di 'a' o ad un blocco di 'b', quindi se itero quante volte voglio 'a' o 'b' non saranno più uguali (stessa lunghezza). y potrebbe essere anche a cavallo tra 'a' e 'b', iterando così blocchi di 'a' e 'b' e mi ritroverei fuori dal linguaggio.
Se però prendessi un 'u' dentro le 'a', e una 'v' dentro le 'b' (con $|a|=|b|$), potrei iterarle parallelamente e non uscire dal linguaggio.

Il lemma di iterazione per i linguaggi non contestuali:

Sia L un linguaggio non contestuale.

Esiste un intero n tale che ogni parola $w \in L$ di lunghezza $|w| > n$

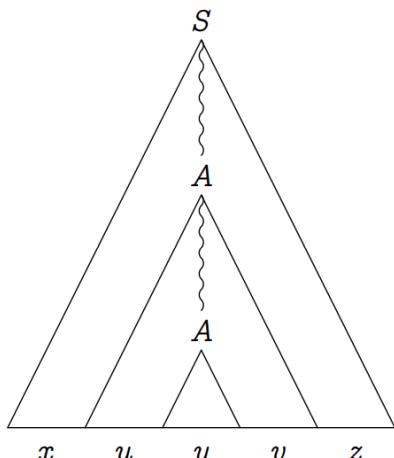
si può fattorizzare $w = xuyvz$ con $uv \neq \varepsilon$ e $xu^k yv^k z \in L$ per ogni $k \geq 0$.

Come si può vedere qui **iteriamo parallelamente** u e v .

Dimostrazione

- Sia $L = L(G)$, con G priva di ε -produzioni e produzioni unarie;
- sia n la massima lunghezza delle parole di L che hanno un albero di derivazione di altezza $\leq \text{Card}(N)$; altezza <= lunghezza delle parole
- Sia $w \in L$, con $|w| > n$;
- in un cammino di lunghezza $> \text{Card}(N)$ dalla radice a una foglia nell'albero di derivazione c'è una variabile che si ripete;
- $S \Rightarrow^* xAz, \quad A \Rightarrow^* uAv, \quad A \Rightarrow^* y,$
 $w = xuyvz, \quad uv \neq \varepsilon;$
- $S \Rightarrow^* xAz \Rightarrow^* xuAvz \Rightarrow^* xuuAvvz$
 $\Rightarrow^* \dots \Rightarrow^* xu^k Av^k z \Rightarrow^* xu^k yv^k z;$
- quindi $xu^k yv^k z \in L$ per ogni $k \geq 0$.

Se L è un linguaggio non contestuale, in ogni parola di L abbastanza lunga si possono trovare due fattori non entrambi vuoti che possono essere iterati simultaneamente (**pumped**) senza uscire dal linguaggio. Pertanto, un linguaggio che non ha questa proprietà non è un linguaggio non contestuale; tuttavia ci sono anche linguaggi che hanno questa proprietà ma non sono linguaggi non contestuali.



Algoritmo di Cocke-Kasami-Younger (CYK)

27.04.21

Algoritmo moderatamente efficiente per risolvere il problema di appartenenza, quindi per vedere se una parola appartiene o meno ad un linguaggio generato da una grammatica non contestuale. Con opportune modifiche questo algoritmo può risolvere anche il problema di parsing.

(Riguarda a pagina 8 la definizione di **ricognizione e parsing**)

Questo algoritmo è ideale per le grammatiche in forma normale di **Chomsky** ma, come abbiamo visto precedentemente, ogni grammatica non contestuale si può ridurre in tale forma normale.

Siano G una grammatica in forma normale di Chomsky e w una parola, scriviamo w come stringa di lettere $w = a_1 a_2 \dots a_n$,

Cerchiamo di calcolare, per $0 \leq i < j \leq n$, tutte le variabili da cui si può derivare il fattore $a_i + 1 a_{i+2} \dots a_j$, quindi vado a cercare per ogni fattore della mia parola, quali sono le variabili.

Per $i = 0, j = n$, avrò le variabili da cui si può derivare w ; quindi $w \in L(G)$ se e solo se tra esse il simbolo iniziale S è una di tali variabili.

Il calcolo

Per ogni coppia di indici i, j devo calcolare l'insieme N_{ij} , insieme delle variabili che mi generano il fattore $a_{i+1}, a_{i+2}, \dots, a_j$.

Dobbiamo calcolare gli insiemi

$$N_{ij} = \{X \in N \mid X \xrightarrow{*} a_{i+1} a_{i+2} \dots a_j\}, \quad 0 \leq i < j \leq n.$$

insieme delle variabili da cui posso derivare la singola lettera a_j . Il nostro linguaggio è però in forma normale di Chomsky, quindi quando da una variabile deriva la concatenazione di più variabili, poi avrà un singolo terminale, e non potrà più recuperare solo una lettera. Quindi la soluzione è fare tutto in un passo: se c'è la produzione $X \xrightarrow{*} a_j$, X sta in N_{ij} .

$$N_{ij} = \{X \in N \mid X \xrightarrow{*} a_j\} = \{X \in N \mid X \xrightarrow{*} a_j \text{ in } P\}. \quad (1)$$

Per $j \geq i + 2$, si ha $X \xrightarrow{*} a_{i+1} a_{i+2} \dots a_j$ se e solo se

$$X \rightarrow YZ, \quad Y \xrightarrow{*} a_{i+1} a_{i+2} \dots a_h, \quad Z \xrightarrow{*} a_{h+1} a_{h+2} \dots a_j,$$

per opportuni $Y, Z \in N$ e $i < h < j$. Otteniamo così

N_{ij} si ottiene facendo variare h da $i+1$ a $j-1$. Se il fattore a_{i+1}, \dots, a_j è generato da X , ci sarà una produzione $X \rightarrow YZ$, con Y in N_{ih} quindi produce qualcosa da a_{i+1} fino ad arrivare ad a_h , e Z in N_{hj} quindi produce il resto della parola.

$$N_{ij} = \bigcup_{h=i+1}^{j-1} \{X \in N \mid \exists Y \in N_{ih}, Z \in N_{hj}, X \rightarrow YZ \text{ in } P\}. \quad (2)$$

Per ogni coppia di variabili Y, Z , poniamo

insieme di tutte le variabili che hanno la produzione $X \rightarrow YZ$. E' come se definissimo un'operazione sull'insieme delle variabili che non è la concatenazione perché prima concateno YZ e poi lo sostituisco con il lato sx della produzione.

$$Y \odot Z = \{X \in N \mid X \rightarrow YZ \text{ in } P\}$$

operazione di concatenazione e poi sostituzione con il lato sx.

e, più in generale, per ogni coppia di insiemi di variabili $H, K \subseteq N$, se abbiamo due insiemi di variabili H e K , definiamo la composizione di H e K come l'unione delle coppie che stanno nella composizione Y e Z , con Y in H e Z in K .

$$H \circ K = \bigcup_{\substack{Y \in H \\ Z \in K}} Y \odot Z.$$

Allora la nostra formula ricorsiva diventa

N_{ij} = somme per h da $i+1$ a $j-1$, di $N_{ih} \circ N_{hj}$.

$$N_{ij} = \bigcup_{h=i+1}^{j-1} N_{ih} \circ N_{hj}.$$

Anzi, se conveniamo che $N_{ij} = \emptyset$ per $0 \leq j \leq i \leq n$.

Quando h è più piccolo, $<$, oppure $>$ di j , N_{ij} = vuoto.

$$N_{ij} = \bigcup_{h=0}^n N_{ih} \circ N_{hj}.$$

Calcolare $N_{ih} \circ N_{hj}$ ha costo 1.

La formula 3 della slide ha costo $j - i - 1 \leq n$ e viene eseguita $1 + 2 + \dots + (n - 1) = n(n - 1)/2$ volte, la dimensione è limitata al numero di variabili della grammatica e non dipende dalla parola che vado ad analizzare.

Il costo totale è quindi $O(n^3)$.

ESEMPIO

Produzioni:

$$S \rightarrow XY \mid AY \mid XB \mid AB, \quad X \rightarrow AS, \\ Y \rightarrow BS, \quad A \rightarrow a, \quad B \rightarrow b.$$

X composto ° Y, sarà S. Stessa cosa per A°Y, ecc...

Per $w = aabb$ si ha Parsing

| | 0 | 1 | 2 | 3 | 4 |
|---|---|-------------|-------------|-------------|---|
| 0 | A | \emptyset | X | S | |
| 1 | A | S | \emptyset | | |
| 2 | | | B | \emptyset | |
| 3 | | | | B | |
| 4 | | | | | |

Nella cella 0,1 ci devo mettere le variabili che mi producono la lettera A, prima lettera della parola.

Nella cella 0,2 devo mettere il risultato della moltiplicazione della riga 0 e la colonna 2: vuoto*vuoto + A*A=vuoto + vuoto*vuoto + vuoto*vuoto.

Cella 2,3 le variabili che producono la terza lettera.

Cella 1,3 prodotto della riga 1 per la colonna 3: vuoto*vuoto + vuoto*vuoto + A*B (che nella mia tabellina di sopra mi dicono sia S)+vuoto*vuoto.

Cella 0,3 prodotto di colonna 3 per riga 0: vuoto*vuoto, A*S=X, B*vuoto, vuoto*vuoto, vuoto*vuoto.

Poiché S compare nella cella in alto a destra, ne deduciamo che

$w \in L(G)$. Per completare la tabella di parsing procediamo per diagonali: prima la diagonale delle lettere della parola (AABB), poi la diagonale sopra (vuoto S vuoto), e così via.

L'output dell'algoritmo CYK è quindi la matrice di ricognizione N_{ij} associata a w. Ritorna TRUE se w appartiene al linguaggio, FALSE altrimenti.

Per poter fare anche il parsing basta fare il procedimento inverso di quello di ricognizione: se da X riesco a derivare $a_{i+1} a_{i+2} \dots a_j$, vuol dire che ci sono le variabili Y e Z per cui $X \rightarrow YZ$, e poi da Y derivo $a_{i+1} \dots a_h$, e da Z derivo $a_{h+1} \dots a_j$.

Queste variabili e questo h le abbiamo già trovate, quindi usando questi dati possiamo costruire la derivazione (parsing).

Function Parse (i, j, X) funzione che ha come parametri gli indici i,j e la variabile X. Mi restituisce una derivazione della parola $a_{i+1} \dots a_j$ a partire dalla variabile X (non dal simbolo iniziale quindi).

- 1 if $j - i = 1$ return $X \rightarrow a_j$
- 2 trova h, Y, Z tali che $Y \in N_{i,h}, Z \in N_{h,j}, X \rightarrow YZ$ in P
- 3 return $(X \rightarrow YZ, \text{Parse}(i, h, Y), \text{Parse}(h, j, Z))$

La chiamata di $\text{Parse}(0, n, S)$ ritorna il parsing di w .

ESEMPIO riprendendo l'esempio precedente...

Produzioni: 1)

$$S \rightarrow XY \mid AY \mid XB \mid AB, \quad X \rightarrow AS, \\ Y \rightarrow BS, \quad A \rightarrow a, \quad B \rightarrow b.$$

3) 0 1 2 3 4

| | 0 | 1 | 2 | 3 | 4 |
|---|---|-------------|-------------|-----|---|
| 0 | A | \emptyset | X | S | |
| 1 | A | S | \emptyset | | |
| 2 | | B | \emptyset | | |
| 3 | | | B | | |
| 4 | | | | | |

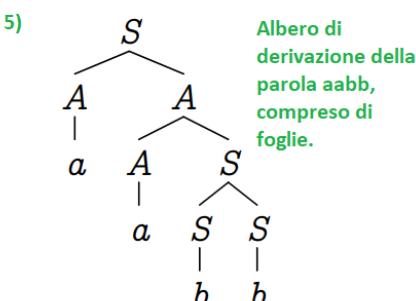
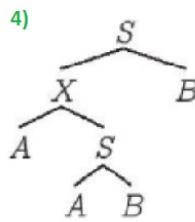
La S in alto a dx l'ho ottenuta grazie a X^0B della colonna 3 e 4 rispettivamente.

Queste due variabili sono dunque i primi due figli dell'albero.

X l'ho ottenuta facendo A^0S della colonna 1 e 3 rispettivamente.

Le righe rosse della tabella, quindi, ci danno l'albero di derivazione escluse le foglie. (Se inclini la testa e lo guardi con S in cima, vedi proprio l'albero come quello della figura del passo 4).

| 2) | o | S | X | Y | A | B |
|----|-------------|-------------|-------------|-------------|-------------|-------------|
| S | \emptyset | \emptyset | \emptyset | \emptyset | \emptyset | \emptyset |
| X | \emptyset | \emptyset | S | \emptyset | S | |
| Y | \emptyset | \emptyset | \emptyset | \emptyset | \emptyset | \emptyset |
| A | X | \emptyset | S | \emptyset | S | |
| B | Y | \emptyset | \emptyset | \emptyset | \emptyset | \emptyset |



Nel caso la mia grammatica non fosse in forma normale di Chomsky, la metto in forma normale con l'algoritmo standard visto in precedenza. Poi eseguo il parsing nella nuova grammatica con l'algoritmo CYK, così da ottenere una derivazione della parola della grammatica in forma normale di Chomsky. A questo punto riesaminiamo al contrario il primo passo e risaliamo al parsing nella grammatica originale.

Parsing Top-Down

28.04.21

In genere l'operazione di parsing richiede un tempo notevole $O(n^3)$ e non è accettabile nel caso dovessimo fare la compilazione di un programma molto complesso (perché come sappiamo, raramente un programma non presenta errori alla prima compilazione, quindi dovremmo fare più tentativi e di conseguenza effettuare la compilazione e il parsing più volte).

Problema di parsing

Input: una grammatica $G = \langle V, \Sigma, P, S \rangle$ e una parola $w \in L(G)$;
Output: una derivazione $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n \Rightarrow w$ di w in G
(o, equivalentemente un albero di derivazione) derivazione della parola nella grammatica, quindi la seq. dei passi di una derivazione.

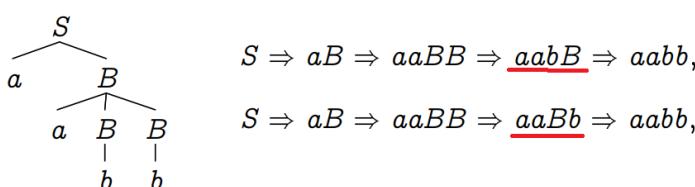
Osservazione

A ogni derivazione corrisponde un albero di derivazione, ma a un albero di derivazione possono corrispondere più derivazioni

come abbiamo già visto, quando per ogni parola c'è un solo albero di derivazione, la grammatica è non ambigua.

Esempio

$S \rightarrow aB \mid bA, \quad A \rightarrow a \mid aS \mid bAA, \quad B \rightarrow b \mid bS \mid aBB.$



Come possiamo vedere nell'esempio della slide, sostituendo prima una variabile invece che un'altra, otteniamo sempre lo stesso albero di parsing.

Sarebbe quindi bene mettersi d'accordo su un ordine da seguire per sostituire le variabili. Per fare ciò usiamo le **derivazioni sx.**

Una derivazione si dice sinistra se ad ogni passo vado a riscrivere la variabile più a sx della mia parola.

$S = \alpha_0 \Rightarrow_L \alpha_1 \Rightarrow_L \dots \Rightarrow_L \alpha_n = w$, dove '=' rappresenta una conseguenza e ' \Rightarrow ' una derivazione,
 $\alpha_0 \Rightarrow_L \alpha_1$ significa che ho ottenuto α_1 sostituendo la variabile più a sx di α_0 con il lato dx di una sua produzione. (ovviamente esistono anche le **derivazioni destre**, andando a sostituire la variabile più a dx)

Se eseguo solo derivazioni sinistre, allora ho una corrispondenza biunivoca tra queste corrispondenze e l'albero di derivazione. Ogni albero di derivazione corrisponde ad una e una sola derivazione sinistra e viceversa.

Inoltre, se ho una derivazione sinistra, per costruire l'albero in realtà mi basta ricordare le produzioni che ho utilizzato, perché la variabile che ho sostituito già la so, è quella più a sinistra.

Ogni parola di un linguaggio si ottiene con derivazioni sinistre. Questo perché le parole si ottengono con una derivazione, alla derivazione corrisponde un albero, e all'albero corrisponde una derivazione sinistra.

Endmaker

Un'altra cosa su cui è bene mettersi d'accordo è quella di utilizzare un marcatore di fine parola (endmaker).

Per far ciò modifichiamo la grammatica aggiungendo un nuovo simbolo iniziale S' , un nuovo terminale $\#$, e una nuova produzione $S' \rightarrow S\#$. Ora, anziché fare il parsing della parola w , faremo il parsing di $w\#$.

Quindi il cancelletto lo genererò partendo da $S' \rightarrow S\#$ e poi avrò S da cui devo derivare w .

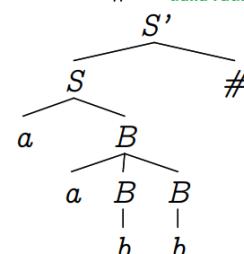
Esempio

$S' \rightarrow S\#, \quad S \rightarrow aB \mid bA, \quad A \rightarrow a \mid aS \mid bAA, \quad B \rightarrow b \mid bS \mid aBB.$

Vogliamo fare il parsing di $aabb\#$

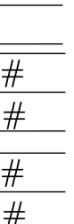
si chiama parsing top-down proprio perché siamo partiti dalla radice per costruire l'albero.

| |
|---|
| $aabb\#$ |
| S' |
| $aabb\#$ |
| $aB\#$ |
| $a \quad abb\#$ |
| $a \quad B\#$ |
| $a \quad abb\# \quad aa \quad bb\# \quad aa \quad bb\# \quad aab \quad b\# \quad aab \quad b\#$ |
| $a \quad aBB\# \quad aa \quad BB\# \quad aa \quad bB\# \quad aab \quad B\# \quad aab \quad b\#$ |



Se fossimo partiti dalle foglie, sarebbe stato un parsing bottom-up.

| |
|-----------------|
| $aabb\#$ |
| $aabb\#$ |
| $aabb \quad \#$ |
| $aabb \quad \#$ |
| $aabb \quad \#$ |



Parsing Predittivo

Abbiamo l'input che dobbiamo derivare dal simbolo iniziale, quindi passo passo esaminiamo il nostro input da sx verso dx e iniziamo a costruire la derivazione dall'alto in basso del nostro albero di derivazione.

Ad ogni passo avremo una parte di input già analizzato (quello alla sx delle tabelline dell'esempio precedente), l'analisi (ciò che avevamo derivato dal nostro simbolo iniziale che faceva match con quello di sopra), a dx abbiamo l'input ancora non analizzato, e sotto quello che ci avanzava dal quale dovevamo derivare la parte superiore (chiamata predizione).

Le regole di questo procedimento sono:

| input analizzato | input da analizzare |
|------------------|---------------------|
| analisi | predizione |

- esaminiamo sempre il simbolo più a sinistra della predizione,
- se è una variabile, dobbiamo fare una predizione: va sostituito con uno dei suoi lati destri,
- se è un terminale, deve coincidere con la prima lettera dell'input da analizzare (simbolo di look-ahead). In caso affermativo, viene spostato nel lato 'analisi'; in caso negativo, l'esecuzione fallisce.

Il fallimento non significa che non possiamo fare il parsing della parola, ma solo che la predizione che abbiamo fatto è sbagliata e che quindi dobbiamo tornare indietro e farne un'altra.

La predizione si gestisce **LIFO**: last in, first out.

Questa procedura è in linea generale non deterministica quindi è necessario esplorare tutte le possibili computazioni. Tale esplorazione può essere fatta in 2 modi:

1. Esplorazione in **ampiezza**: si esaminano parallelamente tutti i possibili sviluppi della computazione.

Richiede molta memoria, per questo a volte è preferibile l'esplorazione in profondità.

2. Esplorazione in **profondità**: si esamina una computazione alla volta, salvo tornare indietro e provare un'altra strada in caso di fallimento (backtracking)

C'è un caso in cui tutto il meccanismo visto negli esempi, però non funziona: quando si presenta il fenomeno della recursione destra.

Ricorsione destra: da una variabile posso derivare una forma sentenziale che comincia con la variabile stessa.

ESEMPIO ESPLORAZIONE IN AMPIEZZA

Dovremo tenere traccia

- della lettera che stiamo esaminando (simbolo di look-ahead)
- delle produzioni utilizzate in ogni tentativo di parsing,
- delle **predizioni** relative a ciascun tentativo.

$$\begin{array}{l} S' \rightarrow S\#, \\ S \rightarrow AB \mid DC, \end{array} \quad \begin{array}{l} A \rightarrow a \mid aA, \\ B \rightarrow bc \mid bBc, \end{array} \quad \begin{array}{l} C \rightarrow c \mid cC, \\ D \rightarrow ab \mid aDb. \end{array}$$

Facciamo il parsing della parola aabc#.

| | | | |
|----|----------------|----|-----------|
| 1. | aabc# | 4. | aabc# |
| | $S'_1 S_1 A_1$ | | $A \#$ |
| | $S'_1 S_1 A_2$ | | $AB \#$ |
| | $S'_1 S_2 D_1$ | | $abC \#$ |
| | $S'_1 S_2 D_2$ | | $aDbC \#$ |

| | | | |
|----|----------------|----|----------|
| 2. | aabc# | 5. | a |
| | S' | | $abc \#$ |
| | S'_1 | | $B \#$ |
| | $S'_1 S_1 A_1$ | | $AB \#$ |
| | $S'_1 S_1 A_2$ | | $bC \#$ |
| | $S'_1 S_2 D_1$ | | $DbC \#$ |

| | | | |
|----|------------|----|----------|
| 3. | aabc# | 6. | a |
| | $S'_1 S_1$ | | $abc \#$ |
| | $AB \#$ | | $bc \#$ |
| | $S'_1 S_2$ | | $bBc \#$ |
| | $DC \#$ | | $aB \#$ |

lookahead: confronto la prima lettera a sx con la parola. Tutte e 4 le predizioni mi vanno bene in questo caso.

porto la a sx.
Dove ho il simbolo terminale come prima lettera non devo fare nulla, mentre dove ho le variabili devo sostituire successivamente con le loro derivazioni. In questo caso sia B, A e D, hanno ciascuno 2 produzioni, quindi nella prossima tabella avrà 7 predizioni (6+1).

Anche qui dato che la prima lettera è un terminale faccio il lookahead, e posso buttare tutte le predizioni il cui terminale è diverso da 'a'.

| | | |
|----|------------------------|-----------|
| 7. | aa | bc# |
| | $S'_1 S_1 A_2 a A_1 a$ | $B \#$ |
| | $S'_1 S_1 A_2 a A_2 a$ | $AB \#$ |
| | $S'_1 S_2 D_2 a D_1 a$ | $bbC \#$ |
| | $S'_1 S_2 D_2 a D_2 a$ | $DbbC \#$ |

| | | |
|----|----------------------------|------------|
| 8. | aa | bc# |
| | $S'_1 S_1 A_2 a A_1 a B_1$ | $bc \#$ |
| | $S'_1 S_1 A_2 a A_1 a B_2$ | $bBc \#$ |
| | $S'_1 S_1 A_2 a A_2 a A_1$ | $aB \#$ |
| | $S'_1 S_1 A_2 a A_2 a A_2$ | $aAB \#$ |
| | $S'_1 S_2 D_2 a D_1 a$ | $bbC \#$ |
| | $S'_1 S_2 D_2 a D_2 a D_1$ | $abbC \#$ |
| | $S'_1 S_2 D_2 a D_2 a D_2$ | $aDbbC \#$ |

lookahead.

| | | |
|----|------------------------------|----------|
| 9. | aab | c# |
| | $S'_1 S_1 A_2 a A_1 a B_1 b$ | $c \#$ |
| | $S'_1 S_1 A_2 a A_1 a B_2 b$ | $bcc \#$ |

| | | |
|-----|----------------------------------|-----------|
| 10. | aab | c# |
| | $S'_1 S_1 A_2 a A_1 a B_1 b$ | $c \#$ |
| | $S'_1 S_1 A_2 a A_1 a B_2 b$ | $bcc \#$ |
| | $S'_1 S_1 A_2 a A_1 a B_2 b B_2$ | $bBcc \#$ |

| | | |
|-----|-----------------------------------|---|
| 11. | aabc | # |
| | $S'_1 S_1 A_2 a A_1 a B_1 b c$ | # |
| | $S'_1 S_1 A_2 a A_1 a B_1 b c \#$ | |

| | | |
|-----|-----------------------------------|--|
| 12. | aabc# | |
| | $S'_1 S_1 A_2 a A_1 a B_1 b c \#$ | |

Risultato:

$S' \Rightarrow S \# \Rightarrow AB \# \Rightarrow aAB \# \Rightarrow aaB \# \Rightarrow aabc\#.$

Questo comporta che nella parte dx della nostra tabella non ci saranno mai solo terminali come prima lettera, e di conseguenza dovremo ripetere all'infinito gli stessi passaggi.

Tuttavia, sappiamo che con la forma normale di Greibach è possibile eliminare le recursioni dx perché i lati dx cominciano con un terminale.

Ogni grammatica non contestuale, quindi, è equivalente a una grammatica priva di recursione destra.

Per esempio, le grammatiche in forma normale di Greibach sono prive di recursione destra.

Esempio ESPLORAZIONE IN PROFONDITA'

$$\begin{array}{lll} S' \rightarrow S\#, & A \rightarrow a \mid aA, & C \rightarrow c \mid cC, \\ S \rightarrow AB \mid DC, & B \rightarrow bc \mid bBc, & D \rightarrow ab \mid aDb. \end{array}$$

| | | | | | |
|-----------|----------|------------|----------|----------------|----------|
| 1. | $aabc\#$ | 2. | $aabc\#$ | 3. | $aabc\#$ |
| S'_1 | $S\#$ | $S'_1 S_1$ | $AB\#$ | $S'_1 S_1 A_1$ | $aB\#$ |

In questo caso non proviamo entrambe le produzioni di S, ma solo la prima.

lookahead.

Se non andrà bene, torneremo indietro.

| | | | | | | | | |
|------------------|-------|---------|----------------------|--------|---------|------------------|---------|---------|
| 4. | a | $abc\#$ | 5. | a | $abc\#$ | 6. | a | $abc\#$ |
| $S'_1 S_1 A_1 a$ | $B\#$ | | $S'_1 S_1 A_1 a B_1$ | $bc\#$ | | $S'_1 S_1 A_1 a$ | $B_1\#$ | |

lookahead.

La predizione non va bene perché la lettera che ci serve è 'a' e non 'b', quindi torniamo indietro.

| | | | | | | | | |
|----------------------|---------|---------|------------------|---------|---------|----------------|----------|---------|
| 7. | a | $abc\#$ | 8. | a | $abc\#$ | 9. | a | $abc\#$ |
| $S'_1 S_1 A_1 a B_2$ | $bBc\#$ | | $S'_1 S_1 A_1 a$ | $B_2\#$ | | $S'_1 S_1 A_1$ | $aB_2\#$ | |

lookahead, anche questa è sbagliata.

| | | | | | | |
|------------|-----------|----------------|----------|------------------|--------|---------|
| 10. | $aabc\#$ | 11. | $aabc\#$ | 12. | a | $abc\#$ |
| $S'_1 S_1$ | $A_1 B\#$ | $S'_1 S_1 A_2$ | $aAB\#$ | $S'_1 S_1 A_2 a$ | $AB\#$ | |

provo A2

| | | | | | | | | |
|----------------------|--------|---------|------------------------|-------|--------|----------------------------|--------|--------|
| 13. | a | $abc\#$ | 14. | aa | $bc\#$ | 15. | aa | $bc\#$ |
| $S'_1 S_1 A_2 a A_1$ | $aB\#$ | | $S'_1 S_1 A_2 a A_1 a$ | $B\#$ | | $S'_1 S_1 A_2 a A_1 a B_1$ | $bc\#$ | |

| | | | | | | | | |
|------------------------------|-------|-------|--------------------------------|--------|------|----------------------------------|----------|--|
| 16. | aab | $c\#$ | 17. | $aabc$ | $\#$ | 18. | $aabc\#$ | |
| $S'_1 S_1 A_2 a A_1 a B_1 b$ | $c\#$ | | $S'_1 S_1 A_2 a A_1 a B_1 b c$ | $\#$ | | $S'_1 S_1 A_2 a A_1 a B_1 b c\#$ | | |

Risultato.

Come Realizzare un Parser

- Usare dei programmi che fanno questo lavoro.
- Costruirsi il Parser da noi.

Nella seconda opzione, la procedura più semplice da seguire è quella di realizzare un **parser a discesa ricorsiva**. Ovvero scrivere tante procedure quante sono le variabili della nostra grammatica.

Interpretiamo la produzione $S \rightarrow aB \mid bA$ come

S ha successo se

a ha successo e poi B ha successo

oppure

b ha successo e poi A ha successo

• ideale per la programmazione di un parser per una grammatica specifica,

• una procedura per ogni variabile,

• dobbiamo tenere traccia

• posizione corrente nella produzione (automatico),

• l'input (globale),

• posizione corrente nell'input (globale),

• posizione nell'input al momento della chiamata (locale).

ESEMPIO:

$$\begin{array}{lll} S' \rightarrow S\#, & A \rightarrow a \mid aA, & C \rightarrow c \mid cC, \\ S \rightarrow DC \mid AB, & B \rightarrow bc \mid bBc, & D \rightarrow ab \mid aDb. \end{array}$$

| Active rules | sentence | parse |
|---------------------------------|-----------------|---|
| 1. $S' \rightarrow S\# \bullet$ | $abc\# \bullet$ | 1. $S' \rightarrow S\#$ 2. $S \rightarrow DC$ 3. $D \rightarrow ab$ 4. $C \rightarrow c$ |

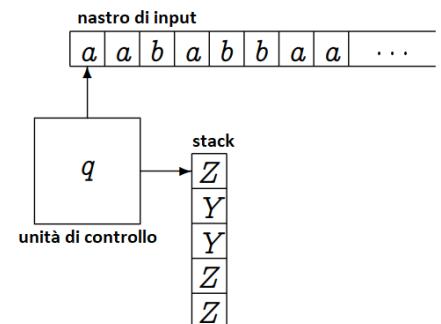
Nel capitolo precedente abbiamo visto come si fa il parsing top-down, dove costruiamo l'albero di parsing partendo dalla radice e producendo le foglie. In sostanza la struttura dati che ci serve è una sequenza di pile (stack), dati che contengono la cosiddetta predizione in cui leggiamo il simbolo che sta più a sx ed eventualmente lo andiamo a sostituire inserendo al posto della prima variabile, la predizione.

Un **automa a pila** è un modello astratto di macchina, un automa a stati finiti con uno **stack**, che ci permette di definire i linguaggi non contestuali, esattamente come i linguaggi che sono accettati da questo nuovo modello di macchina astratta.

È costituito da un'unità di controllo e un nastro di input, come l'automa a stati finiti, con in più uno stack.

Ad ogni passo la macchina legge una lettera dal nastro di input e preleva (**pop**) il simbolo che sta in cima allo stack (se è vuoto, la macchina si blocca). Dopodiché, in base allo stato e ai due simboli letti, la macchina assume un nuovo stato, scrive (**push**) in cima allo stack 0 o più lettere, e poi sposta la testina di lettura dal nastro di input sulla cella a dx.

Sono ammesse delle **ϵ -transizioni** in cui le operazioni di push e pop sono eseguite senza leggere il nastro di input. Quindi estrae e legge dalla pila, ma non tocca il nastro di input.



Un **automa a pila** è una settupla

$$\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$$

dove

- Q è un insieme finito, detto **insieme degli stati**,
- Σ è un alfabeto, detto **alfabeto di input**,
- Γ è un alfabeto, detto **alfabeto di pila**, lettere che scriviamo nello stack. (gamma maiuscola)
specifica che l'insieme di output è finito.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \wp(Q \times \Gamma^*)$ è la **funzione di transizione**,
o anche epsilon.
- $q_0 \in Q$ è lo **stato iniziale**, la funzione prende in input una tripla: stato, lettera alfabeto input, e lettera alfabeto di pila. Restituisce in output un insieme di coppie: **stato-parola**
- $Z_0 \in \Gamma$ è il **simbolo iniziale della pila**, nuovo stato che assume la macchina sull'alfabeto di pila.
- $F \subseteq Q$ è l'insieme degli **stati finali**.

Funzionamento

Inizialmente, l'automa è nello stato iniziale e la pila contiene il simbolo iniziale della pila.

Supponiamo che a un dato istante della computazione l'automa si trovi nello stato p , riceva a in input e Z dall'operazione di pop. Se $\delta(p, a, Z)$ contiene una coppia (q, γ) , allora l'automa si può portare nello stato q eseguendo il push di γ sulla pila (γ è quello che scrivo nella pila al posto della Z che è stata estratta).

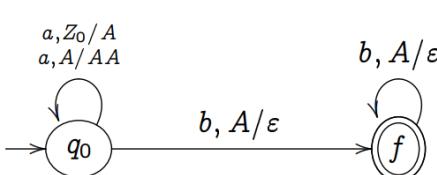
È un modello non deterministico.

ESEMPIO stato iniziale e stato finale

Sia $Q = \{q_0, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{Z_0, A\}$, $F = \{f\}$, e δ data da:

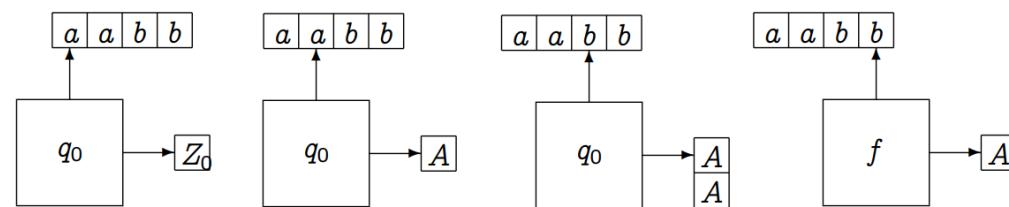
ad ogni tripla stato-lettera-pila, mi deve associare una coppia.

| | a | b | ϵ |
|-------|-------------|-----------------|------------|
| q_0 | (q_0, A) | — | — |
| A | (q_0, AA) | (f, ϵ) | — |
| f | — | — | — |
| Z_0 | — | — | — |
| A | — | (f, ϵ) | — |



Quando mi trovo nello stato q_0 , estraggo Z_0 dalla pila, e ho in input 'a': rimane nello stesso stato e scrive A nella pila.

Se invece estraggo 'A', aggiungo AA alla pila.



Come descrivere l'Automa

Il funzionamento dell'automa a pila è descritto da: stato, contenuto della pila, contenuto del nastro di input non ancora letto.

La **descrizione istantanea** è quindi una tripla $(q, u, \gamma) \in Q \times \Sigma^* \times \Gamma^*$, che mi descrive in ogni istante qual è lo stato della macchina. Le regole descritte in precedenza, in base all'istantanea, mi diranno quale sarà la descrizione istantanea all'istante successivo.

Una **relazione** tra descrizioni istantanee è denotata con: $\vdash_{\mathcal{A}}$.

$(q, aw, Z\alpha) \vdash_{\mathcal{A}} (p, w, \gamma\alpha)$: se prima la macchina era nello stato a sx, nell'istante successivo si troverà nella descrizione istantanea a dx.

Una **sequenza di descrizioni istantanee**, ognuna consecutiva alla precedente (ognuna può essere quella successiva alla precedente), si denota con $D \vdash_{\mathcal{A}}^* D'$.

$$D = D_0 \vdash_{\mathcal{A}} D_1 \vdash_{\mathcal{A}} \dots \vdash_{\mathcal{A}} D_n = D'$$

Quindi, si ha $D \vdash_{\mathcal{A}}^* D'$ se esiste una computazione che porta l'automa \mathcal{A} dalla descrizione istantanea D alla descrizione istantanea D' .

Metodi di Accettazione

Una parola viene **accettata per stato finale** se c'è una computazione che all'inizio parte con la mia parola sul nastro, stato iniziale, e simbolo iniziale della pila (soltanto nella prima); mano mano legge tutta la mia parola (consuma tutto il nastro di input) e alla fine si trova in uno stato finale che appartiene a F .

Quindi c'è una computazione che mi parte da q_0, w, Z_0 e arriva allo stato finale, ϵ (nastro di input vuoto), γ :
 $(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (f, \epsilon, \gamma)$, $f \in F, \gamma \in \Gamma^*$.

Una parola è **accettata per pila vuota** se c'è una computazione che parte con la solita configurazione iniziale (q_0, w, Z_0) e termina in uno stato non-finale perché sia il nastro di input che la pila sono vuote.
 $(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (p, \epsilon, \epsilon)$, $p \in Q$.

Una parola è **accettata per stato finale e pila vuota**, se si verificano entrambe le condizioni viste in precedenza:

$$(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (f, \epsilon, \epsilon), \quad f \in F.$$

Ciò sta ad indicare che ogni automa riconosce 3 linguaggi che potrebbero essere diversi.

L'insieme delle parole accettate per stato finale (risp., per pila vuota, per stato finale e pila vuota) sarà chiamato il **linguaggio accettato da \mathcal{A} per stato finale** (risp., per pila vuota, per stato finale e pila vuota). Essi sono denotati rispettivamente: $L_F(\mathcal{A})$, $L_P(\mathcal{A})$, $L(\mathcal{A})$.

Problema

Realizzare un automa a pila che accetti per pila vuota le palindromi pari sull'alfabeto $\Sigma = \{a, b\}$.

Come procediamo: quello che leggiamo lo aggiungiamo nella pila che, essendo LIFO, mi farà ritrovare quello che estraggo capovolto. Fino a metà scrivo nella pila quello che leggo, da metà in poi verifico che quello che leggo sia uguale a quello che c'è nella pila.

- ➊ copiare nella pila la prima metà dell'input,
- ➋ verificare che ciò che rimane dell'input è uguale a ciò che abbiamo nella pila, letto in ordine inverso.

Il non-determinismo ci consente di non specificare in quale punto della computazione si passa dalla fase 1 alla fase 2.

| | a | b | ϵ |
|-------|-----------------|-----------------|-----------------|
| q_0 | (q_0, A) | (q_0, B) | (f, ϵ) |
| A | (q_0, AA) | (q_0, BA) | (f, A) |
| B | (q_0, AB) | (q_0, BB) | (f, B) |
| f | (f, ϵ) | — | — |
| A | — | (f, ϵ) | — |
| B | — | — | — |

L'automa, essendo non deterministico, fa sì che io possa assumere in ogni passaggio di trovarmi a metà parola: se non è così la computazione fallisce e ne comincio un'altra, altrimenti procedo.

mi serve solo per accettare la parola vuota che è palindroma.

Automa a pila deterministico

Un **automa a pila** è **deterministico** quando: per ogni stato, per ogni simbolo in input, e per ogni simbolo sull'alfabeto di pila, l'insieme $\delta(q, a, Z) \cup \delta(q, \epsilon, Z)$ contiene al più un elemento.

L'automa ad ogni passo, quindi, può fare un solo movimento possibile.

Un linguaggio accettato per stato finale da un automa a pila deterministico si dirà deterministico.

Se \mathcal{A} è deterministico, ogni descrizione istantanea D ammette al più una descrizione istantanea successiva.

In tal caso, si può verificare in **tempo lineare** se una parola è accettata o meno dall'automa.

Metodi di Accettazione

- linguaggi accettati da automi a pila per **pila vuota**,
- linguaggi accettati da automi a pila per **stato finale**,
- linguaggi accettati da automi a pila per **pila vuota e stato finale**.

Pila vuota e stato finale

Se il linguaggio L è accettato da un automa per pila vuota e stato finale. Esiste effettivamente un automa a pila \mathcal{A} tale che L è accettato da un altro automa anche per stato finale, e da un altro ancora per pila vuota.

$$L = L(\mathcal{A}) = L_P(\mathcal{A}) = L_F(\mathcal{A}).$$

Dimostrazione

Sia \mathcal{A}_0 l'automa a pila che accetta L per pila vuota e stato finale.

Costruiamo un automa a pila \mathcal{A} con un unico stato finale f che si comporta nel modo seguente:

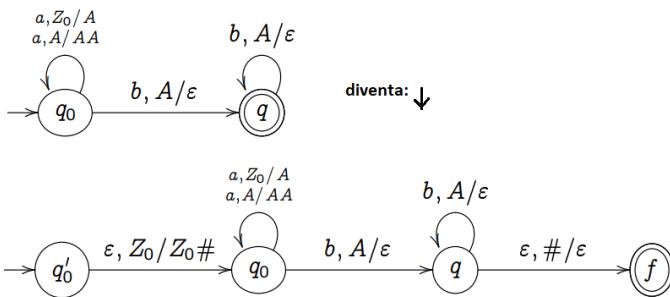
- come primo passo, scrive un simbolo speciale $\#$ all'inizio della pila,
- poi simula il funzionamento dell'automa \mathcal{A}_0 ,
- se, mentre si trova in uno stato finale di \mathcal{A}_0 , estrae $\#$ dalla pila, passa nel suo unico stato finale senza scrivere nulla sulla pila.

In una computazione di \mathcal{A} la pila si svuota se e solo se raggiunge lo stato finale. Quindi

$$L(\mathcal{A}) = L_P(\mathcal{A}) = L_F(\mathcal{A}).$$

Inoltre ciò avviene se e solo se l'automa \mathcal{A}_0 col medesimo input raggiunge uno stato finale con pila vuota. Pertanto

$$L = L(\mathcal{A}) = L_P(\mathcal{A}) = L_F(\mathcal{A}).$$



Pila vuota

Sia L il linguaggio accettato da un automa a pila per pila vuota.

Esiste effettivamente un automa a pila \mathcal{A} tale che $L = L(\mathcal{A}) = L_P(\mathcal{A}) = L_F(\mathcal{A})$.

Dimostrazione:

Sia \mathcal{A}_0 l'automa a pila che accetta L per pila vuota.

Posso supporre, senza perdita di generalità, che tutti gli stati siano finali.

Allora \mathcal{A}_0 accetta L per pila vuota e stato finale.

La conclusione segue dal lemma precedente.

Stato Finale

Sia L il linguaggio accettato da un automa a pila per stato finale. Esiste effettivamente un automa a pila \mathcal{A} tale che

$$L = L(\mathcal{A}) = L_P(\mathcal{A}) = L_F(\mathcal{A}).$$

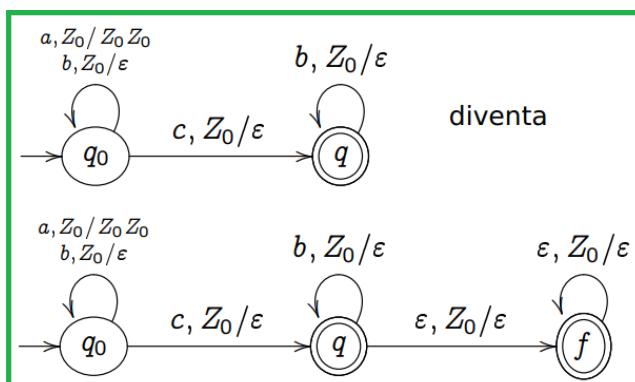
Dimostrazione:

Sia \mathcal{A}_0 l'automa a pila che accetta L per stato finale. Costruiamo un automa a pila \mathcal{A} , aggiungendo ad \mathcal{A}_0 lo stato e le istruzioni necessarie a svuotare la pila quando raggiunge uno stato finale:

- un ulteriore stato finale f ,
- ϵ -transizioni che mi portano da ogni stato finale in f , per qualunque simbolo estratto dalla pila, senza scrivere nulla nella pila,

È facile verificare che $L = L(\mathcal{A})$. La conclusione segue da uno dei lemmi precedenti.

← Esempio.



Il caso deterministico

I linguaggi accettati da automi deterministici per pila vuota, e i linguaggi accettati da automi deterministici per pila vuota e stato finale, coincidono. Se il primo è deterministico, anche il secondo lo è.

La classe dei linguaggi deterministici l'abbiamo definita come la classe dei linguaggi accettati dagli automi a pila deterministici per stato finale, che è una classe strettamente più grande.

Teorema di caratterizzazione dei linguaggi non contestuali 05.05.21

Questo teorema ha la funzione di far vedere che i linguaggi accettati dagli automi finiti, quel modello di macchina astratta introdotta nel capitolo precedente, coincidono con la classe dei linguaggi non contestuali (di **tipo 2**), generati da una grammatica non contestuale.

Come per i linguaggi regolari avevamo l'automa a stati finiti che definiva il modello di computazione per verificare se una parola apparteneva o meno al linguaggio, così il modello di macchina per i linguaggi non contestuali è l'automa a pila.

Il teorema di caratterizzazione vuole quindi dimostrare che un linguaggio è non contestuale se e solo se è riconosciuto da un automa a pila.

1. Dalla grammatica all'automa: sia G una grammatica non contestuale. Esiste effettivamente un automa a pila con un solo stato che accetta per pila vuota (e di conseguenza anche per gli altri due metodi di accettazione come abbiamo visto precedentemente) il linguaggio generato da G .

2. Dall'Automa con un solo stato alla grammatica: sia \mathcal{A} un automa a pila con un solo stato. Esiste effettivamente una grammatica non contestuale G che genera il linguaggio accettato per pila vuota da \mathcal{A} .

3. Dall'Automa generale all'Automa con un solo stato: sia \mathcal{A} un automa a pila. Esiste effettivamente un automa a pila con un solo stato che accetta per pila vuota il medesimo linguaggio di \mathcal{A} .

Dalla grammatica all'automa

Lo sappiamo già fare perché abbiamo già visto il parsing predittivo che non è altro che un automa a pila.

Con la procedura non deterministica: analizzo il simbolo che sta in cima allo stack, se è una variabile la vado a sostituire con il lato dx di una sua produzione, quindi non faccio altro che qualcosa che somiglia ad una ϵ -transizione di un automa a pila (estratto una variabile e ci scrivo il lato dx di una produzione).

Se invece il simbolo più a sx è un terminale, deve coincidere con la prima lettera dell'input da analizzare (simbolo di look-ahead). In caso affermativo viene spostato nel lato 'analisi', altrimenti l'esecuzione fallisce.

Ogni linguaggio non contestuale è accettato per pila vuota da un automa a pila con un solo stato:

Dimostrazione

Sia $L = L(G)$ con $G = \langle V, \Sigma, P, S \rangle$.

Possiamo supporre che i lati destri delle produzioni siano o singoli terminali o parole di N^* . (o un singolo terminale o una sequenza di variabili, anche vuota)

Costruiamo \mathcal{A} con un solo stato e

- alfabeto di input Σ ,
- alfabeto di pila N , costituito dalle variabili della mia grammatica. Il simbolo iniziale della pila è il simbolo iniziale della grammatica.
- simbolo iniziale della pila S ,
- $\delta(\epsilon, A) = \{\gamma \in N^* \mid A \rightarrow \gamma \text{ in } P\}, \quad A \in N$,
- $\delta(a, A) = \begin{cases} \{\epsilon\}, & \text{se } A \rightarrow a \text{ in } P, \\ \emptyset & \text{altrimenti} \end{cases} \quad a \in \Sigma, \quad A \in N$,

Verificheremo che $L(G) = L_P(\mathcal{A})$.

Per le epsilon transizioni:
estraggono dalla pila la variabile A e scrivono sulla pila il lato dx di una produzione.
Per le transizioni che prendono una lettera: non scrivo niente sulla pila (è epsilon) se c'è la produzione $A \rightarrow a$, altrimenti la transiz. non è definita.

ESEMPIO

Sia $L = \{a^n b^n \mid n \geq 0\}$. L è generato dalle produzioni

$$S \rightarrow aSb \mid \epsilon,$$

o, equivalentemente,

$$S \rightarrow ASB \mid \epsilon, \quad A \rightarrow a, \quad B \rightarrow b.$$

L'automa a pila corrispondente avrà alfabeto di input $\{a, b\}$, alfabeto di pila $\{S, A, B\}$, simbolo iniziale della pila S e la funzione di transizione

le epsilon produzioni sostituiscono ad ogni variabile in cima alla pila i lati dx delle produzioni. Per ognuna delle produzioni $A \rightarrow$, $B \rightarrow b$ mettiamo una epsilon all'incrocio tra il lato sx e il dx della produzione.

| | a | b | ϵ |
|---|------------|------------|-----------------|
| S | — | — | ASB, ϵ |
| A | ϵ | — | — |
| B | — | ϵ | — |

Dall'automa alla grammatica

Sia \mathcal{A} un automa a pila con un solo stato. Esiste effettivamente una grammatica non contestuale G che genera il linguaggio accettato per pila vuota da \mathcal{A} .

Dimostrazione

Sia $\mathcal{A} = \langle \{q_0\}, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_0\} \rangle$. Possiamo supporre $\Gamma \cap \Sigma = \emptyset$. Definiamo la grammatica G con che l'alfab. di pila e quello del linguaggio siano distinti.

- l'alfabeto di pila Γ come insieme delle variabili,
- l'alfabeto di input Σ come insieme dei simboli terminali,
- il simbolo iniziale della pila Z_0 come simbolo iniziale,
- per ogni transizione $q_0 \xrightarrow{a, Z/\gamma} q_0$ di \mathcal{A} , G avrà la produzione $Z \rightarrow a\gamma$.

Analogamente al caso precedente, si verifica che 'a' lettera, Z variabile pila, gamma parola che scrivo sulla pila col push.

$$L(G) = L_P(\mathcal{A}).$$

ESEMPIO

Sia \mathcal{A} l'automa a pila con un unico stato, alfabeto di input $\Sigma = \{a, b, \bar{a}, \bar{b}\}$, alfabeto di pila $\{S, A, B\}$, simbolo iniziale della pila S e la funzione di transizione

se S legge 'a' scrive A .
se S legge 'a' barrato' si limita a fare il pop senza il push.

| | (|) |) |) |
|---|----|----|-----------|------------|
| S | a | b | \bar{a} | \bar{b} |
| A | A | B | — | — |
| B | AB | BB | — | ϵ |

La grammatica corrispondente avrà le produzioni

$$S \rightarrow aA | bB, \quad A \rightarrow aAA | bBA | \bar{a}, \quad B \rightarrow aAB | bBB | \bar{b}.$$

Proposizione Sia \mathcal{A} un automa a pila. Esiste effettivamente un automa a pila con un solo stato che accetta per pila vuota il medesimo linguaggio di \mathcal{A} .

Idea della dimostrazione:

Dato un automa a pila \mathcal{A} , realizzare un automa a pila con un solo stato che simula \mathcal{A} , registrando nella pila anche lo stato di \mathcal{A} .

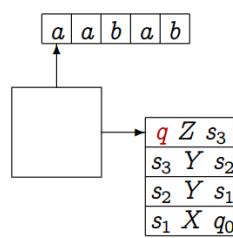
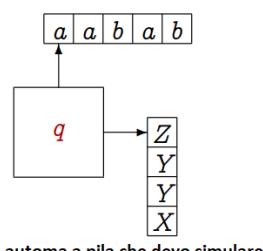
Problema:

Se \mathcal{A} non fa push, nella simulazione non posso registrare lo stato.

Soluzione:

Nell'automa a uno stato ogni cella della pila contiene

- il simbolo di pila di \mathcal{A} ,
- uno stato arbitrario (cioè tutti, per il non-determinismo),
- lo stato arbitrario contenuto nella cella sottostante,
- solo la cella top, al posto dello stato arbitrario contiene lo stato dell'automa simulato.



$$q \xrightarrow{a, Z/XY} p \quad \text{diventa} \quad a, [q \ Z \ s_1] / \boxed{p \ X \ s_2 \ Z \ s_3} \quad \text{stato messo a caso, ma in realtà li metto tutti.}$$

$$q \xrightarrow{a, Z/\epsilon} p \quad \text{diventa} \quad a, [q \ Z \ p] / \epsilon$$

Tutto ciò funziona per la classe di automi a pila con un solo stato.

Perché esiste effettivamente un automa a pila con un solo stato, che accetta per pila vuota lo stesso linguaggio accettato da un automa qualunque con un qualsiasi numero di stati?

Se ho la pila, lo stato lo scrivo sulla pila: l'automa ad ogni passo guarda lo stato, guarda il simbolo in cima alla pila e fa delle cose, e scrive lo stato nuovo sulla pila. Tuttavia, se l'automa non scrive nulla sulla pila, dove scrivo lo stato?

Sulla pila non andrò a scrivere solo lo stato, ma scriverò anche cosa c'è scritto sotto.

La classe dei linguaggi non contestuali è chiusa per le operazioni regolari: unione, concatenazione e chiusura di Kleene. Mentre non è chiusa per intersezione né per complemento.

Unione

Dimostrazione

Siano $L_1 = L(G_1)$, $L_2 = L(G_2)$ con

$$G_1 = \langle V_1, \Sigma, P_1, S_1 \rangle \quad \text{e} \quad G_2 = \langle V_2, \Sigma, P_2, S_2 \rangle.$$

Possiamo supporre $N_1 \cap N_2 = \emptyset$.

esplicare qual è la costruzione per realizzare la grammatica non contestuale che mi genera l'unione, la concatenazione o la chiusura di Kleene, date le grammatiche che mi generano i due linguaggi di partenza.

Costruiamo la grammatica $G = \langle V, \Sigma, P, S \rangle$ con

- $N = \{S\} \cup N_1 \cup N_2$,
- $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}$.

La grammatica G genera $L_1 \cup L_2$.

1. Prendo le due grammatiche e suppongo che non abbiano variabili in comune (se ce l'hanno, cambio il nome delle variabili)

2. Costruisco per unione la nuova grammatica: le variabili e i terminali delle due grammatiche di partenza, più una variabile nuova S (simbolo iniziale), le produzioni delle due grammatiche di partenza, più le due nuove produzioni di S che produce il simbolo iniziale della prima grammatica e il simbolo iniziale della seconda grammatica.

Concatenazione

Dimostrazione

Siano $L_1 = L(G_1)$, $L_2 = L(G_2)$ con

$$G_1 = \langle V_1, \Sigma, P_1, S_1 \rangle \quad \text{e} \quad G_2 = \langle V_2, \Sigma, P_2, S_2 \rangle.$$

Possiamo supporre $N_1 \cap N_2 = \emptyset$.

Costruiamo la grammatica $G = \langle V, \Sigma, P, S \rangle$ con

- $N = \{S\} \cup N_1 \cup N_2$,
- $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\} \cup \{S \rightarrow S_1 S_2 | \varepsilon\}$.

Discorso simile al precedente: le variabili delle due grammatiche più una variabile nuova, i terminali delle due grammatiche, le produzioni delle due grammatiche più la nuova produzione $S \rightarrow$ la concatenazione dei simboli iniziali delle due grammatiche di partenza.

La grammatica G genera $L_1 \cup L_2 L_2^*$.

Chiusura di Kleene

Dimostrazione

Siano $L_1 = L(G_1)$, $L_2 = L(G_2)$ con

$$G_1 = \langle V_1, \Sigma, P_1, S_1 \rangle \quad \text{e} \quad G_2 = \langle V_2, \Sigma, P_2, S_2 \rangle.$$

Possiamo supporre $N_1 \cap N_2 = \emptyset$.

Costruiamo la grammatica $G = \langle V, \Sigma, P, S \rangle$ con

- $N = \{S\} \cup N_1$,
- $P = P_1 \cup \{S \rightarrow S_1 S | \varepsilon\}$.

La grammatica G genera L_1^* .

In questo caso avremo una sola grammatica perché è la chiusura di un linguaggio. Nella nuova grammatica avremo: le variabili della vecchia grammatica G_1 più un nuovo simbolo S che fungerà da simbolo iniziale, i terminali di G_1 , le produzioni di G_1 più due nuove produzioni $S \rightarrow$ la concatenazione di se stesso con il vecchio simbolo iniziale, e $S \rightarrow$ parola vuota.

ESEMPIO

I linguaggi

$$L_1 = \{a^n b^n \mid n \geq 0\}, \quad L_2 = \{b^n c^n \mid n \geq 0\}, \quad a^*, \quad c^*$$

sono non contestuali.

Per la proposizione precedente, lo saranno anche

$$L_3 = L_1 c^* = \{a^n b^n c^m \mid m, n \geq 0\}, \quad \text{otteniamo un}$$

$$L_4 = a^* L_2 = \{a^m b^n c^n \mid m, n \geq 0\}. \quad \text{linguaggio non contestuale}$$

ma non la loro intersezione

$$L_3 \cap L_4 = \{a^n b^n c^n \mid n \geq 0\}.$$

Pertanto

La famiglia dei linguaggi non contestuali non è chiusa per intersezione né per complemento.

Intersezione

Se faccio l'intersezione non è detto che mi venga fuori un linguaggio non contestuale. Infatti quello dell'esempio non è non-contestuale. Però se uno è non contestuale e l'altro è regolare, sono certa che mi venga fuori un linguaggio non contestuale.

Proposizione L'intersezione di un linguaggio non contestuale con un linguaggio regolare è un linguaggio non contestuale. mi basta far vedere che è accettato da un automa a pila.

Dimostrazione

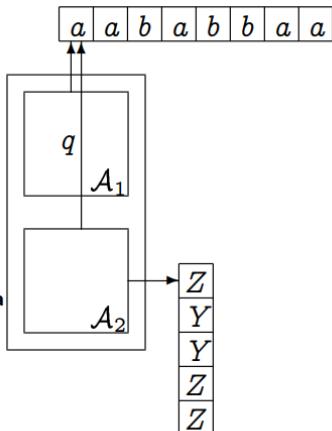
Un linguaggio regolare R è accettato da un automa a stati finiti \mathcal{A}_1 ,

Un linguaggio non contestuale L è accettato da un automa a pila \mathcal{A}_2 con un solo stato,

L'intersezione $R \cap L$ è accettata, per stato finale e pila vuota, da un automa a pila che simula \mathcal{A}_1 e \mathcal{A}_2 in parallelo.

Automa a pila dove gli stati cambiano in base alla legge dell'automa a stati finiti che mi accetta il linguaggio regolare. Le operazioni di pop e push, sono quelle dell'automa a pila che mi accetta il linguaggio non contestuale.

Se alla fine della computazione il nastro è esaurito e l'automa è arrivato in uno stato finale, e la pila è vuota, vuol dire che la parola è accettata da entrambi i tipi di automi (a pila e a stati finiti)



Linguaggio di Dick non contestuale

Il linguaggio di Dick D_n è il linguaggio sull'alfabeto

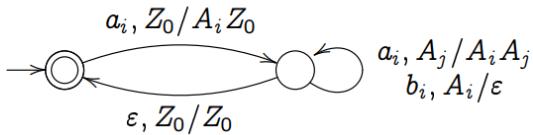
$$\begin{array}{ll} \text{ai=} & \Sigma_n = \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\}. \\ \text{bi=} & \end{array}$$

generato dalla grammatica con le produzioni

$$S \rightarrow a_i S b_i S, \quad i = 1, 2, \dots, n, \quad S \rightarrow \epsilon.$$

Equivalentemente: sequenze di parentesi ben bilanciate

Accettato dall'automa a pila deterministico



Morfismi

Definizione

Un morfismo $f: \Sigma^* \rightarrow \Gamma^*$ è una funzione tale che funzione che definita sull'alfab. sigma, mi restituisce parole sull'alfab. gamba.

• $f(\epsilon) = \{\epsilon\}$, la parola vuota, vale la parola vuota.

• $f(a_1 \cdots a_n) = f(a_1) \cdots f(a_n)$, per ogni $a_1, \dots, a_n \in \Sigma$, $n > 0$. una parola che è la concatenazione di n lettere, l'immagine si trova anche concatenando l'immagine delle lettere.

Osservazione

Un morfismo è completamente determinato dalle immagini delle lettere.

Esempio

Sia $f: \{a, b\}^* \rightarrow \{a, b\}^*$ il morfismo definito da

$$f(a) = a, \quad f(b) = bb.$$

Per es., si ha $f(bab) = bbabb$.

Se $L = \{a^n b^n \mid n \geq 0\}$, allora $f(L) = \{a^n b^{2n} \mid n \geq 0\}$.

Proposizione La classe dei linguaggi non contestuali è chiusa per morfismi. se prendo un ling. non cont. e prendo un morfismo, faccio l'immagine di tutte le parole del ling. non cont., ottengo un nuovo linguaggio anche lui non contestuale.

Dimostrazione

Sia $L \subseteq \Sigma^*$ un linguaggio non contestuale e $f : \Sigma^* \rightarrow \Gamma^*$ un morfismo;

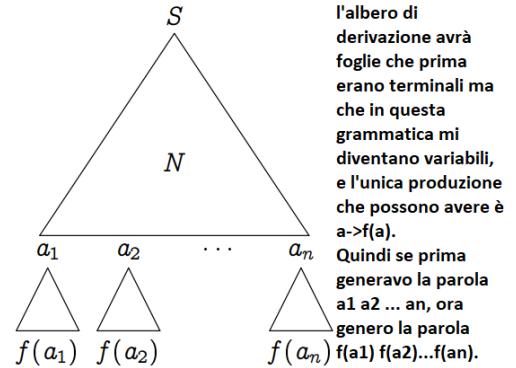
L è generato da una grammatica non contestuale G ;

possiamo supporre che il vocabolario di G non interseca Γ ;

costruiamo la grammatica G' come segue

- le variabili sono tutte le lettere del vocabolario di G (variabili e terminali),
- i terminali sono le lettere di Γ ,
- le produzioni sono quelle di G e, inoltre, le produzioni $a \rightarrow f(a)$, $a \in \Sigma$, in pratica ho aggiunto solo una nuova produzione che a quelli che prima erano terminali, mi sostituisce l'immagine
- il simbolo iniziale è quello di G .

Si verifica che $L(G') = f(L)$.



l'albero di derivazione avrà foglie che prima erano terminali ma che in questa grammatica mi diventano variabili, e l'unica produzione che possono avere è $a \rightarrow f(a)$.
Quindi se prima generavo la parola a1 a2 ... an, ora genero la parola f(a1) f(a2)...f(an).

Sostituzioni

Definizione

Una **sostituzione** $f : \Sigma^* \rightarrow \wp(\Gamma^*)$ è una funzione tale che

- $f(\varepsilon) = \{\varepsilon\}$, ora la funzione non restituisce parola ma linguaggi (insiemi di parole).
-- Linguaggio che contiene solo la parola vuota.
- $f(a_1 \cdots a_n) = f(a_1) \cdots f(a_n)$, per ogni $a_1, \dots, a_n \in \Sigma$, $n > 0$.
concatenazione di linguaggi

Osservazione

Una sostituzione è completamente determinata dalle immagini delle lettere.

$$f(L) = \bigcup_{w \in L} f(w)$$

l'immagine di un linguaggio è l'unione delle immagini delle varie parole che lo compongono.

Definizione

Una sostituzione è **non contestuale** se l'immagine di ogni lettera è un linguaggio non contestuale.

Proposizione La classe dei linguaggi non contestuali è chiusa per sostituzioni non contestuali.

Teorema di rappresentazione

Esempio

Sia D_2 il linguaggio di Dick sull'alfabeto $\Sigma_2 = \{a_1, a_2, b_1, b_2\}$, $R = (a_1 + a_2)^*(b_1 + b_2)^*$ e $f : \Sigma_2^* \rightarrow \{a, b\}^*$ il morfismo definito da

$$f(a_1) = f(b_1) = a, \quad f(a_2) = f(b_2) = b.$$

Allora

$$f(D_2 \cap R)$$

è l'insieme delle palindrome pari sull'alfabeto Γ .

Teorema (Chomsky, Schützenberger) Un linguaggio L è non contestuale se e soltanto se esistono un intero $k > 0$, un linguaggio regolare R e un morfismo f tali che

$$L = f(D_k \cap R).$$