Linguaggi e modelli computazionali M

A cura di Marco Moschettini

Linguaggi e macchine astratte

Algoritmi e programmi

- **Algoritmo:** sequenza **finita** di mosse che risolve in **un tempo finito** una *classe* di problemi
- **Codifica:** scrittura di un algoritmo attraverso un insieme di **istruzioni** di un **linguaggio di programmazione**
- **Programma:** testo scritto in accordo alla **sintassi** e alla **semantica**.

Un programma **può *non essere*** un **algoritmo!** (Ad esempio se il programma *non termina*).
L'esecuzione di un programma presuppone l'esistenza di un **automa esecutore** che sia in grado di eseguire le azioni specificate dall'algoritmo.

Automa esecutore

Caratteristiche di un automa esecutore:

- Deve poter ricevere dall'esterno la **descrizione dell'algoritmo**
- Deve essere in grado di **interpretare** un linguaggio (detto *linguaggio macchina*).

Vincoli di realizzabilità fisica:

- Se l'automa è fatto di parti, queste sono in **numero finito**
- Ingresso e uscita devono essere denotabili attraverso un **insieme finito di simboli**

Sistemi formali

Diversi tipi di approcci matematici per definire il concetto di *commutabilità:*

- **Gerarchia di macchine astratte:** dispositivi con proprio stato interno utilizzabili come memoria e caratterizzato da un certo insieme di mosse elementari. Esempi:
  - Macchine combinatorie
  - Macchine a stati finiti
  - Macchina a stack
  - **Macchina di Turing**
- **Approccio funzionale (Hilbert, Church, Kleene):** fondato sul concetto di **funzione matematica** che mira a caratterizzare il concetto di **funzione computabile.**
- **Sistemi di riscrittura (Thue, Post, Markov):** descrivono l'automa come un insieme di regole di **riscrittura** (o di **inferenza**) che trasformano le **frasi** in **altre frasi.**

Gerarchia di macchine astratte

Diverse capacità di **risolvere i problemi.** Se anche la macchina più potente non riesce a risolvere un dato problema, esso **potrebbe non essere *risolubile***.

La macchina base combinatoria

Formalmente definita dalla tripla:

<I, O, mfn>

con:
- **I** = insieme finito dei **simboli di ingresso**
- **O** = insieme finito dei **simboli di uscita**
- **mfn** = **I —> O (funzione di macchina)** ovvero la funzione che collega ogni ingresso ad ogni uscita

Può risolvere solo problemi in cui è possibile **enumerare tutte le possibili configurazioni di ingresso —> Non adatto a risolvere problemi che richiedono una *memoria interna***

Automa a stati

Formalmente definito dalla tripla:

<I, O, S, mfn, sfn>

con:
- **I** = insieme finito dei **simboli di ingresso**
- **O** = insieme finito dei **simboli di uscita**
- **mfn = I x S —> O (funzione di macchina)**
- **sfn = I x S —> S (funzione di stato)**

L'uscita ora **dipende anche dallo stato;** quindi a parità di ingresso possono esserci uscite diverse nell'arco del tempo. Ne esistono vari tipi:
- automi di *Mealy, Moore*
- *sincroni, asincroni*
- ecc…

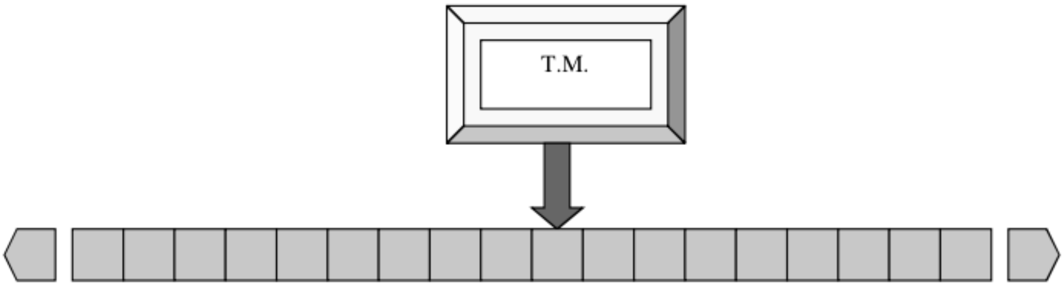
Limitazioni:
- **Inadatto a risolvere problemi che non consentono di limitare a priori la lunghezza delle sequenze d'ingresso (memoria finita)**

Macchina di Turing

Si introduce il concetto di *nastro* come **supporto di memorizzazione esterno**
Formalmente definita dalla quintupla:

<A, S, mfn, sfn, dfn>

con:
- **A =** insieme finito dei **simboli di ingresso e uscita**
- **S =** insieme finito degli **stati** (uno è HALT)
- **mfn = A x S —> A (funzione di macchina)**
- **sfn = A x S —> S (funzione di stato)**
- **dfn = A x S —> D {Left, Right, None} (funzione di direzione)**



La **macchina di Turing,** essendo munita di **testina di lettura/scrittura**, può alternativamente;
- **leggere** un simbolo dal nastro
- **scrivere** il simbolo specificato da **mfn()** sul nastro
- **transitare** in un nuovo stato interno specificato da **sfn()**
- **spostarsi** sul nastro di una posizione nella direzione indicata da **dfn()**

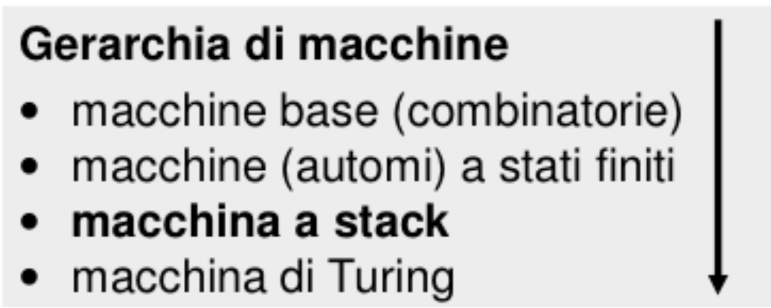
Quando raggiunge lo stato HALT la macchina si *ferma.*

Risolvere un problema con la macchina di Turing:
- definire una rappresentazione dei *dati iniziali* sul nastro
- definire la **parte di controllo** (funzioni mfn, sfn, dfn) in modo da rendere disponibile sul nastro, alla fine, la rappresentazione della soluzione.

Tesi di Church-Turing

NON ESISTE ALCUN FORMALISMO CAPACE DI RISOLVERE UNA CLASSE DI PROBLEMI PIÙ AMPIA DI QUELLA RISOLTA DALLA MACCHINA DI TURING

Ha tuttavia senso definire anche delle **macchine intermedie** fra gli automi a stati finiti e la macchina di Turing



Una volta definita la parte di controllo una **MdT** è capace di risolvere un problema dato (risolubile). Tuttavia essa rimane specifica di quel problema. Soluzione —> **Macchina universale**

Macchina di Turing universale

È una macchina che è in grado di prelevare l'algoritmo stesso dal nastro (memoria).
- La parte di controllo di tale macchina dovrebbe essere in grado di
  - **leggere** dal nastro una descrizione dell'algoritmo (**fetch**)
  - **interpretare** le istruzioni dell'algoritmo (**decode**)
  - **eseguire** le istruzioni interpretate (**execute**)

Quindi è necessario **un linguaggio** per esprimere l'algoritmo e una **macchina** che lo interpreti la **UTM** è **l'interprete del linguaggio.**

Paragonandola alla macchina di Von Neumann

Macchina di Turing		Macchina di Von Neumann
Leggere/scrivere simboli dal/sul nastro	—>	Lettura/scrittura dalla/Sulla memoria RAM/ROM
Transitare in un nuovo stato interno	—>	Nuova configurazione dei registri della CPU
Spostarsi sul nastro di una o più posizioni	—>	Scelta della cella di memoria su cui operare

La **UTM è pura computazione —> non modella la dimensione dell'interazione** (che invece esiste in Von Neumann). **NO I/O**!

Problemi risolubili e computabilità

Se neanche la macchina di Turing riesce a risolvere un problema, **quel problema non è risolubile!**

Problema risolubile

UN PROBLEMA LA CUI SOLUZIONE PUÒ ESSERE ESPRESSA DA UNA MACCHINA DI TURING O UN FORMALISMO EQUIVALENTE

Funzione caratteristica di un problema

Dato un problema **P,** l'insieme **X** dei suoi dati di ingresso, l'insieme **Y** delle sue risposte corrette, si dice **funzione caratteristica** del problema **P** la funzione:

$$f_p: X \longrightarrow Y$$

che associa ad ogni dato d'ingresso la corrispondente risposta corretta.
Quindi **funzione computabile? —> problema risolvibile!**

Funzione computabile

UNA FUNZIONE F: A—>B PER LA QUALE ESISTE UNA MACCHINA DI TURING CHE, DATA SUL NASTRO UNA RAPPRESENTAZIONE DI X_A, DOPO UN NUMERO FINITO DI PASSI, PRODUCE SUL NASTRO UNA RAPPRESENTAZIONE DEL RISULTATO $F(X)_B$.

Procedimento di Gödel

Procedimento che permette di rappresentare una **collezione di numeri naturali** con un **unico numero naturale.**

Procedimento

- Siano N_1, N_2,… N_k i numeri naturali
- Siano P_1, P_2,… $P_{k\,i}$ i primi k numeri primi
- Il numero **R** := P_1^{N1} x P_2^{N2} x … x P_k^{Nk} rappresenta **univocamente** la collezione grazie all'unicità della scomposizione in fattori primi.

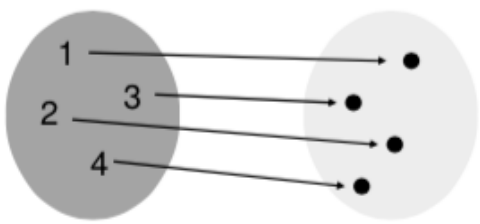
Tuttavia, è noto che l'insieme F: { f: N —> N} **non è enumerabile**, mentre l'insieme delle macchine di Turing **è enumerabile.** Di conseguenza ***la maggioranza delle funzioni NON può essere calcolata!***
Tuttavia le sole funzioni che ci interessano sono quelle che possono essere **definite** tramite un **linguaggio** i cui simboli fanno parte di un **alfabeto** di simboli **finito!** Quindi le funzioni che possiamo realmente calcolare sono molto meno e sono un insieme **enumerabile.** Tuttavia **NON sono tutte calcolabili —> funzioni non computabili**

Funzioni non computabili

Esistono funzioni definibili ma non computabili (esempio **HALT** della macchina di Turing)

Insiemi e linguaggi

Insiemi numerabili



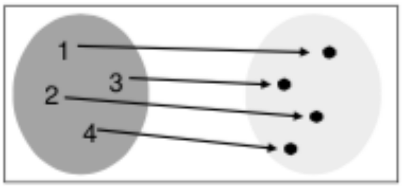
Insieme i cui elementi possono essere **contati,** ossia che possiedono una **funzione biettiva**

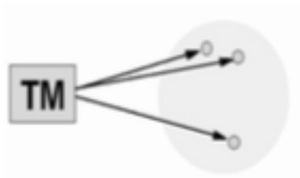
f: N —> S

che mette in corrispondenza i numeri naturali con gli elementi dell'insieme

Insiemi ricorsivamente enumerabili

Detti anche **insiemi semidecidibili,** sono insiemi che possono essere **effettivamente costruiti per enumerazione dei suoi elementi.** In tali insiemi la funzione biettiva f: N —> S non è solo **definibile** ma è anche **calcolabile da una Macchina di Turing**





Tuttavia, il fatto che l'insieme S possa **essere costruito**, **NON SIGNIFICA** che si possa decidere se un certo elemento **x** appartenga all'insieme stesso.

Insiemi ricorsivi

Detti anche **insiemi decidibili** sono insiemi la cui **funzione caratteristica è computabile.**

$$f(x) = \begin{cases} 1, \text{ se } x \in S \\ 0, \text{ se } x \notin S \end{cases}$$

ovvero se esiste una **Macchina di Turing** capace di rispondere "Sì" o "No" alla domanda senza entrare in un *ciclo infinito.*

Teoremi:

1. **TEOREMA 1: Se un insieme è decidibile è anche semidecidibile!**
2. **TEOREMA 2: Un insieme S è decidibile <u>se e solo se</u> sia S, sia il suo complemento (N-S) sono semidecidibili**

Questo è fondamentale perché in un linguaggio di programmazione (**generato da un insieme finito di simboli**) è fondamentale poter **decidere se una frase è giusta o sbagliata** <u>senza entrare in un ciclo infinito.</u>

Linguaggi e grammatiche

Cos'è un linguaggio?

UN LINGUAGGIO È UN INSIEME DI PAROLE E DI METODI DI COMBINAZIONE DELLE PAROLE USATE E COMPRESE DA UNA COMUNITÀ DI PERSONE

Definizione **poco precisa.**

Sintassi e Semantica

Distinzione tra:

- **Sintassi:** insieme di regole **formali** per la scrittura di programmi in un linguaggio, che dettano le modalità per costruire le **frasi corrette.**
  - è espressa tramite notazioni tipo:
    - **BNF**
    - **EBNF**
    - **diagrammi sintattici**
- **Semantica:** insieme dei significati da attribuire le frasi.
  - è espressa:
    - **a parole**
    - **mediante azioni (semantica operazionale)**
    - **mediante funzioni matematiche (semantica denotazionale)**
    - **mediante formule logiche (semantica assiomatica)**

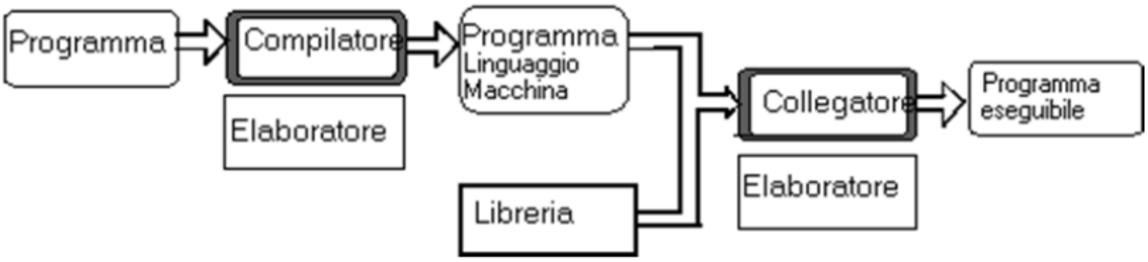
Interpretazione e compilazione

Un **interprete** per un linguaggio L è un programma che:

- Accetta in ingresso le frasi di L
- Le esegue per **una volta**.

Un **compilatore** di un linguaggio L è un programma che:

- Accetta in ingresso un programma scritto in L
- Lo riscrive in un altro linguaggio



Analisi Lessicale

L'analisi lessicale consiste nell'individuazione delle singole parole (token) di una frase. L'analizzatore lessicale (**scanner**) è il componente che, data una sequenza di caratteri, restituisce in sequenza i token che compaiono nella frase

Analisi sintattica

L'analisi sintattica consiste nella **verifica** che una frase possa essere costruita sulla base delle regole grammaticali del linguaggio. L'analizzatore sintattico (**parser**) è il componente che, data la sequenza di token prodotti dallo scanner, produce una rappresentazione interna della frase sotto forma di albero

Analisi semantica

L'analisi semantica consiste nel **calcolo** del significato del linguaggio. Un analizzatore semantico è un componente che, data la rappresentazione intermedia prodotta dal parser, controlla la coerenza logica interna del programma.

Proprietà desiderabili

- Deve essere **effettivamente generabile**
- Deve essere **decidibile** (ossia deve essere possibile se una frase appartiene o no al linguaggio)

Descrizione di un linguaggio (Definizioni)

Alfabeto

UN ALFABETO A È UN INSIEME FINITO E NON VUOTO DI SIMBOLI ATOMICI

Stringa

UNA STRINGA È UNA SEQUENZA DI SIMBOLI, OSSIA UN ELEMENTO DEL PRODOTTO CARTESIANO A^N.

Linguaggio

UN LINGUAGGIO L SU UN ALFABETO A È UN INSIEME DI STRINGHE SU A. UNA FRASE DI UN LINGUAGGIO È UNA STRINGA APPARTENENTE A TALE LINGUAGGIO.

Cardinalità di un linguaggio

NUMERO DI FRASI DI UN LINGUAGGIO

Chiusura di un alfabeto A

Insieme <u>infinito</u> di tutte le stringhe composte con i simboli di A:

$$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots$$

Chiusura positiva di un alfabeto A

Insieme <u>infinito</u> di tutte le stringhe **<u>non nulle</u>** composte con simboli di A:

$$A^+ = A^* - \{\varepsilon\}$$

Grammatica formale

Una grammatica formale è una *notazione* formale con cui esprimere in modo rigoroso la sintassi di un linguaggio.
Una grammatica è una quadrupla

<VT, VN, P, S>

con:
- **VT**: insieme finito di simboli **terminali** (caratteri e stringhe appartenenti a un alfabeto A)
- **VN**: insieme finito di simboli **non terminali** (meta-simboli che rappresentano categorie sintattiche)
- **P:** insieme finito di **produzioni**, ossia regole di riscrittura $\alpha \longrightarrow \beta$ con α e β stringhe.
- **S**: è un particolare simbolo non-terminale detto **simbolo iniziale o scopo** della grammatica.

Gli insiemi VT e VN devono essere **disgiunti** $VT \cap VN = \emptyset$
L'unione di VT e VN si dice **vocabolario** della grammatica.

Convenzioni

- Simboli **terminali** sono rappresentati da lettere minuscole
- **meta-simboli** rappresentati da lettere MAIUSCOLE
- lettere greche indicano **stringhe di terminali e meta-simboli.**

Forme di frasi e frasi

- Si dice **forma di frase** una qualsiasi stringa **σ** comprendente **sia simboli terminali, sia meta-simboli** derivabile dallo scopo S
- Si dice **frase** una **forma di frase** comprendente **solo simboli terminali.**

Derivazione

Se **G** è una grammatica <VT, VN, P, S> e **α** e **β** sono due stringhe appartenenti a $(VT \cup VN)$ con **α != ε,** si dice che che **β <u>deriva direttamente</u>** da **α** se:
- Le stringhe si possono decomporre in **$\alpha = \eta\, A\, \delta$** e **$\beta = \eta\, \gamma\, \delta$**
- esiste la produzione **$A \longrightarrow \gamma$**

Si dice poi che **β deriva** da **α** (in generale) se:
- Esiste una sequenza di N derivazioni dirette che da **possono produrre β**

Inoltre si dice **sequenza di derivazione** la sequenza di passi necessari per produrre una forma di frase a partire dallo scopo S mediante l'applicazione di una o più regole di produzione.

S —> σ	σ deriva direttamente da una sola derivazione di produzioni
S —>$^{+}$ σ	σ deriva da S con una o più applicazioni di produzioni
S —>* σ	σ deriva da S con zero o più applicazioni di produzioni

Definizione di Linguaggio

DEFINISCO LINGUAGGIO LG GENERATO DALLA GRAMMATICA G L'INSIEME DELLE FRASI DERIVABILI DAL SIMBOLO INIZIALE S APPLICANDO LE PRODUZIONI P: OVVERO

LG = { Ƒ APPARTENENTE A VT* TALE CHE S —>* Ƒ}

Linguaggi equivalenti

Due grammatiche sono equivalenti se generano lo stesso linguaggio. Stabilire se due grammatiche sono equivalenti è un problema indecidibile.

Classificazione di Chomsky

Le grammatiche sono classificate in 4 tipi in base alla **struttura delle produzioni:**

- **TIPO 0:** *nessuna restrizione sulle produzioni*
- **TIPO 1** (dipendenti dal contesto)**:** *produzioni vincolate alla forma:*
  - ***ß A δ —> ß α δ** (con ß, δ, α appartenenti alla grammatica)*: quindi A può essere sostituita da α solo nel contesto ß A δ. In questo modo le riscritture **non accorciano mai** la forma di frase corrente.
  - **Non ammettono** la stringa vuota
- **TIPO 2** (libere dal contesto)**:** *produzioni vincolate alla forma:*
  - ***A —> α** (con α appartenente alla grammatica).* In questo caso A può **sempre** essere sostituito da α, indipendentemente dal contesto.
  - **Ammettono** la stringa vuota
- **TIPO 3** (grammatiche regolari)**:** *produzioni vincolate alla forma:*
  - *Lineare a destra:*
    - ***A —> σ***
    - ***A —> σB***
  - *Lineare a sinistra:*
    - ***A —> σ***
    - ***A —>Bσ***
  - **Ammettono** la stringa vuota

Per grammatiche regolari, è sempre possibile e conveniente trasformare la grammatica in forma *strettamente lineare*.

- Si passa da *σ appartenente a VT** ***(stringa di caratteri)*** *a σ appartenente a VT* ***(singolo carattere)***

Relazione gerarchica

Le 4 grammatiche di Chomsky sono in relazione gerarchica (la tipo 3 è un caso particolare della 2 ecc.. ecc…) e per questo motivo un linguaggio può essere generato da più grammatiche, **anche di tipo diverso.** Di conseguenza, se una grammatica G genera il linguaggio L(G) **non è detto** che L(G) sia dello stesso tipo di G (potrebbe essere più semplice)

Il problema della stringa vuota

Le grammatiche di **tipo 1** non ammettono stringa vuota ε sul lato destro delle produzioni, mentre le grammatiche di **tipo 2** la ammettono. Come è possibile che le grammatiche siano in gerarchia tra di loro ma ci sia questa contraddizione? No perché esiste questo **teorema**:

LE PRODUZIONI DI GRAMMATICHE DI TIPO 2 (E DI TIPO 3) POSSONO SEMPRE ESSERE RISCRITTE IN MODO DA EVITARE LA STRINGA VUOTA: AL PIÙ POSSONO CONTENERE LA REGOLA S —> ε

L'eliminazione delle ε-rules

Se G è una grammatica context-free (tipo 2) con produzioni nella forma A —> α (con α che può essere ε), allora esiste una grammatica context-free G' che genera **lo stesso linguaggio L(G)** ma le cui produzioni hanno o la forma A —> α (con α diverso da ε) oppure la forma S —> ε, ed S non compare sulla destra di nessuna produzione.
Come identificare la grammatica equivalente G'?
Ecco un esempio in cui la grammatica G', derivata da G, non contiene la stringa vuota:

Grammatica G (con **ε** -rules)

```
S →   A B | B
A →   a A | ε
B →   b B | c
```

Grammatica G'

```
S →   A B | B
A →   a (A|ε)
B →   b B | c
```

Grammatica G'

```
S →   A B | B
A →   a A | a
B →   b B | c
```

Altrimenti è possibile avere la stringa vuota ma solo al livello dello scopo del linguaggio (1 livello) come nell'esempio seguente:

Grammatica G (con ε -rules)

$S \rightarrow A\ B$
$A \rightarrow a\ A \mid \varepsilon$
$B \rightarrow b\ B \mid \varepsilon$

Grammatica G'

$S \rightarrow (A|\varepsilon)\ (B|\varepsilon)$
$A \rightarrow a\ (A|\varepsilon)$
$B \rightarrow b\ (B|\varepsilon)$

Grammatica G'

$S \rightarrow A\ B \mid B \mid A \mid \varepsilon$
$A \rightarrow a\ A \mid a$
$B \rightarrow b\ B \mid b$

In questo caso il linguaggio L(G) comprende la stringa vuota ma le forme di frase non possono comunque accorciarsi.

Come distinguere le grammatiche

Cosa discrimina un **tipo 1 da un tipo 2 (context-free)**? Nel tipo 1 è possibile definire produzioni che scambiano due simboli: BC —> CB. Nel tipo 2 questo è **impossibile** da definire perché la restrizione del tipo 2 ha un solo simbolo sul lato sinistro della produzione. E una grammatica **regolare da una context-free?** Tramite il **self-embedding!**

Self-embedding

Se una grammatica G contiene un simbolo **non terminale** A tale che

$$A \rightarrow^* \alpha_1\ A\ \alpha_2 \text{ (con } \alpha_1 \text{ e } \alpha_2 \text{ appartenenti a } V^+)$$

si dice che A è **autoinclusiva** (**self-embedded**) e la grammatica G si dice **contenere self-embedding.**

Teorema

UNA GRAMMATICA CONTEXT-FREE CHE NON CONTENGA SELF-EMBEDDING GENERA UN LINGUAGGIO <u>REGOLARE</u>

La presenza di **self-embedding** è dunque la **caratteristica cruciale** che differenzia le grammatiche di tipo 2 da quelle di tipo 3. Il ruolo del self-embedding è di introdurre una ricorsione in cui si aggiungono **contemporaneamente** simboli a **sinistra** e a **destra.** È quindi essenziale per definire linguaggi le cui frasi devono avere simboli bilanciati (esempio: parentesi tonde, graffe, ecc…)

Ad esempio:

$$S \longrightarrow \mathbf{(}S\mathbf{),}\ S \longrightarrow \mathbf{a}$$

genera un linguaggio L(G) = { $(^n$ **a** $)^n$ n ≥ 0 }

A volte, tuttavia, capita che , nonostante G contenga self-embedding, L(G) sia regolare, perché la regola con self-embedding è “disattivata” da altre regole più generali. In questo caso si può parlare di ***finto self-embedding***.
Esempio:

$$\mathbf{S \longrightarrow a\ S\ a\ |}\ \varepsilon$$

in questo caso L(G) è, in realtà, regolare: L(G) = { $(a\ a)^{2n}$, ≥ 0} e la grammatica di **tipo 3 alternativa** potrebbe essere scritta così:

$$\mathbf{S \longrightarrow a\ a\ S\ |}\ \varepsilon$$

Tale proprietà generalizza un teorema:

OGNI LINGUAGGIO CONTENT-FREE DI ALFABETO UNITARIO È UN LINGUAGGIO REGOLARE

Riconoscibilità dei linguaggi

I linguaggi generati da grammatiche di tipo 1 (e quindi anche di tipo 2, 3) sono **riconoscibili** (decidibili), ovvero esiste un algoritmo per decidere se una frase appartiene al linguaggio. Invece per quanto riguarda il tipo 0 questo può non essere vero.

- La sintassi di un linguaggio di programmazione è **sempre** descritta tramite grammatiche content-free (tipo 2), perché ciò assicura che il traduttore possa essere realizzato in modo efficiente —> (**parser**).
- Alcune sottoparti (numeri, identificatori) sono descritte con grammatiche **regolari** (tipo 3) —> **scanner**

Grammatiche	Automi riconoscitori
Tipo 0	Se L(G) è riconoscibile —> **Macchina di Turing**
Tipo 1	**Macchina di Turing** (con nastro proporzionale alla frase da riconoscere
Tipo 2	**Push-Down Automaton** (PDA) = ASF + stack
Tipo 3	**Automa a Stati Finiti** (ASF)

B.N.F. ed E.B.N.F

Esempio 1

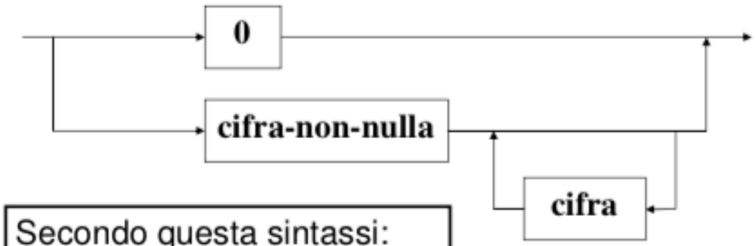
```
G = ⟨VT,VN,P,S⟩,   dove:
VT = { il, gatto, topo, sasso, mangia, beve }
VN = { <frase>, <soggetto>, <verbo>,
         <compl-ogg>, <articolo>, <nome> }
S  = <frase>
P =  {
     <frase> ::= <soggetto> <verbo> <compl-ogg>
     <soggetto> ::= <articolo><nome>
     <articolo> ::= il
     <nome> ::= gatto | topo | sasso
     <verbo> ::= mangia | beve
     <compl-ogg> ::= <articolo> <nome>
}
```

E.B.N.F.
aggiunge alcune notazioni compare per <u>alleggerire</u> la scrittura delle regole di produzione

E.B.N.F	B.N.F.	Significato
X ::= [a] B	**X ::= B l aB**	**a** può comparire 0 o 1 volta
X ::= {a}^{n} B	**X ::= B l aB l … l a^{n}B**	**a** può comparire da 0 a *n* volte
X ::= {a} B	**X ::= B l aX**	**a** può comparire 0 o più volte

Esempio numeri naturali

```
G = ⟨VT,VN,P,S⟩
dove:
VT = { 0,1,2,3,4,5,6,7,8,9 }
VN = { <num>, <cifra>, <cifra-non-nulla> }
S = <num>
                                          EBNF
P =  {
     <num> ::= <cifra> | <cifra-non-nulla> {<cifra>}
     <cifra> ::= 0 | <cifra-non-nulla>
     <cifra-non-nulla> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
}
```

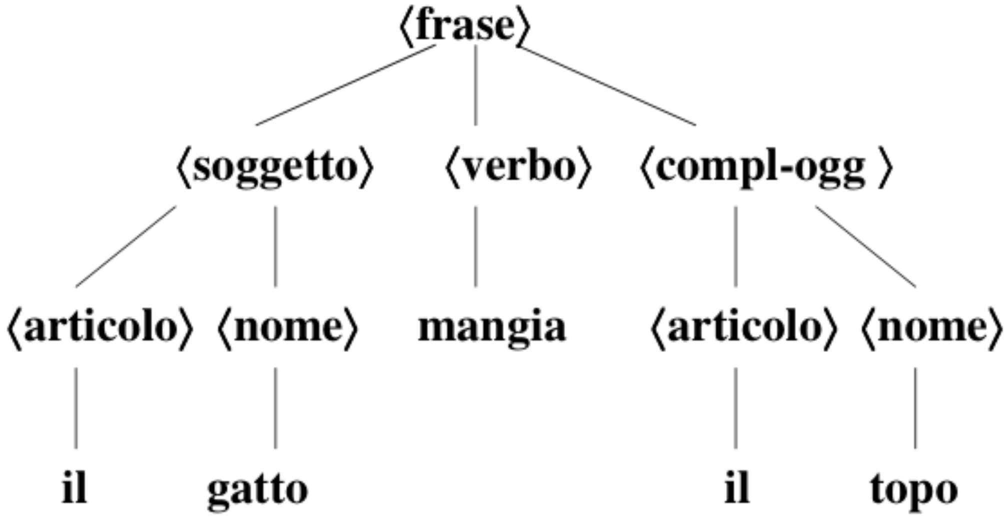


Secondo questa sintassi:
- 0 → sì
- 131 → sì
- 013 → NO

Albero di derivazione

Se G è una grammatica context-free si può introdurre il concetto di **albero di derivazione**.
- La **radice** dell'albero corrisponde allo **scopo**
- ogni **nodo** dell'albero è associato ad un simbolo del vocabolario (V = VT ∪ VN)

Riprendendo l'esempio 1 avremmo un albero di questo tipo:



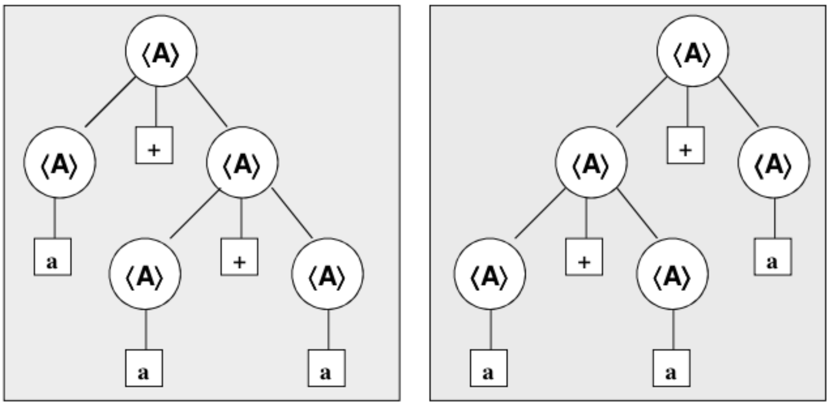
In caso di utilizzo di notazione E.B.N.F. è necessario riscrivere le produzioni in B.N.F. prima di mapparle su un albero di derivazione.

Derivazioni canoniche

- **LEFT-MOST:** derivazione canonica a sinistra
  - A partire dallo scopo della grammatica, **si riscrive sempre il simbolo non-terminale più a sinistra.**
- **RIGHT-MOST:** derivazione canonica a destra
  - A partire dallo scopo della grammatica, **si riscrive sempre il simbolo non-terminale più a destra.**

Grammatiche ambigue

Due grammatiche sono **ambigue** se esiste almeno una frase di L(G) che ammette due o più derivazioni canoniche sinistre distinte, oppure per la quale esistono almeno due alberi sintattici distinti:



Il **grado di ambiguità** è definito dal **numero di alberi sintattici** distinti. Stabilire se una grammatica context-free G è ambigua è un problema **indecidibile.**
Un linguaggio si dice ***inerentemente*** **ambiguo** se tutte le grammatiche che lo generano sono ambigue.

L'inserimento della stringa vuota

Pur non essendo possibile includere la stringa vuota in grammatiche di tipo 1, 2, 3, molto spesso fa comodo avere la stringa vuota ε all'interno del linguaggio pur non dovendo cambiare il tipo di grammatica. Questo si può ottenere **introducendo** la stringa vuota **solo a livello di scopo della grammatica —> in questo modo le stringhe NON possono accorciarsi se non al primo passo di derivazione.**

Forme normali

Un linguaggio context-free non vuoto **può sempre essere generato da una grammatica context-free G tale che:**
- **ogni simbolo, terminale o non terminale, compaia almeno una volta in una frase di L (*Non esistono simboli o meta-simboli inutili*)**
- **non ci sono produzioni della forma A —> B (*Non esistono produzioni che cambiano solo li nome*)**
- **Se il linguaggio non comprende la stringa vuota allora non ci sono produzioni della forma A —>** ε

Forme normali di Chomsky

Forme normali particolari che prevedono solo produzioni del tipo

$$A \rightarrow B\ C \mid a$$

con A, B, C $\in$VN, a$\in$VT $\cup$ ε

Esiste un algoritmo che può trasformare ogni grammatica di tipo 2 in forma di Chomsky

Forme normali di Greibach

Forme normali particolari che prevedono solo produzioni del tipo

$$A \rightarrow a\ \alpha$$

con A$\in$VN, a$\in$VT, $\alpha\in$VN*

La forma normale di Greibach facilità la costruzione di riconoscitori.

Trasformazioni importanti

Vengono utilizzate per rendere le regole di produzione più adatte allo scopo. Le regole più importanti sono:

- **Sostituzione:** Consiste nell'espandere un simbolo non terminale che compare nella parte destra di una regola di produzione, sfruttando un'altra regola di produzione.
- **Fattorizzazione:** Consiste nell'isolare il prefisso più lungo comune a due produzioni
- **Eliminazione della ricorsione sinistra:** trasformazione *sempre possibile* che avviene in 2 passi:
  - Eliminazione dei cicli ricorsivi a sinistra
  - Eliminazione della ricorsione sinistra diretta

Pumping Lemma

Lo scopo del Pumping Lemma (Lemma del pompaggio) è **capire se un linguaggio è context-free.** Quindi il pumping lemma è una **condizione necessaria (non sufficiente) perché un linguaggio sia context-free/regolare.**
L'idea di fondo è che in un linguaggio infinito**, ogni stringa deve avere una parte che si ripete e che, come tale, può essere "*pompata*"** un qualunque numero di volte.

Per linguaggi **context-free** il pumping lemma è definito come segue:

SE L È UN LINGUAGGIO CONTEXT-FREE, ESISTE UN INTERO N TALE CHE, PER OGNI STRINGA z DI LUNGHEZZA PARI A N:
- z PUÒ ESSERE RISCRITTA COME Z = uvwxy
- LA PARTE CENTRALE vwx HA LUNGHEZZA LIMITATA
- v E x NON SONO ENTRAMBI NULLE
- TUTTE LE STRINGHE DELLA FORMA uv^{i}wx^{i}y APPARTENGONO A L

In pratica:
- le due sottostringhe v e x possono essere “pompate” quanto si vuole ottenendo sempre stringhe di L
- il numero N dipende caso per caso dal linguaggio specifico.

Per linguaggi **regolari**, invece è definito come segue:

SE L È UN LINGUAGGIO CONTEXT-FREE, ESISTE UN INTERO M TALE CHE, PER OGNI STRINGA z DI LUNGHEZZA PARI A M:
- z PUÒ ESSERE RISCRITTA COME Z = xyw
- LA PARTE CENTRALE xy HA LUNGHEZZA LIMITATA
- y NON È NULLA
- TUTTE LE STRINGHE DELLA FORMA xy^iw ***APPARTENGONO A L***

In pratica:
- la sottostringa y può essere “pompata” quanto si vuole ottenendo sempre stringhe di L
- il numero M dipende caso per caso dal linguaggio specifico.

Espressioni regolari

Le espressioni regolari sono tutte e sole le espressioni costruibili tramite le seguenti regole:
- **la stringa vuota ε è un’espressione regolare**
- dato un alfabeto A, ogni elemento di A è un espressione regolare
- Se X ed Y sono espressioni regolari lo sono anche:
  - **X+Y (unione)**, *operatore meno prioritario*
    - $X + Y = \{x \mid x \in X, x \in Y\}$
  - **X•Y(concatenazione)**
    - $X \cdot Y = \{x \mid x = a\ b, a \in X, b \in Y\}$
    - $\{\ \} \cdot X = \{\ \}$ per qualsiasi x
  - **X* (chiusura)**, *operatore più prioritario*
    - $X^* = X^0 \cup X^1 \cup X^2 \cup \ldots$ dove $X^0 = \varepsilon$ e $X^k = X^{k-1} \cdot X$

Teorema

L’INSIEME DEI LINGUAGGI GENERATI DA GRAMMATICHE DI TIPO 3 COINCIDE CON L’INSIEME DEI LINGUAGGI DESCRITTI DA ESPRESSIONI REGOLARI.

Per passare dalla grammatica all’espressione regolare si interpretano le produzioni come “equazioni” in cui:
- i simboli terminali sono i termini noti
- linguaggi generati da ogni simbolo non terminale sono le incognite.

Esempio:

ESEMPIO: la grammatica lineare a destra vista in precedenza:

$$S \to a \mid a + S \mid a - S$$

può essere letta come un'equazione con

- **tre termini noti:** a, +, -
- **una incognita,** L_S

che impone il vincolo *(usiamo per l'unione il simbolo $\cup$ anziché +)*

$$L_S = a \cup (a + L_S) \cup (a - L_S) = (a + \cup a -) L_S \cup a$$

la cui soluzione è l'espressione regolare

$$S = (a + \cup a -)^* a$$

Soluzioni di equazioni sintattiche

In generale esiste un algoritmo per risolvere le equazioni sintattiche derivate da grammatiche **lineari** e calcolare il corrispondente linguaggio regolare. Tale algoritmo esiste in due versioni distinte per:

- **grammatiche lineari a destra**
- **grammatiche lineari a sinistra**

Algoritmo per grammatiche lineari a destra

- Riscrivere ogni gruppo di produzioni del tipo X —> $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ come X = $\alpha_1 + \alpha_2 + \dots + \alpha_n$
- Poiché la grammatica è **lineare a destra,** ogni α_k ha la forma $\mathbf{uX_k}$ (dove $X \in VN \cup \varepsilon$, $u \in VT^*$)
  - Quindi **si raccolgono a destra** i simboli non-terminali dei vari $\alpha_1 \dots \alpha_n$ scrivendo $X = (u_1 + u_2 + \dots)X_1 \cup \dots \cup (z_1 + z_2 + \dots)X_n$
  - Ciò porta a un sistema di M equazioni in M incognite dove M è la cardinalità dell'alfabeto VN
- **Eliminare** dalle equazioni le ricorsione dirette mediante:
  - $\mathbf{X = uX \cup \delta \longrightarrow X = (u)^* \delta}$
- **Risolvere il sistema** rispetto a S per eliminazioni successive (metodo di Gauss)
- La soluzione del sistema è il **linguaggio regolare cercato**.

Algoritmo per grammatiche lineari a sinistra

- Riscrivere ogni gruppo di produzioni del tipo X —> $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ come X = $\alpha_1 + \alpha_2 + \dots + \alpha_n$
- Poiché la grammatica è **lineare a sinistra,** ogni α_k ha la forma $\mathbf{X_k u}$ (dove $X \in VN \cup \varepsilon$, $u \in VT^*$)
  - Quindi **si raccolgono a sinistra** i simboli non-terminali dei vari $\alpha_1 \dots \alpha_n$ scrivendo $X = X_1(u_1 + u_2 + \dots) \cup \dots \cup X_n(z_1 + z_2 + \dots)$
  - Ciò porta a un sistema di M equazioni in M incognite dove M è la cardinalità dell'alfabeto VN
- **Eliminare** dalle equazioni le ricorsione dirette mediante
  - $\mathbf{X = uX \cup \delta \longrightarrow X = (u)^* \delta}$
- **Risolvere il sistema** rispetto a S per eliminazioni successive (metodo di Gauss)
- La soluzione del sistema è il linguaggio regolare cercato.

Esempio:

Fase 1 • **scrittura di un'equazione per ogni regola:**	Grammatica data: $S \to a\ B \mid a\ S$ $B \to d\ S \mid b$
Fase 2 • **eventuali raccoglimenti a fattore comune per evidenziare suffissi:** ***qui non ce ne sono***	Equazioni: $S = a\ B + a\ S$ $B = d\ S + b$
Fase 3 • **eliminare la ricorsione diretta** $X = u\ X + \delta$ **riscrivendola come** $X = u^*\ \delta$ **(qui** $\delta = a\ B$**)**	$S = a^*\ a\ B$ $B = d\ S + b$
Fase 4 • **sostituzione della 2ª equazione nella 1ª e sviluppo dei relativi calcoli**	$S = a^*\ a\ (d\ S + b) =$ $= a^*\ a\ d\ S + a^*\ a\ b$
Fase 5 • **nuova eliminazione della ricorsione introdotta al punto precedente: risultato finale.**	$S = a^*\ a\ d\ S + a^*\ a\ b$ $S = (a^*\ a\ d)^*\ a^*\ a\ b$

Notare che uno stesso linguaggio **può essere denotato da più espressioni regolari equivalenti tra di loro**.

Automi riconoscitori

Grammatiche	Automi riconoscitori
Tipo 0	Se L(G) è riconoscibile —> **Macchina di Turing**
Tipo 1	**Macchina di Turing** (con nastro proporzionale alla frase da riconoscere
Tipo 2	**Push-Down Automaton** (PDA) = ASF + stack
Tipo 3	**Automa a Stati Finiti** (ASF)

Riconoscitore a stati finiti

Un linguaggio regolare (tipo 3) p riconoscibile da un Automa a Stati Finiti (ASF) che è definito cos dalla quadrupla:

<I,O, S, mfn, sfn>

- **I** = insieme dei **simboli in ingresso**
- **O** = insieme dei **simboli di uscita**
- **S** = insieme degli stati
- **mfn**: I x S —> O = *machine function*
- **sfn**: I x S —> S = *state function*

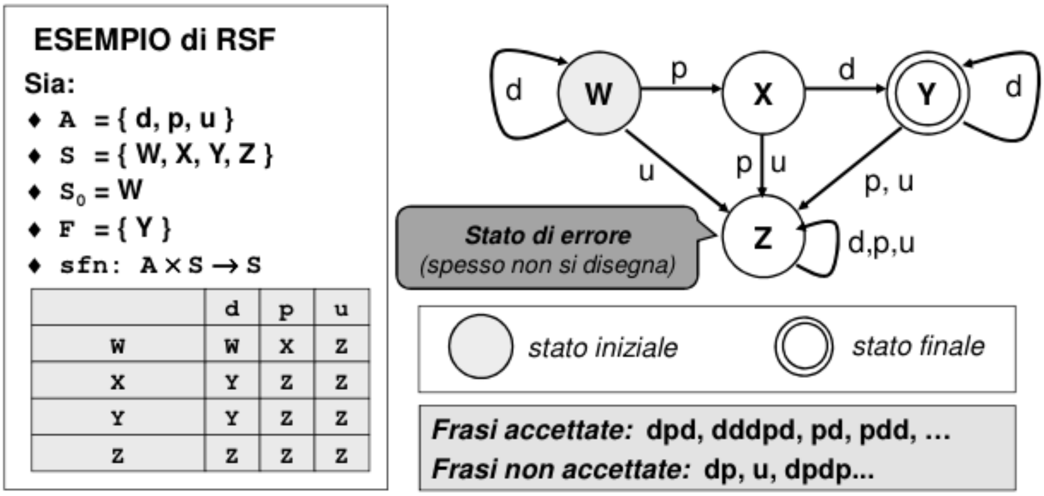
Un Riconoscitore a stati finiti (RSF) è una *specializzazione* di ASF:

<A, S, S_0, F, sfn*>

- **A = alfabeto** (A* = chiusura)
- **S** = **stati**
- **S_0** = **stato iniziale** ∈S
- **F** = insieme degli **stati finali** ⊆ S
- **sfn***: A* x S —> S = *state function*

La funzione sfn*:
- definisce l'evoluzione dell'automa a partire dallo stato iniziale S_0 in corrispondenza di ogni sequenza di ingresso x∈A.
- è definita in termini della funzione sfn: A x S —> S. Tratta quindi una **sequenza di simboli** (stringa) di A.
- data la stringa **x a**, si considera prima la sottostringa **x**, poi si applica la funzione sfn al simbolo **a** e allo stato **s' = sfn*(x,s)**.

Esempio:



Il linguaggio L(R) accettato dal riconoscitore R è infinito se la rappresentazione grafica presenta cicli.

Teoremi

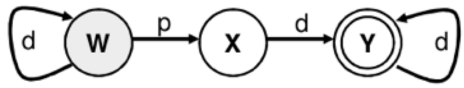
- **TEOREMA 1: Un linguaggio L(R) è *non vuoto* se e solo se il riconoscitore R accetta una stringa x di lunghezza L_x minore del numero di stati N dell'automa.**
  - **Dimostrazione**
- **TEOREMA 2: Un linguaggio L(R) è infinito se e solo se il riconoscitore R accetta una stringa x di lunghezza N ≤ L_x ≤ 2N, (con N numero di stati dell'automa)**

In conseguenza di questi due teoremi, decidere se un linguaggio **regolare** L(R) è vuoto o infinito è un problema **risolubile:**
- Nel primo caso, basta esaminare se esiste una stringa accettata di lunghezza minore di N

- Nel secondo caso, basta verificare se esiste una stringa accettata tra quelle di lunghezza compresa fra N (incluso) e 2N (escluso).

Dai riconoscitori alle grammatiche

Esempio:



Grammatica regolare a <u>destra</u>
- *scopo = stato iniziale: W*
- *stato finale: Y*

```
W → d W | p X
X → d   | d Y
Y → d   | d Y
```

Grammatica regolare a <u>sinistra</u>
- *scopo = stato finale: Y*
- *stato iniziale: W*

```
Y → X d | Y d
X → p   | W p
W → d   | W d
```

$L = d^* p\, d\, d^*$

Il mapping fra automa e grammatica presenta alcuni **gradi di libertà:**
- Se la grammatica è regolare a destra, si ottiene un automa riconoscitore "top down"
- Se la grammatica è regolare a sinistra, si ottiene un automa riconoscitore "bottom up"

Riconoscitori top down

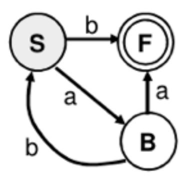
Data una grammatica regolare lineare <u>a destra</u>, il riconoscitore
- ha tanti stati quanti simboli non terminali
- ha come **stato iniziale** lo **scopo s**
- per ogni regola del tipo **X —> xY** l'automa con ingresso x **si porta dallo stato X allo stato Y**
- per ogni regola del tipo **X —> x** l'automa con ingresso x **si porta dallo stato X allo stato finale F**

ESEMPIO

Sia G una grammatica lineare a destra caratterizzata dalle produzioni:

```
S → a B | b
B → b S | a
```

Automa top-down corrispondente:



Riconoscitori bottom up

Data una grammatica regolare lineare <u>a sinistra</u>, il riconoscitore

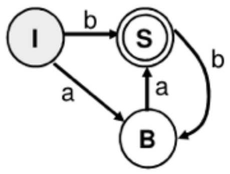
- ha tanti stati quanti simboli non terminali
- ha come **stato finale** lo scopo S
- per ogni regola del tipo **X —> Yx** l'automa con ingresso x, **riduce lo stato Y allo stato X**
- per ogni regola del tipo **X —> x** l'automa con ingresso x, **riduce lo stato iniziale allo stato X**

ESEMPIO

Sia G una grammatica lineare a sinistra con le produzioni:

$$S \to B\,a \mid b$$
$$B \to S\,b \mid a$$

Automa bottom-up relativo:



Dall'automa alle grammatiche

Dato un automa riconoscitore, se ne possono trarre:

- una grammatica **regolare a destra**, interpretandolo **top-down**
- una grammatica **regolare a sinistra**, interpretandolo **bottom-up**

(Vedi esempi pag 4 delle slides)

Nel caso di più stati finali in analisi bottom-up si può considerare ogni stato finale come scopo di una diversa grammatica ed unire le grammatiche alla fine.

Implementazione di RSF deterministici

Un riconoscitore a stati finiti *deterministico* è facilmente realizzabile in un linguaggio imperativo. Possibili soluzioni:

- Possibilità 1: **ciclo while** con **if annidati —> schifo**
- Possibilità 2: **ciclo while** con **switch —> poco meno schifo**
- Possibilità 3: **ciclo while** con **<u>tabella separata</u> —> automa non più cablato nel codice**

Riconoscitori non deterministici

Certe grammatiche possono portare a un automa *non deterministico*, cioè nella cui tabella delle transizioni compaiono più stati futuri per una stessa configurazione

In questo caso l'automa dev'essere *intrinsecamente* in grado di scegliere in quale stato portarsi, quando ha più alternative

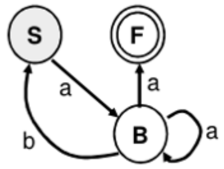
ESEMPIO

Sia G una grammatica lineare a destra caratterizzata dalle produzioni:

`S → a B`

`B → a B | b S | a`

Automa risultante:



In corrispondenza dello stato `B`, l'automa, con ingresso `a`, "sceglie" se portarsi in `B` o in `F`.

La corrispondente **tabella di transizioni** (**completata con uno** *stato di errore E* in cui si transita in presenza di ingressi non previsti) è la seguente:

	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

È un automa *non deterministico:*
per ogni frase di L(G) esiste *<u>almeno una computazione</u>* che porta l'automa dallo stato iniziale S allo stato finale F.

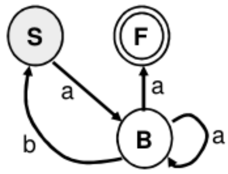
Nell'automa a lato:

- la frase `abaa` è riconosciuta dalla sequenza di transizioni **S → B → S → B → F**
- la frase `abaaa` è riconosciuta invece dalla sequenza **S → B → S → B → B → F**

Nel primo caso, dallo stato `B` con ingresso `a` l'automa deve "scegliere" di portarsi in `F`, mentre nel secondo caso deve "scegliere" di restare in `B` (la prima volta) e successivamente di portarsi in `F` (la seconda volta).

- la frase **`aaba` è invece sbagliata** e non viene riconosciuta (**S → B → B → S → B**)

Esempio:



	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

Se il linguaggio supporta il non determinismo (vedi **prolog**), fare il riconoscitore è molto semplice. In **prolog** infatti, ogni **produzione** diventa una **regola.** La macchina virtuale del linguaggio è in grado di *tentare* la strada e tornare indietro nel caso si riveli sbagliata, provando via via tutte le possibili alternative.

Esempio programma prolog che modella l'esempio sopra descritto:

```
regolaS([a|B]) :- regolaB(B).
regolaB([a|B]) :- regolaB(B).
regolaB([b|S]) :- regolaS(S).
        regolaB([a]).
```

Se il linguaggio utilizzato è, invece, un linguaggio imperativo il riconoscitore non-deterministico:

- è più <u>inefficiente</u>
- deve disporre di strutture dati interne per "ricordare la strada" e poterla **disfare** se necessario per esplorarne un'altra
- occorre ricostruire le capacità del motore prolog.

Da automi non deterministici ad automi deterministici

Teorema:

UN AUTOMA NON DETERMINISTICO PUÒ SEMPRE ESSERE RICONDOTTO AD UN AUTOMA DETERMINISTICO EQUIVALENTE

Procedimento:

- Si definisce un automa i cui stati corrispondono a dei **set di stati** dell'automa originale
- Si costruisce la tabella delle transizione del nuovo automa aggiungendo righe.

	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

	a	b
[S]	[B]	[E]
[B]	[B,F]	[S]
[B,F]	[B,F,E]	[S,E]
[S,E]	[B,E]	[E]
[B,E]	[B,F,E]	[S,E]
[B,F,E]	[B,F,E]	[S,E]
[E]	[E]	[E]

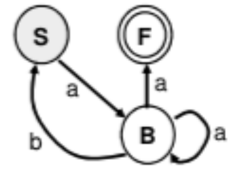
Rinominando gli stati:

	a	b
S0	S1	S6
S1	S2	S0
S2	S5	S3
S3	S4	S6
S4	S5	S3
S5	S5	S3
S6	S6	S6

Minimizzando:

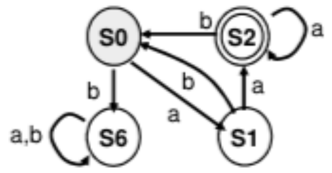
S1	S1/S2,S0/S6					
S2	X	X				
S3	S1/S4	S2/S4,S0/S6	X			
S4	S1/S5,S3/S6	S2/S5,S0/S3	X	S4/S5,S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6,S0/S6	X	S4/S6	S5/S6,S3/S6	X
	S0	S1	S2	S3	S4	S5

Automa non deterministico:



	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

Automa deterministico minimo:



	a	b
S0	S1	S6
S1	S2	S0
S2	S2	S0
S6	S6	S6

Una volta terminata la semplificazione si può implementare il nuovo automa **sia** in linguaggi **imperativi**, sia, più rapidamente, in linguaggi **dichiarativi** (prolog)

Espressioni e linguaggi regolari

Teorema:

L'INSIEME DI LINGUAGGI RICONOSCIUTI DA UN ASF COINCIDE CON L'INSIEME DEI LINGUAGGI REGOLARI, OSSIA QUELLI DESCRITTI DA ESPRESSIONI REGOLARI.

Riconoscitori per grammatiche Context-Free

Consideriamo ora solo il caso delle grammatiche di **tipo 2.** Il riconoscitore può essere definito da il **push-down automata** (PDA). Infatti un linguaggio context-free non può essere riconosciuto solo da un RSF.

Esempio:
Sia A = {0, 1, c} un alfabeto con S —> 0 S 0 l 1 S 1 l c.
Il linguaggio generato è L = {word c word^R}, dove word^R indica il ribaltamento della stringa word e word a sua volta indica tutte le possibili sequenze di 0 e 1, inclusa la stringa vuota ε. ***Questo linguaggio non è riconoscibile da un RSF perché occorre memorizzare la stringa word, la cui lunghezza non è limitata a priori.***

Per questo motivo è necessario introdurre uno **stack.**

PDA (Push-down automota)

Lo **stack** è formalmente definito come una **sequenza di simboli**, quello più a destra è in cima alla pila. Come ASF, PDA legge un simbolo d'ingresso e transita in un nuovo stato; in più produce una nuova **configurazione** dello stack, **in funzione del simbolo d'ingresso e di quello in cima allo stack.**

Possiamo quindi **definire** il PDA come una sestupla:

<A, S, S_0, sfn, Z, Z_0>

- **A = alfabeto** (A* = chiusura)
- **S = insieme degli stati**
- **S_0 = stato iniziale** $\in$S
- **sfn =** (A∪ℇ) x S x Z —> Q con Q sottoinsieme finito di S x Z*
- **Z** = **alfabeto dei simboli interni**
- **Z_0** $\in$**Z = simbolo iniziale sullo stack**

Il linguaggio accettato da un PDA è definibile in 2 modi equivalenti:
- <u>Criterio dello stack vuoto</u>: il linguaggio accettato è il set delle stringhe tali che una sequenza di mosse porta il PDA della **configurazione di stack vuoto.**
- <u>Criterio dello stato finale</u>: il linguaggio accettato è l'insieme di tutte le stringhe di ingresso per cui esiste una sequenza di mosse che porta il PDA in **uno degli stati finali.**

Inoltre, la funzione **sfn,** dati:
- un simbolo di ingresso **a**
- lo stato attuale **s**
- il simbolo interno attualmente in cima allo stack

opera così:
- **consuma** il simbolo in ingresso **a**
- **elimina dallo stack** (pop) il simbolo interno attualmente al top, **z**
- **fa transitare l'automa** nello stato futuro specificato, **s'**
- **pone in cima allo stack** (push) i simboli interni specificati, **Z**

Esempio

Sia A = {0, 1, c} un alfabeto con S —> 0 S 0 | 1 S 1 | c.
Definiamo il PDA come segue:

$\langle$A, S, S_0, sfn, Z, $Z_0\rangle$

A = { 0, 1, c }
S = { Q1, Q2 }, S_0 = Q1
Z = { Z, U, C }, Z_0 = C
sfn: $(A \cup \varepsilon) \times S \times Z \rightarrow S \times Z^*$

$A \cup \varepsilon$	S	Z	$S \times Z^*$
0	Q1	C	Q1 × CZ
1	Q1	C	Q1 × CU
c	Q1	C	Q2 × C
0	Q1	Z	Q1 × ZZ
1	Q1	Z	Q1 × ZU
c	Q1	Z	Q2 × Z
0	Q1	U	Q1 × UZ
1	Q1	U	Q1 × UU
c	Q1	U	Q2 × U
0	Q2	Z	Q2 × ε
1	Q2	U	Q2 × ε
ε	Q2	C	Q2 × ε

Funzionamento:
- All'inizio lo stack contiene il simbolo C
-Si iniziano a consumare i simboli di ingresso e si opera come da tabella
-Alla fine, se la stringa è stata riconosciuta, lo stack contiene solo il simbolo C iniziale, che viene infine consumato con una ℇ-mossa

PDA non deterministici

Anche un PDA può non essere deterministico: in tal caso la funzione sfn produce insiemi di elementi di Q. Il **non determinismo** dell'automa può emergere sotto **due aspetti:**

- L'automa, in un certo stato Q_0, con simbolo interno in cima allo stack z, e con ingresso x**, può portarsi in uno qualunque degli stati futuri:** ***sfn(Q_0, x, z,) = {(Q_1, Z_1), (Q_2, Z_2), … (Q_k, Z_k)}***
- L'automa, in un certo stato Q_0, con simbolo interno in cima allo stack z, e con ingresso x, può decidere di **leggere** o **non leggere** il simbolo di ingresso x. Ciò accade se sono definite **entrambe le mosse:** ***sfn(Q_i, x, z)*** **e** ***sfn(Q_i, ε, z)*** di cui la seconda mossa è una ε-mossa.

Teorema

LA CLASSE DEI LINGUAGGI RICONOSCIUTI DA UN PDA (NON DETERMINISTICO) COINCIDE CON LA CLASSE DEI LINGUAGGI CONTEXT-FREE: PERCIÒ QUALUNQUE LINGUAGGIO CONTEXT-FREE PUÒ ESSERE SEMPRE RICONOSCIUTO DA UN OPPORTUNO PDA

Il problema è che il calcolo del PDA non-deterministico può essere esponenziale. Infatti la complessità degli algoritmi per risolvere i PDA è *sovra-lineare.*

Teorema

ESISTONO LINGUAGGI CONTEXT-FREE RICONOSCIBILI SOLTANTO DA PDA NON-DETERMINISTICI.

Tuttavia:

ESISTE UNA CLASSE DI LINGUAGGI CONTEXT-FREE RICONOSCIBILI DA PDA DETERMINISTICI

e in tal caso la complessità di calcolo del PDA deterministico è *lineare* rispetto alla lunghezza della stringa da riconoscere.

PDA deterministici

Perché l'automa sia **deterministico**, occorre evitare che l'automa, in un certo stato Q_0, con simbolo in cima allo stack z e con ingresso x, possa:

- portarsi in uno qualunque degli stati futuri: *sfn(Q_0, x, Z) = {(Q_1, Z_1), (Q_2, Z_2), … (Q_k, Z_k)}*
- decidere di leggere o non leggere il simbolo in ingresso x grazie alla presenza di una ε-mossa.

Proprietà di PDA deterministici

- **L'unione**, **l'intersezione** e il **concatenamento** di due linguaggi deterministici in generale **non danno luogo a un linguaggio deterministico.**
- Il **complemento** di un linguaggio deterministico **è deterministico**
- Se L è un linguaggio deterministico e R u linguaggio regolare, il linguaggio **quoziente L/R** (ossia l'insieme delle stringhe di L private di un suffisso regolare) **è deterministico**

- Se L è un linguaggio deterministico e R un linguaggio regolare, il **concatenamento L·R** (ossia l'insieme delle stringhe di L con un suffisso regolare) **è deterministico**

Passando da PDA non deterministico a PDA deterministico vengono meno alcune proprietà. Ovvero:
- per un PDA **deterministico**, il riconoscimento con il criterio dello *stack voto* risulta **meno potente** del riconoscimento con il criterio degli *stati finali*
- per un PDA **deterministico**, una limitazione sul numero di stati interni o sul numero di configurazioni finali **riduce** l'insieme dei linguaggi riconoscibili
- per un PDA **deterministico**, l'assenza di ε-mosse **riduce** l'insieme dei linguaggi riconoscibili

Implementazione di PDA deterministici

Molto efficiente utilizzando l'**analisi ricorsiva discendente (analisi top-down)**
- Si introducono tante procedure o funzioni quanti i simboli **non terminali**
- si fa in modo che ognuna di tal funzioni **riconosca** il sotto linguaggio generato dal simbolo non terminale associato.

Esempio:

Il "solito" linguaggio *context-free*:
Alfabeto: A = { **0, 1, c** }   Produzioni: $S \rightarrow 0\,S\,0 \mid 1\,S\,1 \mid c$
Linguaggio: L = { *word* **c** *word*^{R} }

La variabile `first` è <u>locale</u>, quindi allocata nel record di attivazione → l'insieme di tutte le `first` costituisce lo stack del PDA

```
char ch;
boolean S(){
 char first;
 boolean result;
 ch = nextchar();              /* recupera il prossimo carattere di ingresso */
 switch (ch){
  case 'c':  ch = nextchar(); result = true; break;
  case '0':
  case '1':  first = ch;                                  /* push */
             if (S())if(ch==first) {                      /* pop */
                     ch = nextchar(); result = true;
                   }
                   else result = false;
             else result = false;
  default:   result = false;
 }
 return result;
}
```

Siccome l'**analisi ricorsiva discendente** è un processo meccanico, non ha senso "cablare nel codice" questo comportamento. È opportuno **separare il motore e il parser** dalla **descrizione delle regole**. Per farlo si costruisce una t**abella di parsing:**
- È simile alla tabella delle transizioni dell'automa a stati
- Il suo scopo è indicare **la prossima produzione da applicare**.

Il "solito" linguaggio *context-free*: L = { **word c word^R** }
Produzioni: `S → 0 S 0 | 1 S 1 | c`

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

L = { **if c then cmd (endif | else cmd)** }
Produzioni:
S → if c then cmd X   X → endif | else cmd

	if	c	then	endif	else	cmd
S	S → if c then cmd X	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	*error*	X → endif	X → else cmd	*error*

Grammatiche LL(k)

Detto A un meta-simbolo, rendere *deterministica* l'analisi sintattica di linguaggi **context-free** significa realizzare PDA capaci di fare sempre la "mossa giusta" per riconoscere la frase di input. La classe di grammatiche LL(k) è caratterizzata dalla possibilità di analizzare le frasi **left-to-right** applicando la **left-most-derivation** (derivazione canonica a sinistra) utilizzando al più **k simboli** della frase per scegliere con certezza la produzione opportuna per la riscrittura. È ovviamente di particolare importanza la classe di grammatiche **LL(1)** in cui basta *un solo simbolo* della frase per scegliere con certezza la produzione opportuna.

Esempio:

Si consideri la grammatica:
VT = { p, q, a, b, d, x, y }
VN = { S *(scopo)*, X, Y }
Produzioni:
S → p X | q Y
X → a X b | x
Y → a Y d | y

- **Le parti destre delle produzioni *di uno stesso meta-simbolo* iniziano *tutte con un simbolo terminale diverso***
- **È quindi possibile impostare un *parser discendente deterministico* partendo dallo scopo S e applicando la produzione che inizia con il simbolo terminale "giusto": se una tale produzione non esiste, *Errore.***

ESEMPIO: riconoscimento della frase **paaaxbbb**

Frase	Produzione	Derivazione
paaaxbbb	S ::= pX	pX
aaaxbbb	X ::= aXb	paXb
aaxbb	X ::= aXb	paaXbb
axb	X ::= aXb	paaaXbbb
x	X ::= x	paaaxbbb
	nessuna	paaaxbbb

```
char ch;                          /* variabile globale*/

main() {
 ch = nextchar();                 /* recupera il prossimo carattere di ingresso */
 if (S()) printf("Frase accettata."); else printf("Errore");
}

boolean S() {
 switch (ch){
   case 'p':   ch = nextchar(); return X();     /* produzione S ::= pX */
   case 'q':   ch = nextchar(); return Y();     /* produzione S ::= qY */
 }
 return false;
}

boolean X() {   /* analogamente si scrive la funzione Y() per le altre produzioni */
 switch (ch){
   case 'a':   ch = nextchar();                /* produzione X ::= aXb */
               if (X()) {
                 if (ch=='b') { ch=nextchar(); return true; }
                 else return false;
               } else return false;
   case 'x':   ch = nextchar(); return true;    /* produzione X ::= x */
   return false;
 }
}
```

Con la parsing table:

Si consideri la grammatica:

VT = { p, q, a, b, d, x, y }
VN = { S *(scopo)*, X, Y }

Produzioni:

S → p X | q Y
X → a X b | x
Y → a Y d | y

Ogni cella contiene una sola produzione:
• il parser è deterministico
• la grammatica è LL(1)

	p	q	a	b	d	x	y
S	S → p X	S → q Y	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	X → a X b	*error*	*error*	X → x	*error*
Y	*error*	*error*	Y → a Y d	*error*	*error*	*error*	Y → y

- La sequenza di invocazioni alle funzioni S() e X() corrisponde a una visita *depth-first*
- Ogni attivazione di funzione consuma i simboli terminali corrispondenti alle propria produzione, e lascia in ch il primo simbolo terminale non scoperto
- Lo stack interno alla macchina virtuale C fornisce lo stack del PDA riconoscitore, necessario per controllare la corrispondenza tra le **a** e le **b** (caso X) e tra le **a** e le **d** (caso Y)
- Non è necessario memorizzare esplicitamente i simboli **a** tramite variabili locali delle funzioni X() e Y() perché sono tanti quante le attivazione di tale funzione.

Starter symbols

Anche se le parti destre delle produzioni di uno stesso meta-simbolo **NON** iniziano tutte con un simbolo terminale, può ancora essere possibile scegliere la produzione da usare in base al simbolo di ingresso attuale. È evidente che è importante solo il simbolo iniziale.
Quindi:
Se A è un **meta-simbolo** e α, β due **stringhe** di terminali e non terminali, definiamo

- starter simbols del non terminale A l'insieme

$$\mathbf{SS(A) = \{ a \in VT \mid A \rightarrow^{+} a\beta \}} \text{ con } \beta \in V^*$$

- starter symbols della riscrittura α come l'insieme:

$$\mathbf{SS(\alpha) = \{ a \in VT \mid \alpha \rightarrow^{*} a\beta \}} \text{ con } \alpha \in V^+ \text{ e } \beta \in V^*$$

Condizione *necessaria* affinché una grammatica sia LL(1) è che *per ogni meta-simbolo* che appare nella parte più a sinistra di più produzioni, gli starter symbols corrispondenti alle parti destre delle produzioni alternativi siano disgiunti. Perché la condizione sia anche ***sufficiente*** è necessario anche che nessuno dei meta-simboli possa generare la stringa vuota.

Starter symbols: il problema della stringa vuota

ESEMPIO: Grammatica G

Produzioni senza ε -rules:

$S \rightarrow A\ B \mid B$
$A \rightarrow a\ A$
$B \rightarrow b\ B \mid c$

Produzioni **con** ε -rules:

$S \rightarrow A\ B \mid B$
$A \rightarrow a\ A \mid \varepsilon$
$B \rightarrow b\ B \mid c$

Nel primo esempio abbiamo SS(AB) = {a}, SS(B) = {b, c} —> Insiemi disgiunti —> Grammatica LL(1)
Nel secondo caso invece avremmo: SS'(AB) = {a, b, c}, SS'(B) = {b, c} —> insiemi non disgiunti —> grammatica non LL(1).
Di conseguenza non si dovrebbero usare Ɛ-rules se non sullo scopo S.
Come eliminare la stringa vuota? Si utilizza il teorema della grammatica equivalente visto in precedenza.

Director Symbols

Definiamo director symbols della produzione A —>α l'insieme degli **starter-symbols** e dei **following symbols**:

$$DS(A \rightarrow \alpha) = SS(\alpha) \cup FOLLOW(A)$$

dove **FOLLOW(A)** denota l'insieme dei simboli che, **nel casi A generi** ε, possono seguire la frase generata da A. In pratica:

In pratica,

$$DS(A \rightarrow \alpha) = \begin{cases} SS(\alpha) & \textbf{se } \alpha \textbf{ non genera mai } \varepsilon \\ SS(\alpha) \cup FOLLOW(A) & \textbf{se } \alpha \textbf{ può generare } \varepsilon \end{cases}$$

Condizione <u>necessaria e sufficiente</u> perché una grammatica context-free sia LL(1) è che **per ogni meta-simbolo che appare nella parte sinistra di più produzioni, i director symbols relativi a produzioni alternative siano <u>disgiunti</u>.**
(Grammatiche con ricorsioni sinistre non sono <u>mai</u> LL(1))

TEOREMI:
- Stabilire se una grammatica G sia LL(1) è un problema <u>decidibile</u>
- Stabilire se un linguaggio L sia LL(1) è un problema <u>indecidibile</u>
- Non tutti i linguaggi context-free possiedono una grammatica LL(1)
- Se L è un linguaggio context-free ma il suo complemento non lo è, allora L è <u>non deterministico</u>

In molti casi pratici è possibile trasformare una grammatica context-free in una grammatica LL(1) tramite due tecniche:
- **Eliminazione della ricorsione a sinistra**
- **Raccoglimento a fattor comune**

Eliminazione della ricorsione a sinistra

Come già detto, la ricorsione a sinistra è un male, ma molto spesso, è utile per inserire concetti interessanti. Può quindi essere utile cercare di rimuoverla con il seguente algoritmo:
- **Fase 1**: Introdurre una **ricorsione sinistra diretta** al posto dei **cicli ricorsivi a sinistra**
- **Fase 2**: Eliminazione della **ricorsione sinistra diretta mediante riscrittura della regola —> ricorsione destra diretta**.

ESEMPIO
di ciclo ricorsivo a sinistra

```
A → B a
B → C b
C → A c | p
```

Si ottiene quindi:

$A \rightarrow B\ a$

$B \rightarrow C\ b$

$C \rightarrow C\ b\ a\ c \mid p$

Ergo, $C \rightarrow C\ b\ a\ c \mid p$ diventa

$C \rightarrow p \mid p\ Z$

$Z \rightarrow b\ a\ c \mid b\ a\ c\ Z$

Raccoglimento a fattor comune

Questa tecnica **non sempre** consente di rendere la grammatica LL(1). La tecnica consiste nell'**isolare il prefisso più lungo comune a due produzioni**: ciò può, in certi casi, rendere la grammatica LL(1).

ESEMPIO

$S \rightarrow a\ S\ b \mid a\ S\ c$

$S \rightarrow \varepsilon$

ESEMPIO

$S \rightarrow a\ S\ X \mid \varepsilon$

$X \rightarrow b \mid c$

Limiti di grammatiche LL(k)

Caso LL(1) è il solo praticamente utilizzato. Non tutti i linguaggi context-free possiedono una grammatica LL(1). Esistono linguaggi context-free non riconoscibili in modo **deterministico** con le tecniche dell'analisi LL(k). Per risolvere esistono grammatiche più potenti—>LR(k) che analizzano le frasi **L**eft to right adottando una **R**ight-most-derivation e guardando k simboli oltre per scegliere la produzione da usare. L'analisi LR è meno "naturale" dell'analisi LL ma è superiore dal punto di vista teorico.

Interpreti

Un interprete è un sistema che:
- Accetta in ingresso una stringa di caratteri
- **riconosce** se è una frase del linguaggio
  - in caso positivo, **esegue azioni** in accordo al **significato** (semantica) della frase.

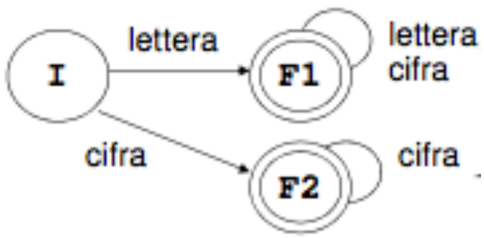
Struttura di un interprete

Solitamente strutturato su **due componenti:**
- *Analizzatore lessicale* (**scanner**)
- *Analizzatore sintattico-semantico* (**parser**)

e in un'architettura **client/server**:
- **lo scanner (server)** analizza le parti regolari del linguaggio e fornisce al parser **singole parole** (token) **già aggregate**, evitandogli di doversi occupare dei dettagli relativi ai singoli caratteri
- **il parser (client)** riceve dallo scanner le **singole parole** (token) e le usa come elementi terminali del suo linguaggio per valutare la **correttezza della sequenza**.

Analisi lessicale

Individuazione delle **singole parole** (token)



Per evitare di incapsulare nella struttura stessa del **RSF** le peculiarità del linguaggio, si utilizzano spesso alcune tabelle che descrivono i dettagli del linguaggio come:
- **Tabella delle parole chiave**
- **Tabella dei simboli speciali**

Ciò semplifica molto la struttura della RSF —> Scanner più modulare.

Analisi sintattica

Con grammatiche context-free LL(1), l'analisi top-down ricorsiva discendente si presta a costruire riconoscitori:
- *modulari*
- *espandibili*

È utile appoggiarsi ad una rappresentazione ***intermedia** di **albero sintattico***. Tuttavia un albero intero, pur rappresentando in maniera *esterna* e *intrinseca* la frase, è una struttura molto pesante da "portarsi dietro". Per questo si utilizzano delle versioni più sintetiche ed astratte chiamate **Alberi Sintattici Astratti (AST)**

Notazioni infisse, prefisse e postfisse

Prendendo in esempio le espressioni aritmetiche, noi le scriviamo utilizzando una notazione infissa: { 3+5*3-1 }. Tuttavia ciò presuppone l'esistenza del concetto di **priorità** tra gli operatori —> utilizzo di parentesi per alterare tale priorità. Tuttavia esistono altri tipi di notazione:

- Prefissa: prima l'operatore e poi gli operandi
- Postfissa: prima gli operandi e poi l'operatore

Entrambe le soluzioni non prevedono l'uso di parentesi perché gli operatori operano direttamente su gli operandi che li **precedono** (*postfissa*) o li **seguono** (*prefissa*).

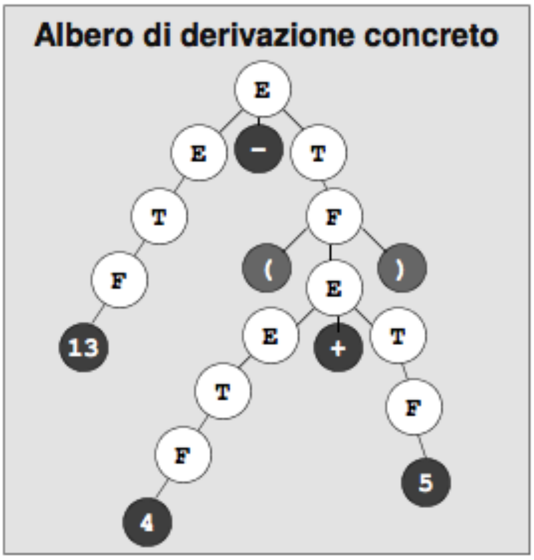
INFISSA:	3 + 4 * 5	(3 + 4) * 5	9 – 4 – 1	9 – (4 – 1)
PREFISSA:	+ 3 * 4 5	* + 3 4 5	– – 9 4 1	– 9 – 4 1
POSTFISSA:	3 4 5 * +	3 4 + 5 *	9 4 – 1 –	9 4 1 – –

Case of study: Espressioni aritmetiche

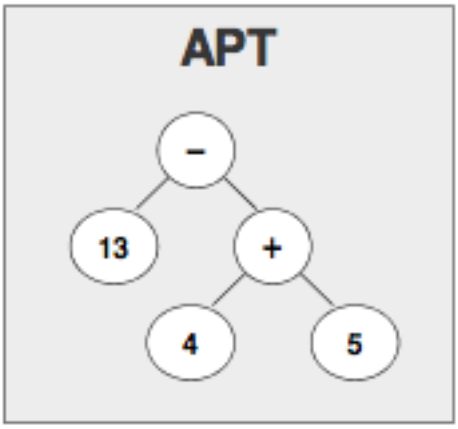
Slides pag 3 pacco 5.

Rappresentazione delle frasi (Alberi sintattici)

Conviene rappresentare una frase con un albero sintattico **astratto** che si ottiene *eliminando* dall'albero di derivazione concreto tutti i nodi non indispensabili e rendendo l'albero quanto più **binario** possibile.

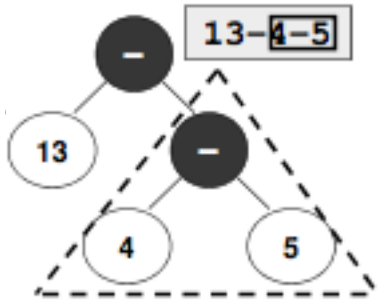


—>



Nel caso delle espressioni:
- Il figlio <u>sinistro</u> è il **primo operando**
- Il figlio <u>destro</u> è il **secondo operando**
- le <u>foglie</u> sono i **valori numerici**

In questo modo la rappresentazione è **univoca —>** la struttura dell'albero fornisce intrinsecamente l'ordine corretto di valutazione (dal basso verso l'alto)

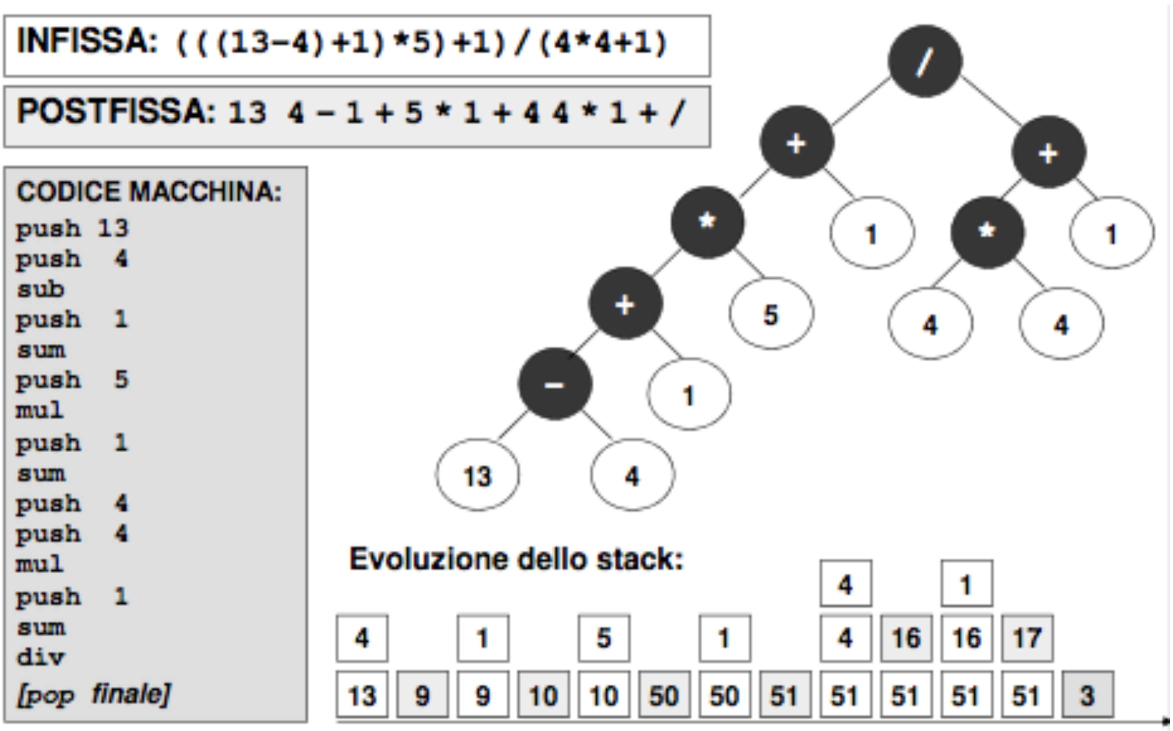


Se un'espressione è un albero, cambiando il tipo di visita cambia la notazione in cui va letto l'albero:
- **pre-order —> prefissa**
- **in-order —> infissa**
- **post-order —> postfissa**

La visita **postfissa** è adattissima a un elaboratore, perché fornisce prima gli operandi (che possono essere inseriti nei registri del processore, pronti per la ALU) e solo **dopo** comanda l'esecuzione dell'operazione.

La macchina a Stack

Nel caso non bastino i registri si può utilizzare una macchina a stack.
- Ogni nodo-valore carica un valore sullo stack (PUSH)
- ogni nodo-operatore causa il prelievo di due valori dallo stack (POP) e il collocamento sullo stack del risultato (PUSH)
- alla fine basta prelevare il risultato dallo stack

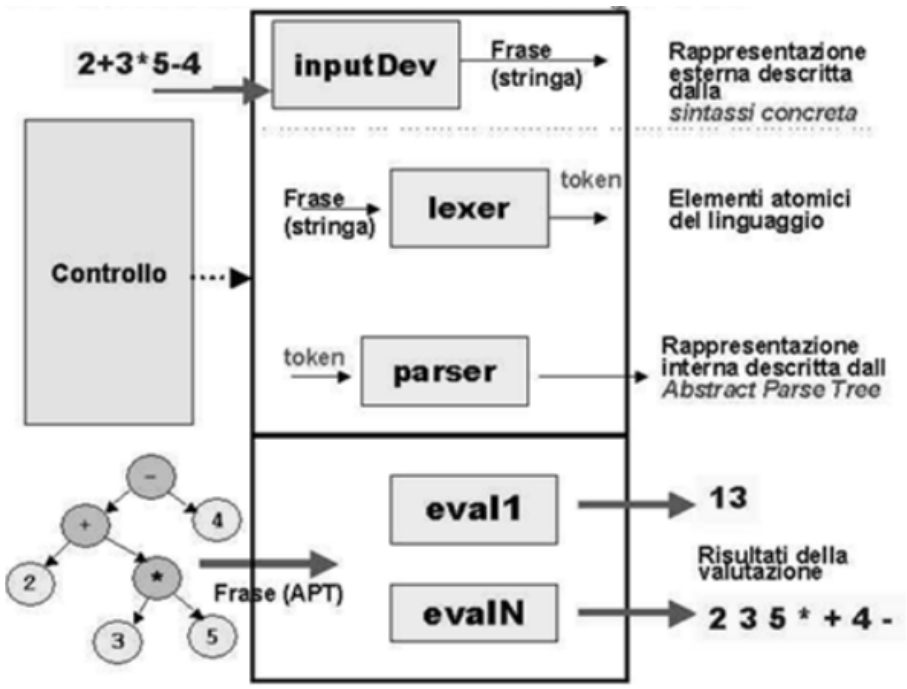


Sintassi astratta

Serve soltanto a rappresentare la frase all'interno del riconoscitore —> può essere ambigua.
Un AST viene costruito dal *riconoscitore* del linguaggio.
- Difetto: difficile produrre errori
- Cura: si organizza l'AST in modo da tenere traccia nei nodi della posizione della parte di frase corrispondente.

Nel caso di espressioni, il riconoscitore è un **PDA** da organizzare mediante **analisi ricorsiva discendente** (sempre se la grammatica è di tipo LL1)
Un puro **riconoscitore** sarebbe un insieme di funzioni (ognuna con risultato boolean). Un **parser** restituirebbe anche la **meta-valutazione del suo sottolinguaggio**

Architettura di un interprete



- **inputDev** denota il dispositivo da cui si riceve la frase in ingresso
- **lexer** è il componente che *individua i token* che costituiscono la frase in ingresso
- **parser** è l'analizzatore sintattico che controlla la corrispondenza dea frase alle regole della sintassi e produce l'AST
- **eval** è il componente valutatore che, ricevuto in ingresso l'AST, effettua la valutazione della frase e produce l'output

Caso di studio: parser per espressioni aritmetiche

Vedi slides pag13, capitolo 5

Stili di interpretazione

Una volta definita una rappresentazione per la grammatica G si possono seguire due strade per definire l'interprete, metodologia:

- **funzionale:** separa nettamente la sintassi (astratta) dall'interpretazione. Si basa su una **funzione di interpretazione** (esterna alle classi) che:
  - determina la classe dell'oggetto F (es: instanceof)
  - opera una cast "sicuro"
  - accede al contenuto informativo di F per "calcolarne" il valore.
- **object-oriented**: sfrutta il **polimorfismo** dei linguaggi ad oggetti. Si basa su un **metodo di interpretazione** (interno alle classi) che:
  - esiste in versione *specializzata per ogni classe.*
  - invocando il metodo sull'oggetto F, grazie al polimorfismo, viene eseguita la versione *appropriata* del metodo.

Stile funzionale

- **PRO**: facilitata l'introduzione di nuove interpretazioni perché basta *scrivere una nuova funzione*
- **CONTRO**: diventa oneroso introdurre una *nuova produzione* perché occorre modificare il codice di *tutte le funzioni di interpretazione* per tenere conto della nuova struttura.

Stile ad oggetti

- **PRO**: diventa facile *aggiungere nuove produzioni*, perché basta definire una nuova sottoclasse con la relativa versione del metodo di valutazione
- **CONTRO**: è difficile introdurre *nuove interpretazioni* perché occorre definire il nuovo metodo in ogni classe.

Approcci a confronto

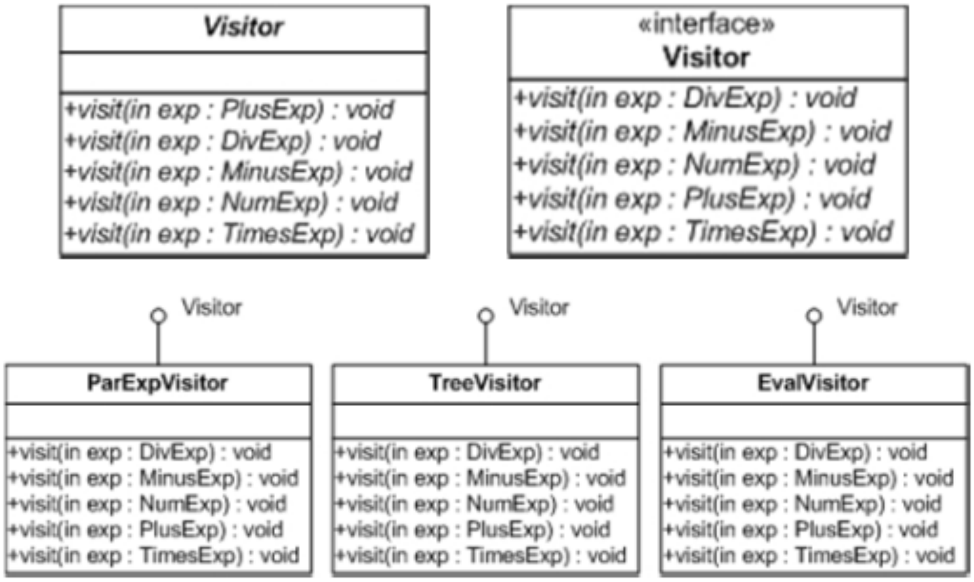
Di solito un linguaggio di programmazione ha una grammatica fissa, richiede molteplici interpretazioni delle frasi, perciò sembra più appropriato utilizzare lo **stile funzionale**. Tuttavia può essere conveniente non abbandonare lo stile ad oggetti.
Soluzione —> **pattern visitor**

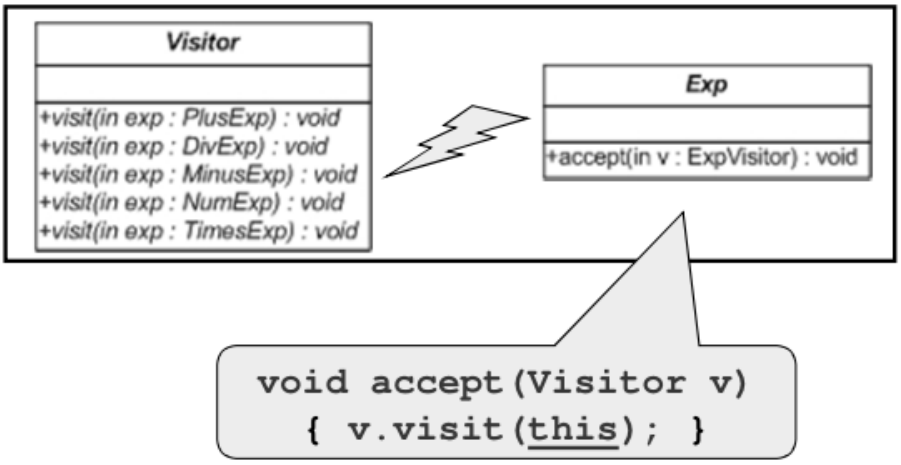
Pattern Visitor

Idee di fondo:

- Mantenere l'unitarietà della funzione di interpretazione localizzandola in un **solo oggetto** (visitor)
- nel contempo mantenere la possibilità di porre in ogni classe il **pezzo di funzione** che lo riguarda.

Realizzazione
Ogni visitor realizza una *funzione di interpretazione* (si scrivono tanti visitor quante le interpretazioni richieste).





io visitatore accetto la tua visita, e lo faccio *ordinandoti di visitarmi.*
VANTAGGI:
- non si “sparpagliano” i metodi di valutazione in giro
- c’è un unico oggetto responsabile della valutazione (visitor)
- evitiamo instanceof, if e cast statici

L'interprete esteso: assegnamenti, ambienti, sequenze

Espressioni di assegnamento

Obiettivo: estendere l'interprete introducendo espressioni le cui azioni *semantiche* producano *effetti collaterali di assegnamento* di valori a *simboli di variabile.*

Sintassi:
- **Astratta:** IDENT = EXP
- **Concreta**: x = *espressione*

Semantica:
- Si valuta il valore *val* dell'espressione che compare a destra del simbolo =
- Si aggiorna l'insieme degli ***identificatori definiti****:*
    - Se in questo insieme <u>non compare</u> già una coppia con primo elemento x, allora si inserisce nell'insieme la coppia (x, val)
    - Se <u>esiste</u> già una coppia (x, v), questa coppia viene eliminata e si inserisce al suo posto la nuova coppia (x, val).

Effetti collaterali

Questa definizione della semantica dell'assegnamento presuppone l'esistenza di un insieme di identificatori *inizialmente vuoto*. Ogni identificatore è caratterizzato dalla coppia (nome, valore).

SI CHIAMA ENVIRONMENT L'INSIEME DEGLI IDENTIFICATORI DEFINITI CHE RISULTANO <u>ACCESSIBILI</u> DURANTE UN PROCESSO D CALCOLO.

Poiché, per ora, questo insieme è <u>unico</u> utilizzeremo il termine di **environment globale.**

L-Value vs R-Value

L'interpretazioni dei simboli varierà a seconda che essi si trovino a sinistra o a destra di =
I linguaggi imperativi introducono i concetti di l-value e r-value:
- **l-value:** l'identificatore indica il ***simbolo*** (o la cella di memoria ad esso associata)
- **r-value:** l'identificatore denota il ***valore*** associato al simbolo (o il contenuto della cella di memoria ad esso associata)

Strumenti per la generazione automatica di riconoscitori LL

Riconoscitori LR(0)

Strumenti per la generazione automatica di riconoscitori LR

Processi computazionali iterativo e ricorsivoBasi di programmazione funzionale

---

Javascript

Lambda calcolo

Scala
