



ALMA MATER STUDIORUM A.D. 1088

**UNIVERSITÀ DI BOLOGNA**

Scuola

**Ingegneria e Architettura**

Corso di studio (laurea magistrale)

**Ingegneria e Scienze informatiche**

Sede didattica

**Cesena**

A.A.

**2017/2018**

Insegnamento

**Linguaggi, compilatori e modelli computazionali**

Titolare

**Prof. Mario Bravetti**

[Versione PDF di questi appunti](#)

## Sommario

Informazioni tecniche	8
Contenuti del corso	8
Materiale e Testi di riferimento	8
Laboratorio	8
Esame	8
Introduzione	9
Parallelismo tra una lingua un linguaggio di programmazione	9
Introduzione ai compilatori	10
Linguaggi regolari	12
Concetti di base	13
Proprietà di chiusura dei linguaggi regolari	15
Problemi legati ai linguaggi regolari	15
Automi	15
Automi a stati finiti deterministici (DFA)	15
Automi a stati finiti non deterministici (NFA)	17
Equivalenza di DFA e NFA	18
FA con transizioni epsilon	19
Epsilon-chiusura	20
Equivalenza di DFA e $\epsilon$ -NFA	20
Espressioni regolari	21
Operazioni sui linguaggi	21
Costruire espressioni regolari, definizione	21
Equivalenza di automi a stati finiti (FA) e espressioni regolari	22
Da espressioni regolari a $\epsilon$ -NFA	23
Leggi algebriche per i linguaggi	24
Proprietà dei linguaggi regolari	24
Pumping Lemma	24
Proprietà di chiusura	26
Proprietà di decisione	28
Tecniche di minimizzazione	30
Perché non si può migliorare un DFA minimizzato	31
Grammatiche e Linguaggi Liberi dal Contesto	32
Esempio informale di CFG	32
Definizione formale di CFG	33
Esempio CFG per $L_{\text{pal}}$	33
Esempio CFG per un semplice linguaggio di programmazione	33
Derivazioni	33

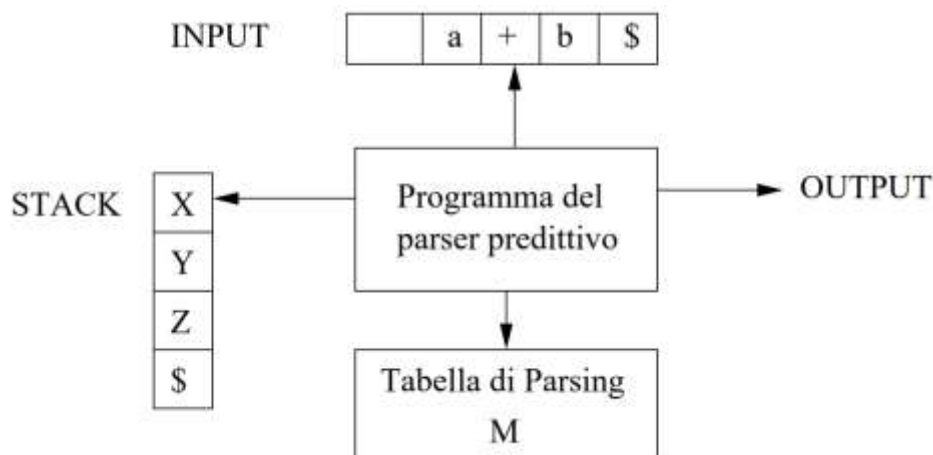
---

Derivazioni a sinistra e a destra (leftmost e rightmost derivations)	34
Il Linguaggio di una grammatica	34
Forme sentenziali	34
Alberi sintattici	35
Costruzione di un albero sintattico	35
Prodotto di un albero sintattico	36
Osservazione	36
Derivazioni da alberi sintattici	37
Ambiguità	37
Rimozione delle ambiguità dalle grammatiche	38
Ambiguità inerente	38
Automi a pila	39
Definizione formale	40
Descrizioni istantanee	40
Accettazione per stato finale	42
Accettazione per pila vuota	42
Trasformazione da pila vuota a stato finale	43
Trasformazione da stato finale a pila vuota	44
Equivalenze PDA e CFG	44
Trasformazione da CFG a PDA	44
Trasformazione da PDA a CFG	45
PDA Deterministici	48
DPDA che accettano per stato finale	49
DPDA che accettano per pila vuota	49
DPDA e non ambiguità	49
Proprietà dei CFL	50
Forma normale di Chomsky	50
Eliminazione simboli inutili	50
Eliminazione produzioni $\epsilon$	51
Eliminazione produzioni unità	51
Forma normale di Chomsky CNF	52
Pumping Lemma per CFL	53
Teorema	53
Dimostrazione del Pumping Lemma	53



Applicazione	54
Proprietà di chiusura per i CFL	56
Operazioni tra CFL e regolari	56
Proprietà di decisione per i CFL	57
Conversione fra CFG a PDA	57
Conversione da CFG a CNF	57
Verificare se un CFL è vuoto	57
Verificare se una stringa appartiene ad un CFL	58
Problemi indecidibili per CFL	60
Analisi lessicale e sintattica	61
Compilatori	61
Struttura di un compilatore	62
Analisi lessicale (scanning, lexer)	62
Analisi sintattica (parsing, parser)	63
Analisi semantica (type checking)	63
Code generation	64
Gestione tabella dei simboli	64
Analisi Lessicale: costruzione di un lexer	64
Introduzione	64
Lexer imperativo	65
Maximal match rule	65

Lexer dichiarativo	66
Full declarative lexer	66
Algoritmo	66
Problemi da gestire nella pratica	67
Regular expression	68
Lexer generators	69
Analisi sintattica: costruzione di un parser	69
Introduzione	69
Parse tree e abstract syntax tree	70
Ambiguità	70
Eliminazione ambiguità pg 187 dragon book	71
Syntax checking	71
Parse tree construction	71
Insiemi First e Follow	71
Esempio di calcolo insiemi FIRST E FOLLOW.	72
Algoritmi di parsing top-down	72
Recursive Descent Parsing	73
Ricorsione a sinistra	73
Predictive Parsing	74
Grammatica LL(1)	75
Left factoring (fattorizzazione a sinistra)	76
Algoritmo per la creazione della tabella di parsing LL(1)	<b>Errore. Il segnalibro non è definito.</b>
Struttura di un predictive non recursive parser	77
Algoritmo predictive non recursive parsing	77



78

Algoritmo predictive non recursive parsing (uso della tabella di parsing)	78
---	----

METODO	78
--------	----

Algoritmi di parsing bottom-up	79
--------------------------------	----

Chaotic bottom-up parser and non deterministic chaotic bottom-up parser	79
---	----

LR parser	80
-----------	----

Generalized Non deterministic LR parser	81
Non deterministic LR parser	81
LR(1) parsing	82
Algoritmo LR(1) parsing	82
Costruzione della tabella di parsing LR	82
Costruzione del DFA	83
Risoluzione dei conflitti	85
LALR	86
Osservazioni conclusive sul parsing	87

POSSIBILI AMBIGUITÀ DI UNA GRAMMATICA	
Ricorsione diretta	INPUT: $A \rightarrow \alpha\alpha_1   \dots   \alpha\alpha_n   \beta_1   \dots   \beta_n$ OUTPUT: $A \rightarrow \beta_1 A^1   \dots   \beta_n A^n$ e $A^i \rightarrow \alpha_i   \dots   \alpha_n$
Ricorsione indiretta	INPUT: $S \rightarrow A\alpha   \beta$ e $A \rightarrow S\delta$ OUTPUT: $S \rightarrow A\alpha   \beta$ e $A \rightarrow \beta\delta A^1$ e $A^i \rightarrow \alpha\beta\delta A^i   \epsilon$
Fattorizzazione a sinistra (left-factoring)	INPUT: $S \rightarrow \alpha\beta_1   \dots   \alpha\beta_n   \gamma_1   \dots   \gamma_n$ OUTPUT: $S \rightarrow \alpha A^1   \gamma_1   \dots   \gamma_n$ e $A^i \rightarrow \beta_i   \dots   \beta_n$

**GRAMMATICA**  
 $G = (V, T, P, S)$

variabili  
 terminali  
 produzioni  
 variabile iniziale

	FIRST	SI	SI	SI
	FOLLOW	SI	SI	SI
	DFA	NO	CLOSURE SET CON LOOKAHEAD	CLOSURE SET CON LOOKAHEAD
	RIGHE	variabili	stati del DFA	stati del DFA
TABELLA DI PARSING	COLONNE	terminali + \$	ACTION GOTO	variabili + \$ terminali
	CELLE	$[v, t] = \text{produzione}$ se $t \in \text{First}(v)$ se $\alpha \in \text{First}(v) \text{ e } \alpha \in \text{Follow}(v)$	se stato = non finale $[s, c] = \text{SHIFT}$ $[s, c] = \text{ACCEPT}$ $[s, c] = \text{REDUCE}$ $[s, c] = \text{GOTO}$ $[s, c] = \text{ERROR}$	CONDIZIONI se ACTION A + non finale * finale su produzione iniziale SRS + sul Follow LR/LALR + sui simboli di lookahead se GOTO altrimenti

	88
Analisi semantica	89
Tabella dei simboli	89
Scoping	89
Scoping statico e scoping dinamico	89
Implementazione della tabella dei simboli	90
Metodo 1: list of hashtables	90
Metodo 2: hashtables of list	91
Type checking	91
Tipi di linguaggi	92
Code Generation	94
Logica di base	100
Introduzione	100
Elementi di base	100

---

Sintassi	100
Semantica	100
Terminologia e notazioni	100
Logica delle proposizioni	100
Concetti generali legati alla logica	101
Una versione di logica delle proposizioni semplificata	103
Sistemi di deduzione	103
Nozioni di deduzione logica	103
Regole e assiomi	103
Teorema di deduzione	104
Il sistema di deduzione di Hilbert	104
Natural deduction	106
Logica dei predicati	106
Sintassi	106
Aritmetica di Peano	107
Variabili free e bound	107
Formule chiuse	108
Semantica per la logica dei predicati (per formule chiuse)	108
Validità	108
Il sistema di deduzione di Hilbert	108
Model Checking	110
Logiche temporali	110
Sintassi LTL	111
Modelli: sequenze di mondi	111
Semantica LTL	111
Operatori minimali	112
Relazioni con operatori classici	112
Notazioni alternative	113
Proprietà tipiche per sistemi software	113
Labeled Transition Systems (NFA) e logiche temporali	114
Sistema di mutua esclusione per l'accesso alla sezione critica (esempi)	114
Modelli per sistemi concorrenti	114
Process Algebra	115
Caratteristiche essenziali	115
Esempi	115
Concorrenza (interleaving) e sincronizzazione	115
Composizione parallela	116
Proprietà della composizione parallela	116

---

Hiding	117
Esempio mutua esclusione	118
Reti di Petri	119
Model checking su reti di Petri	120
Macchine di Turing	121
Introduzione	121
Macchine di Turing	122
Descrizione Istantanea (ID) di una Turing Machine	123
Linguaggio di una Turing Machine	124
Linguaggi ricorsivi e ricorsivamente enumerabili	124
Trucchi di programmazione / estensioni per le Turing Machine	124
Esempio	126
Turing Machine con nastro semi-infinito	128
Turing Machine multinastro	128
Turing Machine non deterministiche	128
Vantaggi delle estensioni	129
Proprietà di chiusura per Linguaggi Ricorsivi e ricorsivamente enumerabili	130
Decidibilità	131
Macchine di Turing codificate come numeri	131
Diagonalizzazione	132
Problemi indecidibili	132
Linguaggio Universale	133
Indecidibilità di $L_u$ (problema della fermata)	134
Riduzioni	134
Teorema di Rice	134
Decidibilità di linguaggi ricorsivamente enumerabili (riconosciuti da Turing Machine)	134
Calcolabilità di funzioni	136
Esercizio	136
Soluzione	136
Esercizio	137
Soluzione	137
Altro esercizio (tipico da esame)	138



## Informazioni tecniche

### Contenuti del corso

- Linguaggi formali e automi
- Compilatori
- Logiche di base
- Modelli computazionali

### Materiale e Testi di riferimento

Per accedere al materiale didattico e ricevere le comunicazioni è necessario iscriversi alla lista di distribuzione **mario.bravetti.LCMC2017**.

I testi di riferimento utilizzati durante il corso sono i seguenti.

- **Automi, linguaggi e calcolabilità, Addison-Wesley**  
J. E. Hopcroft, R. Motwani e J. D. Ullman  
Terza Edizione, 2009  
Capitoli 1–9
- **Compilatori: principi tecniche e strumenti**  
A. V. Aho, M. S. Lam, R. Sethi e J. D. Ullman  
Addison Wesley, Seconda Edizione, 2009  
Capitoli 1–5
- **Logic in Computer Science: Modelling and Reasoning about Systems**  
M. Huth e M. Ryan  
Cambridge University Press, Second Edition, 2004  
Capitoli 1–3

Sito con addendum e soluzioni al libro di testo: <http://infolab.stanford.edu/~ullman/ialc.html>

### Laboratorio

Il laboratorio è parte integrante del corso ed utilizza tool specifici a supporto della parte teorica:

- JFLAP <http://www.jflap.org>  
Argomenti: automi, grammatiche, espressioni regolari
- Tina <http://projects.laas.fr/tina>  
Argomenti: model checking di sistemi di transizione e reti di Petri
- LTSA <http://www.doc.ic.ac.uk/ltsa>  
Argomenti: sistemi di transizione definiti tramite algebre di processi

Eclipse esteso co plug-in ANTLR 4 IDE 0.3.6 viene utilizzato come ambiente di sviluppo per la realizzazione di parser/compilatori.

Per partecipare al laboratorio è necessaria iscrizione tramite mail al docente.

### Esame

L'esame prevede una prova scritta che permette di ottenere fino a 27 punti.

Facoltativa e integrativa all'esame è la realizzazione di un progetto che prevede l'implementazione di un compilatore. Tale progetto può essere svolto singolarmente o in gruppi e deve essere discusso in gruppo, previo appuntamento, dopo lo scritto entro l'inizio della sessione successiva. Il progetto permette di ottenere fino a 7 punti ulteriori.

## Introduzione

**Noam Chomsky** (Filadelfia, 7 dicembre 1928) è un linguista, filosofo, storico, teorico della comunicazione e anarchico statunitense.

Professore emerito di **linguistica** al Massachusetts Institute of Technology, è riconosciuto come il fondatore della **grammatica generativo-trasformativa**.

Le sue considerazioni sulle grammatiche e il linguaggio sono state di fondamentale importanza per lo sviluppo dei compilatori.



## Parallelismo tra una lingua e un linguaggio di programmazione

LINGUA	LINGUAGGIO DI PROGRAMMAZIONE
<b>Alfabeto</b> ↓ <i>regole di lessico</i> <b>Parole</b> → <b>Vocabolario</b> sequenze di lettere dell'alfabeto che rispettano le regole di lessico ↓ <i>regole di sintassi</i> <b>Frase</b> sequenze di parole e spazi del vocabolario che rispettano le regole di sintassi	<b>Alfabeto di simboli</b> ↓ <i>regole di lessico (specifiche del linguaggio)</i> <b>Parole chiave, identificatori</b> → <b>Tokens</b> sequenze di simboli dell'alfabeto che rispettano le regole di lessico ↓ <i>regole di sintassi (standard: EBNF)</i> <b>Programmi corretti</b> Sequenze di tokens che rispettano le regole di sintassi

## THE STRUCTURE OF A COMPILER

### Introduzione ai compilatori

Un compilatore si occupa di verificare le regole di lessico e di sintassi e di segnalare eventuali errori. Se le regole di lessico e sintassi sono rispettate il compilatore può procedere alla generazione del codice eseguibile.

Un compilatore prende in input un **codice sorgente** e genera in output un **codice oggetto** binario, dipendente dall'architettura ed eseguibile in CPU.

Differentemente un interprete prende in input un **codice sorgente** e verifica e manda in esecuzione ogni singola istruzione.

Se un programma è compilato il codice generato è eseguito in CPU, se un programma è interpretato l'interprete è eseguito in CPU.

Il **FRONT END** di un compilatore si occupa dell'analisi **lessicale**, **sintattica** e **semantica** (type checking).

Un compilatore “colma” la distanza tra un codice sorgente del linguaggio e il codice eseguibile in linguaggio macchina: i compilatori per linguaggi del dominio applicativo con un elevato livello di astrazione sono complessi.

Il primo compilatore sviluppato per il linguaggio **FORTRAN**, ha richiesto 18 anni uomo di lavoro ed è stato realizzato in **ASSEMBLER** (in assenza di alternative).

I successivi compilatori potevano essere sviluppati in ASSEMBLER o in FORTRAN.

Il **self hosting** è la metodica per la quale un compilatore è in grado di “compilare sé stesso”.

Il primo compilatore per LISP è stato realizzato sull'interprete stesso di LISP (**bootstrap problem**), le sue successive versioni venivano compilate dallo stesso compilatore (**self hosting**).

Grazie anche ai risultati di Noam Chomsky, agli inizi degli anni '60, vennero inventati i **generatori di compilatori** che automatizzavano tutta la parte di analisi lessicale e sintattica.

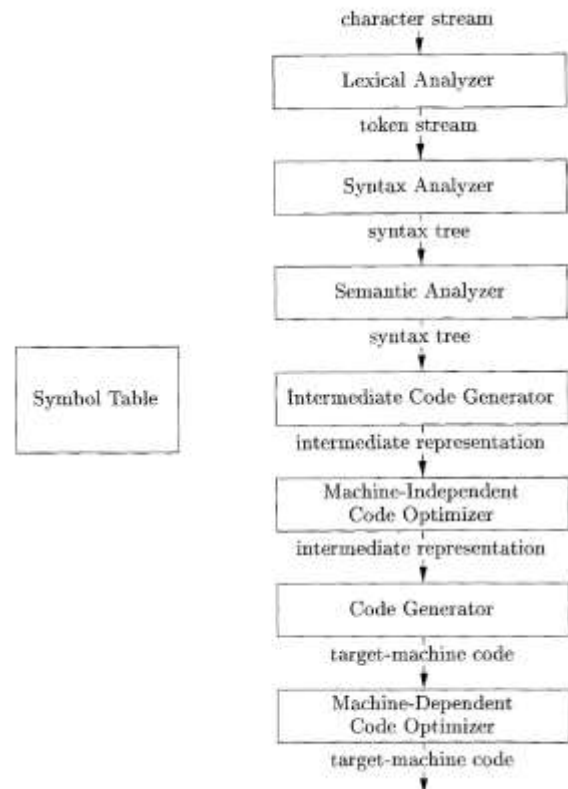
I **generatori di compilatori** utilizzano le specifiche delle regole di lessico e di sintassi e generano compilatori.

Questo **approccio dichiarativo**, che prevede di specificare le regole di lessico e sintassi, utilizza:

1. le **regular expression** per definire le regole di lessico (più semplici)  
Esempio:  $[a-z][a-z0-9]^*$  definisce una regola di lessico per i nomi delle variabili (primo carattere una lettera - in questo caso minuscola - seguito da un numero imprecisato di caratteri alfanumerici).
2. il formalismo **EBNF (Extended Backus-Naur Form)** per definire le regole sintattiche (più complesse e iterative)  
Esempio:  $STATEMENT ::= IF\ EXP\ THEN\ STATEMENT\ ELSE\ STATEMENT$   
Esempio:  $EXP ::= EXP + EXP \mid EXP / EXP \mid ( EXP )$

Per descrivere, studiare e validare le regole di lessico e di sintassi si possono utilizzare gli **automi**

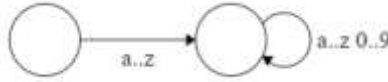
“Dal punto di vista pratico, il concetto di automa a stati finiti equivale a costruire un piccolo dispositivo che mediante una testina legge una stringa di input su un nastro e la elabora, facendo uso di un meccanismo molto semplice di calcolo e di una memoria limitata. L'esame della stringa avviene un carattere alla volta attraverso precisi passi computazionali che comportano l'avanzamento della testina. In sostanza un ASF è un caso particolare di [macchina di Turing](#), utilizzato per l'elaborazione di quei linguaggi che nelle [Grammatiche di Chomsky](#) sono definiti



di Tipo 3 o [Regolari](#). Distinguiamo due tipi di automi a stati finiti: gli automi a stati finiti deterministici (ASFD) e gli automi a stati finiti non deterministici (ASFND).” [Fonte: WIKIPEDIA].

Una regola lessicale può essere descritta utilizzando un **automa a stati finiti** che richiede una quantità costante di memoria, indipendente dalla lunghezza della parola fornita in input.

L'automa a stati finiti illustrato nell'immagine seguente rappresenta la regexp  $[a-z][a-zA-Z0-9]^*$



Una regola sintattica richiede invece una quantità di memoria dipendente dallo statement (problema delle parentesi innestate) in quanto potrebbe includere statement nidificati che richiedono valutazioni indipendenti e che forniscono risultati parziali. Per descrivere una regola sintattica si possono utilizzare **automi a pila** che prevedono l'implementazione di stati nidificati.

## Linguaggio funzionale

“In [informatica](#) la programmazione funzionale è un [paradigma di programmazione](#) in cui il flusso di [esecuzione](#) del [programma](#) assume la forma di una serie di valutazioni di [funzioni matematiche](#). Il punto di forza principale di questo paradigma è la mancanza di [effetti collaterali](#) (side-effect) delle funzioni, il che comporta una più facile verifica della correttezza e della mancanza di [bug](#) del programma e la possibilità di una maggiore ottimizzazione dello stesso. Un uso particolare del paradigma, per l'ottimizzazione dei programmi, è quello di trasformare gli stessi per utilizzarli nella [programmazione parallela](#).

La programmazione funzionale pone maggior accento sulla definizione di funzioni, rispetto ai paradigmi [procedurali](#) e [imperativi](#), che invece prediligono la specifica di una sequenza di [comandi](#) da eseguire. In questi ultimi, i valori vengono calcolati cambiando lo stato del programma attraverso delle [assegnazioni](#); un programma funzionale, invece, è immutabile: i valori non vengono trovati cambiando lo stato del programma, ma costruendo nuovi stati a partire dai precedenti.” [Fonte: WIKIPEDIA].

Il seguente testo è tratto da:

<http://www.dipmat.unict.it/~barba/FONDAMENTI/PROGRAMMI-TESTI/READING-MATERIAL/Ausiello.pdf>

Tra i concetti principali alla base dell'informatica, il concetto di **linguaggio** riveste un ruolo particolarmente importante. Anche se da un punto di vista matematico un linguaggio è semplicemente un insieme di stringhe su un dato alfabeto, **nell'ambito dell'informatica siamo interessati a linguaggi, generalmente infiniti, le cui stringhe sono caratterizzate da qualche particolare proprietà**: ad esempio, il linguaggio dei programmi C è costituito da tutte le stringhe che soddisfano la proprietà di poter essere analizzate da un compilatore C senza che venga rilevato alcun errore sintattico.

Lo studio formale dei linguaggi è una parte importante dell'informatica teorica e trova applicazione in varie direzioni. La prima, più evidente, è appunto lo studio delle proprietà sintattiche dei programmi, cioè lo studio dei metodi di definizione della sintassi di un linguaggio di programmazione, dei metodi per la verifica che un programma soddisfi le proprietà sintattiche volute e dei metodi di traduzione da un linguaggio ad un altro (tipicamente da un linguaggio di programmazione ad alto livello al linguaggio di macchina).

Inoltre, al di là di questa importante applicazione, la familiarità con i concetti di generazione e riconoscimento di linguaggi è importante in quanto stringhe di caratteri aventi struttura particolare vengono frequentemente utilizzate, in informatica, per rappresentare vari aspetti relativi all'elaborazione di dati, come ad esempio sequenze di operazioni eseguite da un programma, sequenze di passi elementari eseguiti da una macchina, protocolli di comunicazione, sequenze di azioni eseguite nell'interazione con un'interfaccia utente, ecc.

Infine, è importante tenere presente che, nella formulazione dei problemi trattati mediante metodi algoritmici (come il calcolo di funzioni, la risoluzione di problemi di ottimizzazione, ecc.) ci si riconduce frequentemente al problema standard di decidere l'appartenenza di una stringa ad un dato linguaggio: ad esempio, anziché porsi il problema di calcolare una funzione  $f: N \rightarrow N$ , ci si può porre il problema di riconoscere il linguaggio  $\{a^n b^{f(n)} \mid n \geq 0\}$ .

**La definizione di linguaggi si può effettuare mediante strumenti diversi.** Un primo esempio di strumento formale per la definizione di linguaggi sono le **ESPRESSIONI REGOLARI**, tuttavia, le espressioni regolari consentono di definire linguaggi di tipo particolare, con una struttura piuttosto semplice, e non sono idonee a rappresentare linguaggi più complessi, come ad esempio i linguaggi di programmazione. Un approccio alternativo è il cosiddetto **APPROCCIO GENERATIVO**, dove si utilizzano opportuni strumenti formali, le **GRAMMATICHE** formali appunto, che consentono di costruire le stringhe di un linguaggio tramite un insieme prefissato di regole, dette regole di produzione. Altro approccio di interesse, infine, è quello **RICONOSCITIVO**, che consiste nell'utilizzare macchine astratte, dette **AUTOMI** riconoscitori, che definiscono algoritmi di riconoscimento dei linguaggi stessi, vale a dire algoritmi che per un dato linguaggio  $L \subseteq \Sigma^*$  stabiliscono se una stringa  $x \in \Sigma^*$  appartiene a  $L$  o meno.

## Linguaggi regolari

In informatica teorica un linguaggio regolare è un linguaggio formale, ossia costituito da un insieme di stringhe costruite con un alfabeto finito, che è descritto da un'espressione regolare, generato da una grammatica generativa regolare (o di tipo 3, secondo la gerarchia di Chomsky) o accettato da un automa a stati finiti (automa a stati finiti deterministico o automa a stati finiti non deterministico).

L'insieme dei linguaggi regolari basati su un alfabeto  $\Sigma$  è definito ricorsivamente come segue:

- il linguaggio vuoto  $\emptyset$  è un linguaggio regolare

- il linguaggio  $\{\epsilon\}$  contenente la sola stringa vuota è un linguaggio regolare
- per ogni carattere  $a \in \Sigma$ , il linguaggio singleton  $\{a\}$  è un linguaggio regolare.
- se  $A$  e  $B$  sono linguaggi regolari allora  $A \cup B, A \cdot B, A^*$  sono linguaggi regolari.
- nessun altro linguaggio su  $\Sigma$  è regolare.

Tutti i linguaggi finiti sono regolari.

Tipici esempi sono:

- linguaggio consistente di tutte le stringhe dell'alfabeto  $\{a, b\}$  e che contiene un numero pari di  $a$
- linguaggio consistente di tutte le stringhe nella forma: zero o più  $a$  seguite da zero o più  $b$

## Concetti di base

<b>ALFABETO</b> Insieme finito e non vuoto di simboli	Esempi Alfabeto binario $\Sigma = \{0,1\}$ Insieme di tutte le lettere minuscole $\Sigma = \{a, b, c, d, \dots, z\}$ Insieme di tutti i caratteri ASCII
<b>STRINGA</b> Sequenza finita di simboli di un alfabeto $\Sigma$	Esempi Stringa binaria: 001010011 Stringa ASCII: A7@é!(=12fz
<b>STRINGA VUOTA</b> Stringa con zero occorrenze di simboli di un alfabeto $\Sigma$ . <b>La stringa vuota è denotata con <math>\epsilon</math>.</b>	
<b>LUNGHEZZA DI UNA STRINGA</b> Numero di posizioni per i simboli nella stringa. <b><math> w </math> denota la lunghezza della stringa <math>w</math>.</b>	Esempi: $ 0110  = 4$ $ \epsilon  = 0$
<b>POTENZE DI UN ALFABETO <math>\Sigma^k</math></b> Insieme delle stringhe di lunghezza $k$ con simboli da $\Sigma$	Esempi: $\Sigma^1 = \{0, 1\}$ $\Sigma^2 = \{00, 01, 10, 11\}$ $\Sigma^0 = \{\epsilon\}$
<b>INSIEME DI TUTTE LE STRINGHE SU <math>\Sigma</math></b> $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$ $\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$ $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$	
<b>CONCATENAZIONE</b> Se $x$ e $y$ sono stringhe, allora $xy$ è la stringa ottenuta collocando una copia di $y$ subito dopo una copia di $x$ . $x = a_1 a_2 \dots a_i$ , $y = b_1 b_2 \dots b_j$ $xy = a_1 a_2 \dots a_i b_1 b_2 \dots b_j$ Nota: per ogni stringa $x$ $x\epsilon = \epsilon x = x$	Esempi: $x = 01101$ $y = 110$ $\Rightarrow$ $xy = 01101110$

<b>LINGUAGGI</b>  Se $\Sigma$ è un alfabeto, e $L \subseteq \Sigma^*$ allora $L$ è un linguaggio.	<p>Esempi di linguaggi:</p> <ul style="list-style-type: none"><li>• L'insieme delle parole italiane legali</li><li>• L'insieme dei programmi C legali</li><li>• L'insieme delle stringhe che consistono di <math>n</math> zeri seguiti da <math>n</math> uni: <math>\{\epsilon, 01, 0011, 000111, \dots\}</math></li><li>• L'insieme delle stringhe con un numero uguale di zeri e di uni: <math>\{\epsilon, 01, 10, 0101, 1010, 1001, \dots\}</math></li><li>• <math>L_p</math>: l'insieme dei numeri binari il cui valore è primo: <math>\{10, 11, 101, 111, 1011, \dots\}</math></li><li>• Il linguaggio vuoto <math>\emptyset</math></li><li>• Il linguaggio <math>\{\epsilon\}</math> consistente della stringa vuota</li></ul> <p>Nota: <math>\emptyset \neq \{\epsilon\}</math></p> <p>Nota: L'alfabeto <math>\Sigma</math> è sempre finito.</p>
---	---

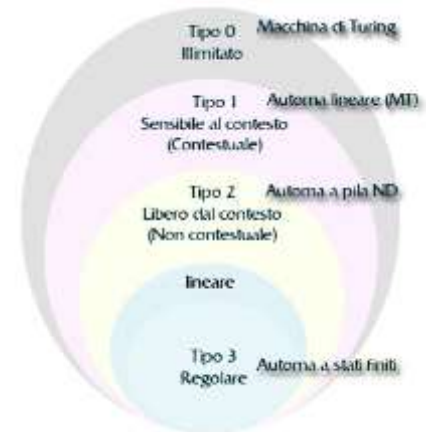
## Proprietà di chiusura dei linguaggi regolari

I linguaggi regolari sono chiusi rispetto alle seguenti operazioni (sia  $L$  un linguaggio regolare):

- $\bar{L}$ : complemento
- $L^*$ : stella di Kleene
- $L_1.L_2$ : concatenazione
- $L_1 \cup L_2$ : unione
- $L_1 \cap L_2$ : intersezione
- $L_1 \setminus L_2$ : differenza
- $L_1^R$ : riflesso

## Problemi legati ai linguaggi regolari

Nella gerarchia di Chomsky i linguaggi regolari corrispondono ai linguaggi generati da grammatiche di tipo 3. È possibile stabilire se un linguaggio è regolare o meno utilizzando il teorema di Myhill-Nerode. È invece possibile dimostrare che un linguaggio non è regolare utilizzando il pumping lemma per i linguaggi regolari. Dati due linguaggi regolari  $L_1$  e  $L_2$  è possibile verificare l'inclusione  $L_1 \subseteq L_2$  utilizzando le proprietà di chiusura. Per questo motivo è possibile stabilire se due linguaggi regolari sono equivalenti.



## Automi

Nella teoria dei sistemi dinamici, un **automa** è un sistema dinamico discreto (nella scansione del tempo e nella descrizione del suo stato) e invariante (il sistema si comporta alla stessa maniera indipendentemente dall'istante di tempo in cui agisce).

- L'insieme dei possibili simboli che possono essere forniti ad un automa costituisce il suo **alfabeto**.
- Quando l'automa si trova in un dato stato, esso può accettare solo un **sottoinsieme dei simboli del suo alfabeto**.
- L'evoluzione di un automa parte da un particolare stato detto **stato iniziale**.
- Un sottoinsieme privilegiato dei suoi stati è detto insieme degli **stati finali** o marcati.

Una sequenza di simboli (detto anche stringa o parola) appartiene al linguaggio se essa viene accettata dal corrispondente automa, ovvero se porta l'automa in uno stato valido, che sia lo stesso o un altro stato. Un sottoinsieme del linguaggio riconosciuto, chiamato linguaggio marcato porta l'automa dal suo stato iniziale ad uno stato finale o marcato.

In genere gli automi sono **deterministici**: dato uno stato ed un simbolo in ingresso è possibile una sola transizione. Esistono comunque anche automi **non deterministici**, o stocastici.

Gli automi sono spesso utilizzati per descrivere linguaggi formali in informatica teorica, e per questo sono chiamati accettori o riconoscitori di un linguaggio.

A diverse classi di automi corrispondono diverse classi di linguaggi, caratterizzate da diversi livelli di complessità.

## Automi a stati finiti deterministici (DFA)

Nella teoria del calcolo, un **automa a stati finiti deterministico** (ASFD) o **deterministic finite automaton** (DFA) è un automa a stati finiti dove **per ogni coppia di stato e simbolo in ingresso c'è una ed una sola transizione allo stato successivo**.

Un **automa a stati finiti deterministico** (DFA) è una quintupla  $A = (Q, \Sigma, \delta, q_0, F)$  dove:

- $Q$  è un insieme finito di stati
- $\Sigma$  è un alfabeto finito (simboli di input)
- $\delta$  è una funzione di transizione  $(q, a) \rightarrow p$
- $q_0 \in Q$  è lo stato iniziale



$F \subseteq Q$  è un insieme di stati finali

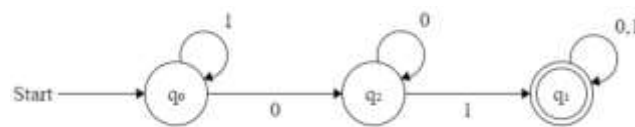
Esempio: un automa  $A$  che accetta  $L = \{x01y: x, y \in \{0,1\}^*\}$

L'automata  $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$

può essere descritto come tabella di transizione

	0	1
$\rightarrow q_0$	$q_2$	$q_0$
$* q_1$	$q_1$	$q_1$
$q_2$	$q_2$	$q_1$

o come un diagramma di transizione



Un automa a stati finiti accetta una stringa  $w = a_1a_2\dots a_n$  se esiste un cammino nel diagramma di transizione che

1. Inizia nello stato iniziale
2. Finisce in uno stato finale (di accettazione)
3. Ha una sequenza di etichette  $a_1a_2\dots a_n$

L'automata a stati finiti descritto sopra accetta, ad esempio, la stringa 01101, infatti:

- a) 0 entra in  $q_0$
- b) 1 resta in  $q_0$
- c) 1 resta in  $q_0$
- d) 0 passa in  $q_2$
- e) 1 passa in  $q_1$

La stringa termina in  $q_1$  che è lo stato di accettazione.

La funzione di transizione  $\delta$  può essere estesa a  $\hat{\delta}$  che opera su stati e stringhe (invece che su stati e simboli).

**Base:**  $\hat{\delta}(q, \epsilon) = q$     **Induzione:**  $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

Formalmente, il **linguaggio accettato** da un automa a stati finiti deterministico  $A$  è:

$$L(A) = \{w: \hat{\delta}(q_0, w) \in F\}$$

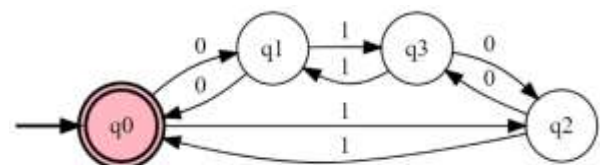
I linguaggi accettati da automi a stati finiti sono detti **linguaggi regolari**.

Esempio: Automa a stati finiti deterministico che accetta tutte e sole le stringhe con un numero pari di zeri e un numero pari di uni.

Diagramma

Tabella di transizione

	0	1
$* \rightarrow q_0$	$q_1$	$q_2$
$q_1$	$q_0$	$q_3$
$q_2$	$q_3$	$q_0$
$q_3$	$q_2$	$q_1$



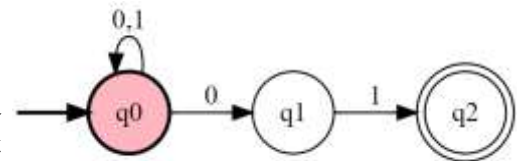
Strumenti online per Automi:

<http://madebyevan.com/fsm/>

[http://ivanzuzak.info/noam/webapps/fsm\\_simulator/](http://ivanzuzak.info/noam/webapps/fsm_simulator/)

## Automi a stati finiti non deterministici (NFA)

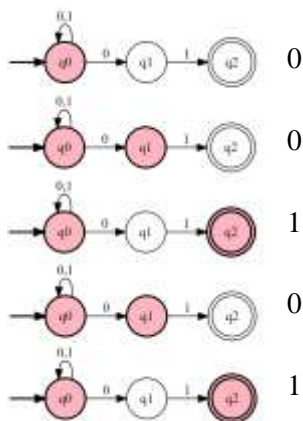
Nella teoria del calcolo, un **automa a stati finiti non deterministico** (ASFND) o **nondeterministic finite automaton** (NFA) è una macchina a stati finiti dove per ogni coppia stato-simbolo in input possono esservi più stati di destinazione.



Un automa a stati finiti non deterministici (NFA) accetta una stringa se esiste un cammino che conduce ad uno stato finale.

Esempio: automa che accetta tutte e solo le stringhe che finiscono con 01.

L'elaborazione della stringa 00101 determina le seguenti fasi:



Formalmente, un automa a stati finiti non deterministici è una quintupla  $A = (Q, \Sigma, \delta, q_0, F)$  dove:

$Q$  è un insieme finito di stati

$\Sigma$  è un alfabeto finito (simboli di input)

$\delta$  è una funzione di transizione da  $Q \times \Sigma$  all'insieme dei sottoinsiemi di  $Q$ , cioè  $(q, a) \rightarrow Q'$  con  $Q' \subseteq Q$

$q_0 \in Q$  è lo stato iniziale

$F \subseteq Q$  è un insieme di stati finali

Esempio: l'automa che accetta tutte e solo le stringhe che finiscono con 01 è  $(\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$ , dove  $\delta$  è la funzione di transizione:

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$q_1$	$\emptyset$	$\{q_2\}$
$* q_2$	$\emptyset$	$\emptyset$

La funzione di transizione estesa  $\hat{\delta}$  è definita: **Base:**  $\hat{\delta}(q, \epsilon) = \{q\}$  **Induzione:**  $\hat{\delta}(q, xa) = \bigcap_{p \in \hat{\delta}(q, x)} \delta(p, a)$

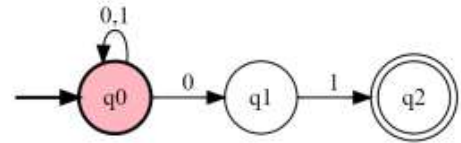
Esempio:  $\hat{\delta}(q_0, 0010) = \{q_0, q_1, q_2\}$

Formalmente, il *linguaggio accettato* da un automa a stati finiti non deterministico Aè

$$L(A) = \{w: \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Gli NFA sono solitamente più semplici da programmare dei DFA.

Esempio: L'NFA qui a fianco accetta il linguaggio  $\{x01: x \in \Sigma^*\}$



## Equivalenza di DFA e NFA

Per ogni NFA  $N$  esiste un DFA  $D$ , tale che  $L(D) = L(N)$  e viceversa.

Dato un NFA  $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$  è possibile costruire un DFA  $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$  tale che  $L(D) = L(N)$

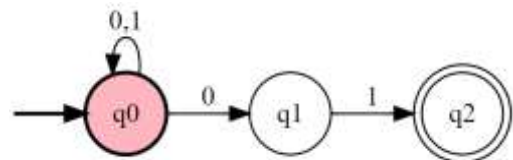
Per farlo si esegue una **costruzione a sottoinsiemi**:

- $Q_D = \{S: S \subseteq Q_N\}$  allora  $|Q_D| = 2^{|Q_N|}$   
 Gli stati in  $Q_D$ , per la maggior parte, non sono raggiungibili dallo stato iniziale e sono detti **garbage**.
- $F_D = \{S \subseteq Q_N: S \cap F_N \neq \emptyset\}$
- Per ogni  $S \subseteq Q_N$  e  $a \in \Sigma$

$$\delta_D(S, a) = \bigcap_{p \in S} \delta_N(p, a)$$

Esempio: il  $\delta_D$  dell'NFA qui a fianco è:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$* \{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$* \{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$* \{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$* \{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$



Nota: gli stati di  $D$  corrispondono a sottoinsiemi di stati di  $N$ , ma si sarebbero potuti denotare in un altro modo, per esempio  $A - H$ , la funzione di transizione può essere riscritta:

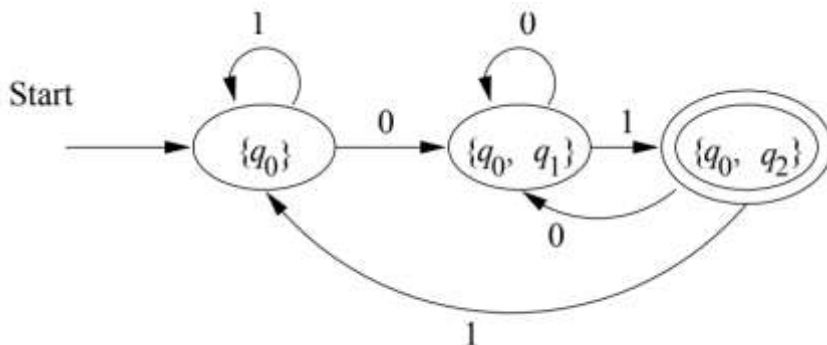
	0	1
$A$	$A$	$A$
$\rightarrow B$	$E$	$B$
$C$	$A$	$D$
$* D$	$A$	$A$
$E$	$E$	$F$
$* F$	$E$	$B$
$* G$	$A$	$D$
$* H$	$E$	$F$

Per evitare la crescita esponenziale degli stati è possibile costruire la tabella di transizione per  $D$  solo per gli stati accessibili  $S$ :

**Base:**  $S = \{q_0\}$

**Induzione:** Se lo stato  $S$  è accessibile, lo sono anche gli stati  $\delta_D(S, a)$  per ogni  $a \in \Sigma$ .

Esempio: il *sottoinsieme* DFA solitamente con stati accessibili.



### Teorema 2.11

Sia  $D$  il DFA ottenuto da un NFA  $N$  con la costruzione a sottoinsiemi. Allora  $L(D) = L(N)$

### Teorema 2.12

Un linguaggio  $L$  è accettato da un DFA se e solo se  $L$  è accettato da un NFA.

Nota: il numero degli stati del DFA equivalente ad un NFA con  $n$  stati è, nel caso peggiore, pari a  $2^n$  stati.

## FA con transizioni epsilon

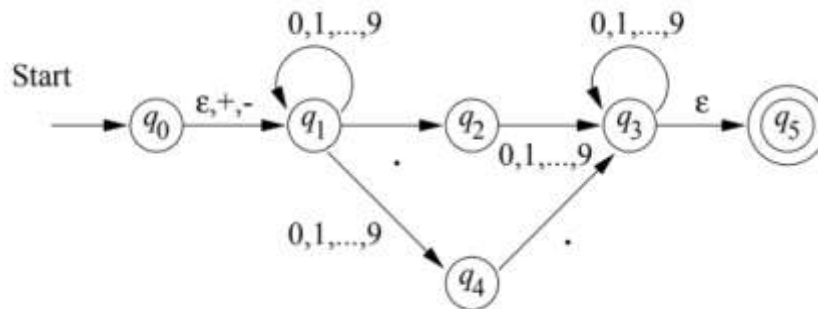
Al contrario degli automi a stati finiti deterministici, gli NFA possono cambiare stato indipendentemente dal simbolo letto, tramite epsilon-transizioni. Gli automi che presentano questo tipo di transizioni sono anche detti  $\epsilon$ -NFA.

Un  $\epsilon$ -NFA è una quintupla  $(Q, \Sigma, \delta, q_0, F)$  dove  $\delta$  è una funzione da  $Q \times \Sigma \cup \{\epsilon\}$  all'insieme dei sottoinsiemi di  $Q$ .

Ad esempio, un  $\epsilon$ -NFA che accetta numeri decimali consiste di:

1. un segno + o -, opzionale
2. una stringa di cifre decimali
3. un punto decimale
4. un'altra stringa di cifre decimali

La stringa descritta al punto 2 o quella descritta al punto 4 sono opzionali.



$$E = (\{q_0, q_1, \dots, q_5\}, \{., +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, q_0, \{q_5\})$$

in cui la tabella delle transizioni è:

	$\epsilon$	$+, -$	$.$	$0, \dots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\emptyset$	$\{q_1\}$	$\{q_2\}$
$\{q_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_1, q_4\}$
$\{q_3\}$	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
$\{q_4\}$	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$* \{q_5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

## Epsilon-chiusura

Una epsilon-chiusura di uno stato  $q$  è denominata  $ECLOSE(q)$  ed è definita induttivamente:

BASE:  $q \in ECLOSE(q)$

INDUZIONE:  $p \in ECLOSE(q)$  and  $r \in \delta(p, \epsilon) \Rightarrow r \in ECLOSE(q)$

Esempio di epsilon-chiusura:

$$ECLOSE(1) = \{1, 2, 3, 4, 6\}$$

$$ECLOSE(5) = \{5, 7\}$$

Definizione induttiva di  $\hat{\delta}$  per automi  $\square$ -NFA

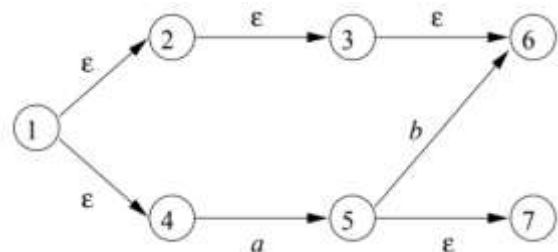
BASE:  $\hat{\delta}(q, \epsilon) = ECLOSE(q)$

INDUZIONE:  $\hat{\delta}(q, xa) = \bigcap_{p \in \hat{\delta}(q, x)} \left( \bigcap_{t \in \hat{\delta}(p, a)} ECLOSE(t) \right)$

Un linguaggio  $L$  accettato è definito  $L = \{w: \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

## Equivalenza di DFA e $\epsilon$ -NFA

Dato un  $\epsilon$ -NFA  $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$  è possibile costruire un DFA  $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$  tale che  $L(D) = L(E)$ . Si ha:



- $Q_D = \{S: S \subseteq Q_E \text{ e } S = \cap_{s \in S} ECLOSE(s)\}$
- $q_D = ECLOSE(q_0)$
- $F_D = \{S: S \in Q_D \text{ e } S \cap F_E \neq \emptyset\}$
- $\delta_D(S, a) = \cap_{t \in S} (\cap_{p \in \delta_E(t, a)} ECLOSE(p))$

## Teorema 2.22

Un linguaggio  $L$  è accettato da un  $\square$ -NFA se e solo se  $L$  è accettato da un DFA.

Equivalenze tra automi

	Insieme degli stati	Alfabeto	Funzione di transizione	Stato iniziale	Insieme degli stati finali
<b>NFA</b>	$Q_N$	$\Sigma$	$\delta_N$	$q_0$	$F_N$
<b><math>\epsilon</math>-NFA</b>	$Q_E$	$\Sigma$	$\delta_E$	$q_0$	$F_E$
<b>DFA</b>	$Q_D =$ $= \{S: S \subseteq Q_N\}$ $= \{S: S \subseteq Q_E \text{ e } S$ $= \cap_{s \in S} ECLOSE(s)\}$	$\Sigma$	$\delta_D =$ $= \cap_{p \in S} \delta_N(p, a)$ $\delta_D(S, a) =$ $= \cap_{t \in S} (\cap_{p \in \delta_E(t, a)} ECLOSE(p))$	$\{q_0\}$	$F_D =$ $= \{S \subseteq Q_N: S \cap F_N \neq \emptyset\}$ $= \{S \subseteq Q_D: S \cap F_E \neq \emptyset\}$

## Espressioni regolari

Una espressione regolare (in lingua inglese regular expression o, in forma abbreviata, regexp, regex o RE) è una sequenza di simboli (quindi una stringa) che identifica un insieme di stringhe. Programmi diversi supportano notazioni diverse per esprimere le stesse espressioni regolari, pertanto non esiste una sintassi "universale".

Le espressioni regolari possono definire tutti e soli i linguaggi regolari. Il teorema di Kleene afferma che la classe dei linguaggi regolari corrisponde alla classe dei linguaggi generati da grammatiche di tipo 3 (nella gerarchia di Chomsky) e riconosciuti da automi a stati finiti. Tuttavia, nella pratica esistono taluni costrutti (ad esempio i costrutti di backreference)[1] che permettono di ampliare l'insieme di linguaggi definibili.

[WIKIPEDIA]

Le espressioni regolari esprimono le stesse classi di linguaggi riconosciuti dai DFA.

## Operazioni sui linguaggi

Dati i linguaggi  $L$  e  $M$  si definiscono:

**UNIONE**  $L \cup M = \{w: w \in L \text{ o } w \in M\}$

**CONCATENAZIONE**  $L.M = \{w: w = xy, x \in L, y \in M\}$

**POTENZA**  $L^0 = \{\epsilon\}$   $L^1 = L$   $L^{k+1} = L.L^k$

**CHIUSURA DI KLEENE**  $L^* = \cap_{i=0}^{\infty} L^i$

Nota:  $\emptyset^0 = \{\epsilon\}$ ;  $\emptyset^1 = \emptyset \forall i$ ;  $\emptyset^* = \{\epsilon\}$

**Proprietà:**

- $(L \cup M).R = (L.R) \cup (M.R)$
- $R.(L \cup M) = (R.L) \cup (R.M)$
- $L.M \neq M.L$
- $L \cup M = M \cup L$

## Costruire espressioni regolari, definizione

Definizione induttiva di espressioni regolari:

BASE:

$\epsilon$  e  $\emptyset$  sono espressioni regolari

$$L(\epsilon) = \{\epsilon\} \quad L(\emptyset) = \emptyset$$

Se  $a \in \Sigma$  allora  $a$  è un'espressione regolare

$$L(a) = \{a\}$$

INDUZIONE:

Se  $E$  è un'espressione regolare, allora  $(E)$  è un'espressione regolare

$$L((E)) = L(E)$$

Se  $E$  e  $F$  sono espressioni regolari, allora  $E + F$  è un'espressione regolare

$$L(E + F) = L(E) \cup L(F)$$

Se  $E$  e  $F$  sono espressioni regolari, allora  $E \cdot F$  è un'espressione regolare

$$L(E \cdot F) = L(E) \cdot L(F)$$

Se  $E$  è un'espressione regolare, allora  $E^*$  è un'espressione regolare

$$L(E^*) = (L(E))^*$$

Esempio

Espressioni regolari per  $L = \{w \in \{0,1\}^* : 0 \text{ e } 1 \text{ alternati in } w\}$

$(01)^* + (10)^* + 0(10)^* + 1(01)^*$  o, equivalentemente:  $(\epsilon + 1)(01)^*(\epsilon + 0)$

L'ordine di precedenza degli operatori è:

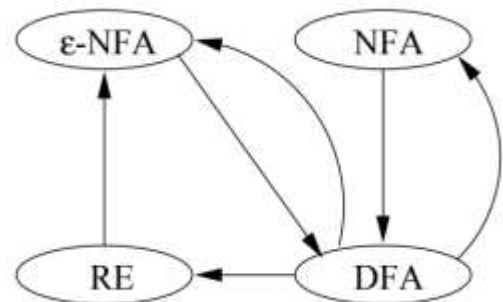
1. Chiusura
2. Concatenazione (punto)
3. Più (+)

Esempio:  $01^* + 1$  è raggruppato in  $(0(1)^*) + 1$

## Equivalenza di automi a stati (FA) e espressioni regolari

Per dimostrare che gli automi a stati sono equivalenti alle espressioni regolari deve essere stabilito che:

1. Per ogni DFA  $A$  è possibile trovare (costruire) un'espressione regolare  $R$ , tale che  $L(R) = L(A)$
2. Per ogni espressione regolare  $R$  esiste un  $\epsilon$ -NFA  $A$ , tale che  $L(A) = L(R)$

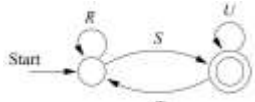


### Teorema 3.4

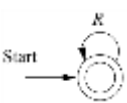
Per ogni DFA  $A = (Q, \Sigma, \delta, q_0, F)$  esiste una espressione regolare  $R$  tale che  $L(R) = L(A)$

Automa	Espressione regolare
Si trasforma un automa etichettando gli archi con espressioni regolari di simboli	Si eliminano gli stati intermedi (in questo caso $s$ ) e si costruiscono le espressioni di cambio stato

Per ogni stato  $q \in F$  si ha un automa

$A_Q$  della forma  che corrisponde all'espressione regolare  $E_Q = (R + SU^*T)^*SU^*$

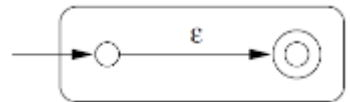
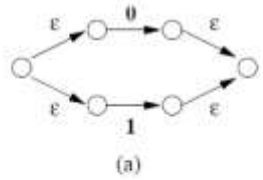
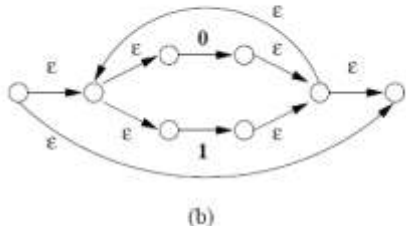
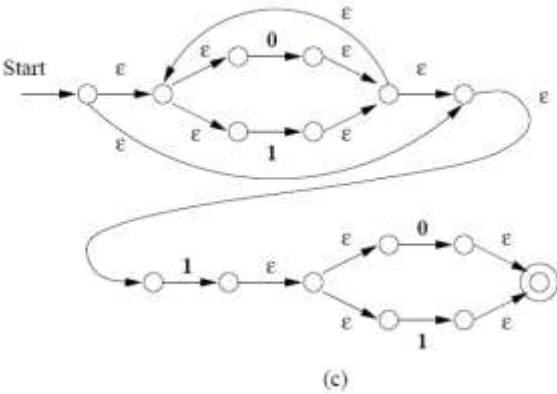

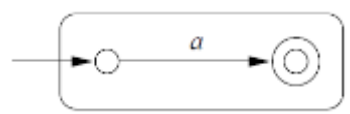
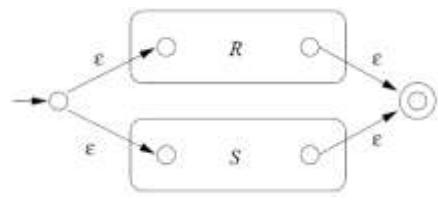
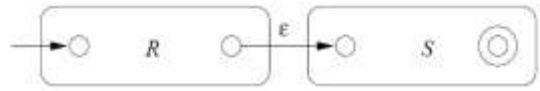
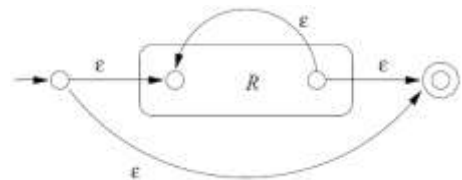
oppure

$A_Q$  della forma  che corrisponde all'espressione  $E_Q = R^*$ .

L'espressione finale è:  $\bigoplus_{q \in F} E_Q$

## Da espressioni regolari a $\square$ -NFA

**Teorema 3.7:** Per ogni espressione regolare  $R$  possiamo costruire un  $\square$ -NFA  $A$  (con un unico stato finale diverso da quello iniziale) tale che  $L(A) = L(R)$ .

Automa	Costruzione	Esempio: Trasformazione per $(0+1)^*1(0+1)$
$\square$		 <p>(a)</p>  <p>(b)</p>  <p>(c)</p>
$\emptyset$		
$a$		
$R + S$		
$RS$		
$R^*$		



## Leggi algebriche per i linguaggi

$L \cup M = M \cup L$	L'unione è commutativa
$(L \cup M) \cup N = L \cup (M \cup N)$	L'unione è associativa
$(LM)N = L(MN)$	La concatenazione è associativa
$LM \neq ML$	La concatenazione non è commutativa
$\emptyset \cup L = L \cup \emptyset = L$	$\emptyset$ è l'identità per l'unione
$\{\epsilon\}L = L\{\epsilon\} = L$	$\{\epsilon\}$ è l'identità sinistra e destra per la concatenazione
$\emptyset L = L\emptyset = \emptyset$	$\emptyset$ è l'annichilatore sinistro e destro per la concatenazione
$L(M \cup N) = LM \cup LN$	La concatenazione è distributiva a sinistra sull'unione
$(M \cup N)L = ML \cup NL$	La concatenazione è distributiva a destra sull'unione
$L \cup L = L$	L'unione è idempotente
$\emptyset^* = \{\epsilon\}, \{\epsilon\}^* = \{\epsilon\}$	
$L^+ = LL^* = L^*L,$ $L^* = L^+ \cup \{\epsilon\}$	
$(L^*)^* = L^*$	

## Proprietà dei linguaggi regolari

### Pumping Lemma

Ogni linguaggio regolare soddisfa il **pumping lemma**.

Dato un linguaggio, se usando il pumping lemma si ottiene una contraddizione allora il linguaggio non è regolare.

*Supponendo, per assurdo, che  $L = \{0^n 1^n : n \geq 0\}$  sia regolare.*

*L contiene tutte le stringhe del tipo 01, 0011, 000111, ...*

*Allora deve essere accettato da un qualche DFA A. Sia k il numero degli stati di A.*

*Se A riceve in input una stringa di lunghezza  $\geq k$*

*allora, lungo il cammino di riconoscimento, incontra due volte uno stesso stato.*

In altre parole, una stringa di lunghezza superiore agli stati dell'automa genera problemi di memoria in quanto l'automa, ad un certo punto della sua valutazione, si ritrova in uno stato precedentemente raggiunto senza sapere di averlo già raggiunto.

Avendo l'automa un numero finito k di stati si ha che, quando riceve una stringa, incontra gli stati:  $\epsilon \rightarrow p_0$  (stato iniziale),  $0 \rightarrow p_1$ ,  $00 \rightarrow p_2, \dots, 0^k \rightarrow p_k$ ; si osserva che gli stati incontrati sono k + 1 pertanto devono

necessariamente esistere due indici  $i, j, i < j: p_i = p_j$  ovvero due stringhe di lunghezze diverse che portano allo stesso stato  $q$  (generando di fatto un ciclo).

Ora, se l'automa va in  $q$  con  $0^i$  e con  $0^j$ , dovendo l'automa accettare la stringa  $0^i 1^i$  sarà tale che  $\delta(q, 1^i) \in F$ , allora è accettata anche la stringa  $0^j 1^i$  con  $i < j$  (**ASSURDO**).

In altre parole, siccome l'automa raggiungerebbe un certo stato con almeno due sequenze diverse di 0 (lunghe rispettivamente  $i$  e  $j$  caratteri), se il linguaggio fosse regolare accadrebbe che l'automa dovrebbe accettare sia sequenze di  $i$  caratteri 1 che di  $j$  caratteri 1, ovvero il linguaggio conterrebbe stringhe non hanno lo stesso numero di 0 e di 1. Pertanto  $L$  non può essere un linguaggio regolare.

#### Teorema 4.1: Pumping Lemma per Linguaggi Regolari

*Sia  $L$  un linguaggio regolare.*

*Allora  $\exists n \geq 1$  che soddisfa  $\forall w \in L: |w| \geq n$  è scomponibile in tre stringhe  $w = xyz$  tali che*

1.  $y \neq \epsilon$
2.  $|xy| \leq n$
3.  $\forall k \geq 0, xy^kz \in L$

#### Dimostrazione

*Supponendo che  $L$  sia regolare, allora  $L$  è riconosciuto da un DFA  $A$ .*

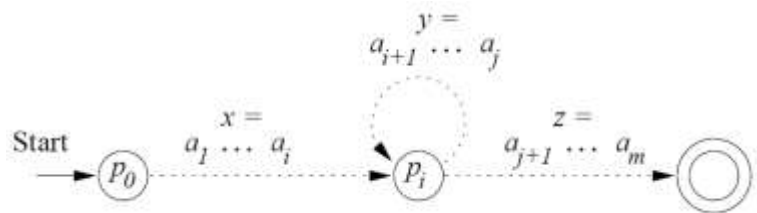
*Sia  $n$  il numero degli stati di  $A$ . Sia  $w = a_1 a_2 \dots a_m \in L, m \geq n$ .*

*Sia  $p_i = \delta(q_0, a_1 a_2 \dots a_i)$  dove  $q_0$  è lo stato iniziale di  $A$ .*

$\Rightarrow \exists i < j \leq n: p_i = p_j$

*Prendendo  $w = xyz$  con:*

1.  $x = a_1 a_2 \dots a_i$
2.  $y = a_{i+1} a_{i+2} \dots a_j$
3.  $z = a_j a_{j+1} \dots a_m$



*Pertanto anche  $xy^kz \in L$ , per ogni  $k \geq 0$*

#### Applicazione

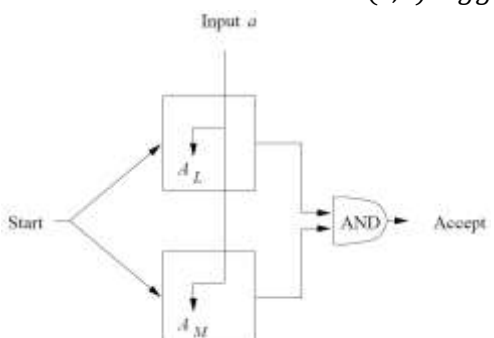
*Sia  $L_{eq} = \{0^n 1^n: n \geq 1\}$ , per il pumping lemma,  $w = xyz$  e  $|w| \geq n$ , in particolare modo  $|w| = 2n$ , con  $n \geq 0$  e  $n \geq 1$*

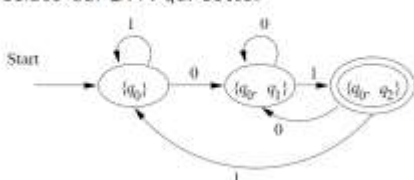
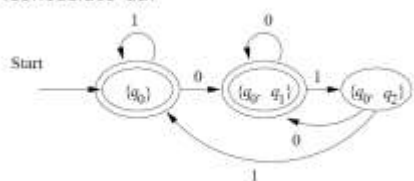
1.  $y \neq \epsilon$
2.  $|xy| \leq n \Rightarrow y = 0^k$  per un qualche valore di  $k$
3.  $\forall k \geq 0, xy^kz \in L_{eq} \Rightarrow$  dato  $k = 0$ , ad esempio,  $xz \in L_{eq}$ , ma  $|x| < n$  e  $|z| = n, \Rightarrow xz \notin L_{eq}$  **ASSURDO!!!**

## Proprietà di chiusura

Permette di definire come costruire automi da componenti usando operazioni e definisce quando non è possibile farlo.

*Siano  $L$  e  $M$  due linguaggi regolari. I seguenti linguaggi sono tutti regolari:*

<i>Unione: <math>L \cup M</math></i>	<i>Sia <math>L = L(E)</math> e <math>M = L(F)</math>, allora <math>L(E + F) = L \cup M</math></i>
<i>Intersezione: <math>L \cap M</math></i>	<p><i>Per la legge di De Morgan: <math>L \cap M = \underline{\underline{L \cup M}}</math></i></p> <p><i>Prova alternativa:</i></p> <p><i>Sia <math>L</math> il linguaggio di <math>A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)</math></i>  <i>e <math>M</math> il linguaggio di <math>A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)</math></i></p> <p><i>Si assume che entrambi gli automi siano deterministici</i></p> <p><i>Si procede costruendo un automa che simula <math>A_L</math> e <math>A_M</math></i>  <i>in parallelo e accetta se e solo se entrambi accettano.</i></p> <p><i>Se <math>A_L</math> va dallo stato <math>p</math> allo stato <math>s</math> leggendo <math>a</math> e <math>A_M</math> va dallo stato <math>q</math> allo stato <math>t</math> leggendo <math>a</math>, allora <math>A_{L \cap M}</math> andrà dallo stato <math>(p, q)</math> allo stato <math>(s, t)</math> leggendo <math>a</math>.</i></p>  <p><i>Formalmente <math>A_{L \cap M} = (Q_L \times Q_M, \Sigma, \delta_{L \cap M}, (q_L, q_M), F_L \times F_M)</math></i>  <i>dove <math>\delta_{L \cap M}((p, q), a) = (\delta_L(p, a), \delta_M(q, a))</math></i>  <i>Si può dimostrare per induzione su <math> w </math> che:</i>  <i><math>\hat{\delta}_{L \cap M}((q_L, q_M), w) = (\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w))</math></i></p>
<i>Complemento: <math>\underline{L}</math></i>	<p><i><math>L</math> linguaggio regolare su <math>\Sigma</math>, allora <math>\underline{L} = \Sigma^* \setminus L</math></i></p> <p><i>Dato l'automata <math>A = (Q, \Sigma, \delta, q_0, F)</math></i></p> <p><i>Sia l'automata <math>B = (Q, \Sigma, \delta, q_0, Q \setminus F)</math></i></p> <p><i>Allora <math>L(B) = \underline{L}</math></i></p>

	<p>Esempio:                  Sia <math>L</math> riconosciuto dal DFA qui sotto:</p>  <p>Allora <math>\bar{L}</math> è riconosciuto da:</p>  <p><b>Domanda:</b> Espressione regolare per <math>\bar{L}</math> a partire da una per <math>L</math>?</p>
--	---

<i>Differenza: <math>L \setminus M</math></i>	$L \setminus M$ $= L \cap \underline{M}$ , sappiamo che i linguaggi regolari sono chiusi rispetto a complemento e intersezione
<i>Inversione: <math>L^R = \{w^R : w \in L\}</math></i>	$L^R$ è il linguaggio che riconosce le stringhe lette a "rovescio" (es. $\text{pippo} \in L \Rightarrow \text{oppip} \in L^R$ ) Allora, dato un FA $A$ che riconosce $L$ :: 1. Si girano tutti gli archi 2. Lo stato iniziale diventa l'unico stato finale 3. Si crea un nuovo stato iniziale $p_0$ con $\delta(p_0, \epsilon) = F$
<i>Chiusura: <math>L^*</math></i>	
<i>Concatenazione: <math>L.M</math></i>	

## Proprietà di decisione

Analisi computazionale di automi (es. valutare se due automi sono equivalenti).

### 1) Come convertire tra diverse rappresentazioni dei linguaggi regolari?

Complessità:

- Da NFA a DFA, supponendo che un  $\epsilon$ -NFA abbia  $n$  stati, il DFA ha  $2^n$  stati: **esponenziale**
- Da DFA a NFA: **lineare**
- Da FA a espressioni regolari: **esponenziale** (ogni volta che si elimina uno stato la lunghezza delle etichette quadruplica)
- Da espressioni regolari a FA: **lineare**; **esponenziale** se si vuole ottenere un DFA.

### 2) E' $L = \emptyset$ ? Testare se un linguaggio è vuoto

$L(A)$

$\neq \emptyset$  per FA  $A$  se e solo se uno stato finale è raggiungibile dallo stato iniziale in  $A$  (totale  $O(n^2)$  passi).

In alternativa, osservando un'espressione regolare  $E$  di lunghezza  $s$  è possibile verificare se  $L(E) = \emptyset$  in  $s$  passi.

$E = F + G \Rightarrow L(E)$  è vuoto se e solo se sia  $L(F)$  che  $L(G)$  sono vuoti.

$E = F.G \Rightarrow L(E)$  è vuoto se e solo se  $L(F)$  o  $L(G)$  sono vuoti.

$E = F^* \Rightarrow L(E)$  non è mai vuoto in quanto  $\epsilon \in L(E)$ .

$E = \epsilon \Rightarrow L(E)$  non è vuoto.

$E = a \Rightarrow L(E)$  non è vuoto.

$E = \emptyset \Rightarrow L(E)$  è vuoto.

### 3) E' $w \in L$ ? Controllare l'appartenenza

Per controllare se  $w \in L(A)$  per DFA  $A$ , si simula  $A$  su  $w$ . Se  $|w| = n$ , la simulazione richiede  $O(n)$  passi.

Se  $A$  è un NFA e ha  $s$  stati, simulare  $A$  su  $w$  richiede  $O(ns^2)$  passi. Lo stesso per  $\epsilon$ -NFA calcolando preventivamente le ECLOSE di tutti gli stati.

Se  $L$

$= L(E)$  per l'espressione regolare  $E$  di lunghezza  $s$ , prima si converte  $E$  in un  $\epsilon$ -NFA con  $2s$  stati, poi

si simula  $w$  sull'automa convertito, in  $O(ns^2)$  passi.

### 4) Due descrizioni definiscono uno stesso linguaggio?

L'algoritmo riempi-tabella permette di individuare le classi di equivalenza nell'insieme degli stati.

Può essere utilizzato sia per valutare l'equivalenza di due descrizioni (due automi) sia per valutare la possibilità di minimizzare un automa e guidare nella minimizzazione.

L'algoritmo è definito in maniera induttiva:

**Passo base:**

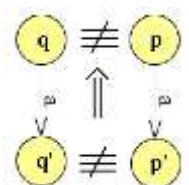
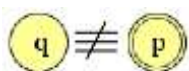
$p \neq q$  ricordando  $F = \{\text{insieme stati finali}\}$

Se  $p \in F$  e  $q \notin F$  allora

**Induzione:**

$\Sigma: \delta(p, a) \neq \delta(q, a)$  allora  $p \neq q$

Se  $\exists a \in \Sigma$



Quindi, se due stati sono uno finale e un non finale, i due stati sono distinti.

Se le due transizioni che partono da due stati, con lo stesso carattere, arrivano a due stati distinti, i due stati di partenza sono distinti.

Per applicare l'algoritmo riempi-tabella in pratica si può descrivere una parte di tabella che permetta di identificare tutte le combinazioni degli stati dell'automa (degli automi).

Se ad esempio gli stati sono A, B, C, D, E, F, G, H, I e gli stati finali sono D, F, H, allora:

B									Per il basso base si segnano, come distinti, tutte le coppie di stati finali / non finali ( $X^1$ )
C									
D	$X^1$	$X^1$	$X^1$						Per il passo di induzione, si valutano, per ogni coppia di stati, le coppie di stati di arrivo per le transizioni con lo stesso simbolo. Se la casella identificativa della coppia dei due stati di arrivo è già stata segnata come distinta ( $X^1$ ) si segna come distinta anche la casella degli stati di partenza ( $X^j$ , j-esimo ciclo sulla tabella)
E				$X^1$					
F	$X^1$	$X^1$	$X^1$		$X^1$				
G				$X^1$		$X^1$			
H	$X^1$	$X^1$	$X^1$		$X^1$		$X^1$		
I				$X^1$		$X^1$		$X^1$	Quando si arriva effettua un ciclo completo sulle caselle vuote, senza aggiungere alcun simbolo, l'algoritmo riempi tabella termina.
	A	B	C	D	E	F	G	H	

I cicli scorrono le caselle dall'alto al basso e da sinistra verso destra.

Al termine della parte di compilazione della tabella, tutte le coppie contraddistinte da un simbolo definiscono gli stati distinti, tutte le coppie senza simbolo definiscono simboli indistinti.

**Teorema 4.20:** *Se  $p$  e  $q$  non sono distinguibili dall'algoritmo allora  $p \equiv q$ .*

**Dimostrazione:**

*Si suppone per assurdo che esiste una coppia sbagliata  $\{p, q\}$  tale che*

1.  $\exists w: \hat{\delta}(p, w) \in F$  e  $\hat{\delta}(q, w) \notin F$  o viceversa
2. *L'algoritmo non distingue tra  $p$  e  $q$*

Si assume che  $\{p, q\}$  sia la coppia sbagliata con la stringa più corta  $w = a_1 a_2 \dots a_n$  che la identifica come tale.

Allora:

$w$

$\neq \epsilon$ , altrimenti l'algoritmo distinguerebbe  $p$  da  $q$  per il caso base ( $p$  è finale o non finale e allo stesso modo  $q$ ).

Quindi  $n \geq 1$ .

Considerando gli stati  $r = \hat{\delta}(p, a_1)$  e  $s$

$= \hat{\delta}(q, a_1)$  si ha che  $\{r, s\}$  non può essere una coppia sbagliata perché sarebbe

identificata dalla stringa  $a_2 a_3 \dots a_n$  che è più corta di  $w$ . Quindi l'algoritmo deve aver scoperto che  $r$  e  $s$  sono dist

Allora, per l'induzione, l'algoritmo distinguerebbe  $p$  da  $q$  pertanto non ci sono coppie sbagliate.

**Verifica equivalenza dei linguaggi definiti da due automi**

Applicare l'algoritmo riempi-tabella sull'insieme somma degli stati di due automi, se al termine dell'algoritmo si rileva che gli stati iniziali dei due automi sono distinguibili allora i due automi non definiscono lo stesso linguaggio, altrimenti il linguaggio è lo stesso.

Siano L e M due linguaggi regolari (definiti in qualche forma)

1. Si convertono i due linguaggi in DFA
2. Si applica l'algoritmo riempi-tabella all'unione degli insiemi degli stati dei due automi
3. Se al termine l'algoritmo distingue i due stati iniziali dei due automi allora i due linguaggi sono distinti, viceversa sono equivalenti.

Esempio:

**DFA Linguaggio 1**

**DFA Linguaggio 2**

Algoritmo riempi tabella:

B	X <sup>1</sup>			
C*		X <sup>1</sup>		
D*		X <sup>1</sup>		
E	X <sup>1</sup>		X <sup>1</sup>	X <sup>1</sup>
	A*	B	C*	D*

\*Stati finali

L'algoritmo termina dopo il primo ciclo.

Le coppie di stati iniziali sono indistinte.

Quindi i due linguaggi sono equivalenti.

## Tecniche di minimizzazione

Costruzione di automi più piccoli senza ridondanze.

Applicando l'algoritmo riempi-tabella si identificano le classi di equivalenza degli stati indistinguibili dell'automata. Si può costruire un automa minimizzato con gli stati limitati alle classi di equivalenza individuate.

Le classi di equivalenza sono definite dagli stati che, nella tabella, sono indistinguibili.

Se q e r sono indistinti, la loro classe di equivalenza contiene q e r, e tutti gli stati non distinguibili da q e da r e via via iterativamente.

Per minimizzare un DFA dopo l'applicazione dell'algoritmo riempi-tabella, si costruisce un nuovo DFA con uno stato per ogni classe di equivalenza individuata. Si identifica come stato iniziale quello della classe di equivalenza che contiene lo stato iniziale del DFA non minimizzato. Si definiscono le transizioni per ogni stato (per un certo simbolo dell'alfabeto), osservando la transizione per quel simbolo di un qualsiasi stato appartenente alla classe di equivalenza per identificare la classe di arrivo (classe di equivalenza a cui appartiene lo stato di arrivo).

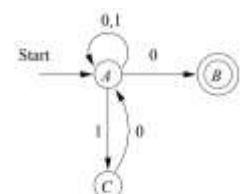
Quindi, per minimizzare un  $DFA = (Q, \Sigma, \delta, q_0, F)$  dove si assume di aver già eliminato tutti gli stati non raggiungibili dallo stato iniziale, si costruisce un DFA  $B = (Q/\equiv, \Sigma, \gamma, q_0/\equiv, F/\equiv)$  dove

$$\gamma(p/\equiv, a) = \delta(p, a)/\equiv$$

$Q/\equiv$  è l'insieme delle classi di equivalenza degli stati di Q.

$F/\equiv$  è l'insieme delle classi di equivalenza degli stati finali.

Se  $\delta(p, a) \not\equiv \delta(q, a)$  l'algoritmo conclude che  $p \not\equiv q$  si ha che se  $p \equiv q \Rightarrow \delta(p, a) \equiv \delta(q, a)$  e il DFA B è ben definito.



L'algoritmo di minimizzazione non può essere applicato agli NFA, per esempio lo stato C del NFA a fianco è evidentemente eliminabile, tuttavia l'algoritmo lo identificherebbe come distinto rispetto ad  $(A \rightarrow_0 B \text{ e } C \dashrightarrow_0 A)$ .

### Perché non si può migliorare un DFA minimizzato

Se il DFA B è minimizzato applicando l'algoritmo al DFA A, allora si sa che  $L(A) = L(B)$ .

Potrebbe esistere un DFA C tale che  $L(C) = L(B)$  che abbiamo meno stati di B?

Applicando l'algoritmo a B "unito con" C, dato che  $L(B) = L(C)$  si avrà che lo stato iniziale è non distinto per i due DFA:  $q_0^B \equiv q_0^A$ .

Inoltre, per ogni stato p in B esiste almeno uno stato q in C tale che  $q \equiv p$ ,

Infatti non ci sono stati raggiungibili, pertanto:  $p = \hat{\delta}(q_0^B, a_1 a_2 \dots a_k)$  per una qualche stringa  $a_1 a_2 \dots a_k$  e considerando  $p = \hat{\delta}(q_0^C, a_1 a_2 \dots a_k)$  si ha che  $q \equiv p$ .

Assumendo che C abbia meno stati di B, devono esistere due stati r e s di B tali che  $r \equiv t \equiv s$  per qualche stato t di C. Ma sapendo che B è stato costruito con l'algoritmo,  $r \equiv s$  è una contraddizione.



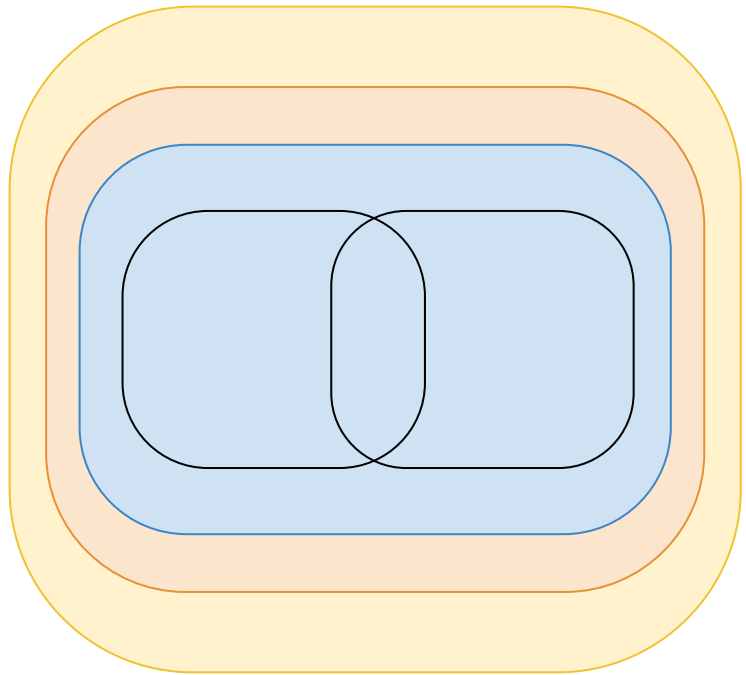
## Grammatiche e Linguaggi Liberi dal Contesto

Non tutti i linguaggi sono regolari, esistono classi più grandi di linguaggi.

I linguaggi liberi dal contesto (CFL, Context Free Languages) sono stati utilizzati per lo studio dei linguaggi naturali dal 1950 e nello studio dei compilatori dal 1960.

In generale, con il termine **grammatica** si intende un formalismo che permette di definire un insieme di stringhe mediante l'imposizione di un particolare metodo per la loro costruzione. Tale costruzione si effettua partendo da una stringa particolare e riscrivendo via via parti di essa secondo una qualche regola specificata nella grammatica stessa.

Le grammatiche libere dal contesto (CFG, Context Free Grammar) sono la base per le sintassi BNF (Backus-Naur-Form). Le grammatiche servono a definire i linguaggi.



### Esempio informale di CFG

Considerando il linguaggio delle stringhe palindromo:  $L_{pal} = \{w \in \Sigma^* : w = w^R\}$ . Dove  $w^R$  è la stringa ottenuta applicando la funzione  $R$ , reverse, alla stringa  $w$ . La funzione reverse inverte l'ordine dei caratteri nella stringa.

Parole come *otto*, *ara*, ... appartengono al linguaggio.

Limitando l'alfabeto ai caratteri 0 e 1, supponendo che  $L_{pal}$  sia regolare allora - per il pumping lemma - dato  $n$ , si ha che  $w = 0^n 10^n \in L_{pal}$  e  $w = xyz : |xy| \leq n \Rightarrow xy = 0^n \Rightarrow y = 0$ , quindi  $xy^kz$  deve  $\in L_{pal}$

ma esiste certamente un valore di  $k$  per cui  $xy^kz = 0^{n+k}10^n$  che non appartiene ad  $L$ .

Una possibile definizione, induttiva, di  $L_{pal}$  è (induttivamente):

**BASE:**

$\epsilon$ , 0, 1 sono palindromi

**INDUZIONE:**

se  $w$  è palindromo, allora

$0w0$  e  $1w1$  sono palindromi.

Nessuna altra stringa è palindromo.

Le CFG sono un modo formale per definire un linguaggio, ad esempio  $L_{pal}$ :

1.  $P \rightarrow \epsilon$
2.  $P \rightarrow 0$
3.  $P \rightarrow 1$
4.  $P \rightarrow 0P0$
5.  $P \rightarrow 1P1$

queste 5 regole (produzioni) descrivono che, 0 e 1 sono simboli terminali,  $P$  è una variabile (non terminale o categoria sintattica),  $P$  è - in questa grammatica - anche il simbolo iniziale.

## Definizione formale di CFG

Una **grammatica libera dal contesto** è una quadrupla  $G = (V, T, P, S)$  dove:

$V$  è un insieme finito e non vuoto di variabili

$T$  è un insieme finito e non vuoto di simboli terminali

$P$  è un insieme finito di produzioni della forma  $A \rightarrow \alpha \in (V \cup T)^*$  dove  $A$  è una variabile e

$S$  è una variabile distinta chiamata simbolo iniziale.

### Esempio CFG per $L_{pal}$

$$G_{pal} = (\{P\}, \{0,1\}, A, P)$$

con  $A = \{P \rightarrow \epsilon, P \rightarrow 0, P \rightarrow 1, P \rightarrow 0P0P \rightarrow 1P1\}$ , in forma contratta:  $A = \{P \rightarrow \epsilon | 0|1|0P0P|1P1\}$

### Esempio CFG per un semplice linguaggio di programmazione

La grammatica per un linguaggio che contenga gli operatori  $+$  e  $*$ , in cui gli operandi sono *identificatori* cioè stringhe in  $L = ((a + b)(a + b + 0 + 1)^*)$  (ovvero stringhe che iniziano con la lettera  $a$  o con la lettera  $b$  e che sono seguite da una serie di  $a$ ,  $b$ ,  $0$  e/o  $1$ ).

La grammatica che definisce queste espressioni è:

$G = (\{E, I\}, \{+, *, (, ), a, b, 0, 1\}, P, E)$  le produzioni che appartengono all'insieme  $P$  sono:

1.  $E \rightarrow I$
2.  $E \rightarrow E + E$
3.  $E \rightarrow E * E$
4.  $E \rightarrow (E)$
5.  $I \rightarrow a$
6.  $I \rightarrow b$
7.  $I \rightarrow Ia$
8.  $I \rightarrow Ib$
9.  $I \rightarrow I0$
10.  $I \rightarrow I1$

## Derivazioni

Sia  $G = (V, T, P, S)$  una CFG,  $A \in V, \{\alpha, \beta\} \subset (V \cup T)^*, e A \rightarrow \gamma \in P$ .

Allora una derivazione si scrive  $\alpha A \beta \Rightarrow_G \alpha \gamma \beta$  o (se ovvia la  $G$ )  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  e si dice che  $\alpha \gamma \beta$  si deriva da  $\alpha A \beta$ .

$\Rightarrow^*$  è definita come la chiusura riflessiva e transitiva di  $\Rightarrow$ , ovvero:

BASE: sia  $\alpha \in (V \cup T)^*$ , allora  $\alpha \Rightarrow^* \alpha$

INDUZIONE: sia  $\alpha \Rightarrow^* \beta$  e  $\beta \Rightarrow \gamma$  allora  $\alpha \Rightarrow^* \gamma$

### Notazione per le derivazioni CFG

Esistono diverse convenzioni abitualmente utilizzate per aiutare a ricordare le regole dei simboli quando si parla di CFG.

Quella utilizzata nel libro e negli appunti prevede:

1. che le prime lettere dell'alfabeto, minuscole, siano utilizzate come simboli terminali (a, b, ...); simboli, cifre e altri caratteri quali +, -, le parentesi sono solitamente simboli terminali;
2. che le prime lettere dell'alfabeto, MAIUSCOLE, siano utilizzate come variabili
3. che le ultime lettere dell'alfabeto, minuscole, siano utilizzate come stringhe di terminali
4. che le ultime lettere dell'alfabeto, MAIUSCOLE, siano utilizzate come terminali che come variabili
5. che le lettere dell'alfabeto greco,  $\alpha$ ,  $\beta$ , ... , siano utilizzate come stringhe che contengono terminali e/o variabili.

Un esempio di derivazione, nel linguaggio visto sopra, per la stringa  $a * (a + b00)$  può essere:

$$\begin{aligned} E &\Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow a * (E) \Rightarrow a * (E + E) \Rightarrow a * (a + E) \Rightarrow a * (a + I) \Rightarrow a * (a + I0) \\ &\Rightarrow a * (a + I00) \Rightarrow \\ &\Rightarrow a * (a + b00) \end{aligned}$$

Si osservi che ad ogni passo potrebbero esserci diverse possibili derivazioni che portano allo stesso risultato.

Si osservi anche che non tutte le scelte determinano derivazioni valide per una certa stringa.

### Derivazioni a sinistra e a destra (leftmost e rightmost derivations)

Le derivazioni a sinistra rimpiazzano sempre la variabile più a sinistra.

Le derivazioni a destra rimpiazzano sempre la variabile più a destra.

Esempio:

**Derivazioni a sinistra:**  $E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow a * (E) \Rightarrow a * (E + E) \Rightarrow a * (a + E) \Rightarrow a * (a + I) \Rightarrow$   
 $\Rightarrow a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow a * (a + b00)$

**Derivazione a destra:**  $E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (E + I) \Rightarrow E * (E + I0) \Rightarrow E * (E + I00) \Rightarrow$   
 $\Rightarrow E * (E + b00) \Rightarrow E * (a + b00) \Rightarrow a * (a + b00)$

### Il Linguaggio di una grammatica

*Sia  $G(V, T, P, S)$  una CFG, allora il LINGUAGGIO DI  $G$  è  $L(G) = \{w \in T^* : S \Rightarrow_G^* w\}$*

Il linguaggio corrisponde cioè all'insieme di tutte le stringhe su  $T^*$  derivabili dal simbolo iniziale.

Se  $G$  è una CFG, allora  $L(G)$  è un **linguaggio libero dal contesto**.

### Forme sentenziali

*Sia  $G(V, T, P, S)$  una CFG e  $\alpha \in (V \cup T)^*$ , se  $S \Rightarrow^* \alpha$  allora  $\alpha$  è una forma sentenziale.*

*$L(G)$  contiene le forme sentenziali che sono in  $T^*$ .*

Esempio, nella grammatica delle espressioni,  $E * (I + E)$  è una forma sentenziale:

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

## Alberi sintattici

Gli alberi sintattici sono rappresentazioni estremamente utili per le derivazioni, mostrano chiaramente come i simboli di una stringa terminale siano raggruppati in sottostringhe, ognuna delle quali appartiene al linguaggio di una delle variabili della grammatica.

Gli alberi sintattici (parse tree), usati nei compilatori, sono la struttura dati utilizzata per rappresentare i sorgenti di un programma; tale struttura dati semplifica la traduzione del codice sorgente in codice eseguibile tramite l'uso di funzioni ricorsive.

Se la CFG è non ambigua, dovrebbe essere possibile definire un solo albero sintattico; purtroppo non è sempre così. Esistono diversi alberi sintattici per una stessa stringa.

## Costruzione di un albero sintattico

*Sia  $G(V, T, P, S)$  una CFG, un albero sintattico è un albero per cui:*

1. ogni **nodo interno** è etichettato con una **variabile** in  $V$
2. ogni foglia è etichettata con un simbolo in  $V \cup T \cup \{\epsilon\}$ ; e ogni foglia etichettata con  $\epsilon$  è l'unico figlio del suo genitore
3. se un nodo interno è etichettato con  $A$  e i suoi figli (da sinistra a destra) sono etichettati con  $X_1, X_2, \dots, X_k$ , allora  $A \rightarrow X_1 X_2 \dots X_k$  è una produzione in  $P$ .

Esempi:

Grammatica	<ol style="list-style-type: none"> <li>1. <math>E \rightarrow I</math></li> <li>2. <math>E \rightarrow E + E</math></li> <li>3. <math>E \rightarrow E * E</math></li> <li>4. <math>E \rightarrow (E)</math></li> </ol>	<ol style="list-style-type: none"> <li>1. <math>P \rightarrow \epsilon</math></li> <li>2. <math>P \rightarrow 0</math></li> <li>3. <math>P \rightarrow 1</math></li> <li>4. <math>P \rightarrow 0P0</math></li> <li>5. <math>P \rightarrow 1P1</math></li> </ol>
Derivazione	$E \Rightarrow^* I + E$	$E \Rightarrow^* 0110$
Albero sintattico	<pre>       E      /\     E + E           I             </pre>	<pre>       P      /\     0 P 0      /\     1 P 1             ε             </pre>

## Prodotto di un albero sintattico

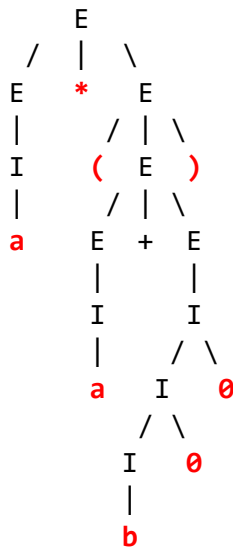
Il **prodotto** di un albero sintattico è la **stringa costituita dai simboli nelle foglie letti da sinistra a destra**.

Sono particolarmente importanti gli alberi sintattici nei quali:

1. il prodotto è una **stringa terminale**
2. la radice è etichettata dal **simbolo iniziale**

in quanto l'insieme dei prodotti di questi alberi sintattici costituisce il **linguaggio della grammatica**.

Ad esempio, l'albero sintattico relativo al prodotto  $a * (a + b00)$  per il linguaggio descritto sopra è:



## Osservazione

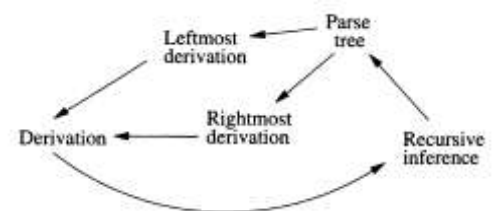
Sia  $G = (V, T, P, S)$  una CFG e  $A \in V$ , le seguenti affermazioni sono equivalenti:

1.  $A \Rightarrow^* w$
2.  $A \Rightarrow_{lm}^* w$
3.  $A \Rightarrow_{rm}^* w$
4. C'è un albero sintattico di  $G$  con radice  $A$  e prodotto  $w$ .

## Dimostrazione

L'immagine a fianco mostra i legami tra le diverse forme e si dimostra che

- A) si possono ottenere derivazioni (a sinistra e a destra) visitando gli alberi sintattici
- B) ovviamente, una derivazione (a sinistra e a destra) è una derivazione
- C) data una derivazione, tramite una inferenza ricorsiva, è possibile costruire l'albero sintattico.



## Derivazioni da alberi sintattici

<p>Partendo da un esempio si costruisce la derivazione a sinistra per l'albero.</p> <p>Supponendo di aver costruito la derivazione a sinistra per il sottoalbero (a):</p> $E \Rightarrow_{lm} I \Rightarrow_{lm} a$ <p>e quella per il sottoalbero (b):</p> $E \Rightarrow_{lm} (E) \Rightarrow_{lm} (E + E) \Rightarrow_{lm} (I + E) \Rightarrow_{lm} (a + E) \Rightarrow_{lm} (a + I) \Rightarrow_{lm} (a + I0) \Rightarrow_{lm} (a + I00) \Rightarrow_{lm} (a + b00)$ <p>è possibile ottenere la derivazione per l'intero albero:</p> $E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E \Rightarrow_{lm} a * (E) \Rightarrow_{lm} a * (E + E) \Rightarrow_{lm} a * (I + E) \Rightarrow_{lm} a * (a + E) \Rightarrow_{lm} a * (a + I) \Rightarrow_{lm} a * (a + I0) \Rightarrow_{lm} a * (a + I00) \Rightarrow_{lm} a * (a + b00)$	<pre>       E     (a)(b)   /     \  E  *  E     /  \  I  ( E )     /  \  a  E + E               I   I        / \     a I  0       / \       I  0               b           </pre>
--	---

## Ambiguità

Data la grammatica:  $E \rightarrow I \mid E + E \mid E * E \mid (E)$        $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

si può notare che la forma sentenziale  $E + E * E$  ha due derivazioni che determinano due alberi sintattici:

$E \Rightarrow E + E \Rightarrow E + E * E$	$E \Rightarrow E * E \Rightarrow E + E * E$
<pre>       E     /     \    E + E     /     \    E * E           </pre>	<pre>       E     /     \    E * E     /     \    E * E           </pre>

Seppure, nella stessa grammatica, la stringa  $a + b$  abbia varie derivazioni, si può notare che i loro alberi sintattici sono gli stessi:

$E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$	$E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$
<pre>       E     /     \    E + E               I   I               a   b           </pre>	<pre>       E     /     \    E + E               I   I               a   b           </pre>

La struttura della stringa  $a + b$ , a differenza di quella di  $E + E * E$ , è non ambigua.

**Definizione:** sia  $G = (V, T, P, S)$  una CFG, si dice che la grammatica  $G$  è **ambigua** se esiste una stringa  $T^*$  che ha più di un albero sintattico. Viceversa, se ogni stringa in  $L(G)$  ha un unico albero sintattico, allora la grammatica  $G$  è **non ambigua**.

## Rimozione delle ambiguità dalle grammatiche

In certi casi è possibile rimuovere le ambiguità dalle grammatiche, tuttavia non esiste alcun algoritmo in grado di farlo in modo sistematico. Inoltre, alcuni CFL hanno solo CFG ambigue.

Ad esempio, si può provare a rendere non ambigua la grammatica:

$$E \rightarrow I \mid E + E \mid E * E \mid ( E ) \quad I \rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid II$$

Per farlo è necessario modificarla stabilendo:

1. quale simbolo abbia precedenza tra  $*$  e  $+$
2. come si devono raggruppare sequenze di uno stesso operatore:  
 $E + E + E$  deve essere inteso come  $( E + E ) + E$  oppure come  $E + ( E + E )$ ?

Si introduce allora una **gerarchia** di variabili che stabilisce un ordine di precedenza tra gli operatori:

1. espressioni E: composizioni di uno o più termini T tramite  $+$
2. termini T: composizioni di uno o più fattori F tramite  $*$
3. fattori F:
  - a. identificatori I
  - b. espressioni E racchiuse tra parentesi.

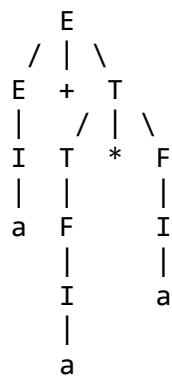
La grammatica diventa:

$$E \rightarrow T \mid E + E \quad T \rightarrow F \mid T * T \quad F \rightarrow I \mid ( E ) \quad I \rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid II$$

Inoltre è necessario imporre un ordinamento per raggruppare gli operatori allo stesso livello (esempio: associatività a sinistra):

$$E \rightarrow T \mid E + T \quad T \rightarrow F \mid T * F \quad F \rightarrow I \mid ( E ) \quad I \rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid II$$

A questo punto la grammatica diventa non ambigua e si ottiene un unico albero sintattico (ad esempio) per la stringa  $a + a * a$ :



Si osservi che in generale:

- ad un albero sintattico corrispondono molte derivazioni, tuttavia:
  - ad ogni (diverso) albero sintattico corrisponde un'unica (diversa) derivazione sinistra
  - ad ogni (diverso) albero sintattico corrisponde un'unica (diversa) derivazione destra

**Teorema 5.29:** data una CFG G, una stringa terminale w ha due distinti alberi sintattici se e solo se w ha due distinte derivazioni a sinistra dal simbolo iniziale.

## Ambiguità inerente

Un CFL L è **inerentemente ambiguo** se tutte le grammatiche per L sono ambigue.

Ad esempio, sia  $L = \{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}$

una grammatica per L è:	Gli alberi per la struttura sintattica della stringa aabbccdd sono:
-------------------------	---

$S \rightarrow AB \mid C$ $A \rightarrow aAb \mid ab$ $B \rightarrow cBd \mid cd$ $C \rightarrow aCd \mid aDd$ $D \rightarrow bDc \mid bc$	<pre>       S      / \     A   B    /  \ /  \   a A b c B d    / \ / \   a  b c  d             </pre>	<pre>       S               C      / \     a C d      / \     a D d      / \     b D c      / \     b  c             </pre>
--	---	---

Dimostrando che qualsiasi grammatica per  $L$  è ambigua, si ottiene che  $L$  è **inerentemente ambiguo**.

## Automi a pila

Gli automi a pila sono in grado di riconoscere i CFL.

Un automa a pila (Push Down Automata, PDA), è un  $\epsilon$ -NFA con una pila: si tratta di un'estensione degli automi a stati finiti non deterministici utilizzati per riconoscere i linguaggi regolari.

Le azioni sulla pila possono essere: lettura, push e pop che, rispettivamente, aggiungono un elemento in testa alla pila o rimuovono l'elemento in cima alla pila.

Gli automi a pila possono *accettare* stringhe di un CFL con due modalità differenti:

- per stato finale (l'automa, dopo aver consumato tutti i caratteri, deve terminare in uno stato finale per accettare la stringa)
- per pila vuota, indipendentemente dallo stato finale (l'automa accetta la stringa se riesce a svuotare completamente la pila).

Entrambe le varianti accettano esattamente i CFL, pertanto le CFG possono essere convertite in PDA e viceversa. Esiste una sottoclasse di automi a pila deterministici.

In una transizione un PDA:

1. consuma un simbolo di input ed esegue una transizione  $\epsilon$
2. va in un nuovo stato (o rimane nello stato attuale)
3. sostituisce il top della pila con una stringa
  - a. consuma il carattere in cima alla pila
  - b. mette al suo posto una stringa, eventualmente vuota o uguale al carattere consumato (in questo caso la pila rimane inalterata).

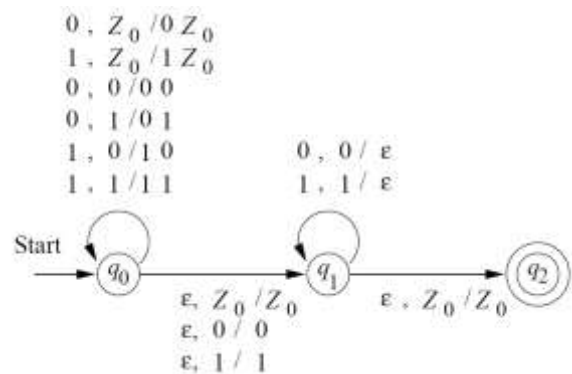


$$\text{Sia } L_{ww^R} = \{ww^R : w \in \{0,1\}^*\}$$

Una grammatica per  $L_{ww^R}$  è:  $P \rightarrow 0P0$ ,  $P \rightarrow 1P1$ ,  $P \rightarrow \epsilon$ .

Si può costruire un PDA per  $L_{ww^R}$  con tre stati:

1.  $q_0$  legge la stringa  $w$ , un simbolo alla volta, rimanendo nello stato  $w$  e aggiungendo il simbolo in input alla pila
2. decide **non deterministicamente** che si trova nel mezzo della stringa  $ww^R$  e si sposta nello stato  $q_1$
3. legge  $w^R$  un simbolo alla volta e lo paragona con il top della pila, se sono uguali effettua un pop della pila e rimane nello stato  $q_1$
4. se la pila è vuota, va nello stato  $q_2$  e accetta la stringa.



## Definizione formale

Un PDA è una tupla di 7 elementi:  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  dove:

- $Q$  è un insieme finito di **stati**
- $\Sigma$  è un alfabeto finito di **simboli in input**
- $\Gamma$  è un alfabeto finito di **pila**
- $\delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$  è la **funzione di transizione**
- $q_0$  è lo **stato iniziale**
- $Z_0 \in \Gamma$  è il **simbolo iniziale per la pila**
- $F \subseteq Q$  è l'insieme degli **stati di accettazione**.

Il PDA dell'immagine sopra è:  $P = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$  con  $\delta$  descritta dalla tabella seguente:

	0, $Z_0$	1, $Z_0$	0, 0	0, 1	1, 0	1, 1	$\epsilon$ , $Z_0$	$\epsilon$ , 0	$\epsilon$ , 1
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$	$\{(q_0, Z_0)\}$	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$
$q_1$	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\{(q_1, Z_0)\}$	$\emptyset$	$\emptyset$
$* q_2$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

## Descrizioni istantanee

Fino a questo punto si è definita solo informalmente la maniera attraverso la quale un PDA “computa”. Intuitivamente un PDA “passa” da una configurazione ad un'altra.

Nel passaggio da una configurazione ad un'altra, un PDA:

1. consuma un simbolo di input oppure effettua una transazione  $\epsilon$
2. consuma il simbolo sulla cima della pila sostituendolo con una **stringa** (eventualmente vuota).

Una **descrizione istantanea** (Instantaneous Description, ID) di un PDA è una tripla  $(q, w, \gamma)$  che definisce una configurazione del PDA in un dato istante, in cui  $q$  è lo stato  $w$  è l'input residuo e  $\gamma$  è il contenuto della pila.

Per descrivere il cambiamento di stato in un PDA si utilizza il simbolo “turnstile” ([https://en.wikipedia.org/wiki/Turnstile\\_\(symbol\)](https://en.wikipedia.org/wiki/Turnstile_(symbol))) che rappresenta la derivabilità di uno stato da un altro.

Esempio:  $(q_0, 0110, Z_0) \vdash (q_0, 110, 0Z_0) \vdash (q_0, 1010, Z_0) \vdash (q_1, 10, 10Z_0) \vdash (q_1, 0, 0Z_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$

Sia  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  un PDA, si definisce  $\vdash_P$ , o semplicemente  $\vdash$  se  $P$  è noto, come segue:

$$\forall w \in \Sigma^*, \beta \in \Gamma^*: (p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

Questa rappresentazione riflette l'idea che, **consumando il carattere  $a$**  (che potrebbe essere  $\epsilon$ ) dalla stringa in input e **rimpiazzando  $X$  dalla cima della pila con  $\alpha$** , il **PDA passa dallo stato  $q$  allo stato  $p$** . Si può notare che  $w$  (la parte residua dell'input) e  $\beta$  il contenuto della pila sotto la cima, non influenzano il passaggio di stato ma saranno utilizzati nei passaggi successivi.

Si utilizza il simbolo  $\vdash^*$  per rappresentare zero o più passaggi del PDA:

BASE:

$$I \vdash^* I \text{ (zero passaggi)}$$

INDUZIONE:

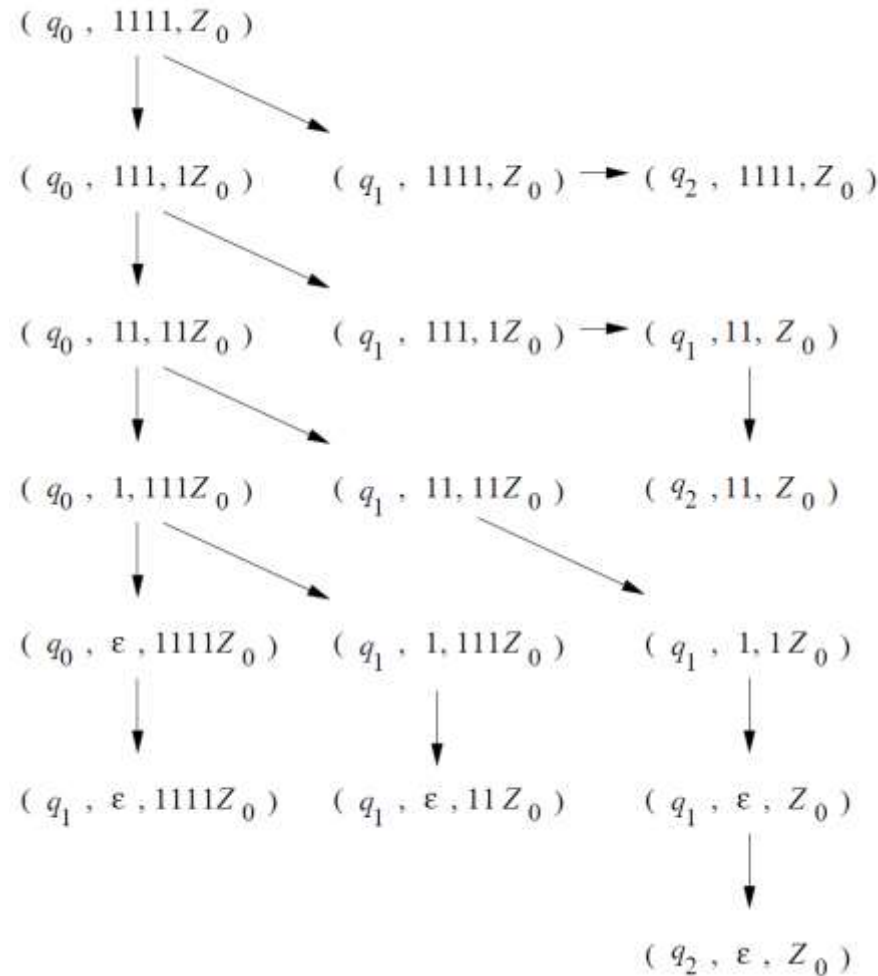
$$I \vdash^* J \text{ se esiste una qualche}$$

ID  $K$  tale che  $I \vdash K$  e  $K \vdash^* J$ .

Si osserva allora che se  $I \vdash^* J$  allora esiste una sequenza di ID  $K_1, K_2, \dots, K_n$  tali che  $I = K_1, J = K_n$  e, per ogni  $i = 1, 2, \dots, n-1$ , si ha che  $K_i \vdash K_{i+1}$ .

<p>Esempio: considerando il PDA...</p> <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p> <math>0, Z_0 / 0 Z_0</math>  <math>1, Z_0 / 1 Z_0</math>  <math>0, 0 / 0 0</math>  <math>0, 1 / 0 1</math>  <math>1, 0 / 1 0</math>  <math>1, 1 / 1 1</math> </p> </div> <div style="width: 45%;"> <p> <math>0, 0 / \epsilon</math>  <math>1, 1 / \epsilon</math> </p> </div> </div>	<p>...con la stringa in input 1111, siccome <math>q_0</math> è lo stato iniziale e <math>Z_0</math> il simbolo iniziale sulla pila, il PDA può “sbagliare” tante volte generando diverse sequenze di passaggi di configurazione. Dalla ID iniziale il PDA può effettuare due scelte, una ipotizzando di essere ancora nella prima metà della stringa, “mangiando” un simbolo in input e aggiungendo un simbolo alla pila, un'altra ipotizzando di aver già raggiunto la metà della stringa, spostandosi nello stato successivo eliminando un simbolo lasciando inalterati stringa e pila.</p>
--	---

...si ottengono le seguenti configurazioni, esiste una sequenza di transizioni per cui il PDA riesce a consumare tutta la stringa in input e a svuotare completamente la pila.



## Accettazione per stato finale

Un PDA che accetta una stringa consumandone ogni carattere ed entrando in uno stato di accettazione finale è un PDA che **accetta per stato finale**.

Formalmente, sia  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  un PDA, allora  $L(P)$  è il linguaggio accettato per stato finale da  $P$  se:  $\{w | (q_0, w, Z_0) \vdash^* (q, \epsilon, a)\}$  per un qualche stato  $q \in F$  e una qualsiasi stringa  $a$ .

Si osservi che il contenuto della pila al raggiungimento dello stato finale (con la stringa interamente consumata) è irrilevante.

## Accettazione per pila vuota

Per ogni PDA  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  si può definire  $N(P) = \{w | (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)\}$  per un qualsiasi stato  $q$ .  $N(P)$  è l'insieme delle stringhe win input che il PDA  $P$  può consumare in modo che al termine della stringa la pila sia vuota. La  $N$  di  $N(P)$  sta per "null stack".

Siccome gli stati finali sono irrilevanti per PDA che accettano per pila vuota, il PDA può essere definito con sei elementi (anziché sette):  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$   $F$  è irrilevante.

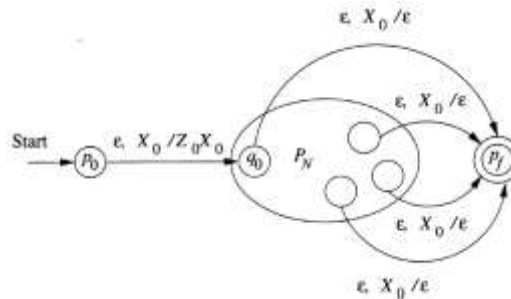
La classe dei linguaggi riconosciuti da PDA che accettano per pila vuota è la stessa classe dei linguaggi riconosciuti da PDA che accettano per stato finale.

**Teorema 6.9:** Se  $L = N(P_N)$  per un qualche PDA  $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$  allora esiste un PDA  $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$  tale che  $L = L(P_F)$ .

La dimostrazione del teorema definisce anche le operazioni necessarie per trasformare un PDA che accetta per pila vuota in un PDA che accetta per stato finale.

## Trasformazione da pila vuota a stato finale

Dato un PDA  $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$ , si può utilizzare un nuovo simbolo  $X_0 \notin \Gamma$  come simbolo iniziale e marcatore di fine della pila per un PDA  $P_F$  che accetti per stato finale. In questo modo, se il PDA  $P_F$  “legge” il simbolo  $X_0$  in cima alla pila, può “passare” ad uno stato finale  $p_f$ . E’ necessario aggiungere un nuovo stato iniziale  $p_0$  nel PDA  $P_F$  che ha l’unico scopo di inserire nella pila lo stato PDA  $Z_0$  del PDA  $P_N$ .



Formalmente, dato un PDA  $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$  che accetta per pila vuota si ottiene un PDA che accetta per stato finale  $P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$ , in cui  $\delta_F$  è definito come:

1.  $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$  nello stato iniziale  $P_F$  effettua una transizione spontanea che mette nella pila il simbolo  $Z_0$
2. per tutti gli stati  $q \in Q$ , per tutti i simboli  $a \in \Sigma$  o  $a = \epsilon$  e per tutti i simboli  $Y \in \Gamma$ ,  $\delta_F(q, a, Y) = \delta_N(q, a, Y)$
3. per tutti gli stati  $q \in Q$ ,  $\delta_F(q, \epsilon, X_0) = \{(p_f, \epsilon)\}$

Esempio:

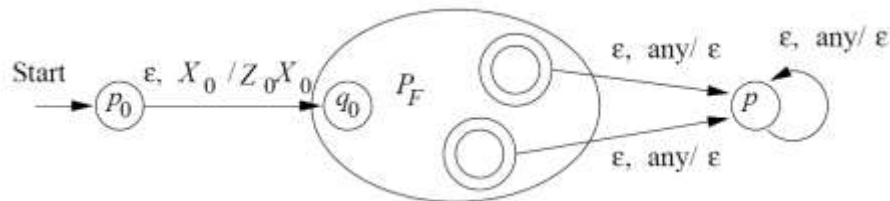
$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ $\delta_N(q, i, Z) = \{(q, ZZ)\}$ e $\delta_N(q, e, Z) = \{(q, \epsilon)\}$	$P_F = (\{q, p, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$ $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$ $\delta_F(q, i, Z) = \{(q, ZZ)\}$ e $\delta_F(q, e, Z) = \{(q, \epsilon)\}$ $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$

## Trasformazione da stato finale a pila vuota

**Teorema 6.11:** Se  $L = L(P_F)$  per un qualche PDA  $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$  allora esiste un PDA  $P_N$  tale che  $L = N(P_N)$ .

$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$  dove  $\delta_N$  è definito come:

1.  $\delta_N(p_0, \epsilon, X_0) = \{(q_0, ZX_0)\}$
2. per tutti gli stati  $q \in Q$ , per tutti i simboli  $a \in \Sigma$  o  $a = \epsilon$  e per tutti i simboli  $Y \in \Gamma$ ,  $\delta_N(q, a, Y) = \delta_F(q, a, Y)$
3. per tutti gli stati  $q \in F$  e per ogni simbolo di pila  $Y \in \Gamma \cup \{X_0\}$ ,  $\delta_N(q, \epsilon, Y) = \{(p, \epsilon)\}$
4. per tutti i simboli di pila  $Y \in \Gamma \cup \{X_0\}$ ,  $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$



## Equivalenze PDA e CFG

Un linguaggio è generato da una Context Free Grammar (CFG)

- se e solo se è accettato da un PDA che accetta per pila vuota
- se e solo se è accettato da un PDA che accetta per stato finale.



## Trasformazione da CFG a PDA

Per mostrare l'equivalenza, data una grammatica si dovranno simulare le derivazioni sinistre di G. Ogni **forma sentenziale** di queste derivazioni può essere scritta come  $x A \alpha$  in cui  $A$  è la variabile a sinistra,  $x$  sono i simboli terminali che appaiono alla sua sinistra e  $\alpha$  è la stringa di terminali e variabili alla destra di  $A$ .

Ad esempio:  $(a + E) = x A \alpha$        $x = (a +$        $A = E$        $\alpha = )$

La parte a destra dei simboli terminali è detta "coda" (tail):  $tail = A \alpha = E)$ , se una forma sentenziale sinistra consiste solo di terminali allora la coda è vuota:  $tail = \epsilon$ .

Ad esempio:  $(a + b00) = x A \alpha$        $x = (a + b00)$        $A \alpha = \epsilon$

## Esempio

$$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow a * (E) \Rightarrow a * (E + E) \Rightarrow a * (a + E) \Rightarrow a * (a + I) \Rightarrow a * (a + IO) \Rightarrow a * (a + IOO) \Rightarrow a * (a + bOO)$$

Forma sentenziale	$x$ (simboli a sinistra di A)	$A$ (variabile a sinistra)	$\alpha$ (simboli a destra di A, tail)
E	$\epsilon$	E	$\epsilon$
$E * E$	$\epsilon$	E	$* E$
$I * E$	$\epsilon$	I	$* E$
$a * E$	$a *$	E	$\epsilon$
$a * (E)$	$a * ($	E	$)$
$a * (E + E)$	$a * ($	E	$+ E )$
$a * (a + E)$	$a * (a +$	E	$)$
$a * (a + I)$	$a * (a +$	I	$)$
$a * (a + IO)$	$a * (a +$	I	$O )$
$a * (a + IOO)$	$a * (a +$	I	$OO )$
$a * (a + bOO)$	$a * (a + bOO )$	$\epsilon$	$\epsilon$

$x A \alpha \Rightarrow_{lm} x \beta \alpha$  corrisponde al PDA che consuma  $x$  e ha  $A \alpha$  sulla pila, poi leggendo  $\epsilon$  elimina  $A$  e mette  $\beta$  sulla pila.

$x$	$A$	$\alpha$	$\Rightarrow_{lm}$	$x$	$\beta$	$\alpha$
$a * ($	E	$+ E )$		$a * ($	a	$+ E )$

Dato un PDA che si trova nella configurazione  $(q, y, A\alpha)$ , con  $w = xy$ , il PDA va non deterministicamente alla configurazione  $(q, y, \beta\alpha)$ . Alla configurazione  $(q, y, \beta\alpha)$  il PDA si comporta come al passo precedente a meno che non ci siano terminali in  $\beta$  in questo caso il PDA li elimina se li legge dall'input. Al termine, se tutte le scommesse sono giuste, il PDA finisce l'input con la pila vuota.

In sintesi la trasformazione da CFG a PDA è la seguente:

Sia  $G = (V, T, Q, S)$  una CFG, si definisce il PDA  $P_G = (\{q\}, T, V \cup T, \delta, q, S)$  dove:

$$\delta(q, \epsilon, A) = \{(q, \beta) : A \rightarrow \beta \in Q\} \text{ per } A \in V \quad e \quad \delta(q, a, a) = \{(q, \epsilon)\} \text{ per } a \in T.$$

## Esempio

Si consideri la grammatica:  $G = (\{S\}, \{i, e\}, Q, S) \quad Q = \{S \rightarrow \epsilon \mid SS \mid iS \mid iSe\}$

Il PDA corrispondente è:

$$P = (\{q\}, \{i, e\}, \{i, e, S\}, \delta, q, S) \text{ con } \delta(q, \epsilon, S) = \{(q, \epsilon), (q, SS), (q, iS), (q, iSe)\}, \delta(q, i, i) = \{(q, \epsilon)\}, \delta(q, e, e) = \{(q, \epsilon)\}$$

## Trasformazione da PDA a CFG

Per completare le dimostrazioni di equivalenza, per ogni PDA si deve trovare una grammatica  $G$  che produca il linguaggio accettato dal PDA (per pila vuota).

L'immagine a fianco mostra il comportamento di un PDA per cui, per una certa sequenza di passaggi di stato si ottiene che un simbolo (ad esempio  $Y_1$ ) viene tolto dalla pila.

L'idea è che per una certa sequenza di input ( $x_1, \dots, x_k$ ) si possono arrivare a togliere tutti i simboli ( $Y_1, \dots, Y_k$ ) dalla pila.

L'immagine mostra come, probabilmente, i simboli nella pila aumentano per una certa sequenza di passaggi di stato ma, ad un certo punto si avrà che il primo simbolo viene tolto dalla pila.

Il PDA parte dallo stato  $p_0$  e termina nello stato  $p_k$ .

Si vuole allora definire una grammatica con variabili della forma  $[p_{i-1}Y_i p_i]$  che rappresentano il passaggio da  $p_{i-1}$  a  $p_i$  con l'effetto di eliminare  $Y_i$ .

Allora, una stringa terminale generata da una variabile  $[pXq]$  rappresenta l'input letto dal PDA andando dallo stato  $p$  allo stato  $q$  rimuovendo  $X$  dalla pila.

Formalmente: sia  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$  un PDA, si definisce la grammatica  $G = (V, \Sigma, R, S)$  con:

- $S$  simbolo di partenza
- tutte le variabili sono espresse nella forma  $[pXq]$  dove  $p \in Q$  e  $q \in Q$ ,  $X \in \Gamma$   

$$V = \{[pXq] : \{p, q\} \in Q, X \in \Gamma\} \cup \{S\}$$
- Le produzioni di  $G$  sono date da:
  - per tutti gli stati  $p$ ,  $G$  ha le produzioni  $S \rightarrow [q_0 Z_0 p]$  intese come tutte le produzioni che generano una stringa  $w$  che toglie  $Z_0$  dalla pila andando dallo stato  $q_0$  allo stato  $p$  nel PDA  $P$ , formalmente:  $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$ . In altri termini da  $S$  si possono generare tutte le stringhe  $w$  che svuotano la pila.
  - sia  $(r, Y_1 Y_2 \dots Y_k) \in \delta(q, a, X)$  in cui:
    - $a$  può essere un simbolo dell'alfabeto  $\Sigma$  oppure  $a = \epsilon$
    - $k$  può essere un qualsiasi numero, anche 0, tale che  $Y_1 Y_2 \dots Y_k = \epsilon$

allora, per ogni stato

$r_1, r_2, \dots, r_k$ ,  $G$  ha le produzioni:

$$[qXr_k] \rightarrow a[rY_1 r_1][r_1 Y_2 r_2] \dots [r_{k-1} Y_k r_k]$$

Questa produzione dice che una via per togliere  $X$  dalla pila e andare dallo stato  $q$  allo stato  $r_k$  è leggere il carattere  $a$  (che può essere  $\epsilon$ ), poi utilizzare un qualche input per togliere  $Y_1$  dalla pila, andando dallo stato  $r$  allo stato  $r_1$ , poi leggere qualche input per togliere  $Y_2$  dalla pila e andare dallo stato  $r_1$  allo stato  $r_2$  e così via.

$$R = \{S \rightarrow [q_0 Z_0 p] : p \in Q\} \cup$$

$$\{[qXr_k] \rightarrow a[rY_1 r_1] \dots [r_{k-1} Y_k r_k] : a \in \Sigma \cup \{\epsilon\}, \{r_1, \dots, r_k\} \subseteq Q, (r, Y_1 Y_2 \dots Y_k) \in \delta(q, a, X)\}$$

dove, in caso  $k = 0$  si ha:  $Y_1 Y_2 \dots Y_k = \epsilon$  e  $r_k = r$

In definitiva:

$$[qXr_k] \Rightarrow a[r_0 Y_1 r_1][r_1 Y_2 r_2] \dots [r_{k-1} Y_k r_k] \Rightarrow^* a w_1 [r_1 Y_2 r_2] \dots [r_{k-1} Y_k r_k] \Rightarrow^* a w_1 w_2 \dots [r_{k-1} Y_k r_k] \Rightarrow^* a w_1 w_2 \dots w_k = w.$$

Dove  $r_k = p$ .

### Esempio

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

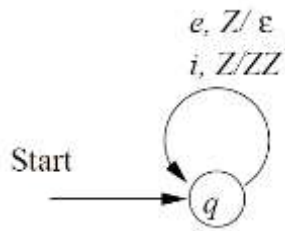
dove  $\delta_N(q, i, Z) = \{(q, ZZ)\}$  e  $\delta_N(q, e, Z) = \{(q, \epsilon)\}$

Per convertirlo in una grammatica  $G = (V, \{i, e\}, R, S)$ , si definiscono:

- l'insieme delle variabili:  $V = \{[qZq], S\}$
- l'insieme delle produzioni:  $R = \{S \rightarrow [qZq], [qZq] \rightarrow i[qZq][qZq], [qZq] \rightarrow e\}$

rimpiazzando  $[qZq]$  con  $A$ , si possono ridefinire le produzioni:  $R = \{S \rightarrow iAA, A \rightarrow e\}$ .

$A, A \rightarrow$





## Esempio

Si converta il PDA  $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$  dove  $\delta$  è definita da:

1.  $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$
2.  $\delta(q, 1, X) = \{(q, XX)\}$
3.  $\delta(q, 0, X) = \{(p, X)\}$
4.  $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
5.  $\delta(p, 1, X) = \{(p, \epsilon)\}$
6.  $\delta(p, 0, Z_0) = \{(q, Z_0)\}$

in una CFG :  $G = (V, \{0, 1\}, R, S)$

Le variabili sono tutte le combinazioni di stati e simboli della pila, più una variabile iniziale S:

$$V = \{[qZ_0q], [pZ_0q], [qZ_0p], [pZ_0p], [qXq], [pXq], [qXp], [pXp], S\}$$

Le produzioni sono date:

1. dalla variabile iniziale alle combinazioni che hanno lo stato iniziale, il simbolo iniziale di pila e ogni stato del PDA:

$$S \rightarrow [qZ_0q] \mid [qZ_0p]$$

2. Dalla transizione (1)  $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$ :

$$[qZ_0q] \rightarrow 1[qXq][qZ_0q]$$

$$[qZ_0q] \rightarrow 1[qXp][pZ_0q]$$

$$[qZ_0p] \rightarrow 1[qXq][qZ_0p]$$

$$[qZ_0p] \rightarrow 1[qXp][pZ_0p]$$

3. Dalla transizione (2)  $\delta(q, 1, X) = \{(q, XX)\}$ :

$$[qXq] \rightarrow 1[qXq][qXq]$$

$$[qXq] \rightarrow 1[qXp][pXq]$$

$$[qXp] \rightarrow 1[qXq][qXp]$$

$$[qXp] \rightarrow 1[qXp][pXp]$$

4. Dalla transizione (3)  $\delta(q, 0, X) = \{(p, X)\}$ :

$$[qXq] \rightarrow 0[pXq]$$

$$[qXp] \rightarrow 0[pXp]$$

5. Dalla transizione (4)  $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$ :

$$[qXq] \rightarrow \epsilon$$

6. Dalla transizione (5)  $\delta(p, 1, X) = \{(p, \epsilon)\}$ :

$$[pXp] \rightarrow 1]$$

7. Dalla transizione (6)  $\delta(p, 0, Z_0) = \{(q, Z_0)\}$ :

$$[pZ_0q] \rightarrow 0[qZ_0q]$$

$$[pZ_0p] \rightarrow 0[qZ_0p]$$

## PDA Deterministici

Nonostante i PDA siano per definizione non deterministici, l'utilizzo del determinismo è una particolarità importante, ad esempio i parser generalmente agiscono come PDA deterministici, cosicché la classe dei linguaggi che sono accettati da questi automi è interessante per le intuizioni suggerite relativamente ai costrutti da utilizzare nei linguaggi di programmazione.

Un PDA  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  è **deterministico** se e solo se:

1. ogni  $\delta(q, a, X)$ ,  $a \in \Sigma \cup \{\epsilon\}$  contiene al più un elemento
2. se  $\exists a \in \Sigma: \delta(q, a, X) \neq \emptyset \Rightarrow \delta(q, \epsilon, X) = \emptyset$

Esempio

$L_{wcw^R} = \{wcw^{R^*} : w \in \{0,1\}^*\}$  è riconosciuto dal PDA a fianco  $\rightarrow$

Il PDA è deterministico, infatti, per ogni tripla  $(q, a, X)$  è possibile effettuare un'unica scelta.

I DPDA (Deterministic Push Down Automata) accettano una classe di linguaggi che sta tra i linguaggi regolari e i linguaggi liberi dal contesto.

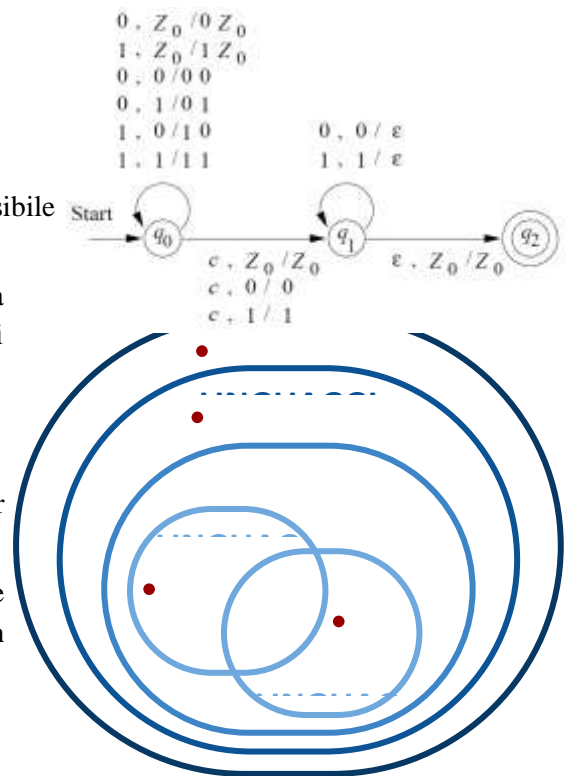
### DPDA che accettano per stato finale

**Teorema 6.17:** Se  $L$  è un linguaggio regolare, allora  $L = L(P)$  per un qualche DPDA  $P$ .

**Dimostrazione:** siccome  $L$  è regolare, allora esiste un DFA  $A$  tale che  $L = L(A)$ , allora dato  $A = (Q, \Sigma, \delta_A, q_0, F)$  si definisce un DPDA  $P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$  dove

$$\delta_P(q, a, Z_0) = \{(\delta_A(q, a), Z_0)\} \text{ per tutti i } p, q \in Q \text{ e } a \in \Sigma$$

Si può osservare che  $(q_0, w, Z_0) \vdash^* (p, \epsilon, Z_0) \Leftrightarrow \hat{\delta}_A(q_0, w) = p$



### DPDA che accettano per pila vuota

I DPDA che accettano per pila vuota sono utili in quanto sono in grado di riconoscere i linguaggi liberi dal contesto che hanno la **proprietà del prefisso**.

**Un linguaggio  $L$  ha la proprietà del prefisso se non esistono due stringhe distinte in  $L$  tali che una è prefisso dell'altra.**

Esempio:  $L_{wcw^R}$  ha la proprietà del prefisso.

Esempio:  $\{0\}^*$  non ha la proprietà del prefisso.

**Teorema 6.19:**  $L \in N(P)$  per qualche DPDA  $P$  se e solo se  $L$  ha la proprietà del prefisso e  $L \in L(P')$  per qualche DPDA  $P'$ .

### DPDA e non ambiguità

La classe dei linguaggi riconosciuti dai DPDA **non** coincide con i CFL che hanno grammatiche **non ambigue** (linguaggi non inerentemente ambigui).

Ad esempio  $L_{ww^R}$  ha una grammatica non ambigua  $S \rightarrow 0S0|1S1|\epsilon$  ma non è  $L(DPDA)$ .

Vale invece l'inverso.

**Teorema 6.20:** se  $L = N(P)$  per qualche DPDA  $P$ , allora  $L$  ha una CFG non ambigua.

**Dimostrazione:** applicando la costruzione da PDA a CFG, se la costruzione è applicata ad un DPDA il risultato è una CFG con derivazioni a sinistra uniche per ogni stringa.

**Teorema 6.21:** se  $L = L(P)$  per qualche DPDA  $P$ , allora  $L$  ha una CFG non ambigua.

**Dimostrazione:** sia  $\$$  un simbolo fuori dall'alfabeto di  $L$  e sia  $L' = L\{\$ \}$ , è possibile modificare  $P$  perché riconosca  $L'$  ( $P$  deterministico) e  $L'$  ha la proprietà del prefisso.

Dal teorema 6.19 si ha che  $L' = N(P')$  per qualche DPDA  $P'$ .

Dal teorema 6.20 si ha che  $N(P')$  può essere generato da una CFG  $G'$  non ambigua.

Modificando  $G'$  in  $G$ , tale che  $L(G) = L$ , aggiungendo la produzione  $\$ \rightarrow \epsilon$  (considerando  $\$$  una variabile anziché un terminale), dato che  $G'$  ha derivazioni a sinistra uniche, anche  $G$  le ha uniche (infatti l'unica cosa aggiunta è  $w\$ \Rightarrow_{lm} w$ ).

## Proprietà dei CFL

Si completa lo studio dei linguaggi liberi dal contesto valutando le loro proprietà. Prima di tutto si vuole provare a semplificare le grammatiche libere dal contesto in modo da rendere più semplice dimostrare alcuni fatti relativi ai CFL.

## Forma normale di Chomsky

Si vuole mostrare che ogni CFL (senza  $\epsilon$ ) è generata da una CFG in cui tutte le produzioni possono essere espresse nella forma  $A \rightarrow BC$  oppure  $A \rightarrow a$ , con  $A, B$  e  $C$  variabili e  $a$  simbolo terminale.

Questa forma è detta **Forma normale di Chomsky**.

Per trasformare una grammatica in forma normale di Chomsky è necessario effettuare una serie ordinata di semplificazioni preliminari per “pulire” la grammatica:

1. eliminare le produzioni  $\epsilon$ , della forma  $A \rightarrow \epsilon$  per una qualche variabile  $A$
2. eliminare le produzioni unità, della forma  $A \rightarrow B$  per qualche variabile  $A$  e  $B$
3. eliminare i simboli inutili: quelle variabili che non appaiono in alcuna derivazione di una stringa terminale da un simbolo di partenza

## Eliminazione simboli inutili

In una grammatica  $G = (V, T, P, S)$ ,  $X$  è un simbolo utile se esiste una qualche derivazione  $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$  dove  $w \in T^*$ . Si osservi che  $X$  può essere sia un simbolo di  $V$  che di  $T$  e la forma sentenziale  $\alpha X \beta$  può essere all'inizio o alla fine della derivazione.

Se un simbolo  $X$  non è utile allora è detto inutile, ed - evidentemente - ometterlo dalla grammatica non modifica il linguaggio generato.

Un simbolo, per essere utile, deve avere due qualità:

1. deve essere un simbolo **generante**, tale che  $X \Rightarrow^* w$  per qualche stringa terminale  $w \in T^*$  (si noti che ogni simbolo terminale è anche generante)
2. deve essere un simbolo **raggiungibile**, tale che esiste una derivazione del tipo  $S \Rightarrow^* \alpha X \beta$  per qualche  $\{\alpha, \beta\} \subseteq (V \cup T)^*$ .

Eliminando i simboli non generanti prima e quelli non raggiungibili successivamente, rimangono solo i simboli utili.

## Esempio

Si consideri la grammatica  $S \rightarrow AB \mid a \quad A \rightarrow b$

1. I simboli  $S$  e  $A$  sono generanti, il simbolo  $B$  invece non lo è.  $a$  e  $b$  sono generanti per se stessi. E' allora possibile eliminare il simbolo  $B$ .  
La grammatica diventa:  $S \rightarrow a \quad A \rightarrow b$
2. A questo punto la variabile  $A$  non è raggiungibile e va eliminata.  
La grammatica diventa:  $S \rightarrow a$ .

Nota: se si invertissero le due fasi, si avrebbe...

1. Si elimina  $B$  in quanto non raggiungibile  
La grammatica diventa:  $S \rightarrow a \quad A \rightarrow b$
2. Tutti i simboli sono generanti e non si elimina nulla.  
La grammatica diventa:  $S \rightarrow a \quad A \rightarrow b$   
Ma contiene simboli inutili.

## Eliminazione produzioni $\epsilon$

Le produzioni  $\epsilon$ , pur essendo convenienti in molti problemi legati alla grammar-design, non sono essenziali; certamente senza produzioni che abbiano  $\epsilon$  nel loro corpo, non è possibile generare stringhe vuote come membri del linguaggio. Si dimostra tuttavia che se un linguaggio  $L$  ha una CFG,  $L - \{\epsilon\}$  ha una CFG senza produzioni  $\epsilon$ . Se  $\epsilon$  non è in  $L$ , allora  $L$  stesso è  $L - \{\epsilon\}$  ed una CFG senza produzioni  $\epsilon$ .

Ciò detto, una variabile  $A$  è annullabile se  $A \Rightarrow^* \epsilon$ .

Se  $A$  è annullabile, allora le regole del tipo  $B \Rightarrow^* \alpha A \beta$  devono essere rimpiazzate con  $B \rightarrow \alpha A \beta, B \rightarrow \alpha \beta$ , cancellando tutte le regole con corpo  $\epsilon$ .

L'insieme di tutti i simboli annullabili di una grammatica  $G = (V, T, P, S)$  è indicato con  $n(G)$ .

### Esempio

Sia  $G$  la grammatica:  $S \rightarrow AB, A \rightarrow aAA|\epsilon, B \rightarrow bBB|\epsilon$

Allora  $n(G) = \{A, B, S\}$

pertanto:

$S \rightarrow AB$	diventa $S \rightarrow AB A B$
$A \rightarrow aAA \epsilon$	diventa $A \rightarrow aAA aA aA a$
semplificato:	$A \rightarrow aAA aA a$
$B \rightarrow bBB \epsilon$	diventa $B \rightarrow bBB bB bB b$
semplificato:	$B \rightarrow bBB bB b$

La grammatica  $G_1$  diventa:  $S \rightarrow AB|A|B, A \rightarrow aAA|aA|a, B \rightarrow bBB|bB|b \dots L(G_1) = L(G) - \{\epsilon\}$

## Eliminazione produzioni unità

Una **produzione unità** è una produzione nella forma  $A \rightarrow B$ , con  $A$  e  $B$  entrambe variabili. Queste produzioni possono essere utili (ad esempio per eliminare ambiguità in una grammatica), tuttavia complicano la realizzazione di alcune dimostrazioni e introducono passi ulteriori nelle derivazioni che, tecnicamente, non sarebbero necessari. Considerando la grammatica:

$$\begin{aligned} I &\rightarrow a|b|Ia|Ib|I0|I1 \\ F &\rightarrow I|(E) \\ T &\rightarrow F|T * F \\ E &\rightarrow T|E + T \end{aligned}$$

si osservano le produzioni unità:  $F \rightarrow I, T \rightarrow F, E \rightarrow T$

$E \rightarrow T$  può essere espansa in  $E \rightarrow F|T * F$ ; espandendo  $E \rightarrow F$  si ha  $E \rightarrow I|(E)|T * F$ ; continuando con  $E \rightarrow I$ , infine, si ha:  $E \rightarrow a|b|Ia|Ib|I0|I1|(E)|T * F$

$T \rightarrow F|T * F$  diventa  $T \rightarrow F|T * F$  poi  $T \rightarrow I|(E)|T * F$  poi  $T \rightarrow a|b|Ia|Ib|I0|I1|(E)|T * F$

$F \rightarrow I$  diventa  $F \rightarrow a|b|Ia|Ib|I0|I1$

**Nota:** il metodo di espansione non funziona se ci sono cicli, ad esempio  $A \rightarrow B, B \rightarrow C, C \rightarrow A$ .

Il seguente metodo basato su **coppie unità**, funziona invece per ogni tipo di grammatica.

$(A, B)$  è una coppia unità se  $A \Rightarrow^* B$  usando solo produzioni unità.

**Nota:** In  $A \rightarrow BC, C \rightarrow \epsilon$  si ha  $A \Rightarrow^* B$  ma non utilizzando solo produzioni unità.

Si indica con  $u(G)$  l'insieme di tutte le coppie unità di  $G = (V, T, P, S)$ .

**Nota:**  $(A, A) \in u(G)$  per ogni variabile  $A \in G$ .

Dato  $G = (V, T, P, S)$  si costruisce  $G_1 = (V, T, P_1, S)$  che non contiene le produzioni unità e tale che  $L(G) = L(G_1)$  definendo  $P_1 = \{A \rightarrow \alpha | \alpha \notin V, B \rightarrow \alpha \in P, (A, B) \in u(G)\}$

Considerando la grammatica  $I \rightarrow a|b|Ia|Ib|I0|I1 \quad F \rightarrow I|(E) \quad T \rightarrow F|T * F \quad E \rightarrow T|E + T$

si ha  $u(G) = \{(I, I), (F, F), (T, T), (E, E), (E, T), (E, F), (T, F), (F, I), (E, I), (T, I)\}$

coppia	produzione
$(I, I)$	$I \rightarrow a b Ia Ib I0 I1$
$(F, F)$	$F \rightarrow (E)$
$(T, T)$	$T \rightarrow T * F$
$(E, E)$	$E \rightarrow E + T$
$(E, T)$	$E \rightarrow T * F$
$(E, F)$	$E \rightarrow (E)$
$(T, F)$	$T \rightarrow (E)$
$(F, I)$	$F \rightarrow a b Ia Ib I0 I1$
$(E, I)$	$E \rightarrow a b Ia Ib I0 I1$
$(T, I)$	$T \rightarrow a b Ia Ib I0 I1$

### Forma normale di Chomsky CNF

Ogni CFL non vuoto, che non contiene  $\epsilon$ , ha una grammatica  $G$  priva di simboli inutili con produzioni nella forma:

- $A \rightarrow BC$  dove  $\{A, B, C\} \subseteq V$
- oppure
- $A \rightarrow a$  dove  $A \in V$  e  $a \in T$

Per ottenere tale grammatica è necessario:

1. pulire la grammatica
    - a. eliminare le produzioni  $\epsilon$
    - b. eliminare le produzioni unità
    - c. eliminare i simboli inutili
  2. modificare le produzioni con 2 o più simboli in modo tale che siano tutte variabili
  3. ridurre il corpo delle regole di lunghezza superiore a 2 in cascate di produzioni con corpi da 2 variabili.
- Per il passo 2, per ogni terminale  $a$  che compare in un corpo di lunghezza  $\geq 2$ , si crea una nuova variabile, ad esempio  $A$  e si sostituisce  $a$  con  $A$  in tutti i corpi, e aggiungendo la nuova regola  $A \rightarrow a$
- Per il passo 3, per ogni regola nella forma  $A \rightarrow B_1 B_2 \dots B_k$ , con  $k \geq 3$ , si introducono le nuove variabili  $C_1, C_2, \dots, C_{k-2}$  e si sostituisce la regola con:
- ◆  $C \rightarrow B_1 C_1$
  - ◆  $C_1 \rightarrow B_2 C_2$
  - ◆ ...
  - ◆  $C_{k-3} \rightarrow B_{k-2} C_{k-2}$
  - ◆  $C_{k-2} \rightarrow B_{k-1} B_k$

A destra è riportata un'illustrazione che rappresenta l'effetto del passo 3.

Partendo dalla grammatica:

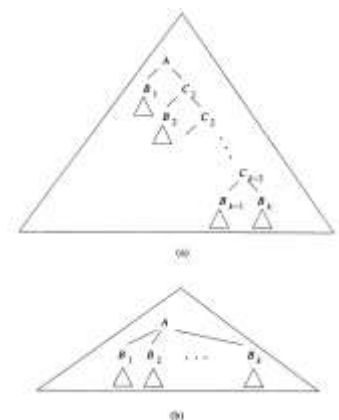
$EE+T|T*F|(E)|a|b|Ia|Ib|I0|I1$   
 $TT*F|(E)|a|b|Ia|Ib|I0|I1$   
 $F(E)|a|b|Ia|Ib|I0|I1$   
 $Ia|b|Ia|Ib|I0|I1$

Per il passo 2 si definiscono le nuove regole:  $A \rightarrow a, B \rightarrow b, Z \rightarrow 0, O \rightarrow 1, P \rightarrow +, M \rightarrow *, L \rightarrow (, R \rightarrow )$  da cui:

$EEPT|TMF|LER|a|b|IA|IB|IZ|IO$   
 $TTMF|LER|a|b|IA|IB|IZ|IO$   
 $FLER|a|b|IA|IB|IZ|IO$   
 $Ia|b|IA|IB|IO|IO$   
 $Aa, Bb, Z0, O1, P+, M*, L(, R)$

Per il passo 3, si sostituiscono:

$EEPT$  con  $EEC1, C1PT$



ETMF, TTMF con ETC2, TTC2, C2MF  
 ELER, TLER, FLER con ELC3, TLC3, FLC3, C3ER

La grammatica CNF finale è:

EEC1|TC2|LC3|a|b|IA|IB|IZ|IO  
 TTC2|LC3|a|b|IA|IB|IZ|IO  
 FLC3|a|b|IA|IB|IZ|IO  
 Ia|b|IA|IB|IO|IO  
 C1PT, C2MF, C3ER  
 Aa, Bb, Z0, 01, P+, M\*, L(, R)

## Pumping Lemma per CFL

Il pumping lemma per i CFL può essere utilizzato per dimostrare che un linguaggio non è libero dal contesto.

Il pumping lemma per CFL dice che in una qualsiasi stringa appartenente al linguaggio, sufficientemente lunga, è possibile trovare al più due stringhe vicine che possono essere “gonfiate” entrambe in coppia ripetendole un certo numero di volte, tali per cui la stringa risultante è ancora appartenente al linguaggio.

## Teorema

<p><b>Teorema 7.18: Pumping Lemma per CFL</b></p> <p><i>Sia <math>L</math> un CFL.</i></p> <p><i>Allora <math>\exists n \geq 1</math> che soddisfa:</i></p> <p><math>\forall z \in L:  z  \geq n</math> è scomponibile in 5 stringhe</p> <p><math>z = uvwxy</math> tali che</p> <ol style="list-style-type: none"> <li>1. <math> vwx  \leq n</math></li> <li>2. <math> vx  &gt; 0</math></li> <li>3. <math>\forall i \geq 0, uv^iwx^iy \in L</math></li> </ol>	<p><b>Teorema 4.1: Pumping Lemma per Linguaggi Regolari</b></p> <p><i>Sia <math>L</math> un linguaggio regolare.</i></p> <p><i>Allora <math>\exists n \geq 1</math> che soddisfa:</i></p> <p><math>\forall w \in L:  w  \geq n</math> è scomponibile in 3 stringhe</p> <p><math>w = xyz</math> tali che</p> <ol style="list-style-type: none"> <li>1. <math>y \neq \epsilon</math></li> <li>2. <math> xy  \leq n</math></li> <li>3. <math>\forall k \geq 0, xy^kz \in L</math></li> </ol>
--	---

Per dimostrare il pumping lemma per CFL è necessario analizzare la forma e la dimensione dei **parse trees**. Uno degli utilizzi della Chomsky Normal Form è trasformare i parse trees in alberi binari. Questi alberi hanno alcune proprietà utili, tra cui:

- supponendo di avere un parse tree relativo alla CNF di una grammatica  $G = (V, T, P, S)$  e supponendo che il prodotto dell'albero sia una stringa terminale  $w$ ; se la lunghezza del più lungo percorso è  $n$ , allora  $|w| \leq 2^{n-1}$  (**Teorema 7.17**).

Infatti: (base) se  $n = 1$ , allora l'albero è formato da un unico nodo e la lunghezza della stringa prodotta è pari a  $2^{1-1} = 2^0 = 1$ ; (induzione) se  $n > 1$  e  $n$  è il percorso più lungo nell'albero, allora la radice dell'albero utilizza una produzione nella forma  $A \rightarrow BC$ . Le produzioni  $B$  e  $C$  rappresentano i due rami dell'albero sotto la radice; i percorsi di massima lunghezza per ognuno di questi due sottoalberi binari, non possono essere più lunghi di  $n-1$ , che per ipotesi induttiva, determinano un prodotto di lunghezza massima pari a  $2^{n-2}$ .

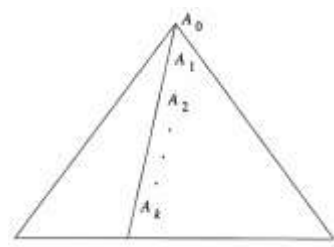
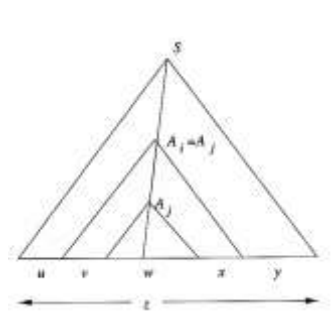
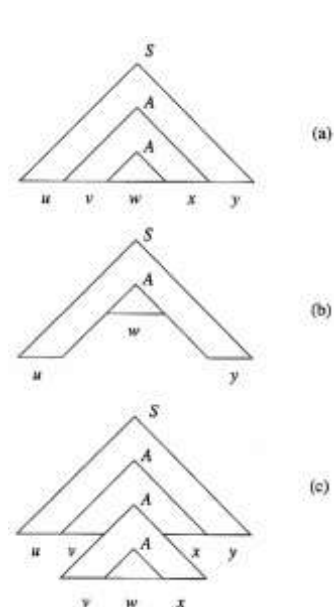
Il prodotto dell'albero completo corrisponde alla concatenazione dei prodotti dei due sottoalberi relativi alle produzioni di  $B$  e  $C$ , che hanno entrambi, al più,  $2^{n-2}$  foglie. Da cui  $2^{n-2} + 2^{n-2} = 2^{n-1}$ .

## Dimostrazione del Pumping Lemma

Il primo passo della dimostrazione consiste nell'individuare una grammatica Chomsky Normal Form per il linguaggio  $L$ . Se  $L$  è il CFL vuoto o  $\{\epsilon\}$  non è possibile individuare una tale grammatica, ciononostante se  $L = \emptyset$  allora sicuramente il teorema non può essere violato in quanto non esiste alcuna stringa  $z$  in  $\emptyset$ , allo stesso modo se  $L = \{\epsilon\}$  il teorema richiede che  $n > 0$ , in tal caso  $z$  non può essere  $\epsilon$ .

A questo punto, sia  $G = (V, T, P, S)$  una grammatica CNF tale che  $L(G) = L - \{\epsilon\}$ , si assuma che  $G$  abbia  $m$  variabili, si scelga  $n = 2^m$ .

Scelta una qualsiasi stringa  $z \in L$  tale che  $|z| \geq n = 2^m$ , per il teorema 7.17, ogni albero sintattico di  $z$  contiene un cammino di lunghezza  $\geq m + 1$ .

<p>La figura a fianco mostra un ipotetico percorso di lunghezza massima nell'albero per cui <math>k \geq m</math> e il percorso ha lunghezza <math>k + 1</math>. Dato che <math>k \geq m</math> e che <math>L</math> ha <math>m</math> variabili, si ha che almeno due variabili devono essere ripetute, supponendo che siano <math>A_i = A_j</math>, con <math>k - m \leq i &lt; j \leq k</math>.</p>	
<p>E' allora possibile dividere l'albero in modo tale che la stringa <math>z</math> sia scomposta in:  <b>w:</b> stringa ottenuta dal prodotto dell'albero con radice <math>A_j</math>  <b>v e x:</b> stringhe a sinistra e destra di <math>w</math> nel prodotto dell'albero con radice in <math>A_i</math>; <math>v</math> e <math>z</math> non possono essere <math>\epsilon</math>  <b>u e y:</b> stringhe a sinistra e destra della stringa prodotto dell'albero con radice in <math>A_i</math>                  Si osservi che la stringa <math>vw x</math> è il prodotto dell'albero radicato in <math>A_i</math> che ha altezza <math>\leq m + 1</math>, pertanto <math> vw x  \leq 2^m = n</math>.</p>	
<p>a) Si può definire un nuovo albero in cui <math>A_i = A_j = A</math></p> <p>b) Sostituendo il sottoalbero con radice in <math>A_i</math> che ha per prodotto <math>vw x</math> con quello con radice in <math>A_j</math> che ha prodotto <math>w</math> si ottiene un nuovo albero che ha prodotto <math>vw y</math> corrispondente al caso <math>i = 0</math>: <math>uv^iwx^i y = uv^0wx^0 y = uwy</math></p> <p>c) Sostituendo il sottoalbero con radice <math>A_j</math> con l'intero sottoalbero con radice in <math>A_i</math>, si ottiene il prodotto <math>uv^2wx^2 y</math>, per il caso <math>i = 2</math>, e si può iterare questa sostituzione in maniera indefinita ottenendo sempre stringhe valide.</p> <p>In definitiva esiste sempre un albero di parsing per le stringhe <math>uv^iwx^i y</math> per ogni valore di <math>i</math>.</p>	

## Applicazione

Dato un linguaggio  $L$ , con il Pumping Lemma è possibile dimostrare che non è CFL.

### Esempio (1)

Sia  $L = \{ 0^m 10^n 10^m : m \geq 1 \}$ . Dimostrare che  $L$  non è CFL.

Assumendo per assurdo che  $L$  sia CFL, si consideri la stringa  $z = 0^n 10^n 10^n$ .

Allora:

- $z \in L$
- $|z| \geq n$

Per il Pumping Lemma si può scrivere  $z = uvwxy$  con  $|vwx| \leq n$  e  $|vx| > 0$  e, per qualsiasi  $i \geq 0$  deve essere  $uv^iwx^iy \in L$ .

Si osservano i seguenti casi:

- 1)  $vwx$  contiene solo 0  $\Rightarrow uvwxy = 0^k10^n10^n$  con  $k \neq n \Rightarrow uvwxy \notin L$
- 2)  $vwx$  contiene un gruppo di 0 vicini e un 1  $\Rightarrow uvwxy = 0^k10^n10^n$  con  $k \neq n \Rightarrow uvwxy \notin L$   
oppure  $uvwxy = 0^n1^k0^n10^n$  con  $k \neq 1 \Rightarrow uvwxy \notin L$
- 3)  $vwx$  contiene due gruppi di 0 divisi da un 1  $\Rightarrow uvwxy = 0^k10^k10^n$  con  $k \neq n \Rightarrow uvwxy \notin L$   
oppure  $uvwxy = 0^n10^k10^k$  con  $k \neq n \Rightarrow uvwxy \notin L$

### Esempio (2)

Sia  $L = \{ 0^{k*2^k} : k \geq 1 \}$ . Dimostrare che  $L$  non è CFL.

Assumendo per assurdo che  $L$  sia CFL, si consideri la stringa  $z = 0^{n*2^n}$ .

Allora:

- $z \in L$
- $|z| \geq n$

Per il Pumping Lemma si può scrivere  $z = uvwxy$  con  $|vwx| \leq n$  e  $|vx| > 0$  e, per qualsiasi  $i \geq 0$  deve essere  $uv^iwx^iy \in L$ .

Si osservi il seguente caso:

- $uv^2wx^2y \in L$
- valutando la lunghezza della stringa:  $n^2 < |uv^2wx^2y| \leq n^2 + n < n^2 + 2n + 1 = (n + 1)^2$ .  
Ma non esistono interi  $m$  per cui  $n^2 < m^2 < (n + 1)^2$ , pertanto  $uv^2wx^2y \notin L$  in contraddizione al punto precedente.



## Proprietà di chiusura per i CFL

Si mostrano alcune delle operazioni sui CFL per le quali è garantito che il risultato sia ancora un CFL.

**Teorema 7.24:** i CFL sono chiusi rispetto ai seguenti operatori: (i) unione, (ii) concatenazione, (iii) chiusura di Kleene e chiusura positiva (+).

Dimostrazione

Siano  $G_1 = (V_1, T_1, P_1, S_1)$  e  $G_2 = (V_2, T_2, P_2, S_2)$

### Unione

$G = G_1 \cup G_2 \Rightarrow G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\})$

$L(G) = L(G_1) \cup L(G_2)$

### Concatenazione

$G = G_1 \cdot G_2 \Rightarrow G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\})$

$L(G) = L(G_1) \cdot L(G_2)$

### Kleene

$G = G_1^* \Rightarrow G = (V_1 \cup \{S\}, T_1, S, P_1 \cup \{S \rightarrow \epsilon | S_1 S\})$

$L(G) = L(G_1)^*$

Osservazione: i CFL non sono chiusi rispetto all'intersezione

Esempio

$L_1 = \{0^n 1^n 2^i : n \geq 1, i \geq 1\} \Rightarrow L_1$  è CFL con grammatica:  $S \rightarrow AB \quad A \rightarrow 0A1 \mid 01 \quad B \rightarrow 2B \mid 2$

$L_2 = \{0^i 1^n 2^n : n \geq 1, i \geq 1\} \Rightarrow L_2$  è CFL con grammatica:  $S \rightarrow AB \quad A \rightarrow 0A0 \mid 0 \quad B \rightarrow 1B2 \mid 12$

$L_1 \cap L_2 = \{0^n 1^n 2^n : n \geq 1\}$  non è CFL, infatti, sia  $z = uvwxy$  si ha che:

- $vw$  è composto da soli 0, da soli 1 o da soli 2, in tal caso si avrebbe  $z = 0^n 1^m 2^l$  con due tra  $n, m$  e  $l$  differenti tra loro.
- $vw$  è composto da gruppi di due simboli "01" o "12", in tal caso si avrebbe  $z = 0^n 1^m 2^l$  con almeno due tra  $n, m$  e  $l$  differenti tra loro.

In entrambi i casi  $z \notin L_1 \cap L_2$ .

## Operazioni tra CFL e regolari

**Teorema 7.27:** se  $L$  è CFL e  $R$  è regolare allora  $L \cap R$  è CFL.

Dimostrazione

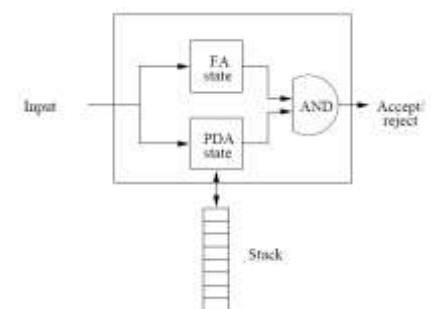
Se  $L$  è accettato dal PDA  $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_P, F_P)$  per stato finale

e se  $R$  è accettato dal DFA  $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$

è possibile provare per induzione che, definito il PDA  $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

in cui  $\delta((q, p), a, X) = \{(r, \delta_A(p, a), \gamma) : (r, \gamma) \in \delta_P(q, a, X)\}$ ,

$(q_P, w, Z_0) \vdash^* (q, \epsilon, \gamma) \text{ in } P \Leftrightarrow ((q_P, q_A), w, Z_0) \vdash^* ((q, \delta_A(q_A, w)), \epsilon, \gamma) \text{ in } P'$



**Teorema 7.29:** Siano  $L, L_1, L_2$  CFL e  $R$  regolare, allora:

1.  $L \setminus R$  è CFL
2.  $\underline{L}$  non è necessariamente CFL
3.  $L_1 \setminus L_2$  non è necessariamente CFL

Dimostrazioni

1.  $\underline{R}$  è regolare,  $L \cap \underline{R}$  per T7.27 è CFL e  $L \cap \underline{R} = L \setminus R$
2. Se  $\underline{L}$  fosse CFL allora  $L_1 \cap L_2 = \underline{L_1} \cup \underline{L_2}$  sarebbe sempre CFL
3. Si noti che  $\Sigma^*$  è CFL, quindi se  $L_1 \setminus L_2$  fosse CFL, allora lo sarebbe anche  $\Sigma^* \setminus L = \underline{L}$ , in contraddizione con 2

## Proprietà di decisione per i CFL

### Conversione fra CFG a PDA

Sia  $n$  la dimensione dell'input allora è valida la seguente tabella:

Conversione	Complessità
CFG $\rightarrow$ PDA	$O(n)$
PDA per stato finale $\rightarrow$ PDA per pila vuota	$O(n)$
PDA per pila vuota $\rightarrow$ PDA per stato finale	$O(n)$

Per convertire un PDA in una CFG si devono gestire, al massimo  $n^3$  variabili del tipo  $[pXq]$ .

Se  $(r, Y_1 Y_2 \dots Y_k) \in \delta(q, a, X)$  allora  $O(n^k) = O(n^n)$  regole della forma  $[qXr_k] \rightarrow a[rY_1 r_1] \dots [r_{k-1} Y_k r_k]$

- Introducendo  $k-2$  nuovi stati è possibile modificare il PDA per inserire al più un simbolo sulla pila per ogni transizione
- Di conseguenza  $k \leq 2$  per ogni regola; la dimensione totale del PDA rimane  $O(n)$
- Ogni transizione genera al più  $n^2$  produzioni
- La dimensione totale della CFG diventa  $O(n^3)$  con pari tempo per il calcolo.

### Conversione da CFG a CNF

Elementi positivi	Elementi negativi
<ul style="list-style-type: none"> <li>★ E' possibile eliminare i simboli inutili con tempo <math>O(n)</math>, calcolando i simboli generanti e raggiungibili</li> <li>★ E' possibile eliminare le produzioni unità in tempo <math>O(n^2)</math> e la grammatica risultante ha dimensione <math>O(n^2)</math></li> <li>★ Modificare i corpi affinché contengano solo variabili richiede tempo <math>O(n)</math></li> <li>★ E' possibile ridurre i corpi - affinché abbiano lunghezza 2 - in tempo <math>O(n)</math></li> </ul>	<ul style="list-style-type: none"> <li>❖ L'eliminazione dei simboli inutili può rendere la grammatica di dimensione <math>O(2^n)</math></li> </ul> <p>E' però possibile ridurre la dimensione dei corpi delle produzioni prima di eliminare i simboli annullabili.</p> <p>In conclusione la conversione ha complessità <math>O(n^2)</math>.</p>

### Verificare se un CFL è vuoto

Data una grammatica  $G$  per un CFL  $L$ , è possibile definire se  $L$  è vuoto tramite un algoritmo che determina se il simbolo iniziale  $S$  della grammatica sia generante o meno.

Sia  $n$  la lunghezza di  $G$ , potrebbero esistere un numero di variabili nell'ordine di  $n$  e ogni passo nella ricerca (induttiva) di generazione di variabili potrebbe richiedere  $O(n)$  tempo per esaminare tutte le produzioni di  $G$ . Se, ad ogni passo, si individuasse una sola variabile generante si avrebbe un costo stimato pari a  $O(n)$  passaggi; in questo modo un'implementazione *naive* la ricerca dei simboli generanti occupa, nel caso generale  $O(n^2)$ .

Utilizzando opportune strutture dati, l'algoritmo per la ricerca dei simboli generanti può essere  $O(n)$ .

## Verificare se una stringa appartiene ad un CFL

Supponendo che  $G$  sia CNF e che la stringa  $w$  abbia lunghezza  $|w| = n$ , da CNF si sa che l'albero sintattico è binario e contiene quindi  $2n - 1$  nodi interni.

Un algoritmo che utilizza queste informazioni potrebbe generare **tutti** gli alberi sintattici di  $G$  con  $2n - 1$  nodi interni e poi verificare se almeno uno di essi genera  $w$ . Questo algoritmo ha complessità esponenziale in  $n$ .

### Algoritmo CYK

Una tecnica più efficiente, basata sull'idea di programmazione dinamica, è implementata nell'algoritmo CYK<sup>1</sup>, noto anche come "algoritmo riempi tabella".

Data una grammatica  $G$  in CNF e data una stringa  $w = a_1a_2\dots a_n$  si costruisce una tabella triangolare nella quale  $X_{ij}$  è l'insieme di tutte le variabili  $A$  tali che  $A \Rightarrow_G^* a_ia_{i+1}\dots a_j$

Si riempie la tabella riga per riga, dal basso verso l'alto.

La prima riga è calcolata al passo base.

BASE:  $X_{ii} = \{A: A \rightarrow a_i \text{ è in } G\}$

Le successive al passo induttivo.

INDUZIONE

Calcolando  $X_{ij}$  (nella riga  $j - i + 1$ ), si ha che  $A \in X_{ij}$  se e solo se:

$$A \Rightarrow^* a_ia_{i+1}\dots a_j$$

ovvero  $A \rightarrow BCe$ , per qualche  $i \leq k < j$  si ha  
 $B \Rightarrow^* a_ia_{i+1}\dots a_k$  e  $C \Rightarrow^* a_{k+1}a_{k+2}\dots a_j$

che equivale a controllare che esista una produzione  $A \rightarrow BC$  con qualche  $i \leq k < j$  tale che  $B \in X_{ik}$  e  $C \in X_{(k+1)j}$

Osservazione: per calcolare  $X_{ij}$  si devono analizzare, al più,  $n$  coppie di insiemi precedentemente calcolati:  $(X_{ii}, X_{i+1,j}), (X_{i,i+1}, X_{i+2,j}), \dots, (X_{i,j-1}, X_{jj})$ . Per  $w = a_1a_2\dots a_n$  ci sono  $O(n^2)$  insiemi di  $X_{ij}$  da calcolare. Il costo computazionale totale è quindi  $O(n^3)$ .

$X_{15}$					
$X_{14}$	$X_{25}$				
$X_{13}$	$X_{24}$	$X_{35}$			
$X_{12}$	$X_{23}$	$X_{34}$	$X_{45}$		
$X_{11}$	$X_{22}$	$X_{33}$	$X_{44}$	$X_{55}$	
	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$

### Esempio

Si vuole testare se la stringa baaba appartiene alla grammatica  $G$  avente le seguenti produzioni:

$S \rightarrow AB|BC$

$A \rightarrow BA|a$

$B \rightarrow CC|b$

$C \rightarrow AB|a$

La prima riga si compila seguendo il passo base considerando le produzioni che hanno il terminale nel proprio corpo

{ S, A, C }				
-	{ S, C, A }			
-	{ B }	{ B }		
{ S, A }	{ B }	{ S, C }	{ S, A }	
{ B }	{ A, C }	{ A, C }	{ B }	{ A, C }
b	a	a	b	a

<sup>1</sup> Il nome deriva dalle iniziali delle tre persone che hanno, seppure indipendentemente, scoperto la stessa idea, J. Cocke, D. Younger and T. Kasami.

Per riempire le successive righe (verso l'alto) della tabella è necessario valutare come comporre la stringa di lunghezza sempre superiore considerando il contributo dei set di variabili precedentemente individuati.

Il significato della prima riga è che, ad esempio, per ottenere il terminale  $b$  è possibile utilizzare una produzione con la variabile  $B$  in testa.

Per ottenere la stringa  $ba$  formata da due terminali è necessario combinare le produzioni che hanno in testata le variabili identificate al primo passo:  $\{ B \}$  e  $\{ A, C \}$ , queste determinano le combinazioni  $BA$  e  $BC$ , il valore da mettere nella cella alla riga superiore è pertanto l'insieme delle variabili che sono in testa a produzioni che hanno  $BA$  o  $BC$  nel corpo, queste sono  $S$  e  $A$ .

Si tratta poi di proseguire valutando tutte le combinazioni binarie:

$ba = b \cup A$

$aa = a \cup a \quad ab = a \cup b$

$baa = b \cup aa \mid ba \cup a$   
 $\cup a$

$aab = a \cup ab \mid aa \cup b \quad aba = a \cup ba \mid ab$

$baab = b \cup aab \mid ba \cup ab \mid baa \cup b$

$aaba = a \cup aba \mid aa \cup ba \mid aab \cup a$

$baaba = b \cup aaba \mid ba \cup aba \mid baa \cup ba \mid baab \cup a$

Ognuna di queste combinazioni determina una combinazione binaria di variabili per le quali, se esiste una produzione che abbia la produzione nel suo corpo, è necessario inserire la variabile della testa nella cella corrispondente alla produzione della stringa, ricordando che i set identificati nella prima riga contribuiscono per un carattere, quelli della seconda per due caratteri e così via.

## Problemi indecidibili per CFL

I seguenti problemi per CFL sono indecidibili:

1. Una Context Free Grammar  $G$  è ambigua?
2. Un dato Context Free Language  $L$  inerentemente ambiguo?
3. L'intersezione di due Context Free Languages è vuota?
4. Due Context Free Languages sono uguali?
5. Un CFL è universale (cioè uguale a  $\Sigma^*$ )?

Per il 5: il complementare di un CFL non è CFL, pertanto, anche se si mostra che il complementare è o non è vuoto, non è possibile dimostrare che CFL è universale.

## Analisi lessicale e sintattica

I calcolatori sono in grado di eseguire programmi scritti in un linguaggio macchina, tale linguaggio - di basso livello - utilizza le istruzioni primitive per sfruttare le componenti delle specifiche architetture hardware.

Per rendere lo sviluppo software un'attività più semplice e nascondere alla programmazione la gestione di dettagli dipendenti dalle specifiche architetture, nel tempo si sono realizzati linguaggi di programmazione sempre più "distanti" dal linguaggio macchina (linguaggi di alto livello).

I programmi scritti in questi linguaggi hanno bisogno, per essere eseguiti, di essere **tradotti** nel linguaggio macchina.

La traduzione viene effettuata da altri programmi appositamente realizzati, questi sono **compilatori** e **interpreti** e definiscono la distinzione tra linguaggi compilati e linguaggi interpretati.

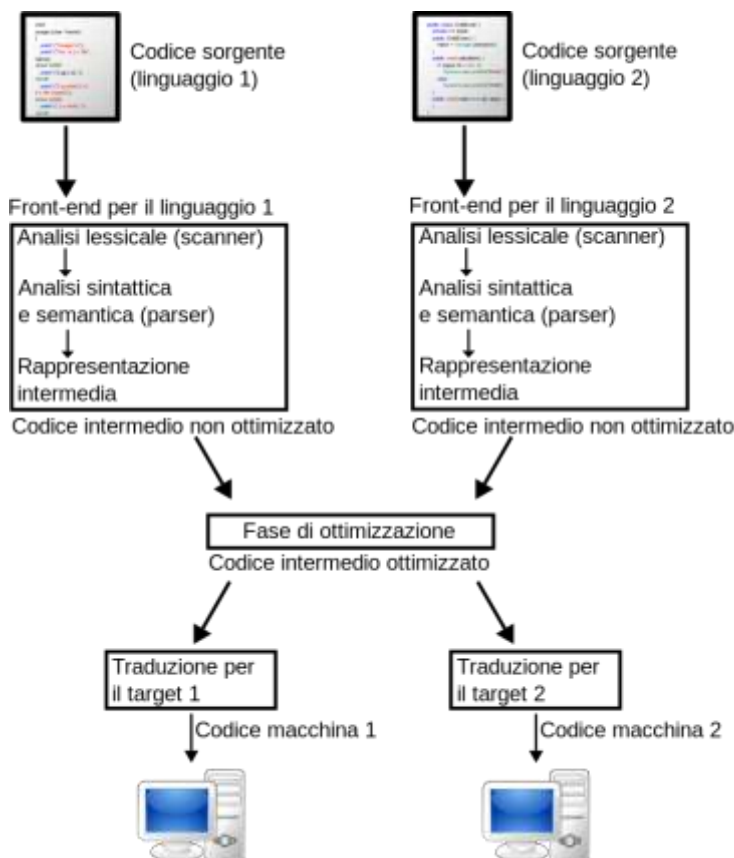
I **linguaggi compilati** prevedono che il codice sorgente di alto livello venga interamente tradotto nel linguaggio di destinazione (linguaggio macchina), il codice generato viene poi eseguito per ogni istanza richiesta.

I **linguaggi interpretati** prevedono invece che il programma sorgente venga tradotto nel linguaggio macchina istruzione dopo istruzione.

I linguaggi compilati generano solitamente codice più veloce mentre quelli interpretati forniscono una diagnostica migliore sugli errori.

Esiste un'implementazione dei compilatori che genera **codice intermedio** (es. Java), tale codice è pronto per essere eseguito su una qualsiasi architettura tramite una **macchina virtuale** specificatamente predisposta.

## Compilatori



## Struttura di un compilatore

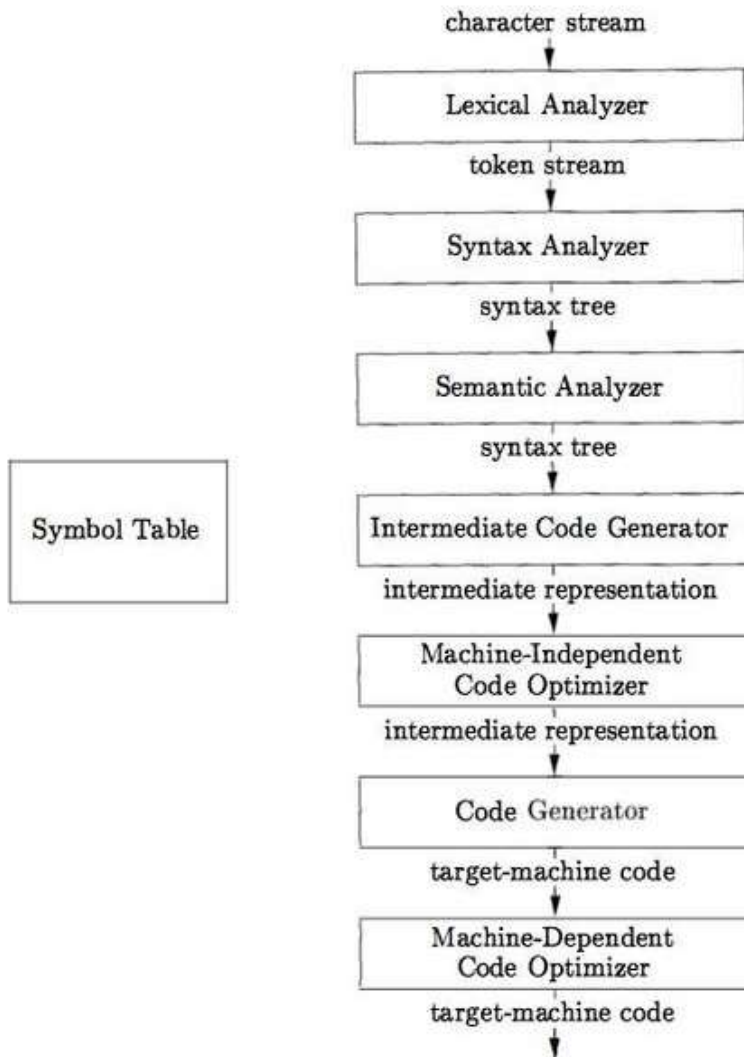


Figure 1.6: Phases of a compiler

### Analisi lessicale (scanning, lexer)

**Input:** charset stream

**Output:** tokens stream

**Input:** if x == y then z = 1; else z = 2;  
, z , = , 1 , ; , else , z , = , 2 , ;

**Output:** if , x , == , y , , then

La prima fase della compilazione è detta analisi lessicale o scanning ed è eseguita tramite un componente detto **lexer**.

Il lexer legge il codice sorgente come uno stream di caratteri e tenta di raggrupparli secondo il lessico del linguaggio e produce una serie di tokens (parole chiave riconosciute dal linguaggio).

Gli obiettivi del lexer sono:

1. partizionare i caratteri in input in sottostringhe dette **lessemi**
2. classificare i lessemi in conseguenza dei loro ruoli (ruoli = **tokens**).

Il lexer, nella pratica, deve quindi partizionare l'input e mapparli in una sequenza di tokens ignorando eventuali **whitespaces** e gestendo eventuali attributi associati ai lessemi.

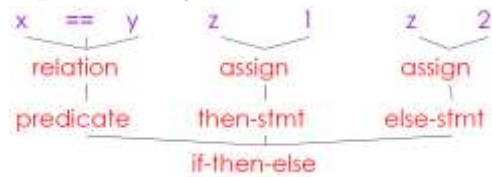
## Analisi sintattica (parsing, parser)

**Input:** tokens stream  
tree

**Output:** abstract syntax

**Input:** if x == y then z = 1; else z = 2;

**Output:** diagram



Ogni linguaggio di programmazione ha regole ben precise che definiscono quale debba essere la struttura di un programma sintatticamente corretto. La **sintassi** dei costrutti di un linguaggio di programmazione può essere descritta utilizzando una **grammatica libera dal contesto** o utilizzando la notazione **BNF (Backus-Naur-Form)**.

Alcuni vantaggi offerti dalle grammatiche:

- ★ una grammatica fornisce una specifica della sintassi di un linguaggio precisa e di facile comprensione
- ★ le specifiche della grammatica sono i punti di partenza utili per costruire un parser efficiente in grado di determinare la struttura sintattica di un programma, il processo di costruzione può rilevare eventuali ambiguità sintattiche e potenziali problemi non emersi nella fase iniziale della definizione di un linguaggio
- ★ la struttura imposta da una grammatica è utile per la traduzione di un programma sorgente in codice oggetto e per il rilevamento degli errori
- ★ una grammatica permette di sviluppare/estendere iterativamente un linguaggio, arricchendolo con nuovi costrutti in grado di svolgere nuove operazioni.

## Analisi semantica (type checking)

**Input:** abstract syntax tree  
enriched

**Output:** abstract syntax tree

L'analisi semantica parte dalle informazioni descritte dall'albero sintattico e nella tabella dei simboli per **verificare la consistenza** del programma sorgente rispetto alla definizione del linguaggio.

Durante questa fase l'albero sintattico e la tabella dei simboli vengono arricchiti di informazioni sui tipi utili alla successiva fase di generazione del codice.

L'analisi semantica effettua anche il **controllo sui tipi (type checking)** che assicura che ogni operatore abbia operandi di tipo adeguato, prima di procedere alla successiva fase. In certi casi, la specifica di un linguaggio potrebbe prevedere conversioni di tipo dette **coercizioni** per cui tipi differenti risultano comunque compatibili in certi casi (ad esempio con operatori di confronto su booleani e interi, interi e decimali).



## Code generation

Input: syntax tree enriched

Output: code

La fase di code generation può essere costituita da tre passi:

- generazione codice intermedio  
*molti compilatori generano una rappresentazione intermedia a basso livello* che può essere vista come un programma per una macchina virtuale. Tale rappresentazione dovrebbe avere due importanti proprietà: essere semplice da generare ed essere semplice da tradurre nel linguaggio macchina di destinazione.
- ottimizzazione
- generazione del codice  
in questa fase il codice intermedio viene tradotto nel programma destinazione in codice macchina, direttamente eseguibile sull'architettura hardware disponibile.

## Gestione tabella dei simboli

La tabella dei simboli è una struttura dati appositamente predisposta per memorizzare i nomi delle variabili utilizzate nel programma sorgente e le informazioni (attributi) necessarie ad essi collegati.

Le informazioni riguardano diversi aspetti:

- la classe di memorizzazione allocata
- la visibilità o scope (porzioni di programma in cui il nome può essere usato)
- numero, tipi e metodi di passaggio degli argomenti, tipo restituito (in caso di procedure).

Tale tabella deve essere efficientemente accessibile sia in lettura che in modifica.

## Analisi Lessicale: costruzione di un lexer

### Introduzione

I lexer vengono generati utilizzando dei programmi detti **lexer generator** che si occupano di implementare tutte le funzioni necessarie all'analisi lessicale data una grammatica.

Il generatore di lexer permette al programmatore di concentrarsi su **cosa** il lexer dovrebbe fare anziché sul **come** dovrebbe comportarsi.

**COSA:** tramite programmazione  
dichiarativa

**COME:** tramite programmazione  
imperativa

Si definisce allora una tabella di traduzione lessema → token.

Il codice generato per il lexer analizza il charset stream e, quando identifica un lessema, aggiunge il corrispondente token nello stream di tokens.

Tre concetti importanti nella descrizione dell'analisi lessicale sono:

- **token:** coppia costituita da un **nome** e un **attributo** opzionale; il token è il simbolo astratto che rappresenta uno specifico tipo di unità lessicale
- **pattern:** (o modello) è una descrizione compatta della forma che il lessema di un token può assumere
- **lessema:** è una sequenza di caratteri del programma che corrisponde al pattern di un token identificata dall'analizzatore lessicale come una specifica istanza di quel token.

Esempio:

token	pattern	lessemi di esempio
<b>if</b>	sequenza di caratteri "i" e "f"	if

<b>else</b>	sequenza di caratteri “e”, “l”, “s”, “e”	else
<b>comparison</b>	“<”, “>”, “<=”, “>=”, “==”, “!=”	<=, !=
<b>id</b>	lettera seguita da cifre	pi, score, D2
<b>number</b>	costante numerica qualsiasi	3.14159, 0, 6.02e23
<b>literal</b>	tutto tranne “ racchiuso tra “”	“core dumped”

Nella maggior parte dei linguaggi di programmazione i token possono essere suddivisi in cinque classi:

- **parole chiave:** per ogni parola chiave serve uno specifico token, il pattern coincide con la parola stessa
- **operatori:** un token può essere usato per rappresentare un singolo operatore o una classe di operatori
- **identificatori:** un token per tutti
- **costanti:** uno o più token (esempio numeri interi o stringhe)
- **segni d’interpunzione:** ad esempio i vari tipi di parentesi, la virgola, il punto e virgola, ...

## Lexer imperativo

Nella spiegazione si considerano solo questi tokens.

Token	Lessemi
ID	sequenza di una o più lettere e numeri che inizia con una lettera
EQUALS	“==”
PLUS	“+”
TIMES	“*”

## Maximal match rule

La funzione **undoNextChar** mostra una caratteristica dei lexer legata al fatto che i lessemi devono essere identificati nella loro lunghezza totale per essere tradotti in tokens, ma questo potrebbe non essere banale. Diventa necessario implementare una regola di **maximal match** per cui il lexer deve continuare a leggere caratteri finché non trova un carattere che non può essere parte del lessema, quando questo avviene è possibile determinare con certezza se il subset di caratteri letti fino a quel punto è un lessema valido e quale lessema sia.

Una tecnica per analizzare il programma sotto riportato, può seguire l’idea di continuare a leggere caratteri dallo stream finché non individua un pattern associato ad un token, se lo trova lo restituisce continua a leggere fino alla fine del programma. Un automa a stati finiti può implementare questo comportamento.

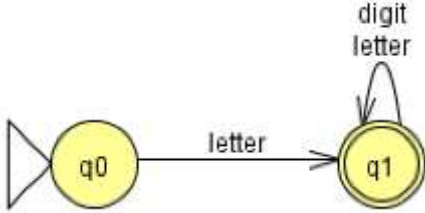
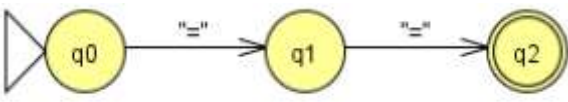
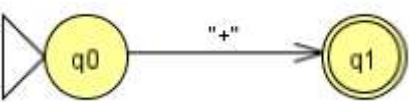

```

c=nextChar();
if (c == “=”) { c=nextChar(); if (c == “=”) {return EQUALS;}}
if (c == “+”) { return PLUS; }
if (c == “*”) { return TIMES; }
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) { c=NextChar(); }
    undoNextChar(c);
    return ID;
}
    
```

## Lexer dichiarativo

Come prima cosa si crea un automa a stati finiti per ogni token.

### Esempio:

Token	Lessemi	Automa a stati finiti
ID	sequenza di una o più lettere e numeri che inizia con una lettera	
EQUALS	"=="	
PLUS	"+"	
TIMES	"*"	

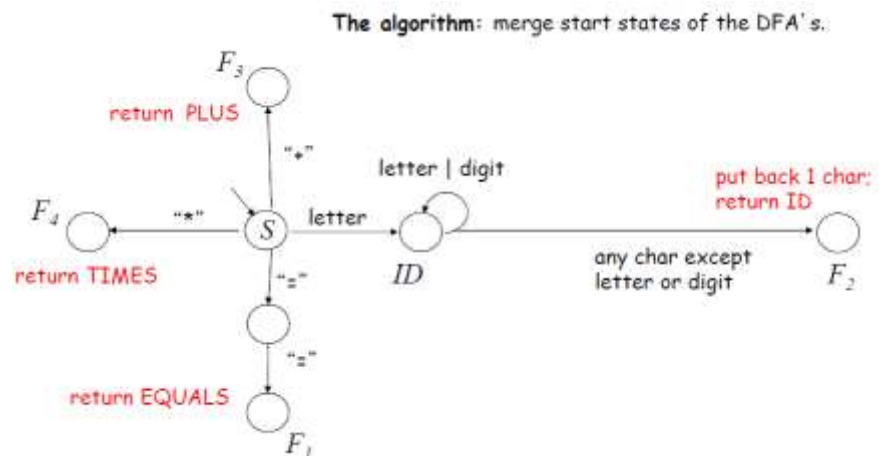
La seconda azione, quando un lessema viene accettato da un automa, può essere:

1. tornare indietro di un carattere (se è necessario effettuare look-ahead)
2. ritorarne il token corrispondente al lessema individuato.

Effettuando il merge dei diversi automi si ottiene un automa in grado di riconoscere i lessemi.

Tuttavia esiste un problema legato all'individuazione di particolari lessemi che potrebbero essere parzialmente ambigui, ad esempio come riconoscere un "=" di assegnazione da un "==" di confronto.

La soluzione si ottiene tramite un "full declarative lexer" descritto di seguito.



## Full declarative lexer

### Algoritmo

1. Si costruisce un automa per ogni lessema
2. Si effettua un merge degli automi a partire dallo stato iniziale
3. Se l'automato ottenuto è non deterministico si effettua una trasformazione per renderlo deterministico
4. Si applica una delle seguenti regole:
  - a. Quando si raggiunge uno stato finale:
    - i. Si memorizza la posizione dell'input (per un'eventuale "undo")

- ii. Si continuano a leggere ulteriori caratteri muovendo gli altri stati
- b. Quando ci si blocca non potendo procedere con la lettura del carattere successivo:
  - i. Si ritorna all'ultimo stato finale (posizione memorizzata al punt 4.a.i)
  - ii. Si restituisce il token associato allo stato finale memorizzato

## **Problemi da gestire nella pratica**

## Ambiguità

Problema: un lexer può raggiungere diversi stati finali contemporaneamente

Esempio: la stringa “if” può essere riconosciuta sia come ID che come IF (keyword)

Soluzione: definire una priorità tra i tokens (es. IF è prioritario rispetto ad iD).

## Scarto di whitespace

In caso di raggiungimento di uno stato finale per un whitespace (lessemi speciali da scartare), non si ritorna alcun token e si riparte da uno stato iniziale.

## Errori nell'input

Problema: come scartare lessemi illegali e stampare messaggi di errore.

Soluzione semplice (scarto carattere per carattere): aggiungere un lessema riconosciuto con qualsiasi carattere assegnandogli la priorità minima in modo tale che se nessun'altro lessema corrisponde all'input allora il carattere viene scartato.

## Regular expression

La modalità pratica di traduzione degli automi (adatti per realizzare una buona rappresentazione grafica ma di complessa gestione in forma testuale) consiste nella realizzazione delle specifiche in forma di **espressioni regolari**: una modalità compatta di definire un linguaggio che possa essere accettato da un automa. Le espressioni regolari sono un input per un lexer generator (definiscono ogni token, white-space, commenti che devono essere riconosciuti e ignorati).

In generale esistono alcuni operatori per le espressioni regolari:

- | significa “oppure”
- . significa “seguito da”
- \* significa “zero o più istanze di”
- + significa “una o più istanze di”
- () sono utilizzate per raggruppare

In alcuni “dialetti” per le espressioni regolari, si utilizzano delle abbreviazioni, ad esempio:

[a-z] è equivalente ad a|b|c|d|...|z

[0-9] è equivalente a 0|1|2|...|9

si possono usare gli apici per definire i caratteri speciali (es. ‘|’ per indicare un pipe anziché l’operatore “oppure”).

Ad esempio, l’espressione regolare per un token per un identificatore (una lettera seguita da 0 o più lettere o cifre) è [a-zA-Z].[a-zA-Z][0-9]\*

Gli operatori seguono un ordine di precedenza implicito:

Operatore per le espressioni regolari	Operatore aritmetico analogo	Precedenza
	+ (somma)	minima
.	x (prodotto)	media
*	^ (potenza)	massima

Si osservino ad esempio le due seguenti espressioni regolari che, ad una prima vista, potrebbero apparire equivalenti:

[a-zA-Z].[a-zA-Z][0-9]\*

lettera o da una sequenza di cifre

una lettera seguita da una

$[a-zA-Z].([a-zA-Z][0-9])^*$   
sequenza di lettere o cifre

una lettera seguita da una

## Lexer generators

Un **generatore lessicale** riceve in input tutte le espressioni regolari che descrivono i lessemi del linguaggio. I lexer generators generano codice (C, C++, Java, ...) che implementa l'algoritmo full declarative lexer a partire da lessemi forniti in input.

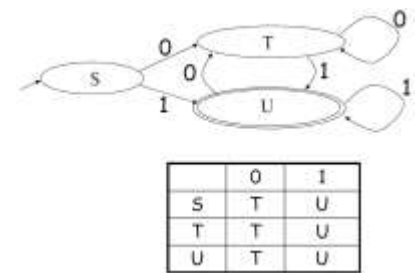
- traduce le espressioni regolari in automi a stati finiti
- effettua il merge di tutti gli automi in un unico automa
- traduce l'automa ottenuto in un automa deterministico (più efficiente)
- produce il codice che implementa la specifica simulazione per l'automa deterministico:
  - lookahead per la regola del maximal match
  - priorità in caso di match multipli
  - operazioni da eseguire durante il matching
  - ritorno allo stato iniziale.

Un DFA può essere implementato tramite una tabella bidimensionale T:

- una dimensione è definita gli **stati** dell'automa
- l'altra dimensione è definita dai **simboli in input**, tradizionalmente divisa in:
  - **terminali**: azioni
  - **non terminali**: goto
- per ogni transizione  $S_i \xrightarrow{a} S_k$ , è definita la cella  $T[i,a] = k$

L'esecuzione (molto efficiente) del DFA utilizza la tabella T:

- dallo stato  $S_i$  l'automa va allo stato  $S_k$  leggendo il simbolo a.



## Analisi sintattica: costruzione di un parser

### Introduzione

Un **parser** riceve una sequenza di **tokens** dal **lexer** e verifica se tale sequenza può essere generata dalla grammatica del linguaggio sorgente. Se un programma è ben formato, il parser costruisce un **albero di parsing** e lo *passa* alla parte restante del compilatore.

Esistono tre tipi di parser per le grammatiche:

- **universali**  
(algoritmo di Cocke, Younger e Kasami, algoritmo di Earley)  
possono trattare qualsiasi grammatica ma sono troppo inefficienti per l'utilizzo nei compilatori reali
- **top-down**  
costruiscono l'albero di parsing partendo dalla radice e scendendo verso le foglie
- **bottom-up**  
partono dalle foglie e procedono verso l'alto.

Nei parser top-down e bottom-up, la sequenza d'ingresso viene elaborata da sinistra verso destra, un simbolo alla volta.

Durante il parsing possono essere svolte molte attività:

- composizione della tabella dei simboli con le informazioni relative ai token
- controllo dei tipi
- altri tipi di analisi semantiche
- generazione di codice intermedio
- ...

Un parser effettua due azioni principali:

1. Syntax checking: un programma che contiene errori di sintassi è rifiutato e vengono fornite informazioni circa gli errori individuati
2. Costruzione del parse tree: finalizzata alla costruzione dell'abstract syntax tree.

Tuttavia sia possibile costruire un parser implementandolo a mano (possibile per linguaggi “piccoli” ma molto complesso per linguaggi di programmazione reali) esistono, come per i lexer, dei generatori in grado di scrivere il codice di un parser in automatico.

## Parse tree e abstract syntax tree

Un **albero di parsing (parse tree)** è una rappresentazione grafica di una derivazione che non dipende dall'ordine in cui le produzioni sono utilizzate per rimpiazzare i non-terminali:

- la radice è la variabile iniziale
- ogni nodo interno rappresenta l'applicazione di una produzione è etichettato con il non-terminale A che costituisce la testa della produzione
- i figli di un nodo interno, presi ordinatamente da sinistra verso destra, sono etichettati con i simboli che appaiono nel corpo della produzione utilizzata per sostituire la specifica occorrenza di A nella derivazione
- le foglie sono tutti simboli terminali

E	E: variabile iniziale
/ \	
- E	Produzione: $E \rightarrow -E$
/ \	
( E )	Produzione: $E \rightarrow ( E )$
/ \	
E + E	Produzione: $E \rightarrow E + E$
id id	Produzione: $E \rightarrow id$

Un parse tree contiene tutti i tokens, inclusi quelli che il parser individua ma che non sono utili per le fasi successive (parentesi, terminazioni di riga, ...).

In un **albero sintattico astratto (abstract syntax tree, AST)** ogni nodo rappresenta un operatore e i figli del nodo rappresentano gli operandi.

-   + / \ id id	Molti terminali di una grammatica rappresentano costrutti di programmazione mentre altri sono simboli di supporto di varia natura, questi vengono omessi negli AST in quanto generalmente non sono necessari.
-----------------------------	---

## Ambiguità

Una grammatica che produce più di un albero di parsing per una stessa frase è una grammatica ambigua. In altri termini una grammatica è ambigua se ammette più di una derivazione sinistra o più di una derivazione destra per una stessa frase.

Ad esempio una grammatica per il costrutto if-then-else può essere ambigua:

STMT  $\rightarrow$  **if** expr **then** STMT

| **if** expr **then** STMT **else** STMT

In questa grammatica lo statement condizionale **if** E1 **then** **if** E2 **then** S1 **else** S2 ammette due alberi di parsing in quanto, intuitivamente senza l'uso di parentesi, il ramo **else** S2 può essere l'alternativa sia per la condizione E1 che per la condizione E2, a seconda dell'interpretazione:

**if** E1 **then** ( **if** E2 **then** S1 **else** S2 )

oppure

**if E1 then ( if E2 then S1 ) else S2**

Se possibile allora si procede con la disambiguazione di una grammatica. Talvolta è però necessario definire **regole di disambiguazione** che scartino alberi di parsing indesiderati, definendo ad esempio, priorità tra gli operatori.

## Eliminazione ambiguità pg 187 dragon book

### Syntax checking

La sintassi di un linguaggio di programmazione è un linguaggio libero che richiede un automa a pila per il suo riconoscimento.

Il formalismo utilizzato per descrivere le regole sintattiche di un linguaggio è costituito dalle grammatiche (input per i parser generators).

Un programma non presenta errori sintattici se può essere derivato da una grammatica, tuttavia una grammatica definisce come derivare le stringhe: non come verificare se una stringa sia o meno derivabile.

Una possibile soluzione a questo problema consiste nella derivazione di ogni possibile stringa e nella verifica che il programma sia uguale ad una stringa generata: sebbene appaia come una soluzione tutt'altro che ottima ed efficiente esistono parser che la implementano.

### Parse tree construction

### Insiemi First e Follow

L'algoritmo per la costruzione della tabella di parsing si avvale di due funzioni ausiliarie: la FIRST e la FOLLOW.

**FIRST( $\alpha$ )**, definita su stringhe  $\alpha \in (V \cup \Sigma)^*$ , restituisce l'insieme dei terminali con cui iniziano stringhe derivabili da  $\alpha$ ,  $\text{FIRST}(\alpha) = \{ b \mid \alpha \rightarrow^* b\beta \} \cup \{ \epsilon \mid \alpha \rightarrow^* \epsilon \}$ , ossia:

- se  $\alpha \rightarrow^* a\beta$  allora  $a \in \text{FIRST}(\alpha)$
- può anche contenere la stringa:  $\alpha \rightarrow^* \epsilon$  implica  $\epsilon \in \text{FIRST}(\alpha)$

In base alla definizione il FIRST di un simbolo si calcola così:

1.  $\text{FIRST}(b) = \{ b \}$  se  $b$  è un simbolo terminale
2. Per tutte le produzioni del tipo  $X \rightarrow A_1 \dots A_n$  con  $n \geq 0$ :
  - a.  $\text{FIRST}(X) = \emptyset$
  - b.  $\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(A_1) - \{ \epsilon \}$ . Se  $\epsilon \notin \text{FIRST}(A_1)$  termina.
  - c.  $\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(A_2) - \{ \epsilon \}$ . Se  $\epsilon \notin \text{FIRST}(A_2)$  termina.
  - d. ...
  - e.  $\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(A_n) - \{ \epsilon \}$ . Se  $\epsilon \notin \text{FIRST}(A_n)$  termina.
  - f.  $\text{FIRST}(X) = \text{FIRST}(X) \cup \{ \epsilon \}$

**FOLLOW(A)**, definita su non terminali della grammatica, restituisce l'insieme dei terminali che compaiono immediatamente a destra di  $A$  in qualche forma sentenziale della grammatica,  $\text{FOLLOW}(A) = \{ b \mid S\$\rightarrow^* \beta A b \delta \}$ :

- Se  $S \rightarrow^* \alpha A a \beta$  allora  $a \in \text{FOLLOW}(A)$
- può anche contenere il simbolo  $\$$ :  $S \rightarrow^* \alpha A$  implica  $\$ \in \text{FOLLOW}(A)$

Secondo la definizione il FOLLOW di un simbolo non terminale si calcola così:

1. se  $S$  è il simbolo iniziale della grammatica, allora  $\$$  sta in  $\text{FOLLOW}(S)$
2. se  $A \rightarrow \alpha B \beta$  è una produzione della grammatica, tutti i simboli terminali in  $\text{FIRST}(\beta) - \{ \epsilon \}$  stanno in  $\text{FOLLOW}(B)$
3. se  $A \rightarrow \alpha B$  è una produzione della grammatica, oppure se  $A \rightarrow \alpha B \beta$  è una produzione della grammatica e  $\epsilon \in \text{FIRST}(\beta)$ , tutti i simboli in  $\text{FOLLOW}(A)$  stanno in  $\text{FOLLOW}(B)$ .



## Esempio di calcolo insiemi FIRST E FOLLOW.

Si calcolano FIRST e FOLLOW per la grammatica:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Simbolo	Terminale?	FIRST	FOLLOW
E	No	$\{ (, id \}$ Produzioni osservate: $E \rightarrow TE' \Rightarrow T \rightarrow FT' \Rightarrow F \rightarrow (E) \mid id$ <b>E</b> è il simbolo iniziale $\Rightarrow$ ogni stringa del linguaggio inizia con ( oppure con <b>id</b>	$\{ \$, ) \}$ E è il simbolo iniziale $\Rightarrow$ \$ è in FOLLOW Dalla produzione $F \rightarrow (E) \mid id$ si ha che il simbolo ) è nel FOLLOW.
E'	No	$\{ +, \epsilon \}$ Produzioni osservate: $E' \rightarrow +TE' \mid \epsilon$	$\{ \$, ) \}$ Le produzioni che hanno E' nel corpo: $E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$ determinano che FOLLOW(E) sia nel FOLLOW(E')
T	No	$\{ (, id \}$ Produzioni osservate: $T \rightarrow FT' \Rightarrow F \rightarrow (E) \mid id$ <b>E</b> è il simbolo iniziale $\Rightarrow$ ogni stringa del linguaggio inizia con ( oppure con <b>id</b>	$\{ +, ), \$ \}$ Nelle produzioni che hanno T nel corpo $E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$ il simbolo successivo è E' quindi, va aggiunto il FIRST(E')- $\{ \epsilon \}$ e, siccome il FIRST(E') contiene $\epsilon$ si aggiunge anche il FOLLOW(E') e il FOLLOW(E)
T'	No	$\{ *, \epsilon \}$ Produzioni osservate: $T' \rightarrow *TE' \mid \epsilon$	$\{ +, ), \$ \}$ Delle produzioni $T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$ si deduce che va aggiunto, al FOLLOW(T'), il FOLLOW(T)
F	No	$\{ (, id \}$ Produzioni osservate: $F \rightarrow (E) \mid id$ <b>E</b> è il simbolo iniziale $\Rightarrow$ ogni stringa del linguaggio inizia con ( oppure con <b>id</b>	$\{ *, +, ), \$ \}$ Nelle produzioni che hanno F nel corpo $T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$ il simbolo successivo è T' quindi, va aggiunto il FIRST(T')- $\{ \epsilon \}$ e, siccome il FIRST(T') contiene $\epsilon$ si aggiunge anche il FOLLOW(T') e il FOLLOW(T)
+	Si	$\{ + \}$	
*	Si	$\{ * \}$	
(	Si	$\{ ( \}$	
)	Si	$\{ ) \}$	
id	Si	$\{ id \}$	

## Algoritmi di parsing top-down

L'idea dei parser top-down consiste nella costruzione dell'albero AST partendo dalla radice andando verso le foglie, è un tentativo di costruzione di una derivazione **leftmost** della stringa in input: ad ogni passo si espande il non terminale più a sinistra che non ha figli.

Il problema principale consiste nell'operare una scelta corretta della produzione con la quale espandere il nodo selezionato.

**Se esiste un albero di derivazione e la grammatica non è ambigua, allora ad ogni passo esiste una ed una sola scelta corretta (una sola derivazione leftmost possibile).**

Per effettuare tale scelta si possono utilizzare:

- **parser con backtracking**
  - provano a derivare la stringa e quando arrivano ad una derivazione impossibile devono “tornare indietro” e tentare un’altra strada
  - sono piuttosto inefficienti in quanto, nel caso pessimo, devono operare tutte le scelte possibili per tutti i non terminali possibili
- **parser predittivi (predictive parsing)**
  - “predicono” la produzione che porta alla derivazione della stringa
  - analizzano un numero di terminali minimo (tipicamente 1) necessari per prendere la scelta giusta (simboli di lookahead)
  - per ogni non terminale  $A$  con alternative  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  e per ogni simbolo di lookahead  $a$ , i parser predittivi identificano la sola alternativa di  $A$  in grado di generare stringhe che cominciano per  $a$

## Recursive Descent Parsing

Considerando la grammatica di esempio:

$E \rightarrow T + E \mid T$

$T \rightarrow (E) * T \mid (E) \mid \text{int} * T \mid \text{int}$

e lo stream di tokens: **int \* int**

Un parser con backtracking prova tutte le regole finché non si blocca o non produce la stringa:

$E \rightarrow T + E \Rightarrow E \rightarrow (E) * T + E$	FAIL
$E \rightarrow T + E \Rightarrow E \rightarrow \text{int} * T + E \Rightarrow E \rightarrow \text{int} * (E) * T + E$	FAIL
$E \rightarrow T + E \Rightarrow E \rightarrow \text{int} + E$	FAIL
$E \rightarrow T \Rightarrow (E) * T$	FAIL
$E \rightarrow T \Rightarrow (E)$	FAIL
$E \rightarrow T \Rightarrow \text{int} * T \Rightarrow \text{int} * (E) * T$	FAIL
$E \rightarrow T \Rightarrow \text{int} * T \Rightarrow \text{int} * (E)$	FAIL
$E \rightarrow T \Rightarrow \text{int} * T \Rightarrow \text{int} * \text{int} * T$	FAIL
$E \rightarrow T \Rightarrow \text{int} * T \Rightarrow \text{int} * \text{int}$	SUCCESS

L’implementazione di un recursive descent parser può essere fatta creando una funzione booleana per ogni possibile produzione che “avanzi” l’elaborazione in caso di match su una stringa data partendo dal simbolo iniziale ed esplori ogni singola possibile produzione successiva.

L’elaborazione termina con successo se riesce a produrre la stringa (si suppone di utilizzare un simbolo speciale \$ di fine stringa per identificare il raggiungimento della fine).

L’implementazione è semplice ma non funziona per tutti i casi.

## Ricorsione a sinistra

Una grammatica si dice **ricorsiva a sinistra** se esiste un non terminale  $A$  tale che:  $A \rightarrow^+ A\alpha$ .

Se una grammatica è ricorsiva a sinistra l’algoritmo non termina mai e pertanto non può essere applicato. Ricorsione può essere

- **immediata**, es.  $A \rightarrow A\alpha \mid \beta$
- **non immediata**, es.  $A \rightarrow S\alpha \mid \beta \quad S \rightarrow A\delta$

Nel caso della ricorsione immediata, la grammatica  $A \rightarrow A\alpha \mid \beta$  può essere riscritta eliminando la ricorsione a sinistra:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

In generale una ricorsione immediata a sinistra può essere eliminata seguendo queste regole:

- sia  $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_n$
- tutte le stringhe derivate dal simbolo  $A$  iniziano con  $\beta_1, \dots, \beta_n$  e proseguono con diverse istanze di  $\alpha_1, \dots, \alpha_n$
- pertanto si può riscrivere la grammatica come:  
$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \epsilon$$

Nel caso della ricorsione non immediata, una grammatica tipo

$$S \rightarrow A\alpha \mid \beta$$

$$A \rightarrow S\delta$$

è ricorsiva a sinistra e può essere ricondotta ad una grammatica non ricorsiva

$$S \rightarrow A\alpha \mid \beta$$

ric conducendo al caso della ricorsione immediata  $A \rightarrow S\delta$  si ottiene  $A \rightarrow A\alpha\delta \mid \beta\delta$

e applicando le regole per la ricorsione immediata

$$S \rightarrow A\alpha \mid \beta$$

$$A \rightarrow \beta\delta A'$$

$$A' \rightarrow \alpha\delta A' \mid \epsilon$$

Algoritmo per eliminare le ricorsioni:

INPUT: **grammatica con ricorsione a sinistra senza cicli ( $A \rightarrow^+ A$ ) ed  $\epsilon$ -produzioni ( $A \rightarrow \epsilon$ )**

OUTPUT: **grammatica equivalente senza ricorsione a sinistra**

1. Ordinare i non terminali  $A_1, A_2, \dots, A_n$
2. Eseguire il seguente algoritmo  
**for**  $k:=1$  **to**  $n$  **do begin**  
    **for**  $j:=1$  **to**  $k-1$  **do begin**  
        sostituisci ogni produzione  $A_k \rightarrow A_j\gamma$  con le produzioni:  
         $A_k \rightarrow \delta_1\gamma \mid \dots \mid \delta_m\gamma$    dove  
         $A_j \rightarrow \delta_1 \mid \dots \mid \delta_m$    sono le attuali produzioni per  $A_j$   
    **end**  
    elimina la ricorsione immediata per il non terminale  $A_k$   
**end**

In conclusione, i descent recursive parser:

- implementano una strategia di parsing molto semplice
- richiedono l'eliminazione della ricorsione a sinistra (che può essere fatta automaticamente)
- sono impopolari a causa del backtracking (inefficiente)
- sono abbastanza validi per linguaggi "piccoli"
- sono condizionati significativamente dall'ordine seguito per valutare le produzioni.

## Predictive Parsing

Ciò che manca ad un descent recursive parser è un "qualcosa" in grado di aiutarlo a scegliere quale produzione - tra quelle disponibili - deve essere utilizzata per ottenere la stringa.

Quel "qualcosa" può essere un simbolo di **lookahead** che permette di predire la giusta produzione da utilizzare, questo permette di evitare le dinamiche e le conseguenze collegate al backtracking.

I parser predittivi, per ogni non terminale  $A$  con alternative  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  sono in grado di identificare, per ogni simbolo di lookahead **a**, la sola produzione in grado di generare stringhe che cominciano per **a**. Esistono parser predittivi ricorsivi e non ricorsivi (iterativi, utilizzano uno stack).

## Grammatica LL(1)

Una grammatica si dice LL(1) se esiste una tabella di parsing predittivo che non ha entrate multiple.

Significato della sigla LL(1):

L (Left): indica che l'input è analizzato a partire da sinistra

L (Leftmost): indica che il parsing produce derivazioni leftmost della stringa

1: rappresenta il lookahead e indica che si effettua un lookahead di 1 simbolo

La classe di grammatiche LL(1) è sufficientemente ampia da comprendere la maggior parte dei linguaggi di programmazione, è però necessario scrivere con attenzione una tale grammatica in quanto, ad esempio, nessuna grammatica ambigua o che presenti ricorsione a sinistra può essere LL(1).

Una grammatica  $G$  è LL(1) se e solo se, date due produzioni distinte  $A \rightarrow \alpha | \beta$  appartenente a  $G$ , le seguenti condizioni sono verificate:

1. non esiste alcuna terminale  $a$  tale che le stringhe che iniziano per  $a$  siano derivate sia da  $\alpha$  che da  $\beta$
2. al più una tra  $\alpha$  e  $\beta$  deriva la stringa nulla  $\epsilon$
3. se  $\beta \Rightarrow^* \epsilon$  allora  $\alpha$  non deriva alcuna stringa che inizia con un terminale appartenente al  $\text{Follow}(A)$  e viceversa  $\alpha \Rightarrow^* \epsilon$  allora  $\beta$  non deriva alcuna stringa che inizia con un terminale appartenente al  $\text{Follow}(A)$ .

## Left factoring (fattorizzazione a sinistra)

Le produzioni di alcune grammatiche potrebbero introdurre un'ulteriore grado di valutazione, ad esempio la grammatica

$$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid a$$

$$E \rightarrow b$$

rende difficile la scelta di una delle due produzioni.

La soluzione è allora quella di "fattorizzare" le produzioni in base al prefisso comune:

$$S \rightarrow \text{if } E \text{ then } S' \mid a$$

$$S' \rightarrow \text{else } S \mid \epsilon$$

$$E \rightarrow b$$

Algoritmo per la fattorizzazione a sinistra:

INPUT: **grammatica**

OUTPUT: **grammatica equivalente fattorizzata a sinistra**

1. per ogni non terminale  $A$ 
  - a. individuare il prefisso più lungo  $\alpha$  per le sue diverse produzioni
  - b. se  $\alpha \neq \epsilon$  sostituire  $A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \dots \mid \gamma_k$  con
 
$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_k$$

$$A' \rightarrow \beta_1 \mid \dots \mid \beta_n$$

Algoritmo per la creazione della tabella di parsing LL(1)

INPUT: **grammatica  $G$**

OUTPUT: **tabella di parsing  $M$  per la grammatica  $G$**

1. Per ogni produzione  $A \rightarrow \alpha$ 
  - a. per ogni terminale  $a \in \text{FIRST}(\alpha)$ , aggiungere  $A \rightarrow \alpha$  nella cella  $M(A, a)$  della tabella
  - b. se  $\epsilon \in \text{FIRST}(\alpha)$ 
    - i. per ogni  $b \in \text{FOLLOW}(A)$  aggiungere  $A \rightarrow \alpha$  nella cella  $M(A, b)$  della tabella
2. Ogni entrata senza simboli (indefinita) è in errore.

Ogni intestazione di riga è un simbolo non terminale.

Ogni intestazione di colonna è un simbolo terminale o \$.

Esempio: costruzione tabella di parsing per una grammatica.

In tabella si riportano la grammatica e i FIRST e i FOLLOW.

Grammatica	Simbolo	FIRST	FOLLOW
$E \rightarrow TE'$	E	{ ( , id }	{ \$ , ) }
$E' \rightarrow +TE' \mid \epsilon$	E'	{ +, $\epsilon$ }	{ \$ , ) }
$T \rightarrow FT'$	T	{ ( , id }	{ +, ), \$ }
$T' \rightarrow *FT' \mid \epsilon$	T'	{ *, $\epsilon$ }	{ +, ), \$ }
$F \rightarrow (E) \mid \text{id}$	F	{ ( , id }	{ *, +, ), \$ }

Si ricorda che i FIRST dei terminali contengono solo i terminali.

## Tabella di Parsing:

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Il procedimento di creazione della tabella di parsing può essere applicato ad una qualsiasi grammatica context-free. Tuttavia, per alcune grammatiche, la tabella può avere delle entrate indefinite (ovvero più di un valore nella stessa casella  $M(A,a)$  della tabella).

Se una grammatica  $G$  è ambigua o ricorsiva a sinistra, allora la tabella  $M$  avrà almeno un'entrata indefinita.

### Struttura di un predictive non recursive parser

LL(1) è una tecnica di parsing predittiva ricorsiva.

Un parser predittivo non ricorsivo può essere costruito gestendo uno stack esplicitamente e riproducendo il processo di derivazione sinistra.

Un parser predittivo ha la seguente struttura:

- uno stack che contiene in simboli terminali più il simbolo  $\$$
- una tabella di parsing  $M$  indicizzata da non terminali (righe) e dai simboli terminali più il simbolo  $\$$  (colonne)
- la entry  $M[A,a]$  della tabella indica quale mossa seguire data la variabile  $A$  e il carattere in input  $a$ 
  - $M[A,a]$  restituisce o una produzione oppure un errore
- il comportamento del parser dipende dal simbolo  $X$  in testa allo stack e dal simbolo in input corrente
- l'output del programma è:
  - un albero di derivazione per la stringa in input oppure
  - un messaggio di errore

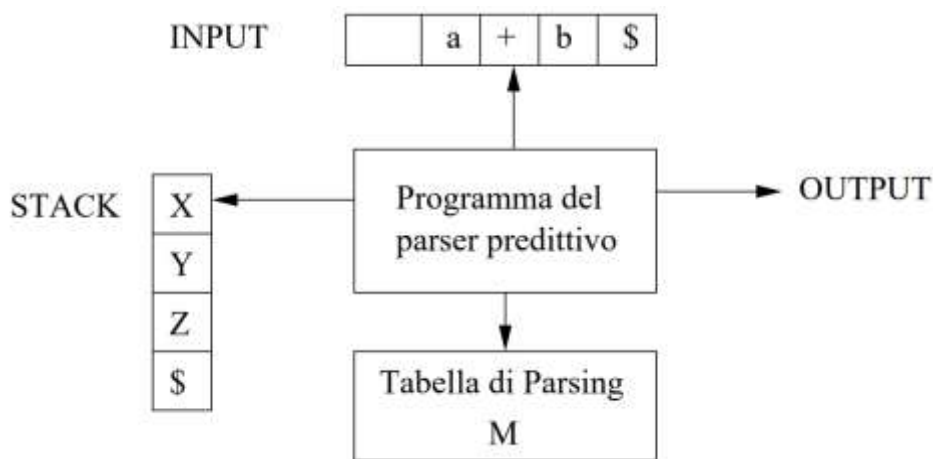
### Algoritmo predictive non recursive parsing

**INPUT:** una stringa  $w$  e una tabella di parsing  $M$  relativa ad una grammatica  $G$

**OUTPUT:** appartenenza di  $w$  a  $L(G)$ , se  $w \in L(G)$  una derivazione sinistra di  $w$  altrimenti un errore

### METODO

Inizialmente il parser è in una configurazione con  $w\$$  nel buffer d'ingresso e il simbolo iniziale  $S$  della grammatica  $G$  sullo stack, al di sopra del marcatore  $\$$ .



Algoritmo predictive non recursive parsing (uso della tabella di parsing)

INPUT: una stringa  $w$  e una tabella di parsing  $M$  relativa ad una grammatica  $G$

OUTPUT: appartenenza di  $w$  a  $L(G)$ , se  $w \in L(G)$  una derivazione sinistra di  $w$  altrimenti un errore

METODO

Un programma di parsing usa la tabella di parsing per analizzare la stringa in input.

Inizialmente:

- lo stack contiene  $SS$  ( $S$  è la variabile iniziale)
- l'input è  $w\$$ .

Sia  $X$  il simbolo al top dello stack e  $a$  il simbolo in input corrente:

1. se  $X = a = \$ \Rightarrow$  il parser termina con successo
2. se  $X = a \neq \$ \Rightarrow$  elimina  $a$  dallo stack e fa avanzare il simbolo di lookahead
3. se  $X$  è un non terminale  $\Rightarrow$  consulta l'entry  $M[X,a]$  della tabella di parsing:
  - a. se  $M[X,a] = X \rightarrow \alpha \Rightarrow$  elimina  $X$  dallo stack e inserisce i simboli di  $\alpha$  in modo tale che il simbolo più a sinistra sia in testa allo stack  
 Es. se  $X \rightarrow UVW$  esegue  $\text{pop}()$ ;  $\text{push}(W)$ ;  $\text{push}(V)$ ;  $\text{push}(U)$ ; e stampa  $X \rightarrow \alpha$
  - b. Se  $M[X,a] = \text{error} \Rightarrow$  termina e stampa un errore

Tabella di parsing						
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

id + id * id \$			out
STACK	w	out	
\$E	id+id*id\$	$E \rightarrow TE'$	
\$E'T	id+id*id\$	$T \rightarrow FT'$	
\$E'T'F	id+id*id\$	$F \rightarrow \text{id}$	id
\$E'T'	+id*id\$	$T' \rightarrow \epsilon$	
\$E'	+id*id\$	$E' \rightarrow +TE'$	+
\$E'T	id*id\$	$T \rightarrow FT'$	
\$E'T'F	id*id\$	$F \rightarrow \text{id}$	id
\$E'T'	*id\$	$T' \rightarrow *FT'$	*
\$E'T'F	id\$	$F \rightarrow \text{id}$	id
\$E'T'	\$	$T' \rightarrow \epsilon$	
\$E'	\$	$E' \rightarrow \epsilon$	
\$	\$	FINE	OK

Il risultato è un AST:

```

      E
     /\
    T  E'
   /\ /\
  F T' + T E'
  | | /\ |
id ∈ F T' ∈
    | /\
    id * F T'
    | |
    id ∈
    
```

## Algoritmi di parsing bottom-up

Come suggerito dal nome, gli algoritmi di parsing bottom-up costruiscono l'albero di derivazione partendo dalle foglie verso la radice. Questi algoritmi utilizzano una particolare tecnica detta **shift-reduce parsing** che rappresenta un tentativo di ridurre la stringa in input al non terminale iniziale della grammatica.

Ogni passo di riduzione consiste nell'individuare una sottostringa  $\alpha$  che faccia match con la parte destra di una qualche produzione  $A \rightarrow \alpha$  e nel sostituire tale sottostringa con il non terminale  $A$ .

Il procedimento corrisponde ad una derivazione **rightmost** al contrario della stringa di input.

Considerando ad esempio la grammatica:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

e la stringa **abbcede**, una derivazione rightmost è:  $S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcede$

## Chaotic bottom-up parser and non deterministic chaotic bottom-up parser

Idea di un algoritmo

1. Fissata una stringa  $s$  in input ed una grammatica
2. trovare in  $s$  il corpo  $r$  di una produzione  $N \rightarrow r$
3. **ridurre** la stringa  $r$  alla sua variabile non terminale  $N$
4. se tutta la stringa è ridotta a non terminali si è ottenuto un parse tree
5. altrimenti ritorna allo step 1 (con la stringa ridotta).

L'algoritmo descritto potrebbe non riuscire ad effettuare il parsing corretto della stringa.

Può essere d'aiuto utilizzare il non determinismo:

1. Fissata una stringa  $s$  in input ed una grammatica
2. trovare in  $s$  tutte le  $k$  stringhe che possono essere ridotte
3. creare  $k$  copie dell'input (eventualmente già parzialmente ridotto)
4. in ogni istanza, eseguire una delle  $k$  riduzioni
5. se un'istanza del parser riduce la stringa si è ottenuto un parse tree
6. altrimenti ritorna allo step 1 (per ogni istanza).

Il parse avrà esito positivo in almeno una delle istanze, tuttavia - a causa delle copie ( $k*k*k*...*k$ ) - l'algoritmo ha una crescita esponenziale di uso delle risorse e tempo. Inoltre molte istanze del parsers producono il parse tree corretto, con spreco di attività.



## LR parser

Il tipo più comune di parsing adottato si basa sul concetto noto come LR(k), come i parser LL non ricorsivi si basa sull'uso di tabelle: quando è possibile creare una tabella con un metodo "LR" per una grammatica G si può affermare che G sia LR.

Il parsing LR è interessante in quanto:

- si può costruire un parser LR capace di riconoscere virtualmente tutti i costrutti dei linguaggi di programmazione con grammatiche libere dal contesto (esistono grammatiche libere dal contesto non LR ma non strettamente necessarie per i tipici costrutti dei linguaggi di programmazione)
- il parsing LR è il metodo di parsing shift-reduce più generale privo di backtracking e può essere implementato in maniera efficiente
- un parser LR è in grado di rilevare un errore sintattico appena possibile in una scansione da sinistra a destra della stringa di ingresso
- la classe delle grammatiche riconosciute da un parser LR è un sovrainsieme proprio di quelle riconosciute con metodi predittivi o LL.

Il principale inconveniente del metodo LR è che la costruzione manuale di un parser richiede troppo lavoro, è necessario uno strumento apposito: un generatore di parser LR.

In un parser LR non deterministico si suddivide l'input in due parti:

- **destra:** la parte di stringa ancora da esaminare
- **sinistra:** la parte di stringa già esaminata (oggetto delle riduzioni)

e può effettuare le riduzioni solo in parti adiacenti allo split (punto di divisione delle parti sinistra e destra).

La parte destra è memorizzata in un buffer (la stringa  $w\$$ ), la parte sinistra in uno stack (inizialmente  $\$$ ).

La sigla LR definisce la logica del parser:

- L (left): effettua lo scan della stringa in input da sinistra a destra
- R (rightmost): cerca la derivazione più a destra

Un parser LR costruisce un parse tree, se esiste.

Azioni di un parser LR non deterministico:

- |                  |  |
|------------------|--|
| • <b>reduce:</b> | ricosce un handle al top dello stack e decide con quale non terminale rimpiazzarlo                     |
| • <b>shift:</b>  | sposta il prossimo simbolo in input al top dello stack (abilitando, potenzialmente, riduzioni diverse) |
| • <b>accept:</b> | definisce il completamento, con successo, del parsing  |
| • <b>error:</b>  | individua un errore sintattico ed avvia una procedura di recovery dell'errore                          |

Queste azioni sono scelte non deterministicamente.

Algoritmo per un parser LR non deterministico	Algoritmo per un caotico parser non deterministico
<ol style="list-style-type: none"> <li>1. Fissata una stringa in input e una grammatica</li> <li>2. trovare in s tutte le k stringhe che possono essere ridotte</li> <li>3. creare k copie dell'input</li> <li>4. in ogni istanza, eseguire non effettuare riduzioni ma shift e tornare allo step 2</li> <li>5. quando un'istanza del parser riduce la stringa al non terminale di partenza, interrompere</li> </ol>	<ol style="list-style-type: none"> <li>1. Fissata una stringa s in input ed una grammatica</li> <li>2. trovare in s tutte le k stringhe che possono essere ridotte</li> <li>3. creare k copie dell'input (eventualmente già parzialmente ridotto)</li> <li>4. in ogni istanza, eseguire una delle k riduzioni</li> <li>5. se un'istanza del parser riduce la stringa si è ottenuto un parse tree</li> <li>6. altrimenti ritorna allo step 1 (per ogni istanza).</li> </ol>

## Generalized Non deterministic LR parser

Modificando l'algoritmo, in modo tale che effettui le riduzioni solo su configurazioni valide dello stack, si ottiene un algoritmo generalizzato che "ignora" le istanze del parser che produrrebbero sequenze di produzioni non valide. Considerando ad esempio la grammatica:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

e la stringa **abbcede**, si presenta di seguito la simulazione di un parser:

STACK	INPUT	AZIONE
\$	abbcede\$	shift
\$a	bbcede\$	shift
\$ab	bcde\$	reduce $A \rightarrow b$
\$aA	bcde\$	shift
\$aAb	cde\$	shift
\$aAbc	de\$	reduce $A \rightarrow Abc$
\$aA	de\$	shift
\$aAd	e\$	reduce $B \rightarrow d$
\$aAB	e\$	shift
\$aABe	\$	reduce $S \rightarrow aABe$
\$S	\$	accept

Da cui:  $S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow \mathbf{abbcede}$

## Non deterministic LR parser

Il parsing LR riduce le stringhe fino alla variabile iniziale invertendo le produzioni.

$STR \leftarrow$  stringa di terminali in input

while str != S:

- identificare  $\beta$  in str tale che  $A \rightarrow \beta$  sia una produzione e  $S \rightarrow^* \alpha A \gamma \rightarrow \alpha \beta \gamma = str$
- sostituire  $\beta$  con A in str (in modo tale che  $\alpha A \gamma$  diventi la nuova str)

Questa modalità è più potente del parsing top-down in quanto prende decisioni dopo aver visto tutti i simboli  $\beta$  (LL(1) prende decisioni conoscendo al più uno dei simboli: il lookahead).

## LR(1) parsing

- aggiungere \$ alla fine dell'input
- eseguire l'algoritmo LR (uguale a quello SLR) utilizzando un automa per decidere quando effettuare shift o reduce
  - ogni volta che si può prendere una decisione, utilizzare lo stack come input per l'automa
  - l'automa indica quando effettuare shift o reduce (in base al lookahead)

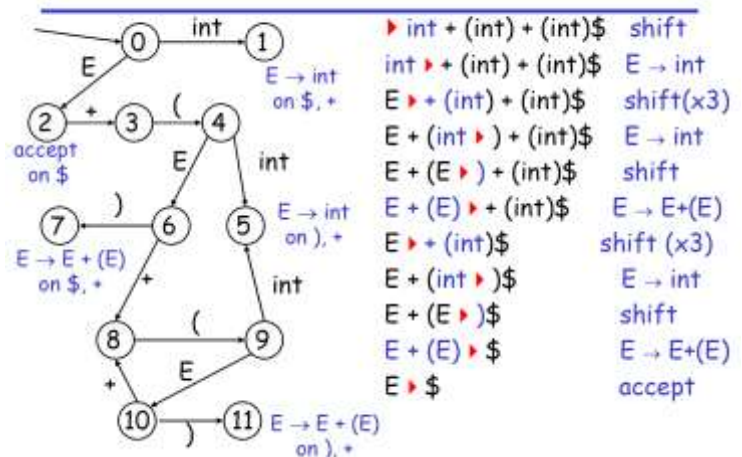
Dopo ogni azione di shift o reduce si ri-esegue il DFA sull'intero stack (si tratta di uno spreco: molte azioni sono ripetute).

### Algoritmo LR(1) parsing

```

I := w1w2...wn$           //input
iniziale
j = 1
DFAstate := 0                //stato
iniziale
stack := < dummy, 0 >        //stack iniziale
ripeti
    case action[top_state(stack), I[j]]
        shift k      : push < I[j], k >; j++;
        reduce X → α : pop |α| pairs, push < X, goto[top_state(stack), X] >
        accept       : termina con successo
        error        : termina con errore
    
```

### LR(1) Parsing. An Example



### Osservazioni:

- Il parsing LR può essere utilizzato per effettuare il parse su un numero maggiore di grammatiche rispetto a LL
- Le grammatiche di molti linguaggi di programmazione sono LR
- Il parsing LR può essere descritto con una semplice tabella
- Esistono tools per creare la tabella di parsing

### Costruzione della tabella di parsing LR

La tabella di parsing si costruisce partendo dalla costruzione di un DFA nel quale ogni stato descrive un contesto di parsing (una configurazione sul top dello stack).

Ogni elemento LR(1) è una coppia:  $X \rightarrow \alpha \cdot \beta$ , a in cui:

- $X \rightarrow \alpha \cdot \beta$  è una produzione
- a è un terminale.

$[X \rightarrow \alpha \cdot \beta, a]$  descrive un'informazione di contesto:

- si cerca una X seguita da una a
- $\alpha$  è nel top dello stack
- si vuole trovare il prossimo prefisso descritto da  $\beta a$ .

### Osservazioni:

- si usa il simbolo  $\blacktriangleright$  per separare lo stack dal resto dell'input:
  - $\alpha\blacktriangleright\gamma$ :  $\alpha$  è lo stack e  $\gamma$  la parte rimanente dell'input
- si usa il simbolo  $\bullet$  per “marcare” il prefisso di una produzione rhs:
  - $X \rightarrow \alpha\bullet\beta$ , a  $\beta$  può contenere anche non terminali
- in entrambi i casi lo stack sta a sinistra dei simboli.

Convenzioni (considerando la grammatica  $E \rightarrow E + ( E ) \mid \text{int}$ ):

- si aggiunge alla grammatica un nuovo simbolo iniziale  $S$  e una produzione  $S \rightarrow E$ , dove  $E$  è il simbolo iniziale originale della grammatica
- il contesto di parsing iniziale è  $S \rightarrow \bullet E, \$$ 
  - lo stack è vuoto
  - si cerca una  $S$  come stringa derivata da  $E\$$
- **shift**: in contesti nei quali il  $\bullet$  punta ad un terminale, se quel terminale è il prossimo dell'input si effettua uno **shift** spostando il  $\bullet$  al simbolo successivo e mettendo il simbolo letto nello stack
  - $E \rightarrow E + \bullet ( E ), +$  se il simbolo successivo è  $($  allora  $E \rightarrow E + ( \bullet E ), +$
- **reduce**: in contesti nei quali il  $\bullet$  è al termine si può effettuare una riduzione se il top dello stack corrisponde al prossimo simbolo
  - $E \rightarrow E + ( E ) \bullet, +$  se il simbolo successivo è  $+$  allora si riduce  $E \rightarrow E + ( E )$
- **closure**: in contesti nei quali il  $\bullet$  punta ad un non terminale, si effettua un'estensione del contesto con su tutti gli item relativi al non terminale
  - $E \rightarrow E + ( \bullet E ), +$  ci si aspetta una nuova stringa derivata da  $E ) +$
  - si aggiungono eventuali produzioni per la variabile  $E$  con il simbolo  $)$ :
    - $E \rightarrow \bullet \text{int}, )$
    - $E \rightarrow \bullet E + ( E ), )$

## Costruzione del DFA

L'operazione di **chiusura (closure)** estende il contesto con gli elementi necessari:

### Function Closure(Items)

ripeti

per ogni  $[X \rightarrow \alpha\bullet Y\beta, a]$  in Items

per ogni produzione  $Y \rightarrow \gamma$

per ogni  $b \in \text{First}(\beta a)$

aggiungi  $[Y \rightarrow \bullet\gamma, b]$  ad Items

finché Items non rimane immutato

return Items

Considerando la grammatica  $E \rightarrow E + ( E ) \mid \text{int}$  va aggiunta una produzione di partenza:  $S \rightarrow E$

L'operazione di closure per il contesto iniziale  $[S \rightarrow \bullet E, \$]$  è:

$S \rightarrow \bullet E, \$$

$E \rightarrow \bullet E + ( E ), \$$

$E \rightarrow \bullet \text{int}, \$$

$E \rightarrow \bullet E + ( E ), +$

$E \rightarrow \bullet \text{int}, +$

che può essere scritta in forma abbreviata:

$S \rightarrow \bullet E, \$$

$E \rightarrow \bullet E + ( E ), \$/+$

$E \rightarrow \bullet \text{int}, \$/+$

Gli stati del DFA:

- ogni stato è un insieme closure degli items LR(1).
- lo stato iniziale è l'insieme closure( $[S \rightarrow \bullet E, \$]$ ) della produzione di partenza aggiunta.
- uno stato che contiene  $[X \rightarrow \alpha\bullet, b]$  definisce la “riduzione con  $X \rightarrow \alpha$  su  $b$ ).

Le transizioni del DFA:

- ogni stato che contiene  $[X \rightarrow \alpha \bullet Y\beta, b]$  ha transizioni etichettate con Y verso uno stato che contiene gli items ottenuti con la funzione "Transition(stato, Y)"
  - ◆ Y può essere sia un terminale che una variabile

**Function Transition(stato, Y)**

Items  $\leftarrow \emptyset$

per ogni  $[X \rightarrow \alpha \bullet Y\beta, b] \in \text{stato}$

aggiungi  $[X \rightarrow \alpha \bullet Y\beta, b]$  all'insieme Items

return Closure(Items)

Esempio: costruzione del DFA per la seguente grammatica

$S \rightarrow L = R \mid R$

$E \rightarrow *R \mid id$

$R \rightarrow L$

**1. Calcolo dei Follow (utilizzati per decidere quando effettuare le riduzioni):**

Follow(S) = { \$ }

Follow(L) = { =, \$ }

Follow(R) = { \$, = }

**2. Modifica della grammatica (aggiungendo la produzione iniziale):**

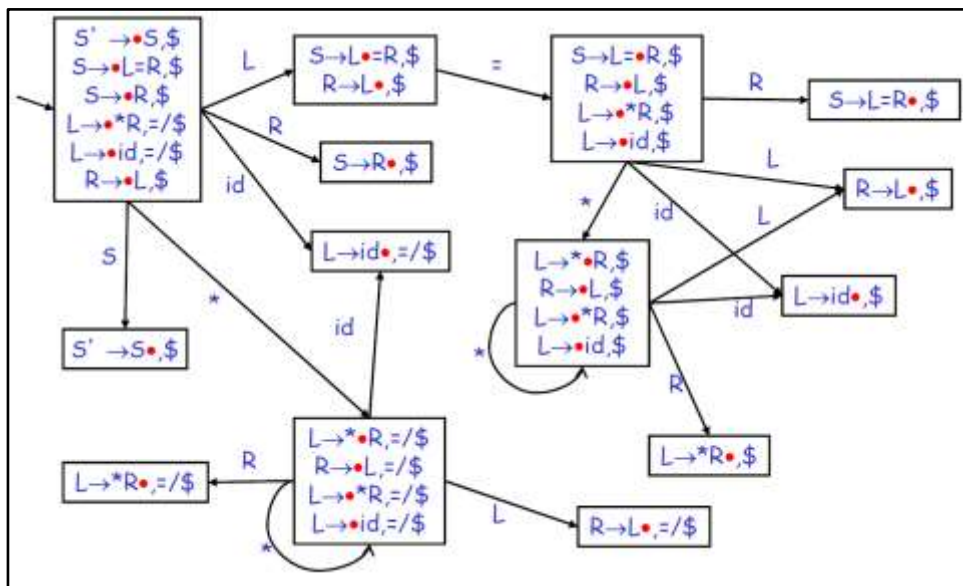
$S' \rightarrow S$

$S \rightarrow L = R \mid R$

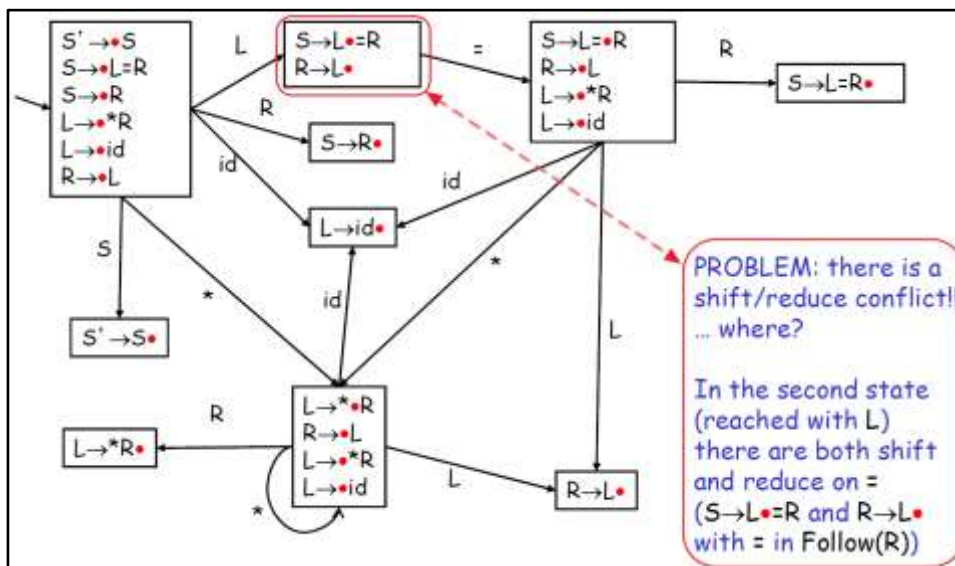
$L \rightarrow *R \mid id$

$R \rightarrow L$

**3. Costruzione dell'automa:**



Si osserva che LR risolve il problema di conflitto (shift/reduce) che si verifica con SLR:



### Osservazioni su LR:

- la tabella di parsing/il DFA possono essere costruiti automaticamente per una CFG
- la costruzione del DFA LR ha tuttavia un problema legato all'aumentare (spesso drammatico nella pratica) degli stati
- per avere un parser predittivo deterministico è necessario risolvere alcuni conflitti.

### Risoluzione dei conflitti

Se lo stato di un DFA contiene produzioni del tipo  $[X \rightarrow \alpha \cdot a\beta, b]$  si effettua uno shift.

Se lo stato di un DFA contiene produzioni del tipo che  $[Y \rightarrow \cdot \gamma, a]$ , determina una riduzione.

### Conflitti shift/reduce SLR

Se lo stato di un DFA contiene sia produzioni del tipo  $[X \rightarrow \alpha \cdot a\beta, b]$  che  $[Y \rightarrow \cdot \gamma, a]$ , in caso di input "a" non è possibile determinare quale sia l'azione da eseguire.

Ciò è solitamente dovuto ad ambiguità delle grammatiche, ad esempio il caso del "dangling else":

$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S$  determina un DFA per cui sia uno stato che contiene le produzioni:

$[S \rightarrow \text{if } E \text{ then } S \cdot, \text{ else}]$

$[S \rightarrow \text{if } E \text{ then } S \cdot \text{ else } S, \$]$

che, in caso riceva il token "else" determina un conflitto.

Un altro esempio è collegato alla grammatica  $E \rightarrow E + E \mid E * E \mid \text{int}$  che determina in uno stato:

$[E \rightarrow E * \cdot E, +]$

$[E \rightarrow \cdot E + E, +]$

dal quale si arriva allo stato

$[E \rightarrow E * E \cdot, +]$

$[E \rightarrow E \cdot + E, +]$

che ancora è uno stato di conflitto shift/reduce per il simbolo in input "+".

Una possibile soluzione allora consiste nel dichiarare una precedenza tra i simboli (\* e +).

Nei generatori di parser è possibile dichiarare la precedenza e l'associatività di simboli terminali, ad esempio con sintassi di questo tipo:

%left +

%left \*

In caso di conflitti shift/reduce il terminale di input ha precedenza rispetto alla regola e si può adottare una tecnica di associatività (a sinistra).

In questo modo,

[  $E \rightarrow E * \bullet E, +$  ]

[  $E \rightarrow \bullet E + E, +$  ]

dando una precedenza a  $E \rightarrow E * E$  si può decidere di effettuare una riduzione.

In caso di stesso simbolo

[  $E \rightarrow E + \bullet E, +$  ]

[  $E \rightarrow \bullet E + E, +$  ]

si può scegliere di applicare l'associatività a sinistra e quindi di effettuare ancora una riduzione.

### Conflitti reduce/reduce SLR

Uno stato del DFA genera un conflitto reduce/reduce se contiene due le produzioni:

[ $X \rightarrow \alpha \bullet, a$ ] e [ $Y \rightarrow \beta \bullet, b$ ]

Questo caso è solitamente determinato da grammatiche con ambiguità. In questi casi è necessario procedere con una riscrittura della grammatica con una equivalente senza ambiguità.

In conclusione i generatori di parser costruiscono il DFA di una CFG utilizzando precedenza ed associatività. Tuttavia non costruiscono un DFA LR(1) in quanto - anche per semplici grammatiche - il numero di stati è significativamente elevato.

### LALR

Il **core** di un insieme LR è costituito dall'insieme dei primi componenti senza il simbolo di lookahead. Ad esempio: il core di { [ $X \rightarrow \alpha \bullet \beta, b$ ], [ $Y \rightarrow \gamma \bullet \delta, d$ ] } è {  $X \rightarrow \alpha \bullet \beta, Y \rightarrow \gamma \bullet \delta$  }.

Partendo dagli stati del DFA LR(1) si possono ottenere gli stati di un DFA **LALR(1)** utilizzando la funzione core, ottenendo solitamente 10 volte in meno gli stati del DFA LR.

Finché esistono stati LR con core condivisi:

- scegliere due stati distinti con lo stesso core
- unire i due stati creandone uno nuovo tramite la funzione core
- le transizioni che arrivavano ai due vecchi stati, arrivano al nuovo stato
- le transizioni che partivano dal vecchio stato, partono dal nuovo stato.



La costruzione LALR potrebbe introdurre nuovi conflitti, ad esempio due stati:

$s1 = \{ [X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, b] \}$

$s2 = \{ [X \rightarrow \alpha \bullet, c], [Y \rightarrow \beta \bullet, b] \}$

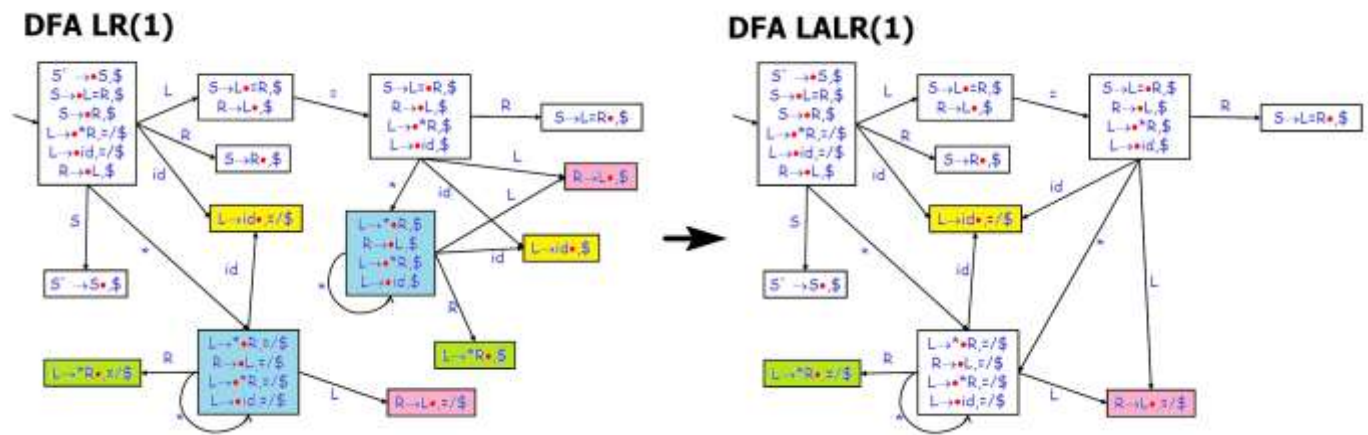
verrebbero fusi nello stato:

$s = \{ [X \rightarrow \alpha \bullet, a/c], [Y \rightarrow \beta \bullet, a/b] \}$

generando un conflitto reduce/reduce.

Nella pratica, tuttavia, questi casi sono piuttosto rari.





Osservazioni:

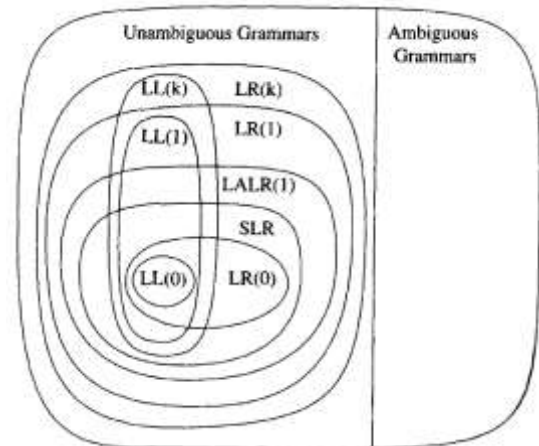
- I linguaggi LALR non sono naturali ma rappresentano una efficiente modifica dei linguaggi LR
- Ogni linguaggio di programmazione ragionevole utilizza grammatiche LALR(1)
- LALR(1) è diventato uno standard per i linguaggi di programmazione e per i generatori di parser.

## Osservazioni conclusive sul parsing

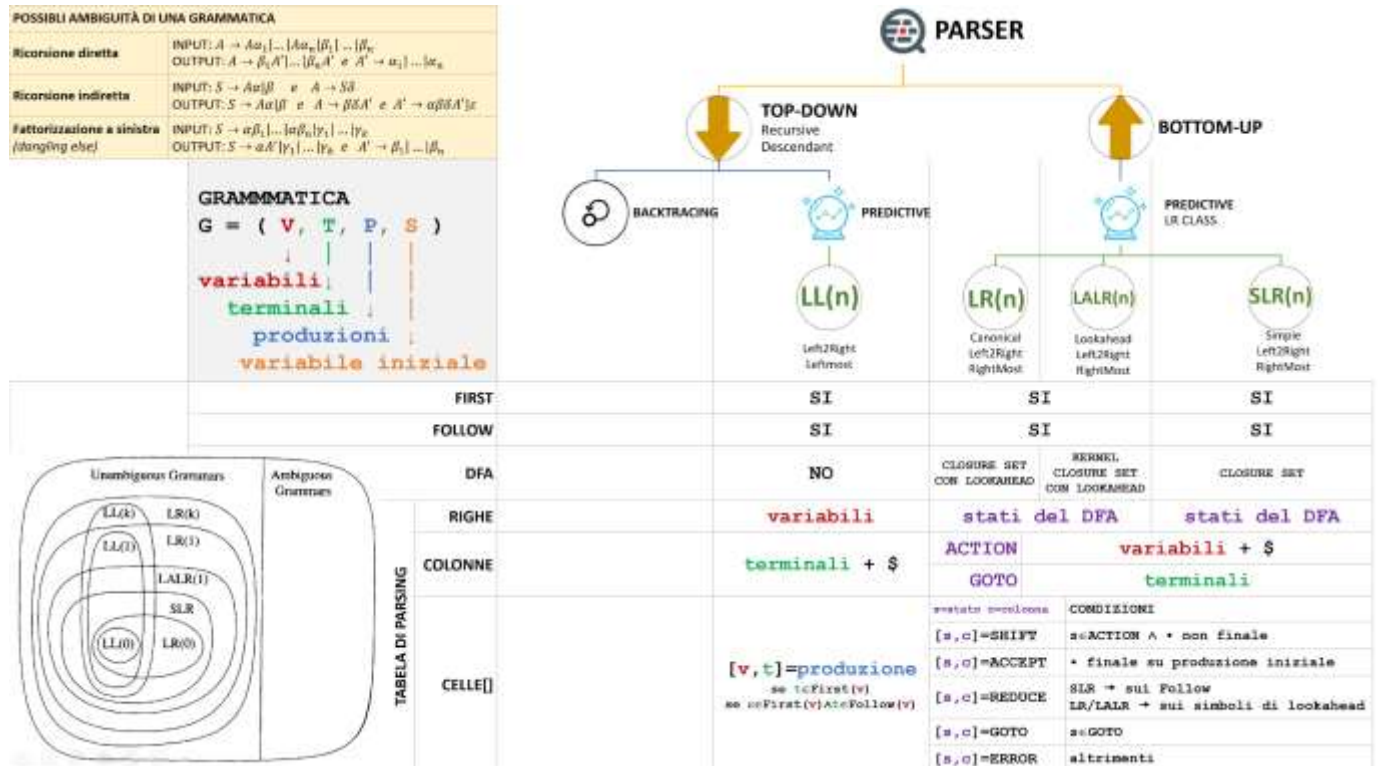
L'immagine a fianco, tratta da “Modern Compiler Implementation in Java” di Andrew Appel, rappresenta la classificazione delle grammatiche non ambigue.

In conclusione:

- le grammatiche libere dal contesto sono una solida base sulla quale costruire un linguaggio di programmazione e sulla quale ha senso ragionare sull'implementazione di parser.
- LL(1) è un semplice parser top-down
- SLR(1) è un semplice parser bottom-up
- LR(1) è un parser bottom-up più potente di SLR(1)
- LALR(1) consiste di modifiche che migliorano l'efficienza di LR(1) e per questo è ampiamente diffuso nei generatori di parser.







## Analisi semantica

L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso.

Azioni di verifica tipiche di questa fase riguardano:

- il controllo dei tipi (**type checking**)
- le dichiarazioni multiple
- l'uso di variabili non dichiarate
- le chiamate a metodi con argomenti sbagliati
- ...

Il risultato di questa fase è l'albero sintattico astratto (AST).

Un analizzatore semantico solitamente lavora in due fasi:

1. Per ogni **scope** del programma
  - a. Processa le **dichiarazioni**:
    - i. Aggiungendo nuovi elementi alla symbol table
    - ii. Segnalando eventuali dichiarazioni multiple di variabili
  - b. Processa gli **statements**:
    - i. Individua l'uso di variabili non dichiarate
    - ii. Aggiorna i nodi "ID" (identificativi) dell'AST per farli puntare agli elementi appropriati della symbol table
2. Processa ogni statement nuovamente
  - a. Utilizzando la symbol table per determinare il tipo di ogni espressione e individuare errori.

## Tabella dei simboli (symbol table)

La tabella dei simboli è costituita da un insieme di "entries" che tengono traccia dei nomi (variabili, classi, campi, metodi, ...) dichiarati nel programma.

Ogni entry della symbol table associa un nome ad un insieme di attributi:

- Tipo del nome: variabile, classe, campo, metodo, ....
- Tipo: int, float, boolean, ...
- Nesting level (per la valutazione degli scope)
- Locazione di memoria (dove individuare il dato a runtime).

## Scoping

La progettazione della symbol table dipende dal tipo di scoping utilizzato dal linguaggio compilato.

Linguaggi differenti supportano la possibilità di effettuare dichiarazioni multiple con lo stesso nome (ad esempio in diversi scope o per diversi tipi di nomi).

Ad esempio in Java è possibile utilizzare uno stesso nome per una classe, un campo della classe, un metodo della classe, una variabile locale del metodo; inoltre Java implementa l'overloading.

Lo scoping è necessario per determinare la giusta dichiarazione da utilizzare.

Lo scoping è concettualizzato a livelli gerarchici per cui la ricerca di un nome utilizzato è solitamente effettuata a partire dallo scope attuale risalendo agli scope "più esterni".

## Scoping statico e scoping dinamico

Utilizzando lo **scoping statico** (o lessicale), il riferimento al nome è risolto basandosi sulla struttura sintattica di nidificazione di blocchi e sottoprogrammi. Quindi l'ambiente non locale sarà quello di definizione del sottoprogramma (e quindi 'statico'), e non quello di applicazione (e quindi 'dinamico').

Utilizzando lo **scoping dinamico**, il riferimento ad un nome è risolto con l'ultima associazione per il nome presente nella catena di chiamate per P.

Quindi in ogni attivazione di sottoprogramma o blocco viene generato un nuovo ambiente: in questo modo non è possibile stabilire a priori (staticamente) quale sarà l'associazione impiegata, in quanto sarà dipendente dal flusso di esecuzione del programma.

In altri termini:

Lo **scope statico** richiede che il riferimento a un nome non locale e non globale venga risolto utilizzando il legame testualmente più vicino, procedendo dall'interno verso l'esterno. Per risolvere un riferimento a un nome si esamina il blocco locale e tutti quelli via via più esterni che staticamente lo includono fino ad individuarne il legame. La determinazione degli scope può essere effettuata dal compilatore, scegliendo il più recente legame attivo elaborato a tempo di compilazione.

Invece, lo **scope dinamico** richiede la scelta del più recente legame ancora attivo stabilito a tempo di esecuzione.

Generalmente i linguaggi compilati usano lo scope statico.

## Implementazione della tabella dei simboli

Le azioni necessarie per la gestione della symbol table prevedono:

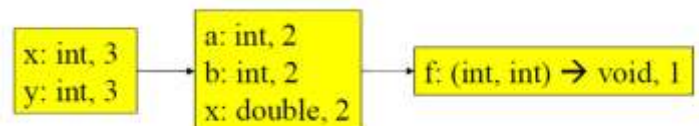
1. La ricerca di un nome nello scope corrente (per verificare se il nome è già stato dichiarato)
2. L'inserimento di un nuovo nome con i suoi attributi nella tabella
3. La ricerca di un nome nello scope corrente e negli scopes che lo includono (per individuare l'eventuale uso di nomi non dichiarati e per collegare un uso con le entry corrispondenti della symbol table)
4. L'esecuzione delle azioni necessarie quando si entra in un nuovo scope
5. L'esecuzione delle azioni necessarie quando si esce da uno scope.

Esistono due possibili implementazioni della symbol table:

- Tramite una lista di hashtable
- Tramite un'hashtable di liste

### Metodo 1: list of hashtables

Questa implementazione prevede che la testa della lista punti ad un'hashtable che contiene le dichiarazioni effettuate nello scope attuale; gli elementi successivi della lista puntano alle hashtable che contengono i nomi dichiarati negli scope che racchiudono quello attuale.



Quando si entra in un nuovo scope si incrementa il numero di **nesting level** e si aggiunge una nuova hashtable (in cima alla lista).

La processazione di ogni dichiarazione prevede una ricerca del nome nell'hashtable in cima alla lista: se si trova il nome si deve segnalare un errore di dichiarazione multipla, viceversa si può aggiungere il nome alla symbol table.

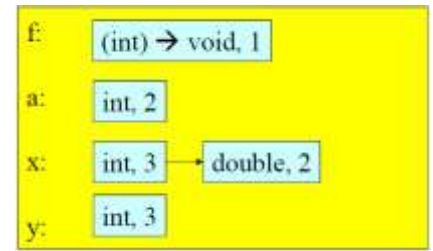
La processazione di un utilizzo di un nome determina una ricerca del nome a partire dall'hashtable in testa e iterando la ricerca nelle hashtable che costituiscono la lista: se l'elemento non è presente nell'hashtable si effettua una ricerca nell'hashtable successiva (nella lista), se non si trova il nome in nessuna hashtable si segnala un errore di uso di variabile non dichiarata.

Quando si esce da uno scope è sufficiente rimuovere la prima hashtable dalla lista e decrementare il numero di nesting level.

Nota: si ricordi che i nomi dei metodi appartengono all'hashtable dello scope più esterno: non alla stessa tabella che contiene le variabili dei metodi.

## Metodo 2: hashtables of list

Questa implementazione prevede di mantenere un'hashtable di liste nella quale ogni elemento dell'hashtable rappresenta una lista legata ad un nome e ogni lista contiene l'elenco delle dichiarazioni di quel nome presentando, in cima alla lista, la dichiarazione più esterna.



Lista di hashtables		Hashtable di liste	
QUANDO SI ENTRA IN UN NUOVO SCOPE			
1. incrementare nestingLevel 2. creare una nuova hashtable (sul nestingLevel)		1. incremantare nestingLevel	
QUANDO SI TROVA UNA DICHIARAZIONE DI UN NOME (x)			
cercare il nome x al nestinglevel corrente * se esiste => errore dichiarazione multipla * se non esiste => si aggiunge x alla tabella	O(1)	cercare il nome x nell'hashtable (se nuovo lo si crea) * se il numero di nestingLevel nella cima della lista è uguale al nestingLevel attuale => errore dichiarazione multipla * viceversa => si aggiunge x in cima alla lista con indicazione del nestingLevel corrente	(O(1))
QUANDO SI TROVA UN UTILIZZO DI UN NOME (x)			
1. si cerca il nome x a partire dal primo livello della symbol table * se si trova => return offset * se non si trova => errore variabile non definita	O(nestingLevel)	1. si cerca il nome x nell'hashtable * se si trova => return offset * se non si trova => errore variabile non definita	(O(1))
QUANDO SI ESCE DA UNO SCOPE			
1. rimuovere la hashtable in testa 2. decrementare il nesting level	O(1)	1. scansione di tutti gli elementi dell'hashtable osservando solo il primo elemento delle liste: se il nestingLevel in cima alla lista è uguale al nestingLevel corrente, si elimina la testa dalla lista 2. si decrementa il nestingLevel	O(names)

## Type checking

La nozione di **tipo** varia da linguaggio a linguaggio, in generale si parla di insiemi di valori e insiemi di operazioni su tali valori. Le classi sono un esempio di moderna nozione di tipo.

E' necessario conoscere e valutare la congruità di tipi per valutare la correttezza semantica di un programma, infatti molte operazioni sono *legali* solo per valori di certi tipi (ad esempio potrebbe non aver senso effettuare operazioni aritmetiche su stringhe).

Il **type system** di un linguaggio specifica quali operazioni sono valide per ogni tipo.

L'obiettivo del **type checking** è di assicurare che le operazioni siano utilizzate con i corretti tipi.

Il **type system** di un linguaggio fornisce una formalizzazione concisa delle regole di controllo semantico.

Il **type checking** permette l'individuazione di certi tipi di errori:

- errori di memoria (es. letture da indirizzi di memoria non validi)
- violazione dei confini di astrazione
- ...

## Tipi di linguaggi

- **Statically typed:** tutto o quasi tutto il controllo dei tipi è effettuato come parte della compilazione (es. C)
  - Molti errori di programmazione vengono individuati al momento della compilazioneSi evita sovraccarico durante l'esecuzione per il controllo di tipi
- **Dynamically typed:** Quasi tutto il controllo dei tipi è effettuato durante l'esecuzione del programma (es. Scheme)
  - I sistemi di controllo dinamico dei tipi sono meno restrittivi di quelli statici
  - La prototipazione rapida è più semplice in un type system dinamico
- **Untyped:** non viene effettuato controllo di tipi (es: codice macchina)

La fase di type checking è nota anche come processo di **type inference** in quanto spesso prevede che vengano effettuate valutazioni di inferenza per parti del programma. Il type checking verifica che il programma rispetti il type system.

Un sistema di tipi statico permette l'individuazione da parte del compilatore di molti errori di programmazione limitando però la libertà di programmazione.

Il **tipo dinamico** di un oggetto è la classe C che è utilizzata per creare l'oggetto (new C): una nozione che riguarda l'esecuzione a run-time.

Il **tipo statico** di un'espressione cattura tutti i possibili tipi dinamici che l'espressione può assumere: riguarda il momento della compilazione.

Nei primi sistemi l'insieme dei tipi statici corrispondeva direttamente a quello dei tipi dinamici.

**Soundness theorem:** per ogni espressione E  $\text{tipoDinamico}(E) = \text{tipoStatico}(E)$ .

Note: una variabile di tipo statico A può contenere valori di tipo dinamico B se  $B \leq A$ .

```
Class A {}  
Class B extends A {}  
Main () {  
    A x;  
    X = new (A);  
    ...  
    X = new (B);  
    ...  
}
```

**Principio di sostituzione di Liskov:** i sottotipi possono essere utilizzati al posto dei supertipi.

## Subtyping

Si definisce una relazione  $X \leq Y$  che esprime il fatto che un oggetto di tipo X può essere utilizzato quando un oggetto di tipo Y è accettabile o, equivalentemente, X è conforme a Y.

Quando la relazione è valida si dice che X è una sottoclasse di Y.

Tra classi,  $X \leq Y$  se X eredita da Y e  $X \leq Z$  se  $X \leq Y$  e  $Y \leq Z$ .

**Soundness theorem:** per ogni espressione E  $\text{tipoDinamico}(E) \leq \text{tipoStatico}(E)$ .

Spiegazione: dato E, il compilatore usa  $\text{tipoDinamico}(E) = C$ ; ogni operazione che può essere effettuata su un oggetto di tipo C può essere effettuata su un oggetto  $C' \leq C$ :

- gli oggetti di tipo C' devono esporre tutti gli stessi metodi pubblici esposti da C
- gli oggetti di tipo C' possono avere metodi pubblici o campi addizionali ma che non vengono utilizzati se il tipo statico è C.

## Esempio

```
Class A { public int A() { return 0; } }  
Class B inherits A { int b() { return 1; } }
```

- un'istanza di B ha i metodi a e b
- un'istanza di A ha il metodo a (in caso di chiamata del metodo b si genera un errore di tipo)
- i supertipi non possono essere utilizzati al posto dei sottotipi.

## Una regola per la gestione dei subtyping sbagliata

Dato un array di A (array di oggetti di tipo A), intuitivamente si può assumere che se  $B \leq A$  allora **array of B**  $\leq$  **array of A** (detti array covarianti).

Considerando però il seguente programma:

```
Let  
  function f(x: array of A) {  
    x[1] ← new A();  
  }
```

```
In  
  let z:array of B  
  in {  
    f(z);  
    z[1].b();  
  }
```

si può osservare che diventa possibile inserire un supertipo A nell'array e quando si utilizza il supertipo ( $z[1].b()$ ) si genera un type error.

Tuttavia, se gli array non possono essere modifica la covarianza è valida.

## Class subtyping

Le sottoclassi possono solitamente effettuare override di alcune dichiarazioni delle superclassi. Di seguito si presentano alcune osservazioni relative all'override di campi e metodi cambiando anche i loro tipi.

### Field overriding

```
Class A { ... public Type field; ... }  
Class B inherits A { ... public Type' field; ... }
```

Come per gli array la definizione di tipi ha senso se i campi sono immutabili, in tal caso:

**se  $Type' \leq Type$  allora  $B \leq A$**  (campi covarianti).

### Method overriding

```
Class A { ...  
  public Type method(Type1 param1, ..., TypeN paramN) { e };  
  ... }  
Class B inherits A { ...  
  public Type' method(Type1' param1', ..., TypeN' paramN') { e' };  
  ... }
```

Il tipo di ritorno di B deve essere utilizzabile al posto del tipo ritornato da A (  $Type' \leq Type$  ) e per ogni parametro deve essere possibile utilizzare un parametro di A al posto parametro di B (  $Type_i \leq Type_i'$  ).

In sintesi deve valere la covarianza sul tipo di output e la controvarianza sui tipi dei parametri.

## Code Generation

La fase di generazione del codice prende in ingresso la rappresentazione intermedia (AST) e la traduce nel programma destinazione. Se il linguaggio destinazione è il linguaggio macchina è necessario gestire tutte le caratteristiche collegate all'utilizzo di registri e memoria. Le istruzioni del linguaggio intermedio vengono tradotte in opportune sequenze di linguaggio macchina che realizzano le operazioni desiderate. Per questo motivo è necessario assumere consapevolezza circa gli aspetti da gestire nell'ambiente di runtime, come ad esempio:

- gestione delle risorse a runtime
- corrispondenza tra strutture statiche (compile-time) e dinamiche (runtime)
- organizzazione e gestione della memoria.

## Gestione della memoria

L'esecuzione del programma è, inizialmente, sotto il controllo completo del sistema operativo, quando un programma viene invocato:

- il sistema operativo alloca lo spazio per il programma
- il codice viene caricato in una parte dello spazio allocato
- il sistema operativo “salta” all'entry point del programma (es. main).

Una parte dello spazio allocato dal sistema operativo (che potrebbe non essere contiguo) contiene il codice, l'altra parte corrisponde allo spazio dei dati.

Il compilatore è responsabile di generare il codice e organizzare l'uso dello spazio dei dati con due obiettivi: **correttezza** e **velocità**. Molte complicazioni nella generazione del codice derivano dal tentativo di rendere veloce ciò che già è stato realizzato per essere corretto.

Assunzioni sull'esecuzione:

1. l'esecuzione è sequenziale: il flusso di controllo si muove da un punto del programma ad un altro secondo un ordine chiaramente definito;
2. al termine dell'esecuzione di una procedura chiamata, il controllo ritorna (eventualmente) al punto immediatamente successivo la chiamata.

L'invocazione di una procedura P è detta **attivazione di P**.

Il **lifetime** di un'attivazione di P comprende tutti i passi di esecuzione della procedura P inclusi i passi delle chiamate della procedura P.

Il **lifetime** di una variabile x è la porzione di esecuzione nella quale x è definita.

Il **lifetime** è un concetto dinamico mentre lo **scoping** è un concetto statico.

## Activation trees

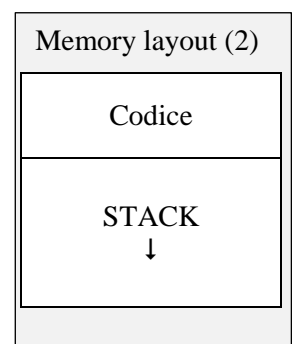
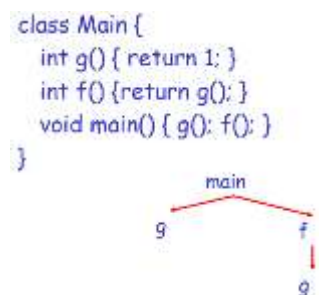
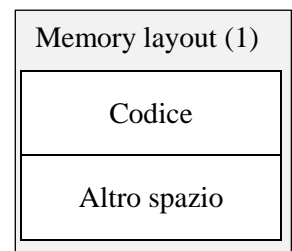
I lifetimes delle attivazioni di procedure sono innestati tra loro (assunzione 2), gli alberi sono strutture dati appropriate per rappresentare i lifetimes.

L'activation tree dipende dal comportamento a runtime e può cambiare in base agli input forniti; è possibile tenere traccia delle attivazioni (innestate tra loro) con uno stack.

## Activation records

Le informazioni necessarie per gestire l'attivazione di una procedura sono dette **activation record (AR)** o **frame**.

Quando una procedura f chiama una procedura g, l'activation record di g contiene un mix di informazioni riguardo f e g. La procedura f è sospesa fino al completamento di g, a quel punto f riprende la sua esecuzione. Il record di attivazione di g contiene informazioni necessarie per garantire la ripresa di f.





Il record di attivazione di una procedura chiamata contiene quindi:

- il valore del punto di ritorno
- il puntatore all'activation record precedente (*control link*)
- i parametri della procedura (forniti dal chiamante)
- lo spazio per le variabili locali di g
- altre variabili temporanee.

Activation record	
Result	
Arguments	
Control link	
Return address	

Il compilatore deve determinare il layout dell'activation record in fase di runtime e generare il codice che garantisce l'accesso alle corrette locazioni di memoria, per questo motivo il *code generator* e l'*activation record* devono essere progettati insieme.

## Variabili globali

Tutti i riferimenti ad una variabile globale puntano allo stesso elemento, pertanto non ha senso memorizzare una variabile globale in un activation record. Le variabili globali sono assegnate a indirizzi fissi una sola volta (allocazione statica).

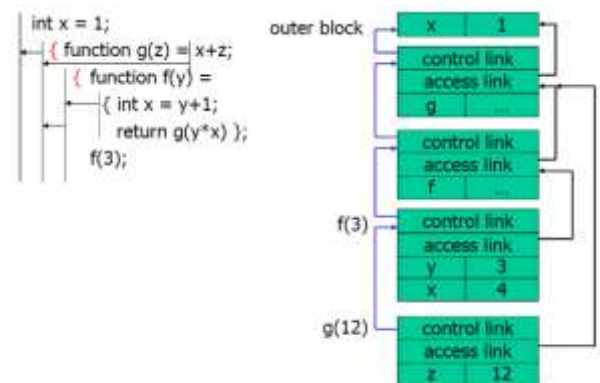
## Variabili dichiarate in altri scopes • Access link

I riferimenti a variabili dichiarate in scopes esterni devono puntare alle variabili memorizzate in altri activation record. In accordo con la “most closely nested rule”, un activation record dovrebbe puntare al più recente activation record dello scope che lo racchiude direttamente (padre).

Per individuare variabili dichiarate in scopes esterni si utilizza l'**access link**: punta sempre al frame che racchiude il blocco sintattico (per corpi delle funzioni l'access link punta al blocco che contiene la dichiarazione della funzione).

Il valore dell'access link di un nuovo activation record è definito come segue:

- per un blocco innestato o una funzioni dichiarata nello scope attuale:
  - access link = (indirizzo dell'access link) dell'activation record attuale
- se una funzione chiama se stessa ricorsivamente o chiama un'altra funzione dichiarata nel blocco sintattico che la racchiude:
  - access link = valore dell'access link dell'activation record attuale
- in generale, chiamando una funzione fuori dallo scope attuale
  - access link = valore dell'access link dell'activation record ottenuto risalendo la catena di access link per un numero di passi pari alla differenza tra nesting level attuale e quello in cui la funzione è stata dichiarata.



## Gestione dell'heap

Qualsiasi valore che “sopravvive” all'esecuzione di una procedura non può essere memorizzato nell'activation record (ad esempio `return new Object();`).

I linguaggi che gestiscono allocazioni dinamiche di dati utilizzano uno heap. Mentre però gli activation records vengono deallocati dallo stack quando l'esecuzione esce dal loro scope, per quanto riguarda l'heap viene utilizzata una tecnica chiamata **garbage collection** per cui una locazione di memoria diventa “garbage” se non è raggiungibile dalla continuazione dell'esecuzione di un programma. Quando è necessario liberare memoria si attiva la procedura di pulizia della garbage collection che rimuove i dati non più necessari.

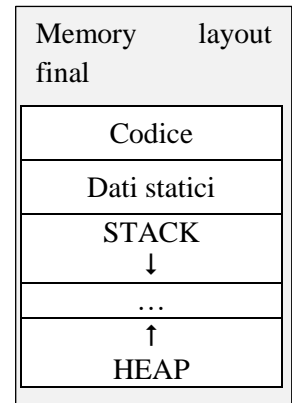


L'**algoritmo mark-and-sweep** utilizza un bit per classificare ogni dato: inizialmente tutti i dati hanno il bit valorizzato a 0, per ogni locazione, se risulta essere raggiungibile dal programma, si imposta il bit a 1; tutte le celle di memoria con flag=0.

L'**algoritmo reference counting** associa ad ogni dato un contatore: inizialmente il contatore è posto a zero quando il dato è allocato in memoria, quando viene settato un puntatore ad un dato il contatore viene incrementato, quando il dato viene resettato si decrementa il contatore; i dati con contatore = 0 possono essere ripuliti dalla memoria.

In sintesi la memoria è costituita da:

- l'area dedicata al codice contiene codice oggetto eseguibile (per molti linguaggi quest'area ha dimensioni fisse ed è di sola lettura)
- l'area statica che contiene dati (non codice) con indirizzi fissi (es. variabili globali), ha solitamente dimensioni fisse ma può essere letta e scritta
- lo stack che contiene un activation record per ogni procedura attualmente attiva (ogni activation record ha dimensioni fisse e contiene le variabili locali)
- lo heap che contiene tutti gli altri dati
- sia lo stack che lo heap crescono e si deve fare in modo che non si sovrappongano (una possibile soluzione: far partire heap e stack da punti di partenza opposti della memoria..)

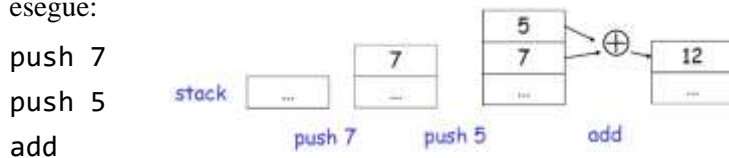


## Code generation strategy

### Stack machines

La stack machine è un'implementazione semplificata per la generazione del codice, non utilizza variabili o registri, memorizza i valori intermedi sullo stack.

Considerando due istruzioni di base: push e add, un programma che computa una somma (ad esempio  $7 + 5$ ) esegue:



L'operazione di add preleva due operandi dallo stack e mette il risultato dell'addizione sulla cima dello stack.

La stack machine è utile in quanto preleva gli operandi e mette il risultato nello stesso posto definendo uno schema di compilazione uniforme.

La locazione degli operandi è pertanto implicita: sempre in cima allo stack.

Esempio:

CODICE		STACK			
					<init>
push 13				13	<init>
push 7			7	13	<init>
push 5		5	7	13	<init>
add			12	13	<init>
sub				1	<init>

**Invariante:** dopo aver computata un'espressione (o una sub-espressione) lo stack è identico a prima tranne per il fatto che in cima, anziché esserci i due operandi c'è il risultato.

Le operazioni della stack machine possono essere ottimizzate con l'uso di un registro **accumulatore** (l'accesso al registro è più rapido) come cima dello stack in modo tale da dimezzare l'accesso alla memoria per le operazioni (la add legge la cima dello stack e la somma all'accumulatore).

**Invarianti:** il risultato è sempre nell'accumulatore, dopo la computazione di un'espressione, lo stack è uguale a prima.

## The MIPS assembly language

Il codice generato viene messo in esecuzione su un processore. MIPS (Microprocessor without Interlocked Pipeline Stages) è un'architettura informatica per microprocessori RISC sviluppata dalla MIPS Computer Systems Inc. (oggi MIPS Technologies Inc.).

Lo stack, in memoria, cresce dall'indirizzo più basso assegnato al programma.

Nell'architettura MIPS, lo stack pointer è memorizzato in un registro denominato fp.

Alcune istruzioni utilizzabili su architettura MIPS sono:

- `lw reg0 offset(reg1)`: carica una parola (32 bit) dall'indirizzo `reg1 + offset` nel registro `reg0`
- `add reg0 reg1 reg2`: somma i valori di `reg1` e `reg2` e mette il risultato in `reg0`
- `sw reg0 offset(reg1)`: salva la parola (32 bit) presente nel registro `reg0` all'indirizzo `reg1 + offset`
- `addiu reg0 reg1 imm`: somma i valori di `reg1` e `imm` e mette il risultato in `reg0`
- `li reg0 imm`: copia il valore di `imm` in `reg0`

Alcune macro/abbreviazioni:

- `push $t`      `addiu $sp $sp -4`  
                  `sw $t 0($sp)`
- `pop`            `addiu $sp $sp 4`
- `$t ← pop`      `lw $t 0($sp)`  
                  `addiu $p $p 4`
- `pop*n`         `addiut $sp $sp z`      con `z = 4 * n`

## Costanti, add, sub, if, function definition, function call, parameters usage, high order functions

Per descrivere il codice da generare si fa riferimento ad una funzione di generazione del codice chiamata `cgen()`.

Oggetto	Funzione	Codice generato	Note
Costanti	<code>cgen(i)</code>	<code>li \$t0 i</code> <code>push \$t0</code>	Le costanti sono semplicemente impilate nello stack
Add	<code>cgen(e<sub>1</sub> + e<sub>2</sub>)</code>	<code>cgen(e<sub>1</sub>)</code> <code>cgen(e<sub>2</sub>)</code> <code>\$t2 ← pop</code> <code>\$t1 ← pop</code> <code>add \$t0 \$t1 \$t2</code> <code>push \$t0</code>	Si potrebbe immaginare di ottimizzare il codice mettendo il risultato di <code>e<sub>1</sub></code> immediatamente nel registro <code>\$t1</code> . Tuttavia ciò potrebbe generare dei problemi in quanto non è possibile fare valutazioni di preservazione del valore sui registri.
Sub	<code>cgen(e<sub>1</sub> - e<sub>2</sub>)</code>	<code>cgen(e<sub>1</sub>)</code> <code>cgen(e<sub>2</sub>)</code> <code>\$t2 ← pop</code> <code>\$t1 ← pop</code> <code>sub \$t0 \$t1 \$t2</code> <code>push \$t0</code>	
Conditional	<code>cgen(if e<sub>1</sub> == e<sub>2</sub>) then e<sub>3</sub> else e<sub>4</sub></code>	<code>cgen(e<sub>1</sub>)</code> <code>cgen(e<sub>2</sub>)</code> <code>\$t2 ← pop</code> <code>\$t1 ← pop</code> <code>beq \$t1 \$t2 true_branch</code> <code>cgen(e<sub>4</sub>)</code> <code>b end_if</code> <b>true_branch:</b> <code>cgen(e<sub>3</sub>)</code> <b>end_if:</b>	Nuove istruzioni: <ul style="list-style-type: none"> <li>• <code>beq</code>: salto condizionato</li> <li>• <code>b</code>: salto incondizionato</li> <li>• utilizzo di etichette (freshly names) generate automaticamente</li> </ul>

Function call	$cgen(f(e_1, \dots, e_n))$	<pre> push \$f cgen(e<sub>n</sub>) ... cgen(e<sub>2</sub>) jal f_entry                     </pre>	<p>Nuova istruzioni: <b>jal</b>: jump and link, salta all'etichetta e salva l'indirizzo della prossima istruzione in \$ra</p> <ul style="list-style-type: none"> <li>Il chiamante salva il suo valore nel frame pointer</li> <li>Successivamente salva i parametri attuali in ordine inverso</li> <li>Il chiamante salva l'indirizzo di ritorno nel registro \$ra</li> <li>L'activation record ha dimensioni pari a <math>4*n+4</math> bytes</li> </ul>
Function definition	$cgen(f(x_1, \dots, x_n)=e)$	<pre> f_entry: move \$fp \$sp push \$ra cgen(e) \$t0 ← pop \$ra ← pop pop*n \$fp ← pop push \$t0 jr \$ra                     </pre>	<p>Nuova istruzioni: <b>jr</b>: jump reg, salta all'indirizzo memorizzato nel registro reg</p> <ul style="list-style-type: none"> <li>Il frame pointer non punta alla fine del frame</li> <li>Il chiamato effettua delle pop per                             <ul style="list-style-type: none"> <li>il valore di ritorno</li> <li>l'indirizzo di ritorno</li> <li>gli argomenti</li> <li>il valore salvato nel frame pointer</li> </ul> </li> <li>\$t0 è un registro utilizzato per mantenere provvisoriamente il valore di ritorno</li> </ul>
<div style="text-align: center;"> <p>Before call      On entry      In body      After call</p> </div>			
Parameters usage	$cgen(x_i)$	<pre> lw \$t0 z(\$fp) push \$t0 cgen(e)  con z = 4 * (i-1)                     </pre>	<p>In questa semplificazione, le variabili di una funzione sono gestite come i parametri: sono tutte nel record di attivazione e sono allocate dal chiamante.</p> <p>Si osservi che, siccome lo stack cresce quando i risultati intermedi sono salvati, le variabili non possono essere associate ad un offset fisso (da \$sp); è pertanto necessario un <b>frame pointer</b>.</p>

## Generazione del codice per funzioni “higher order”

In alcuni linguaggi è possibile passare delle funzioni come parametri:

```
fun bool f ( bool x(int, int), int a, int b) { ... x(a,b); ... }
```

La funzione  $f$  dovrebbe preparare il record di attivazione per l'esecuzione di  $x(a, b)$ , tuttavia essa non ha informazioni sufficienti per impostare l'access link (dovrebbe puntare al record di attivazione della dove la funzione chiamata è stata dichiarata, in base alla differenza di nesting level).

Soluzione: chi invoca la funzione  $f$ , che passa  $g$  (id di una funzione dichiarata) come parametro della funzione, dovrebbe passare una coppia di valori contenenti:

- l'indirizzo **g\_entry** del codice della funzione  $g$  (valore assegnato all'identificatore quando la funzione è stata dichiarata)
- l'indirizzo del record di attivazione più recente in cui  $g$  è stato dichiarato.

Inoltre: il valore del parametro viene impostato come una coppia (valore di un identificatore di un tipo funzione è una coppia); quando la funzione  $g$  viene eseguita l'access link viene impostato al secondo elemento della coppia.

## Generazione del codice per linguaggi object-oriented

È necessario affrontare due problemi:

1. in quale modo si rappresentano gli oggetti in memoria?
2. in quale modo si deve implementare il *dispatch* dinamico?

Un oggetto è simile ad una struct del C. Il riferimento `object.field` è un indice (dentro a `object`) all'offset relativo a `field`.

Gli oggetti sono memorizzati in spazi contigui di memoria, ogni campo è memorizzato ad uno specifico offset (che deve essere inserito nella corrispondente entry della symbol table).

Alla creazione dell'oggetto, il layout corrispondente è istanziato nell'heap.

Per quanto riguarda le sottoclassi, il layout della sottoclasse può essere definito estendendo il layout della superclasse, lasciando immutata la parte di layout relativa alla superclasse.

L'implementazione dei metodi e il **dynamic dispatch** assomigliano molto all'implementazione dei campi:

- ogni classe un insieme limitato di metodi (inclusi i metodi ereditati)
- una tabella di dispatch indicizza questi metodi:
  - un array di indirizzi ai metodi
  - un metodo  $f$  è memorizzato ad un fissato offset nella tabella di dispatch per una classe e per tutte le sue sottoclassi.

Esempio dispatch table.

<pre>Class A {   int a = 0;   int d = 1;   int f() { return a+a*d } }</pre>	<pre>Class B extends A {   int b = 2;   int f() { return a }   int g() { return a-b } }</pre>	<pre>Class C extends A {   int c = 3;   int h() { return a*a*c } }</pre>
---	---	--

Offset Class	0	4	
A	A.f()		Per la classe A, la dispatch table ha un solo metodo
B	B.f()	B.g()	Per la classe B, la tabella è estesa verso destra. Il metodo f sovrascrive il corrispondente metodo di A.
C	A.f()	C.h()	Per la classe C, la tabella è estesa verso destra con l'aggiunta del metodo h.

Il **dispatch pointer** in un oggetto punta alla dispatch table della classe. Ogni metodo  $f$  di classe  $C$  è assegnato ad un offset  $O_f$  nella dispatch table durante la compilazione. L'offset è inserito nella entry della symbol table del metodo  $f$  della classe  $C$ .

Per implementare una dispatch table per  $e.f()$  è necessario:

- definire  $O_f$  come l'offset del metodo  $f$  nella dispatch table associata al tipo statico di  $e$
- valutare  $e$ , ottenendo un oggetto  $o$  (che può essere una sottoclasse)
- definire  $D$  come la dispatch table di  $o$
- eseguire il metodo puntato da  $D[O_f]$ .

# Logica di base

## Introduzione

La logica classica è un linguaggio formale nato per descrivere la matematica. La logica è anche alla base di concetti fondamentali dell'informatica quali la nozione di circuito e schemi di operazioni su informazioni digitali (and, or, ecc.) e la nozione di problema di decisione alla base della teoria della complessità. Negli ultimi decenni la logica classica ha trovato anche numerose altre applicazioni all'informatica, ed intelligenza artificiale in particolare, quali rappresentazione della conoscenza, sistemi esperti, ragionamento automatico, specifica di sistemi, programmazione, e molti altri.

Obiettivi:

- comprendere la struttura generale della logica
- conoscere le logiche delle proposizioni e dei predicati (del prim'ordine)
- comprendere i sistemi deduttivi.

La logica è una teoria fondamentale basata sulla teoria degli insiemi.

Permette di esprimere proposizioni e loro relazioni.

In informatica fornisce un linguaggio formale per esprimere le proprietà di interesse di un sistema e una modalità per attuare le proprietà nel sistema.

I concetti fondamentali della semantica sono:

- nozione di soddisfacibilità (verità rispetto ad un modello)
- validità (verità rispetto a tutti i modelli)
- conseguenza logica (i modelli di una formula sono modelli anche della formula che ne segue logicamente).

La semantica della logica classica è basata su una visione statica della realtà. Ad esempio, data una formula  $F$  e fissato il dominio di interpretazione: (in questo momento)  $F$  è vera o falsa.

## Elementi di base

### Sintassi

La sintassi è un insieme di formule ammissibili  $F = \{\varphi_0, \varphi_1, \dots\}$  tipicamente definite tramite una Context Free Grammar.

### Semantica

La semantica è definita da due elementi:  $\langle M, \models \rangle$ :

- $M$  è un insieme di modelli  $M, M \in M$
- $\models \subset M \times F$  è la relazione di soddisfacibilità

### Terminologia e notazioni

$M \models \varphi$  significa che il modello  $M$  soddisfa la formula  $\varphi$   
o anche che la formula  $\varphi$  è soddisfatta dall'interpretazione/valutazione di  $M$

$M \models \varphi, \varphi', \dots$  equivale a  $M \models \varphi$  and  $M \models \varphi'$  and  $\dots$

## Logica delle proposizioni

La logica proposizionale (o enunciativa) è un linguaggio formale con una semplice struttura sintattica, basata fondamentalmente su proposizioni elementari (atomi) e su connettivi logici di tipo vero-funzionale, che restituiscono il valore di verità di una proposizione in base al valore di verità delle proposizioni connesse (solitamente noti come AND, OR, NOT...). La semantica della logica proposizionale definisce il significato dei simboli e di qualsiasi proposizione che rispetti le regole sintattiche del linguaggio, basandosi sui valori di verità associati agli atomi. Data una interpretazione (o modello) di una proposizione (in generale di un insieme di

proposizioni), e cioè una associazione tra le proposizioni elementari e le realtà rappresentate, possiamo generare un insieme infinito di proposizioni con significato definito che riguardino quella realtà. Ciascuna proposizione si riferisce quindi a uno o più oggetti della realtà rappresentata (anche astratta, ovviamente) e permette di descrivere o ragionare su quell'oggetto, utilizzando i due soli valori "Vero" e "Falso".

La logica delle proposizioni è la logica più elementare.

Dato un insieme finito di simboli proposizionali ( $P = \{p, q, r, \dots\}$ ), una **formula** può essere:

- $\top$  (costante VERA)
- $\perp$  (costante FALSA)
- un simbolo proposizionale  $p \in P$
- del tipo  $\neg\varphi$  (non  $\varphi$ )
- del tipo  $\varphi \vee \varphi'$  ( $\varphi$  oppure  $\varphi'$ )
- del tipo  $\varphi \wedge \varphi'$  ( $\varphi$  e  $\varphi'$ )
- del tipo  $\varphi \rightarrow \varphi'$  ( $\varphi$  implica  $\varphi'$ )

Null'altro può essere una formula  $\varphi$ .

In maniera più formale una logica delle proposizioni è definita da una **grammatica**:

$\varphi ::= p \mid \top \mid \perp \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \varphi \rightarrow \varphi'$

dove  $p$  è un simbolo proposizionale  $p \in P = \{p, q, r, \dots\}$

Un **modello** è una funzione  $M : P \rightarrow \{F, T\}$ . Un modello associa un valore booleano ad ogni proposizione. La **semantica** può essere definita tramite **tabelle di verità**.

$\varphi$	$\psi$	$\neg\varphi$	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Le relazioni di soddisfacibilità  $M \models \varphi$  sono definite come segue:

- $M \models p \Leftrightarrow M(p) = T$
- $M \models \top$
- $M \models \perp$
- $M \models \neg\varphi \Leftrightarrow \text{not } M \models \varphi \quad (M \not\models \varphi)$
- $M \models \varphi \vee \varphi' \Leftrightarrow M \models \varphi \text{ or } M \models \varphi'$
- $M \models \varphi \wedge \varphi' \Leftrightarrow M \models \varphi \text{ and } M \models \varphi'$
- $M \models \varphi \rightarrow \varphi' \Leftrightarrow M \models \varphi \text{ implica } M \models \varphi'$
- $M \models \varphi \leftrightarrow \varphi' \Leftrightarrow M(\varphi) = M(\varphi')$

Ad esempio  $\{p \rightarrow F, q \rightarrow T\} \models \top, \neg\perp, q, \neg p, q \vee p, q \wedge \neg p, p \rightarrow q$

Il metodo delle tabelle di verità è un metodo "semantico", cioè che esplora tutte le possibili configurazioni dell'universo per verificarne lo stato, si vogliono però costruire dei metodi basati più sulla deduzione che non sull'esplorazione e che si basino sull'attività di costruire derivazioni e dimostrazioni, cioè sul ragionamento e non sulla forza bruta.

## Concetti generali legati alla logica

Si svilupperanno dei metodi puramente sintattici, cioè dei calcoli che trasformano simboli in altri simboli, che permetteranno di creare dimostrazioni sulle sentenze logiche.

Si introduce un nuovo simbolo  $\vdash$  per indicare che da una sentenza possiamo derivare sintatticamente un'altra sentenza (deduzione logica).

Si hanno due concetti:

$A \models B$  significa che semanticamente se  $A$  è vera anche  $B$  è vera, mentre

$A \vdash B$  significa che se  $A$  è vera possiamo costruire una dimostrazione di  $B$ .

Il grande risultato della logica moderna è stato legare tra loro questi due concetti nel teorema di completezza:  $A \models B \equiv A \vdash B$ .

<b>Conseguenza logica</b>	Definizione	Una formula $\varphi$ è conseguenza logica di $\varphi_1, \dots, \varphi_n$ se, per ogni modello $M$ , quando $M \models \varphi_1, \dots, \varphi_n$ allora anche $M \models \varphi$
	Esempi	<ul style="list-style-type: none"> <li><math>p, q \models p \wedge q, p \vee q</math></li> <li><math>\alpha \models \beta \rightarrow \alpha</math></li> <li><math>\alpha \rightarrow (\beta \rightarrow \gamma) \models (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)</math></li> <li><math>\neg \alpha \rightarrow \neg \beta \models \beta \rightarrow \alpha</math></li> </ul>
<b>Tautologia/Validità</b>	Definizione	Una formula $\varphi$ che sia conseguenza logica di nessuna formula (sempre vera, per qualsiasi modello $M$ ), è detta tautologia.
	Notazione	$\models \varphi$
	Esempi	$\models \top, \neg \perp, p \rightarrow p, p \vee \neg p$
<b>Dimostrazione per contraddizione</b>	Definizione	Se $\varphi_1, \dots, \varphi_n, \neg \varphi \models \perp$ allora $\varphi_1, \dots, \varphi_n \models \varphi$
	Esercizio	$\varphi, \neg \varphi \models \alpha$ cos'è $\alpha$ ? $\alpha = \perp$ infatti $\varphi, \neg \varphi \models \perp$ per cui $\varphi \models \varphi$
<b>Equivalenza logica</b>	Definizione	Due formule $\varphi$ e $\varphi'$ si dicono <b>logicamente equivalenti</b> se $\varphi \models \varphi'$ e $\varphi' \models \varphi$
	Notazione	$\varphi \equiv \varphi'$
	Esempi	$\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$ Legge di De Morgan $p \rightarrow q \equiv (\neg p) \vee q \equiv (\neg q) \rightarrow (\neg p)$
<b>Soddisfacibilità</b>	Definizione	Una formula $\varphi$ è soddisfacibile se $M \models \varphi$ per qualche modello $M$
	Esempi	$\top, p, p \vee q$ sono soddisfacibili $\neg \top, \perp, p \wedge \neg p$ non sono soddisfacibili
	Note	$\varphi$ è valida se e solo se $\neg \varphi$ è non soddisfacibile

Una sentenza viene detta **soddisfacibile** quando un assegnamento di valori alle frasi atomiche che la compongono porta ad un valore di Vero per tutta la sentenza. Le sentenze che risultano sempre vere indipendentemente dal valore di verità che viene assegnato alle frasi atomiche vengono dette **tautologie**.

Il simbolo  $\models$  indica che una sentenza è soddisfacibile. Alla sinistra elenchiamo le proposizioni che devono essere vere, alla destra la proposizione che viene soddisfatta da queste condizioni.

$A \models B$  indica che  $B$  è vera quando  $A$  è vera, o meglio che da  $A$  è possibile derivare (semanticamente)  $B$ . Un modo alternativo per indicare questo fatto è dire che  $A$  è un modello per  $B$ ; forse poco comprensibile nel contesto del calcolo proposizionale, vedremo in seguito come invece sia molto importante nel calcolo dei predicati.

$\models A$  indica che  $A$  è una tautologia perché non ha bisogno di nessuna condizione per essere vera.

### Calcolo dell'equivalenza logica e verifica delle tautologie

Per valutare se due formule  $\varphi, \varphi'$  sono equivalenti, bisognerebbe verificare che  $\varphi \models \varphi'$  sia verificato per tutti gli infiniti modelli  $M$ . Fortunatamente però, nella logica delle proposizioni, i modelli significativi sono quelli che usano le variabili solo in  $\varphi$  e  $\varphi'$  (che sono finite). Si tratta pertanto di costruire le tabelle di verità di  $\varphi$  e  $\varphi'$ . Allo stesso modo si può ragionare per le tautologie:  $\models \varphi$ .

Se siamo in grado di dimostrare che  $\varphi_1 \dots \varphi_n \models \varphi$ , significa che la formula  $\varphi$  è valida per un vasto insieme di modelli (quelli nei quali sono valide  $\varphi_1 \dots \varphi_n$ ).



Sulla conseguenza logica si osserva che:

$$\varphi_1 \dots \varphi_n \models \varphi \quad \text{se e solo se} \quad \models (\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \varphi$$
$$\text{se e solo se} \quad \models (\varphi_1 \rightarrow (\varphi_2 \rightarrow \dots (\varphi_n \rightarrow \varphi) \dots))$$

Sull'equivalenza logica si osserva che:

$$\varphi \equiv \varphi' \quad \text{se e solo se} \quad \models (\varphi \rightarrow \varphi') \wedge (\varphi' \rightarrow \varphi)$$

## Una versione di logica delle proposizioni semplificata

Tutti gli operatori possono essere definiti solo utilizzando i termini  $\neg$  e  $\rightarrow$ :

$$\top \equiv p \rightarrow p$$

$$\perp \equiv \neg(p \rightarrow p)$$

$$\varphi \vee \varphi' \equiv (\neg\varphi) \rightarrow \varphi'$$

$$\varphi \wedge \varphi' \equiv \neg(\varphi \rightarrow \neg\varphi')$$

Sintassi:  $\varphi ::= p \mid \neg\varphi \mid \varphi \rightarrow \varphi$

Il calcolo della validità di una formula può tuttavia essere piuttosto oneroso, ad esempio una formula potrebbe essere  $\models \neg((a \wedge b \wedge c \wedge d \wedge e) \rightarrow c) \rightarrow (f \vee g \vee h \vee i)$ , per calcolarne la validità con le tabelle di verità occorrerebbe sviluppare 512 combinazioni (righe) e valutarle. Una tecnica più efficiente consiste nel trasformare la formula in una **Conjunctive Normal Form (CNF)**:

- una congiunzione di clausole ( $\wedge$ )
- una clausola è una disgiunzione di simboli ( $\vee$ )
- un simbolo è un positivo o negativo.

Al termine si tratterebbe solo di valutare se ogni clausola contiene una coppia di proposizioni opposte.

## Sistemi di deduzione

La **validità** può essere dedotta adottando un ragionamento per la dimostrazione:

- $(a \wedge b \wedge c \wedge d \wedge e) \rightarrow c$  è vero in quanto a sinistra è presente una congiunzione che include la formula  $c$
- $\neg((a \wedge b \wedge c \wedge d \wedge e) \rightarrow c)$  è falso
- una formula falsa implica qualsiasi formula

Si utilizzano gli **assiomi**:  $\varphi \wedge \varphi' \rightarrow \varphi$ ,  $\neg\neg\top$ ,  $\perp \rightarrow \varphi$

## Nozioni di deduzione logica

Una relazione di deduzione si indica con il simbolo  $\vdash$ :

- $\varphi_1, \dots, \varphi_n \vdash \varphi$  significa che  $\varphi$  può essere dedotto da  $\varphi_1, \dots, \varphi_n$
- in particolare  $\vdash \varphi$  significa che  $\varphi$  può essere dedotto (da nessuna formula)

$\models$  riguarda la semantica,  $\vdash$  riguarda la sintassi.

## Regole e assiomi

Se la relazione  $\models$  può essere definita in modo matematico, la relazione  $\vdash$  deve essere sempre definita in maniera costruttiva tramite assiomi e regole:

- Assiomi nella forma:  $\vdash \varphi$
- Regole nella forma:  $\frac{\vdash \varphi_1 \dots \vdash \varphi_n}{\vdash \varphi}$  che significa  $\varphi$  è verificata se tutte le precondizioni  $\varphi_1, \dots, \varphi_n$  sono vere.

Esempi:

- Assioma:  $\vdash \varphi \rightarrow \varphi$



- Regola: 
$$\frac{\vdash \varphi \rightarrow \varphi'' \quad \vdash \varphi'' \rightarrow \varphi'}{\vdash \varphi \rightarrow \varphi'}$$

Per calcolare una derivazione per  $\varphi_1 \dots \varphi_n \vdash \varphi$  si usa un insieme S di formule:

- inizialmente S è l'insieme delle formule  $\varphi_1 \dots \varphi_n$
- considerando un assioma o una regola le cui precondizioni corrispondono con S, si aggiunge ad S la formula ottenuta:
  - ◆ dall'assioma, sostituendo le sue variabili con qualche formula, oppure
  - ◆ dalla conseguenza della regola, sostituendo le sue variabili coerentemente con la precondizione
- si continua ad applicare gli step precedenti finché  $\varphi$  viene aggiunto alla formula S.

Alcune considerazioni:

- poiché gli assiomi sono schemi di formule, le formule da aggiungere a S sono selezionabili da un insieme infinito
- la procedura di selezione delle formule non è deterministica e può non terminare (ha esito positivo se esiste un modo per applicare assiomi o regole in modo tale che  $\varphi$  venga aggiunto a S)
- assomiglia davvero alla dimostrazione del teorema
- per controllare operativamente se  $\varphi_1 \dots \varphi_n \vdash \varphi$  sia derivabile, assiomi o regole dovrebbero essere applicati in modo intelligente
- per la logica proposizionale (ma non, ad esempio, per logica dei predicati) c'è un algoritmo che calcola una derivazione, se esiste.

## Teorema di deduzione

Definito il concetto di derivazione semantica, vediamo un importante teorema che permette di ridurre le ipotesi di cui abbiamo bisogno o di sfruttare come ipotesi una parte della sentenza che stiamo analizzando: il teorema di deduzione.

$A \models B$  è equivalente a  $\models A \rightarrow B$ , cioè il teorema dice che se A è un modello per B allora  $A \rightarrow B$  è una tautologia.

Se la cosa può sembrare a prima vista sorprendente, ad una analisi più attenta si può notare come sia la semantica profonda dell'implicazione. Vediamo con un semplice esempio in linguaggio naturale: da "Oggi piove" è possibile derivare ( $\models$ ) prendo l'ombrello, da ciò - senza nessuna precondizione - si può derivare ( $\models$ ) "Oggi piove allora ( $\rightarrow$ ) prendo l'ombrello".

Questo teorema sarà particolarmente utile nelle dimostrazioni, perché ci permette di spostare a destra o a sinistra del simbolo di derivazione l'antecedente dell'implicazione. Quindi, se dobbiamo dimostrare che A implica B, la strada più semplice sarà supporre A come ipotesi e quindi dimostrare B.

## Il sistema di deduzione di Hilbert

Sintassi (minimale)	$\varphi ::= p \mid \neg\varphi \mid \varphi \rightarrow \varphi$
Assiomi	$\vdash \alpha \rightarrow (\beta \rightarrow \alpha)$ $\vdash (\alpha \rightarrow (\beta \rightarrow \Box)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \Box))$ $\vdash (\neg\alpha \rightarrow \neg\beta) \rightarrow (\beta \rightarrow \alpha)$
Regola (modus ponens)	$\frac{\vdash \alpha \quad \vdash \alpha \rightarrow \beta}{\vdash \beta}$

Ad esempio, si può dimostrare:  $\alpha \rightarrow \Box, \Box \rightarrow \beta \vdash \alpha \rightarrow \beta$

We prove  $\alpha \rightarrow \gamma, \gamma \rightarrow \beta \vdash \alpha \rightarrow \beta$  that is  $\frac{\vdash \alpha \rightarrow \gamma \quad \vdash \gamma \rightarrow \beta}{\vdash \alpha \rightarrow \beta}$

$$\begin{array}{c} \vdash \alpha \rightarrow \gamma \quad [MP] \quad \frac{\vdash \gamma \rightarrow \beta \quad [A1] \quad \frac{\vdash (\gamma \rightarrow \beta) \rightarrow (\alpha \rightarrow (\gamma \rightarrow \beta))}{\vdash \alpha \rightarrow (\gamma \rightarrow \beta)}}{[A2] \quad \frac{(\alpha \rightarrow (\gamma \rightarrow \beta)) \rightarrow ((\alpha \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta))}{\vdash (\alpha \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta)}} \\ [MP] \frac{}{\vdash \alpha \rightarrow \beta} \end{array}$$

We prove  $\vdash (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow (\beta \rightarrow (\alpha \rightarrow \gamma))$

Exploiting soundness and completeness we can instead prove:

$\alpha \rightarrow (\beta \rightarrow \gamma) \vdash \beta \rightarrow (\alpha \rightarrow \gamma)$

$$\begin{array}{c} [A1] \frac{}{\vdash \beta \rightarrow (\alpha \rightarrow \beta)} \quad [MP] \frac{\vdash \alpha \rightarrow (\beta \rightarrow \gamma) \quad [A2] \frac{(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))}{\vdash (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)}}{\vdash \beta \rightarrow (\alpha \rightarrow \gamma)} \\ [TI] \frac{}{\vdash \beta \rightarrow (\alpha \rightarrow \gamma)} \end{array}$$

## Soundness di $\vdash$ rispetto a $\models$

Se  $\varphi_1, \dots, \varphi_n \vdash \varphi$  allora:  $\varphi_1, \dots, \varphi_n \models \varphi$       Le deduzioni derivate sono infatti conseguenze logiche.

## Completeness di $\vdash$ rispetto a $\models$

Se  $\varphi_1, \dots, \varphi_n \models \varphi$  allora:  $\varphi_1, \dots, \varphi_n \vdash \varphi$       Tutte le deduzioni che sono conseguenze logiche possono essere derivate.

I sistemi deduttivi sono, in sintesi, sistemi puramente sintattici, forniscono una modalità “tipografica” per verificare la validità di una formula. Non è necessario conoscere il significato dei simboli. Gli assiomi e le regole sono ciò di cui è sufficiente conoscere il significato.

## Natural deduction

Un sistema di deduzione alternativo per la logica delle proposizioni, riguarda la **natural deduction**:

- lavora con tutti gli operatori logici
- è molto più complessa
- permette di effettuare derivazioni di deduzioni più brevi
- è *sound* e *complete*.

## Logica dei predicati

La logica delle proposizioni permette di “lavorare” con sentenze tipo “se piove, allora è vero che piove oppure nevicata”: con  $p$  = piove,  $q$  = nevicata si ha  $\models p \rightarrow p \vee q$ . La **logica dei predicati** è più strutturata e permette di discutere proprietà individuali o globali, ad esempio:

- “tutti gli uomini sono mortali, Socrate è un uomo, quindi Socrate è mortale”
  - $\models (\forall X (\text{man}(X) \rightarrow \text{mort}(X)) \wedge \text{man}(\text{socrat}) \rightarrow \text{mort}(\text{socrat}))$
- “ogni studente è più giovane di qualche insegnante”
  - $M \models (\forall X (\text{stud}(X) \rightarrow (\exists Y (\text{instr}(Y) \wedge \text{younger}(X,Y))))$

Gli elementi della logica dei predicati sono:

- quantificatore universale  $\forall$
- quantificatore esistenziale  $\exists$
- le formule atomiche (non più semplici simboli proposizionali) sono predicati applicati a termini (es.  $\text{man}/1$ ,  $\text{younger}/2$ , ...)
- i termini possono essere variabili ( $X$ ) oppure costanti ( $\text{socrat}$ ).

Si può osservare che il primo esempio  $\models (\forall X (\text{man}(X) \rightarrow \text{mort}(X)) \wedge \text{man}(\text{socrat}) \rightarrow \text{mort}(\text{socrat}))$ , è una tautologia, indipendente dal significato di  $\text{man}$ ,  $\text{mort}$  e della costante  $\text{socrat}$ .

Il secondo esempio dipende dal modello  $M$  che deve fornire un’interpretazione per i predicati e i termini:  $M \models (\forall X (\text{stud}(X) \rightarrow (\exists Y (\text{instr}(Y) \wedge \text{younger}(X,Y))))$  è vera se la classe descritta dal modello  $M$  garantisce i vincoli.

## Sintassi

**Termini:** un termine su  $(C, F)$  ha sintassi  $t ::= x \mid c \mid f(t_1, \dots, t_n)$ , con:

- una variabile ( $x, y, z, \dots$ )
- una costante ( $c, d, e, \dots \in C$ )
- un termine composto: una funzione  $n$ -aria ( $f \in F$ ) con  $t_i$  termini e argomenti.

Un termine produce un individuo nel dominio applicativo.

**Formule:**  $\varphi ::= p(t_1, \dots, t_n) \mid \top \mid \perp \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \neg \varphi \mid \forall x \varphi \mid \exists x \varphi$ , con:

- una formula atomica: predicato  $n$ -ario ( $p \in P$ ) con  $t_i$  termini su  $(C, F)$  come argomenti ( $n$  può essere 0)
- una formula  $\top, \perp, \varphi \wedge \varphi, \varphi \vee \varphi, \varphi \rightarrow \varphi, \neg \varphi$  come per le proposizioni
- un quantificatore universale per  $x$  in  $\varphi$

- un quantificatore esistenziale per  $x$  in  $\varphi$ .

Un **modello** per le formule su  $(C, F, P)$  è del tipo  $(A, C, F, P)$ :

- $A$  è un universo non vuoto di valori concreti (gli individui)
- $C: C \rightarrow A$  è un'associazione di ogni costante ad un valore concreto
- $F: F \rightarrow (A^n \rightarrow A)$  è un'associazione di ogni funzione ad un nome concreto
- $P: P \rightarrow (A^n \rightarrow \{F, T\})$  è un'associazione di ogni nome predicato ad un predicato concreto.

Uno specifico modello associa:

- ogni termine  $t$  ad un valore concreto
- ogni formula  $p(t_1, \dots, t_n)$  ad un valore booleano.

Esempio di modello per la formula: "ogni studente è più giovane di qualche insegnante"

$M \models (\forall X (stud(X) \rightarrow (\exists Y (instr(Y) \wedge younger(X, Y))))$

$A = \{ \text{bravetti, rossi, omicini, bianchi, natali} \}$

$C = \{ \}$

$F = \{ \}$

$P = \{ \text{stud/1, instr/1, younger/2} \}$  // rappresentazione delle funzioni

$P = \{ \}$  // istanze

$\text{stud}(\text{bravetti}) = F, \text{stud}(\text{rossi}) = T, \text{stud}(\text{omicini}) = F, \text{stud}(\text{bianchi}) = T, \text{stud}(\text{natali}) = F,$

$\text{instr}(\text{bravetti}) = T, \text{instr}(\text{rossi}) = F, \text{instr}(\text{omicini}) = T, \text{instr}(\text{bianchi}) = F, \text{instr}(\text{natali}) = T,$

$\text{younger}(\text{rossi}, \text{natali}) = T, \text{younger}(\text{natali}, \text{omicini}) = F, \text{younger}(\text{bianchi}, \text{bravetti}) = T, \dots$

$\}$

## Aritmetica di Peano

### Formule:

- costante  $z$
- funzioni  $s/1, +/2, */2$
- predicatori  $>/2, =/2$

**Modello  $M_P = (A, C, F, P)$**

- $A =$  insieme  $N$  dei numeri naturali  $= \{0, 1, 2, \dots\}$
- $C = \{ (z, 0) \}$
- $F = \{ (s/1, \text{successore del termine in input}), (+/2, \text{somma matematica}), (* /2, \text{prodotto}) \}$
- $P = \{ (>/2, \text{maggiore tra due numeri}), (= /2, \text{uguaglianza tra due numeri}) \}$

Esempi di formule:

$M_P \models \exists y \neg \exists x (x * x = y)$  I numeri naturali non sono chiusi rispetto alla radice quadrata

$M_P \models \forall x \exists y (y > x)$  Per ogni numero naturale esiste sempre un numero naturale più grande

$M_P \models \forall q \exists p \forall x \forall y (p > q \wedge ((x > s(z) \wedge y > s(z)) \rightarrow \neg(x * y = p)))$

Per ogni numero naturale  $q$ , esiste un n.n.  $p$  più grande di  $q$  che non è calcolabile come prodotto di due nessuna coppia di n.n. maggiori di 1, ovvero per ogni n.n. esiste un numero primo più grande di esso (quindi i numeri primi sono infiniti).

## Variabili free e bound

Una variabile  $x$  utilizzata in una formula  $\varphi$  è detta **bound** se è utilizzata con un quantificatore:  $\forall x$  oppure  $\exists x$ .

Una variabile  $x$  utilizzata in una formula  $\varphi$  è detta **free** se non è bound.

Ad esempio: nella formula  $p(x) \wedge \forall x (p(x) \vee c)$  la prima occorrenza di  $x$  è free, l'occorrenza a destra è bound.

## Formule chiuse

Una formula  $\varphi$  è chiusa se non include occorrenze di variabili free.

## Semantica per la logica dei predicati (per formule chiuse)

Definizione delle relazioni di soddisfazione  $M \models \varphi$

- $M \models p(t_1, \dots, t_n) \Leftrightarrow M(p(t_1, \dots, t_n)) = T$
- $M \models \top, M \not\models \perp$
- $M \models \neg\varphi \Leftrightarrow M \not\models \varphi$
- $M \models \varphi \wedge \varphi' \Leftrightarrow M \models \varphi \text{ and } M \models \varphi'$
- $M \models \varphi \vee \varphi' \Leftrightarrow M \models \varphi \text{ or } M \models \varphi'$
- $M \models \varphi \rightarrow \varphi' \Leftrightarrow M \models \varphi \text{ implica } M \models \varphi'$
- $M \models \forall x\varphi \Leftrightarrow \text{per ogni } v \in A \text{ si ha } M \models \varphi[v/x]$
- $M \models \exists x\varphi \Leftrightarrow \text{esiste } v \in A \text{ per cui } M \models \varphi[v/x]$

dove  $\varphi[v/x]$  rappresenta la sostituzione di occorrenze della variabile free  $x$  in  $\varphi$  con  $v$ .

Ricordando che un modello  $M = (A, C, F, P)$  si osserva che:

1.  $M()$  per le formule atomiche:
  - a.  $M(p(t_1, \dots, t_n)) = P(p)(M(t_1), \dots, M(t_n))$
2.  $M()$  applicato ai termini  $t$  (con variabili  $x$  sostituite dai valori  $v$ )
  - a.  $M(v) = v$
  - b.  $M(c) = C(c)$
  - c.  $M(f(t_1, \dots, t_n)) = F(f)(M(t_1), \dots, M(t_n))$

## Validità

Nella logica dei predicati non è decidibile la validità di un problema. Turing e Church, in maniera indipendente, hanno dimostrato che:

- il problema  $\models \varphi$  è, in generale, indecidibile
- si può mostrare che valutare se una Turing Machine termina la propria esecuzione (espressa come un programma logico) può essere ricondotto al problema della validità:
  - l'esecuzione di un programma logico (es. PROLOG) è equivalente a stabilire la validità di una formula  $\varphi$  nella logica dei predicati (insoddisfacibilità di  $\neg\varphi$ )
  - basato su una procedura di risoluzione: la formula deve essere in CNF solo con clausole di Horn

## Semi-decidibilità della validità per logica dei predicati

Esiste una procedura, per una generica formula  $\varphi$ , per valutare  $\models \varphi$ :

- se  $\models \varphi$ , in una quantità finita di tempo si ottiene una risposta positiva
- se  $\not\models \varphi$ , la procedura non termina.

## Il sistema di deduzione di Hilbert

Sintassi (minimale)	$\varphi ::= p(t_1, \dots, t_n) \mid \neg\varphi \mid \varphi \rightarrow \varphi \mid \forall x\varphi$
Assiomi	$\vdash \alpha \rightarrow (\beta \rightarrow \alpha)$ $\vdash (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$ $\vdash (\neg\alpha \rightarrow \neg\beta) \rightarrow (\beta \rightarrow \alpha)$

	$\vdash ( \forall x \alpha ) \rightarrow \alpha[t/x]$ con variabili non free, t vincolato $\vdash \forall x ( \alpha \rightarrow \beta ) \rightarrow ( \alpha \rightarrow \forall x \beta )$ con x bound in $\alpha$
Regola	$\frac{\vdash \alpha \quad \vdash \alpha \rightarrow \beta}{\vdash \beta} \text{Modus Ponens}$ $\frac{\vdash \varphi}{\vdash \forall x \varphi} \text{Generalizzazione Universale}$

Si può dimostrare che  $\vdash$  è ancora corretto e completo (una formula è valida se e solo se è dimostrabile).

In conseguenza dell'indcidibilità di  $\models$ ,  $\vdash$  è indecidibile.

## Model Checking

Il model checking è un metodo per verificare algoritmicamente i sistemi formali. Viene realizzato mediante la verifica del modello, spesso derivato dal modello hardware o software, soddisfacendo una specifica formale. La specifica è spesso scritta come formule logiche temporali.

Il modello solitamente viene espresso come un sistema di transizioni, cioè grafo orientato formato da nodi (o vertici) e archi. Un insieme di proposizioni atomiche è associato ad ogni nodo. I nodi rappresentano gli stati di un sistema, gli archi rappresentano le possibili esecuzioni che alterino lo stato, mentre le proposizioni atomiche rappresentano le proprietà fondamentali che caratterizzano un punto di esecuzione.

Formalmente il problema è posto così: scelta una proprietà da verificare, espressa come una formula logica temporale  $p$ , e un modello  $M$  avente stato iniziale  $s$ , decidere se  $M \models p$ .

Gli strumenti del model checking si scontrano con la crescita esponenziale dell'insieme degli stati, comunemente conosciuto come il problema dell'esplosione combinatoria, che deve servire a risolvere la maggior parte dei problemi del mondo reale. I ricercatori hanno sviluppato algoritmi simbolici, riduzione parziale dell'ordine, diagrammi decisionali, astrazioni e model checking al volo per risolvere il problema. Questi strumenti furono inizialmente sviluppati per la correttezza logica dei sistemi a stati discreti, ma da allora sono stati estesi per trattare sistemi real-time e forme limitate di sistemi ibridi. (WIKIPEDIA)

### Logiche temporali

La logica temporale lineare o LTL (Linear Temporal Logic) è un'estensione della logica modale. A differenza delle logiche classiche che considerano un unico “mondo” sul quale si esprimono proprietà *sempre vere* o *sempre false*, le LTL considerano diversi “mondi” sono organizzati in una struttura lineare infinita: ogni mondo può così rappresentare un istante di tempo discreto.

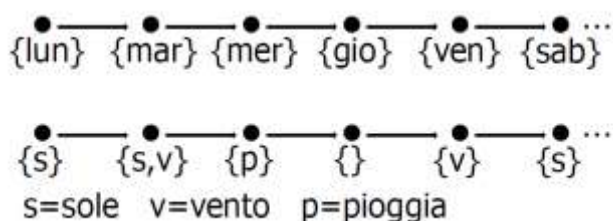
Questa logica trova impiego nella analisi dei sistemi i cui modelli possono essere sviluppati secondo le caratteristiche citate, sebbene l'algoritmo di Model Checking per LTL sia particolarmente complesso e dunque poco utilizzato. (rivisto da WIKIPEDIA)

Le logiche temporali permettono di effettuare valutazione utili per i software, sistemi dinamici nei quali una proprietà può assumere valori di verità differenti in differenti momenti.

Ogni mondo è quindi un diverso istante nel tempo.

Una sequenza lineare di “mondi” è una sequenza di insiemi di proposizioni che indicano i fatti veri in quell'istante temporale.

Esempi:



Le logiche temporali lineari (LTL) estendono la logica proposizionale con operatori utili a specificare proprietà su sequenze lineari di mondi:

$\circ\varphi$	NEXT	PROSSIMO	$\varphi$ è vero nel mondo successivo
$\square\varphi$	FOREVER	SEMPRE	$\varphi$ è vero in tutti i mondi futuri
$\diamond\varphi$	SOMETIME	UNA VOLTA	$\varphi$ è vero in qualche mondo futuro
$\varphi \text{ u } \psi$	UNTIL	FINCHE'	$\varphi$ è vero finché $\psi$ diventa vero

Esempio:  $\square ( (\neg \text{passaporto} \vee \neg \text{biglietto}) \Rightarrow \circ \neg \text{viaggio\_aereo} )$

Esempio per un sistema che richiede l'esecuzione di un'attività che viene richiesta, ricevuta, elaborata ed eseguita:

$\square ( \text{richiesta} \Rightarrow \diamond \text{ricevuta} )$	Il sistema garantisce che sempre, a fronte di una richiesta, questa sarà prima o poi ricevuta
$\square ( \text{ricevuta} \Rightarrow \circ \text{elaborata} )$	Il sistema garantisce che quando viene ricevuta una richiesta, all'istante successivo essa venga elaborata
$\square ( \text{elaborata} \Rightarrow \diamond \square \text{completata} )$	Il sistema garantisce che dal momento in cui una richiesta è in elaborazione, prima o poi - e poi per sempre - la richiesta sarà completata. Quindi la proprietà "richiesta $\wedge \square \neg \text{completata}$ " è falsa.

## Sintassi LTL

La sintassi per una logica LTL comprende:

$G_{LTL} ::=$	$\varphi, \psi \rightarrow p$	(proposizione atomica)
$\top$		(costante VERA)
$\perp$		(costante FALSA)
$\neg\varphi$		(complemento)
$\varphi \wedge \psi$		(congiunzione)
$\varphi \vee \psi$		(disgiunzione)
$\circ$		(next time, prossimo istante)
$\square$		(always, sempre)
$\diamond$		(sometime, una volta)
$\varphi \text{ u } \psi$		(until, finché)

## Modelli: sequenze di mondi

Considerando formule LTL che utilizzano proposizioni  $p \in P$ , un **modello**  $\pi$  è una **sequenza infinita**  $\pi_1, \pi_2, \pi_3, \dots$  di mondi in cui  $\pi_i$  è una funzione  $\pi_i: P \rightarrow \{ T, F \}$ .

## Semantica LTL

Definizione della relazione  $\pi \models \varphi$



$\pi \models p \Leftrightarrow \pi_1(p) = T$  *unico caso differente dalla logica proposizionale*

$\pi \models T$

$\pi \not\models \perp$

$\pi \models \neg \varphi \Leftrightarrow \text{not } \pi \models \varphi$

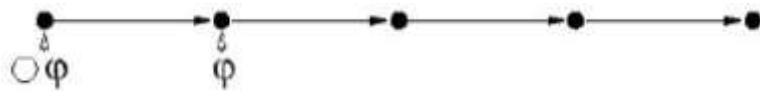
$\pi \models \varphi \vee \varphi' \Leftrightarrow \pi \models \varphi \text{ or } \pi \models \varphi'$

$\pi \models \varphi \wedge \varphi' \Leftrightarrow \pi \models \varphi \text{ and } \pi \models \varphi'$

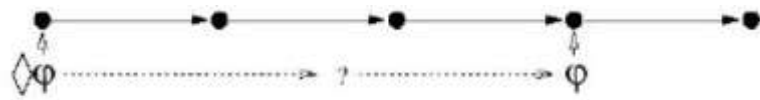
$\pi \models \varphi \rightarrow \varphi' \Leftrightarrow \pi \models \varphi \text{ implica } \pi \models \varphi'$

La semantica inoltre include gli operatori temporali.

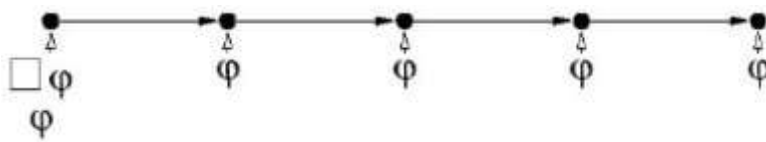
$\pi \models O \varphi \Leftrightarrow \pi^2 \models \varphi$  con  $\pi^i$  si rappresenta la sequenza infinita ottenuta partendo dal mondo i-esimo



$\pi \models \Diamond \varphi \Leftrightarrow \exists i \geq 1 : \pi^i \models \varphi$  indica che una proposizione è vera ora o in un qualche momento futuro

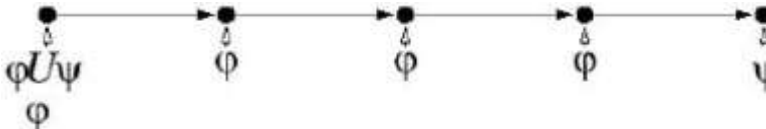


$\pi \models \Box \varphi \Leftrightarrow \forall i \geq 1 : \pi^i \models \varphi$  indica che una proposizione è vera ora e in tutti i momenti futuri



$\pi \models \varphi \text{ u } \psi \Leftrightarrow \exists i \geq 1 : \pi^i \models \psi \wedge \forall j < i : \pi^j \models \varphi$

mette in relazione due proposizioni, una vera da ora e in tutti i successivi mondi fino a quello in cui diventa vera la seconda (e falsa la prima)



## Operatori minimali

Tutti gli operatori possono essere espressi solo utilizzando NEXT e UNTIL.

ALWAYS e SOMETIME sono infatti duali:

- $\neg \Box \varphi \equiv \Diamond \neg \varphi$
- $\Box \varphi \equiv \neg \Diamond \neg \varphi$
- $\Diamond \varphi \equiv \neg \Box \neg \varphi$

SOMETIME può essere espresso usando UNTIL:

- $\Diamond \varphi \equiv T \text{ u } \varphi$

## Relazioni con operatori classici

- $\Diamond (\varphi \vee \psi) \equiv \Diamond \varphi \vee \Diamond \psi$
- $\Box (\varphi \wedge \psi) \equiv \Box \varphi \wedge \Box \psi$
- $\neg O \varphi \equiv O \neg \varphi$

distributività SOMETIME/OR

distributività ALWAYS/AND

“non è vero che nel prossimo mondo è vera  $\varphi$ ”  
 equivale a dire che “nel prossimo mondo non è vera  $\varphi$ ”

- $\neg (\varphi \text{ u } \psi) \equiv (\neg \psi \text{ u } (\neg \varphi \wedge \neg \psi)) \vee \Box \neg \psi$

## Notazioni alternative

SOMETIME	$\Diamond$	F (sometime in the Future)	$\Diamond$
ALWAYS	$\Box$	G (Globally in the future)	$\Box$
NEXT	$\bigcirc$	x (neXt time)	$()$

## Proprietà tipiche per sistemi software

Esistono alcuni concetti tipicamente importanti per i sistemi informatici:

- **safety:** proprietà che garantisce che non si raggiungono mai stati con errori  
La safety si esprime solitamente con la formula:  $\Box \neg \dots$
- **liveness:** proprietà per cui un sistema prima o poi eseguirà una certa azione  
La liveness si esprime solitamente con la formula:  $\Diamond \dots$  oppure  $\Box(\dots \Rightarrow \Diamond \dots)$
- **fairness:** proprietà che garantisce che se si richiede ad un sistema una azione infinite volte, il sistema la eseguirà infinite volte  
Esempio per fairness:  $\Box \Diamond \text{ready} \Rightarrow \Box \Diamond \text{run}$

## LTL come frammento decidibile della logica dei predicati

- Le proposizioni  $\pi_i(p)$  per  $i \geq 1$  sono codificate tramite predicati unari  $p(i)$  su numeri naturali
  - costante  $z$  associata a 0 e  $s(i)$  funzione successore

Una formula LTL  $\varphi$  è codificata dalla formula  $\llbracket \varphi \rrbracket_{s(z)}$ , dove:

$$\llbracket p \rrbracket_t = p(t)$$

$$\llbracket \bigcirc \varphi \rrbracket_t = \llbracket \varphi \rrbracket_{s(t)}$$

$$\llbracket \Diamond \varphi \rrbracket_t = \exists t' \ t' \geq t \wedge \llbracket \varphi \rrbracket_{t'}$$

$$\llbracket \Box \varphi \rrbracket_t = \forall t' \ t' \geq t \rightarrow \llbracket \varphi \rrbracket_{t'}$$

$$\llbracket \varphi \cup \psi \rrbracket_t = \exists t' \ t' \geq t \wedge \llbracket \psi \rrbracket_{t'} \wedge \forall t'' \ (t' \geq s(t'') \wedge t'' \geq 1) \rightarrow \llbracket \varphi \rrbracket_{t''}$$

## Labeled Transition Systems (NFA) e logiche temporali

Un **labeled transition system (LTS)** è un automa NFA senza stati di accettazione **F**.

$LTS = (Q, \Sigma, \delta, q_0)$  con  $\Sigma$  = insieme delle etichette

Una **traccia** di un LS è una sequenza di etichette:  $\bullet \xrightarrow{l_1} \bullet \xrightarrow{l_2} \bullet \xrightarrow{l_3} \bullet \xrightarrow{l_4} \bullet \xrightarrow{l_5} \bullet \dots$

Le tracce ricordano le stringhe degli NFA, iniziano dallo stato iniziale. La valutazione delle proposizioni viene fatta considerando le **tracce massimali** ovvero quelle di lunghezza infinita o quelle che terminano in uno stato di blocco.

Un LTS soddisfa una formula LTL se **tutte le sue tracce massimali** soddisfano tale formula.

Formalmente:

- una **traccia**  $\sigma$  di un  $LTS = (Q, \Sigma, \delta, q_0)$  è una sequenza di etichette  $l_1, l_2, \dots$  tale che esistono stati  $q_1, q_2, \dots$  con  $q_i \in \delta(q_{i-1}, l_i)$  per ogni  $i = 1 \dots \text{len}(\sigma)$
- una **traccia**  $\sigma$  è **massimale** se:
  - $\text{len}(\sigma) = \infty$
  - oppure
  - se  $q_{\text{len}(\sigma)}$  è uno stato di blocco, cioè se  $\delta(q_{\text{len}(\sigma)}, l_i) = \emptyset$  per ogni  $l \in \Sigma$
- una **traccia**  $\sigma = l_1, l_2, \dots$  di un  $LTS = (Q, \Sigma, \delta, q_0)$  **individua il modello**  $\pi$  definito da:
 
$$\pi_i(p) = T \Leftrightarrow i \leq \text{len}(\sigma) \wedge p = l_i$$
- un LTS soddisfa una formula  $\varphi$  di LTL se i modelli  $\pi$  individuati da **tutte le sue tracce massimali** sono tali che:  $\pi \models \varphi$

### Sistema di mutua esclusione per l'accesso alla sezione critica (esempi)

<p>Proprietà:</p> <p>✓ <b>safety</b></p> $\square (en1 \Rightarrow (\neg en2 \vee ex1)) \wedge \square (ene \Rightarrow (\neg en1 \vee ex2))$ <p>sezione critica protetta da accessi contemporanei</p> <p>✓ <b>liveness</b></p> $\square (re1 \Rightarrow (\diamond en1)) \wedge \square (re2 \Rightarrow (\diamond en2))$ <p>ogni volta che un utente chiede di accedere, prima o poi accederà</p> <p>✗ <b>fairness</b></p> $\square (\diamond re1 \wedge \diamond re2)$ <p>Esempio di traccia massimale:              re2-en2-ex2-re2-en2-ex2-re2-en2-ex2-...              si tratta di un ciclo con una traccia infinita che non esegue mai re1</p>	<p>Proprietà:</p> <p>✓ <b>safety</b></p> $\square (en1 \Rightarrow (\neg en2 \vee ex1)) \wedge \square (ene \Rightarrow (\neg en1 \vee ex2))$ <p>sezione critica protetta da accessi contemporanei</p> <p>✓ <b>liveness</b></p> $\square (re1 \Rightarrow (\diamond en1)) \wedge \square (re2 \Rightarrow (\diamond en2))$ <p>ogni volta che un utente chiede di accedere, prima o poi accederà</p> <p>✓ <b>fairness</b></p> $\square (\diamond re1 \wedge \diamond re2)$

## Modelli per sistemi concorrenti

I sistemi modellati da automi LTS sono, solitamente, **sistemi concorrenti** nei quali i vari processi **interagiscono** (come ad esempio i due utenti che vogliono accedere alla sezione critica).

## Process Algebra

Le **process algebra** sono approcci per modellare formalmente i sistemi concorrenti. Forniscono strumenti per la descrizione ad alto livello delle interazioni, comunicazioni e sincronizzazione tra insiemi di diversi processi.

1. Descrivono **ogni singolo processo**
2. **Combinano** le descrizioni per ottenere l'intero sistema

Principali esempi di process algebra sono CSP, CCS, ACP, LOTOS,  $\pi$ -calculus, ambient calculus, PEPA, fusion calculus, join calculus.

### Caratteristiche essenziali

Tutte le process algebra esistenti hanno alcune caratteristiche in comune:

- rappresentano le interazioni tra processi indipendenti come comunicazione (message-passing) piuttosto che come modifica della variabile condivisa
- descrivono i processi e i sistemi utilizzando un piccolo insieme di primitive e operatori
- definiscono leggi algebriche per gli operatori che permettono di processare espressioni che possono essere manipolate utilizzando ragionamenti di equivalenza.

### Esempi

Interruttore	<pre> SWITCH = OFF, OFF    = (on -&gt; ON), ON     = (off -&gt; OFF).         </pre> <pre> graph LR     0((0)) -- on --&gt; 1((1))     1 -- off --&gt; 0         </pre> <p>▪ Oppure, in modo più conciso:  <code>SWITCH = (on-&gt;off-&gt;SWITCH).</code></p>
Semaforo	<pre> TRAFFICLIGHT = (red-&gt;orange-&gt;green-&gt;orange -&gt; TRAFFICLIGHT).         </pre> <pre> graph LR     0((0)) -- red --&gt; 1((1))     1 -- orange --&gt; 2((2))     2 -- green --&gt; 3((3))     3 -- orange --&gt; 0         </pre>
Lancio di una moneta	<pre> COIN = (toss-&gt;HEADS   toss-&gt;TAILS), HEADS = (heads-&gt;COIN), TAILS = (tails-&gt;COIN).         </pre> <pre> graph LR     0((0)) -- toss --&gt; 1((1))     0 -- tails --&gt; 2((2))     1 -- heads --&gt; 0     2 -- tails --&gt; 0         </pre>

### Concorrenza (interleaving) e sincronizzazione

<b>Concorrenza (interleaving)</b>	Esecuzioni “logicamente” parallela di processi (LTS), come in un sistema multitasking	Transizioni specifiche di un processo (non presenti in altri processi) sono eseguite in <b>interleaving</b> : una azione alla volta in <b>ordine arbitrario</b> (non ci sono assunzioni sulla velocità relativa dei processi)
<b>Sincronizzazione</b>	Esecuzione “fisicamente”	La sincronizzazione avviene su <b>etichette comuni</b>

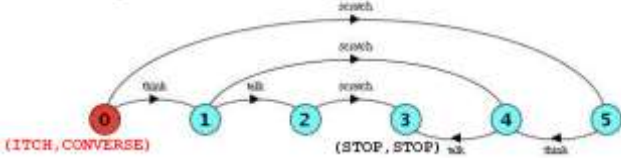
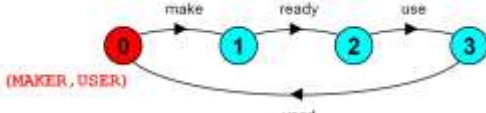
	contemporanea di azioni	dell'alfabeto degli LTS eseguiti in parallelo. Prevede l'esecuzione <b>simultanea</b> di una azione da parte di più LTS.
--	-------------------------	--

## Composizione parallela

Formalmente, dati due LTS  $A_1 = (Q_1, \Sigma_1, \delta_1, q_1^0)$  e  $A_2 = (Q_2, \Sigma_2, \delta_2, q_2^0)$ , la loro **composizione parallela**  $A_1 || A_2$  è un LTS  $(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_1^0, q_2^0))$ , dove  $\delta$  è definita come segue:

- caso della **sincronizzazione**:  
 per ogni  $a \in \Sigma_1 \cap \Sigma_2$   $\delta((p, q), a) = \delta_1(p, a) \times \delta_2(q, a)$   
 cioè: se  $p \xrightarrow{a} p'$  e  $q \xrightarrow{a} q'$  allora  $(p, q) \xrightarrow{a} (p', q')$
- caso dell'**interleaving**  
 per ogni  $a \notin \Sigma_1 \cap \Sigma_2$ 
  - $\delta((p, q), a) = \delta_1(p, a) \times \{q\}$  se  $a \in \Sigma_1$   
 cioè: se  $p \xrightarrow{a} p'$  allora  $(p, q) \xrightarrow{a} (p', q)$
  - $\delta((p, q), a) = \{p\} \times \delta_2(q, a)$  se  $a \in \Sigma_2$   
 cioè: se  $q \xrightarrow{a} q'$  allora  $(p, q) \xrightarrow{a} (p, q')$

Esempi:

Interleaving	Sincronizzazione
<p>ITCH = (scratch-&gt;STOP).</p> <p>CONVERSE = (think-&gt;talk-&gt;STOP).</p> <p>  CONVERSE_ITCH = (ITCH    CONVERSE).</p> <p>▪ Avendo <b>alfabeti disgiunti</b>, le azioni vengono eseguite in interleaving (STOP è stato di blocco):</p> 	<p>MAKER = (make-&gt;ready-&gt;used-&gt;MAKER).</p> <p>USER = (ready-&gt;use-&gt;used-&gt;USER).</p> <p>  MAKER_USER = (MAKER    USER).</p> <p>▪ "ready" e "used" sono <b>in entrambi gli alfabeti</b>, quindi tali transizioni devono <b>sincronizzarsi</b>:</p> 

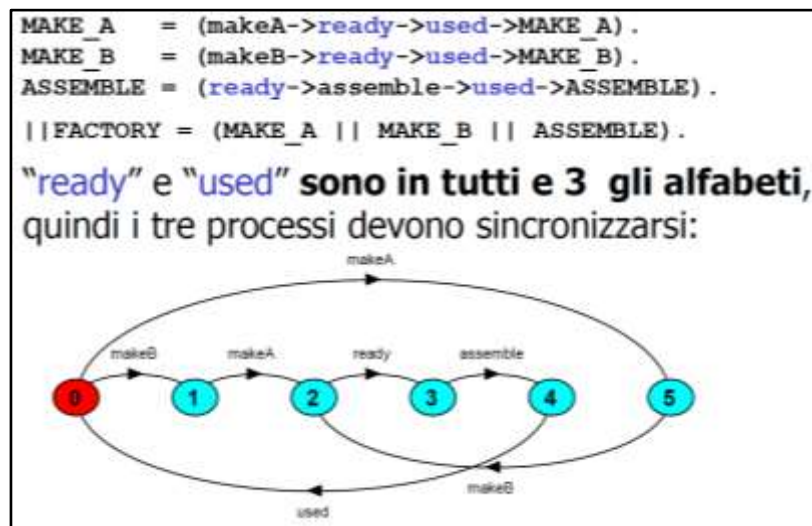
## Proprietà della composizione parallela

La composizione parallela è:

- **commutativa**: LTS di  $A_1 || A_2 = \text{LTS di } A_2 || A_1$
- **associativa**: LTS di  $(A_1 || A_2) || A_3 = \text{LTS di } A_1 || (A_2 || A_3)$

Quindi, la **composizione parallela di multipli LTS**  $A_1, A_2, \dots, A_n$  si può scrivere  $A_1 || A_2 || \dots || A_n$ .

Esempio di composizione multipla.



## Hiding

Dato un LTS  $A = (Q, \Sigma, \delta, q_0)$  l'LTS ottenuto **limitando il suo alfabeto** a  $\Sigma' \subseteq \Sigma$  è  $A@ \Sigma' = (Q, \Sigma', \delta', q_0)$ , dove  $\delta'$  è definita:

- per l'**azione speciale tau** (caso di azioni che si nascondono)  
 $\delta' (p, \tau) = \bigcup_{a \notin \Sigma'} \delta (p, a)$   
 cioè se  $p \xrightarrow{a} p' \wedge a \notin \Sigma'$  allora in  $A@ \Sigma'$  si ha  $p \xrightarrow{\tau} p'$
- per ogni  $a \in \Sigma'$  (caso di azioni che non si nascondono)  
 $\delta' (p, a) = \delta (p, a)$   
 cioè se  $p \xrightarrow{a} p' \wedge a \in \Sigma'$  allora in  $A@ \Sigma'$  si ha  $p \xrightarrow{a} p'$

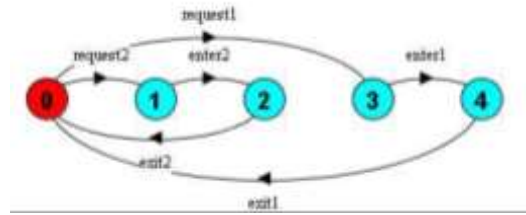
Esempio



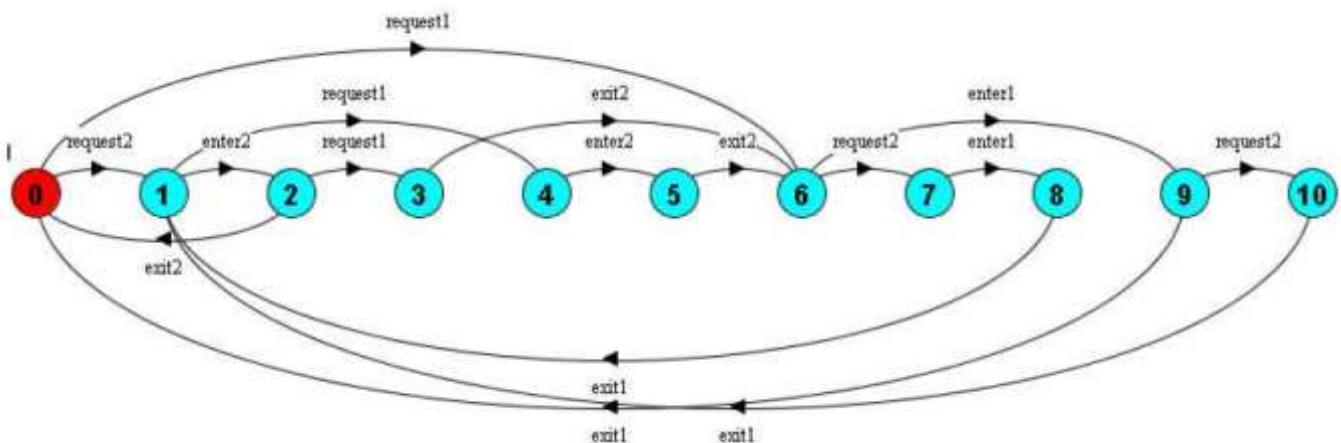


## Esempio mutua esclusione

$USER1 = (request1 \rightarrow enter1 \rightarrow exit1 \rightarrow USER1).$   
 $USER2 = (request2 \rightarrow enter2 \rightarrow exit2 \rightarrow USER2).$   
 $CONTROLLER = ( request1 \rightarrow enter1 \rightarrow exit1 \rightarrow USER1 \mid$   
 $request2 \rightarrow enter2 \rightarrow exit2 \rightarrow USER2 )$   
 $||MUTUAL\_EXCLUSION = ( USER1 \mid \mid USER2 \mid \mid CONTROLLER ).$

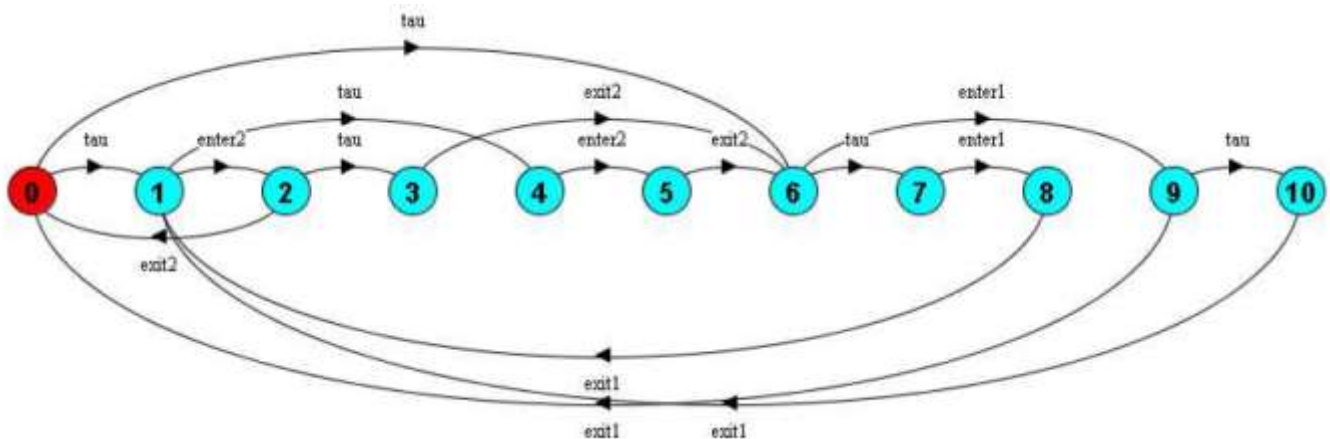


$USER1 = (request1 \rightarrow enter1 \rightarrow exit1 \rightarrow USER1).$   
 $USER2 = (request2 \rightarrow enter2 \rightarrow exit2 \rightarrow USER2).$   
 $CONTROLLER = ( request1 \rightarrow TURN1 \mid request2 \rightarrow TURN2 )$   
 $TURN1 = ( request2 \rightarrow enter1 \rightarrow exit1 \rightarrow TURN2 \mid$   
 $enter1 \rightarrow ( request2 \rightarrow exit1 \rightarrow TURN2 \mid$   
 $exit1 \rightarrow CONTROLLER ) ).$   
 $TURN2 = ( request1 \rightarrow enter2 \rightarrow exit2 \rightarrow TURN1 \mid$   
 $enter2 \rightarrow ( request1 \rightarrow exit2 \rightarrow TURN1 \mid$   
 $exit2 \rightarrow CONTROLLER ) ).$   
 $||MUTUAL\_EXCLUSION = ( USER1 \mid \mid USER2 \mid \mid CONTROLLER ).$



E' possibile nascondere le azioni di richiesta::

$||MUTUAL\_EXCLUSION = ( USER1 \mid \mid USER2 \mid \mid CONTROLLER )@ \{enter1, enter2, exit1, exit2\}.$



## Reti di Petri

Una rete di Petri (conosciuta anche come rete posto/transizione o rete P/T) è una delle varie rappresentazioni matematiche di un sistema distribuito discreto. Come un linguaggio di modellazione, esso descrive la struttura di un sistema distribuito come un grafo bipartito con delle annotazioni. Ovvero, una rete di Petri ha dei nodi posti, dei nodi transizioni e degli archi diretti che connettono posti e transizioni.

Furono inventate nel 1962 durante la tesi di dottorato dell'autore Carl Adam Petri. [WIKIPEDIA]



Una rete di Petri (Petri Net, PN), formalmente, è una quintupla  $PN = (P, T, F, W, M_0)$  così definita:

- $P = \{p_0, p_1, \dots, p_m\}$  è un insieme finito di piazze
- $T = \{t_1, t_2, \dots, t_n\}$  è un insieme finito di transizioni
- $F \subseteq (P \times T) \cup (T \times P)$  è un insieme di archi
- $W: F \rightarrow \{1, 2, 3, \dots\}$  è una funzione che associa peso o costo ad ogni arco.
- $M_0: P \rightarrow \{0, 1, 2, \dots\}$  è una mappatura, detta **initial marking**, che associa ad ogni piazza un valore iniziale

tale che  $P \cap T = \emptyset$  e  $P \cup T \neq \emptyset$ .

Le reti di Petri possono essere considerate come un'estensione naturale degli automi in cui le piazze rappresentano gli stati e le transizioni modificano gli stati attivi. Un sistema di **tokens** permette di associare ad ogni piazza un certo numero di tokens rappresentando lo stato complessivo del sistema. Le transizioni **consumano** tokens dall'insieme degli stati attivi e **generano** nuovi tokens modificando il **multi-insieme degli stati attivi**.

Il **marking graph** di una rete di Petri è un LTS che ne definisce il comportamento.  $M_0$  rappresenta l'initial marking, ogni passaggio di configurazione definisce un nuovo marking. Ogni marking definisce una **configurazione** della rete di Petri indicando la quantità di tokens presenti in ciascuna piazza.

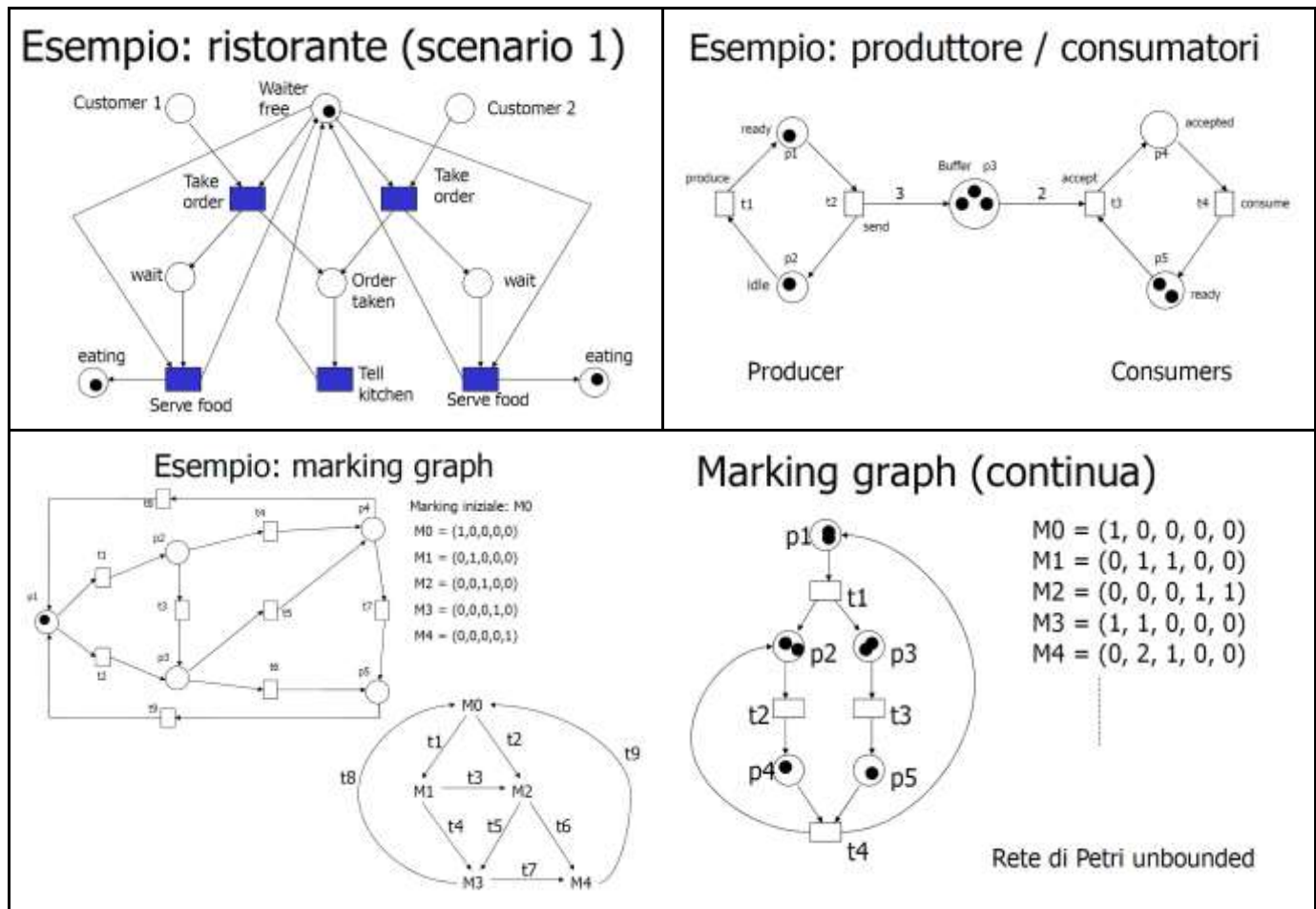
Formalmente:

- un marking  $M$ , è un multi-insieme di piazze:  $M: P \rightarrow \{0, 1, 2, \dots\}$
- una **transizione**  $t \in T$  è **abilitata** in un marking se  $\forall p \in P \quad M(p) \geq W(p, t)$
- l'**esecuzione** di una transizione  $t \in T$  abilitata nel marking, fa *passare* la rete di Petri al marking  $M'$  dato da:  
$$\forall p \in P \quad M'(p) = M(p) - W(p, t) + W(t, p).$$

Una rete di Petri si dice **unbounded** se il suo marking graph risulta essere **infinito**, ad esempio in presenza di cicli.



## Esempi di reti di Petri



## Model checking su reti di Petri

Il model checking su reti di Petri consiste nel verificare se una **formula** LTL è **soddisfatta**:

- se la rete è bounded si considera il marking graph come un LTS che rappresenta il comportamento della rete di Petri e si valutano i **modelli**  $\pi$  per tutte le sue tracce massimali considerando ogni marking come uno stato. Le proposizioni “p” che risultano vere se la corrispondente piazza “p” contiene almeno un token.

Nell’esempio a fianco relativo al problema dei 4 filosofi a tavola, è possibile verificare alcune proprietà del sistema:

$\square (\neg (p_1 \wedge p_3))$

“non si verificherà mai che i due filosofi vicini mangeranno contemporaneamente” ✓ VERO

$\square (\neg (p_8) \Rightarrow (\diamond p_8))$

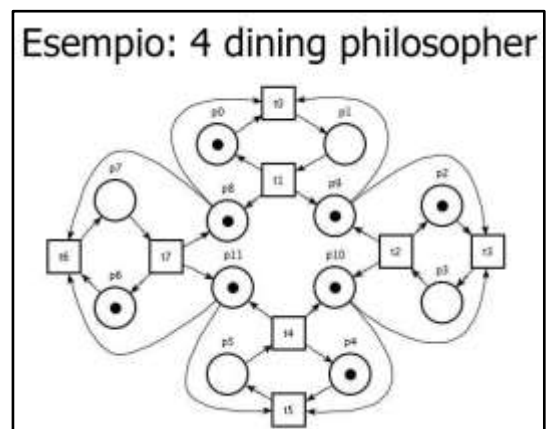
“accade sempre che se una forchetta (ad esempio p8) è in uso, prima o poi sarà rilasciata” ✗ FALSO

$\diamond p_1$

“il filosofo (p1) mangerà” ✗ FALSO

$\neg (\diamond (p_1 \wedge p_5))$

“non si raggiunge uno stato in cui due filosofi lontani mangiano insieme” ✗ FALSO.



# Macchine di Turing

## Introduzione

I linguaggi finora analizzati (regolari, liberi, ...) permettono di delimitare e risolvere alcuni problemi (analisi di protocolli, ricerca nei testi, parsing, ...). Si vuole a questo punto cercare di capire quali linguaggi possono essere definiti da un qualsiasi apparato computazionale, ovvero cosa possono fare i computer?

Il riconoscimento di una stringa è paragonabile alla descrizione di un problema; ciò che è eseguito da un computer è la risoluzione del problema posto.

Esistono specifici problemi che non possono essere risolti utilizzando un calcolatore, questi problemi sono detti **indecidibili**. Esistono programmi che richiedono quantità indeterminate di tempo per fornire un output; il problema legato al fatto di non sapere quando un programma termina definisce la causa dell'incapacità di dire cosa fa il programma stesso.

Per dimostrare che non esiste un programma in grado di eseguire un determinato compito è necessario definire alcuni formalismi, per ora si può pensare che potrebbe non essere possibile sapere se determinate azioni saranno svolte da un certo programma osservando - ad esempio - i due programmi C di seguito esposti, chiedendosi: "il programma stampa la stringa hello, world?".

<pre>main () {     printf("hello, world\n"); }</pre>	<pre>int exp(int i, n) /* computes i to the power n {     int ans, j;     ans = 1;     for (j=1; j&lt;=n; j++) ans *= i;     return ans; }  main () {     int n, total, x, y, z;     scanf("%d", &amp;n);     total = 3;     while (1) {         for (x=1; x&lt;=total-2; x++)         {             z = total - x - y;             if (exp(x,n) + exp(y,n) == exp(z,n)                 printf("hello, world\n");             }         total++;     } }</pre>
--	--

Mentre il primo programma stampa certamente la stringa, il secondo cerca soluzioni intere all'equazione  $x^n + y^n = z^n$  dato in input il valore di  $n$ . In assenza di dimostrazioni (solo recentemente fornite) al Teorema di Fermat per cui non esistono soluzioni all'equazione per  $n > 2$ , non si può sapere se e quando il problema riuscirà a stampare la stringa.

Si può osservare che i **problemi** sono espressi come stringhe di un linguaggio e che il numero di diversi linguaggi su un alfabeto costituito da più di un simbolo non è conteggiabile.

D'altra parte, i **programmi** sono stringhe di lunghezza finita su un linguaggio e sono conteggiabili: è possibile ordinarli per lunghezza e, a parità di lunghezza, secondo l'ordine lessicografico; in questo modo è possibile parlare del primo programma, del secondo programma, ...

Risulta che esistono infinitamente meno programmi che problemi e l'unica ragione per la quale i problemi sembrano spesso **decidibili** è che raramente si affrontano problemi casuali, spesso si approssimano problemi piuttosto semplici e ben strutturati che sono spesso decidibili.

## Macchine di Turing

In informatica una macchina di Turing (o più brevemente MdT) è una macchina ideale che manipola i dati contenuti su un nastro di lunghezza potenzialmente infinita, secondo un insieme prefissato di regole ben definite. Si tratta di un modello astratto che definisce una macchina in grado di eseguire algoritmi dotata di un nastro potenzialmente infinito su cui può leggere e/o scrivere dei simboli.

È un potente strumento teorico che viene largamente usato nella teoria della calcolabilità e nello studio della complessità degli algoritmi in quanto è di notevole aiuto agli studiosi per comprendere i limiti del calcolo meccanico. La sua importanza è tale che, per definire in modo formalmente preciso la nozione di algoritmo, la si riconduce alle elaborazioni effettuabili con Turing Machine..

La MdT come modello di calcolo è stata introdotta nel 1936 da Alan Turing per dare risposta all'Entscheidungsproblem (problema di decisione) proposto da Hilbert nel suo programma di fondazione formalista della matematica. [WIKIPEDIA]

**“Se un problema è intuitivamente calcolabile - risolvibile tramite l'esecuzione di una procedura - allora esiste una Turing Machine in grado di risolverlo”.**

Formalmente una **Turing Machine** è una 7-pla  $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$  dove

- $Q$  è un **insieme finito di stati**
- $\Sigma$  è un **alfabeto di input**
- $\Gamma$  è un **alfabeto del nastro** (contiene  $\Sigma$ )
- $\delta$  è una **funzione di transizione**
- $q_0 \in Q$  è lo **stato iniziale**
- $B \in \Gamma - \Sigma$  è un **simbolo blank** (all'inizio il nastro contiene solo l'input circondato da blank)
- $F \subseteq Q$  è un insieme di **stati finali**.

La **funzione di transizione** è definita come  $\delta(q, Z) = (p, Y, D)$  dove:

- $q \in Q$  è uno **stato (di partenza)**
- $Z \in \Sigma$  è un **simbolo del nastro (letto)**
- $p \in Q$  è uno **stato (di arrivo)**
- $Y \in \Sigma$  è un **simbolo del nastro (scritto)**
- $D$  è una **direzione**: Left o Right

$(p, Y, D)$  può essere indefinito.

L'azione  $\delta(q, Z) = (p, Y, D)$ :

- cambia lo stato da  $q$  a  $p$  (eventualmente  $p = q$ )
- rimpiazza il simbolo  $Z$  con il simbolo  $Y$  sul nastro (eventualmente  $Z = Y$ )
- muove la testina di una cella nella direzione  $D$ .

Si può immaginare una Turing Machine costituita da una testina (finite control) che può trovarsi in uno qualsiasi degli stati definiti in  $Q$ . La testina “punta” ad una cella del nastro che può contenere simboli appartenenti all'input oltre a simboli aggiuntivi del nastro. Inizialmente l'input è una stringa finita di simboli dell'alfabeto e la testina punta all'inizio della stringa. Una mossa (azione) della Turing Machine è una funzione dello stato e del simbolo a cui punta la testina, l'azione determina:

1. un cambiamento di stato (potrebbe essere lo stesso stato prima dell'azione)
2. la scrittura di un simbolo nella cella letta, tale simbolo sostituisce quello pre-esistente, eventualmente la Turing Machine può riscrivere lo stesso simbolo letto
3. un movimento del nastro verso destra o verso sinistra.

## Convenzioni

- I simboli in input sono indicati da lettere maiuscole dell'alfabeto, a partire dalle prime: a, b, c, ...
- I simboli del nastro sono identificati da lettere maiuscole a partire dalle ultime: Z, Y, X, ...
- le stringhe di simboli in input sono identificate da lettere minuscole, a partire dalle ultime: z, y, x, ...
- Le stringhe di simboli del nastro sono identificate da lettere greche minuscole:  $\alpha$ ,  $\beta$ ,  $\gamma$ , ...

## Esempio

Si considera una TM che legge una stringa in input binario ( { 0, 1 } ) in attesa di trovare un 1.

1. La testina punta inizialmente al primo carattere dell'input
2. Se trova uno 0, si sposta a destra di una cella
3. Se trova un 1, lo cambia in 0, entra in uno stato finale f, e si ferma
4. Se trova un blank, scrive un 1 e cambia direzione

Formalmente:  $TM = ( \{q, f\}, \{0,1\}, \{0,1,B\}, \delta, q, B, f )$  con

- $\delta(q, 0) = (q, 0, R)$
- $\delta(q, 1) = (f, 0, R)$
- $\delta(q, B) = (q, 1, L)$

Azione	Input				↓				
$\square(q, 0) = (q, 0, R)$	...	B	B	0	0	0	B	B	...
					↓				
$\square(q, 0) = (q, 0, R)$	...	B	B	0	0	0	B	B	...
						↓			
$\square(q, B) = (q, 1, L)$	...	B	B	0	0	B	B	B	...
					↓				
$\square(q, 0) = (q, 0, R)$	...	B	B	0	0	0	1	B	...
						↓			
$\square(q, 1) = (f, 0, R)$	...	B	B	0	0	1	1	B	...
							↓		
La TM è in uno stato finale: si ferma e accetta.	...	B	B	0	0	0	B	B	...

## Descrizione Istantanea (ID) di una Turing Machine

In generale, una **descrizione istantanea (Instantaneous Description, ID)** è rappresentata tramite una stringa  $\alpha q \beta$  dove:

- a sinistra di  $\alpha \beta$  il nastro contiene solo BLANK
- a destra di  $\alpha \beta$  il nastro contiene solo BLANK
- la testina punta al primo carattere di  $\beta$  (se  $\beta$  è vuota la testina punta a B)
- $q$  è lo stato della Turing Machine.

Si utilizzano i simboli  $\vdash$  e  $\vdash^*$  per rappresentare i **passaggi di ID**:

- $\vdash$  può essere letto come “diventa in una mossa”
- $\vdash^*$  può essere letto come “diventa in zero o più mosse”

Le descrizioni istantanee per la TM dell'esempio precedente sono:  
 $q00\vdash 0q0\vdash 00q\vdash 0q01\vdash 00q1\vdash 000f$ .

Formalmente una **mossa** può essere definita come:

- se  $\delta(q, Z) = (p, Y, D) \Rightarrow \alpha q Z \beta \vdash \alpha Y p \beta$ 
  - nota: se  $Z = B \Rightarrow \alpha q \vdash \alpha Y p$
- se  $\delta(q, Z) = (p, Y, L) \Rightarrow \alpha q Z \beta \vdash \gamma p X Y \beta$       dove  $\alpha = \gamma X$  oppure  $\alpha = \gamma = \varepsilon$  e  $X = B$ 
  - nota: se  $Z = B \Rightarrow \alpha q \vdash \gamma p X Y$       dove  $\alpha = \gamma X$  oppure  $\alpha = \gamma = \varepsilon$  e  $X = B$

## Linguaggio di una Turing Machine

Una Turing Machine definisce un linguaggio per stato finale:  $L(M) = \{ w \mid q_0 w \vdash^* \alpha q \beta, \text{ con } q \text{ finale} \}$

Si osservi che per una stringa  $w \notin L(M)$  la Turing Machine può:

- **bloccarsi** in uno stato non finale
- **continuare** indefinitamente a computare senza mai raggiungere uno stato finale.

## Linguaggi ricorsivi e ricorsivamente enumerabili

I linguaggi riconosciuti dalle Turing Machine sono detti **linguaggi ricorsivamente enumerabili**.

Il termine, inventato prima delle Turing Machine, fa riferimento da una nozione di calcolo basata sulle funzioni.

“Un insieme  $A$  è detto ricorsivamente enumerabile quando esiste una funzione di enumerazione  $f$  di cui  $A$  è il codominio. Essendoci una corrispondenza biunivoca tra l'insieme delle funzioni calcolabili e l'insieme dei programmi in un qualsiasi linguaggio di programmazione, un insieme è quindi ricorsivamente enumerabile se è possibile generare i suoi elementi attraverso un programma per calcolatore (per la tesi di Church-Turing è indifferente il linguaggio di programmazione scelto). Un insieme ricorsivamente enumerabile è anche detto **semidecidibile** in quanto è possibile stabilire (in un tempo non quantificabile) se un elemento generico appartiene ad  $A$ , ma non è possibile stabilire la non appartenenza di un elemento.” [WIKIPEDIA]

Un **algoritmo è una Turing Machine che sicuramente termina** (per ogni input fornito).

I linguaggi riconosciuti da algoritmi sono detti **linguaggi ricorsivi**.

In altri termini, qualsiasi linguaggio per il quale sia possibile verificare se una data stringa vi appartiene oppure no in modo automatico è ricorsivo. Ciò significa che stabilire se una certa stringa  $w$  appartiene al linguaggio  $L(M)$  è un problema **decidibile**.

Esempio di linguaggi ricorsivi: ogni Context Free Language (CFL) è ricorsivo in quanto è sempre possibile utilizzare l'algoritmo CYK per riconoscerlo.

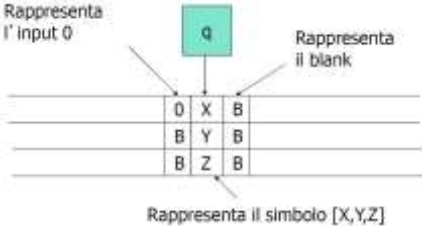
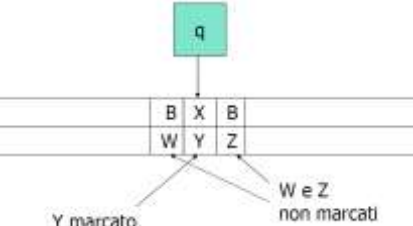
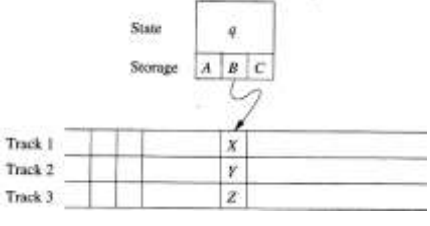
In matematica, logica e informatica teorica, i linguaggi decidibili o ricorsivi sono una classe di linguaggi formali che corrisponde alla classe dei problemi decidibili. Esistono due definizioni principali equivalenti per questa classe:

1. Un linguaggio ricorsivo è un linguaggio per il quale esiste una macchina di Turing che, data una qualsiasi stringa di input, termina accettando la stringa se essa appartiene al linguaggio, e termina rifiutando la stringa in caso contrario.
2. Un linguaggio ricorsivo è un sottoinsieme ricorsivo dell'insieme di tutte le possibili stringhe sull'alfabeto del linguaggio.

Tutti i linguaggi ricorsivi sono ricorsivamente enumerabili. Sono ricorsivi tutti i linguaggi regolari, libero dal contesto e dipendenti dal contesto. È degno di nota il fatto che questa categoria non abbia un corrispondente diretto nella classificazione di Chomsky.

## Trucchi di programmazione / estensioni per le Turing Machine

Tracce multiple	Marcature	Memorie di stati
I simboli del nastro possono essere considerati come vettori di $k$	Utilizzando le tracce multiple, una traccia viene utilizzata per <b>marcare</b>	Si possono implementare vettori di stati: il primo elemento è lo <b>stato di</b>

<p>elementi. Ogni elemento appartiene ad un alfabeto finito.                  E' come il nastro avesse k tracce.                  L'input è costituito da vettori che hanno solo un elemento diverso da BLANK.</p>	<p>alcune posizioni. In questa traccia, inizialmente tutta BLANK, alcune celle possono contenere un simbolo speciale.</p>	<p><b>controllo</b>, gli altri contengono simboli presi da un alfabeto finito.</p>
		

## Esempio

Si descrive una Turing Machine che **copia l'input** (stringa binaria) **infinite volte**.

Descrizione degli stati di controllo:

q: marca la posizione e memorizza il simbolo letto

p: va a destra, continuando a ricordarsi il simbolo letto, finché non trova un BLANK, dove scriverà il simbolo

r: va a sinistra fino alla marcatura

- Gli **stati** della Turing Machine hanno la forma  $[x, Y]$ :
  - $x \in \{q, p, r\}$
  - $Y \in \{0, 1\}$
- I **simboli** del nastro hanno forma  $[U, V]$ :
  - $U \in \{X, B\}$  con  $X := \text{marcatura}$ ,  $B := \text{BLANK}$
  - $V \in \{0, 1, B\}$ 
    - $[B, B]$  è il BLANK della Turing Machine
    - $[B, 0]$  è l'input 0
    - $[B, 1]$  è l'input 1

- la **funzione di transizione**  $\square$  è:  
 (convenzione: a e b sono simboli dell'alfabeto di input  $\{0, 1\}$ )

$\square([q, B], [B, a]) = ([p, a], [X, a], R)$	Nello stato q: → copia il simbolo in input sotto la testina (a) nello stato → marca la posizione letta → imposta lo stato di controllo p → muove a destra
$\square([p, a], [B, b]) = ([p, a], [B, b], R)$	Nello stato p: → va a destra in attesa di trovare un simbolo BLANK
$\square([p, a], [B, B]) = ([r, B], [B, a], L)$	Nello stato p: → se trova un BLANK lo rimpiazza con il simbolo memorizzato nello stato (a) → imposta lo stato di controllo r → muove a sinistra
$\square([r, B], [B, a]) = ([r, B], [B, a], L)$	Nello stato r: → va a sinistra in attesa di trovare la marcatura
$\square([r, B], [X, a]) = ([q, B], [B, a], R)$	Nello stato r: → se trova la marcatura la rimuove → imposta lo stato di controllo q → muove a destra

INPUT

...	B	B	B	B	B	B	...
...	0	1	B	B	B	B	...

q
B

$\square([q,B], [B,a]) = ([p,a], [X,a], R)$   
 $a = 0$

...	B	B	B	B	B	B	...
...	0	1	B	B	B	B	...

p
0

$\square([p,a], [B,b]) = ([p,a], [B,b], R)$   
 $a = 0$

...	X	B	B	B	B	B	...
...	0	1	B	B	B	B	...

p
0

$\square([p,a], [B,B]) = ([r,B], [B,a], L)$   
 $a = 0$

...	X	B	B	B	B	B	...
...	0	1	B	B	B	B	...

r
B

$\square([r,B], [B,a]) = ([r,B], [B,a], L)$   
 $a = 0$

...	X	B	B	B	B	B	...
...	0	1	0	B	B	B	...

r
B

$\square([r,B], [X,a]) = ([q,B], [B,a], R)$   
 $a = 0$

...	X	B	B	B	B	B	...
...	0	1	0	B	B	B	...

q
B

$\square([q,B], [B,a]) = ([p,a], [X,a], R)$   
 $a = 0$

...	B	B	B	B	B	B	...
...	0	1	0	B	B	B	...

p
1

$\square([p,a], [B,b]) = ([p,a], [B,b], R)$   
 $a = 1$

...	B	X	B	B	B	B	...
...	0	1	0	B	B	B	...

p
1

$\square([p,a], [B,B]) = ([r,B], [B,a], L)$   
 $a = 1$

...	B	X	B	B	B	B	...
...	0	1	0	B	B	B	...

r
B

$\square([r,B], [B,a]) = ([r,B], [B,a], L)$   
 $a = 1$

...	X	B	B	B	B	B	...
...	0	1	0	1	B	B	...

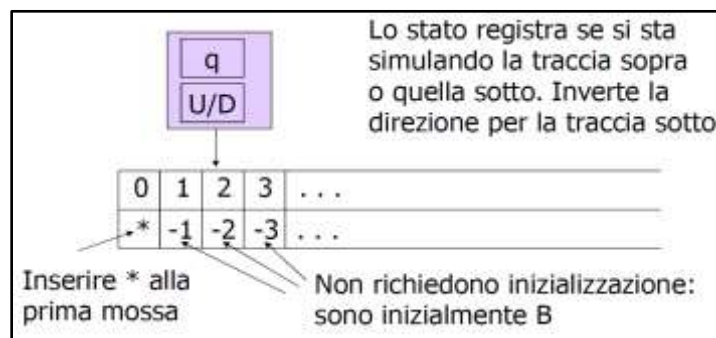


## Turing Machine con nastro semi-infinito

Le Turing Machine finora descritte utilizzano un nastro infinito (infatti possono effettuare arbitrari movimenti a sinistra o a destra). Tuttavia si può pensare ad una Turing Machine con nastro semi-infinito senza restrizioni di computabilità. Per farlo è sufficiente strutturare una Turing Machine con due tracce:

1. la prima traccia contiene i simboli nelle posizioni 0, 1, 2, ...
2. la seconda traccia contiene i simboli nelle posizioni -1, -2, ... (a sinistra)

Lo stato registra se si sta utilizzando la prima o la seconda traccia (Up/Down). Gli spostamenti sulla seconda traccia (Down) sono invertiti.



Il nastro può inoltre essere simulato tramite due stack che contengono rispettivamente le posizioni a sinistra e a destra della testina. Questo permette di vedere una Turing Machine come PDA con due pile.

## Turing Machine multinastro

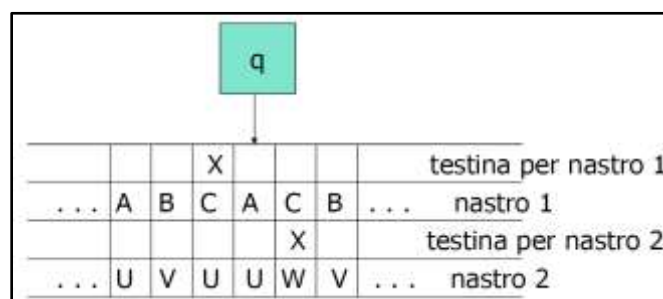
Una estensione delle Turing Machine permette di avere **k nastri e k testine**. Le mosse sono definite a partire dallo stato e dal simbolo sotto la testina di ogni nastro.

In una mossa la Turing Machine può:

- cambiare stato
- scrivere simboli sotto le testine
- muovere ogni testina indipendentemente.

E' possibile simulare i k nastri utilizzando un solo nastro multitraccia con  $2k$  tracce:

- ogni nastro è rappresentato da una traccia
- ogni nastro utilizza una traccia aggiuntiva per determinare la posizione della propria testina
  - attraverso la marcatura
- si effettuano spostamenti avanti/indietro simulando i passi di tutte le singole testine
  - ogni passata simula una mossa.



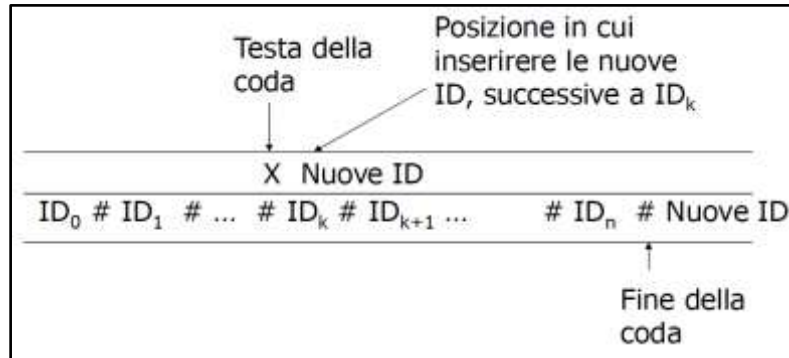
## Turing Machine non deterministiche

Le Turing Machine definite precedentemente hanno una sola possibile scelta di triple (stato da impostare, simbolo da scrivere, direzione da assumere) per ogni coppia (stato attuale, simbolo letto) e sono per questo motivo **deterministiche**.

Le **Turing Machine non deterministiche** invece possono avere più triple e possono quindi agire con diverse opzioni, prendendo diverse strade: accettano gli input se esiste una possibile sequenza di scelte che porta ad uno stato accettante.

E' possibile simulare una Turing Machine non deterministica (NTM) tramite una Turing Machine Deterministica (DTM):

- il nastro della DTM contiene una **coda di ID della NTM**
- una **seconda traccia** è utilizzata per:
  - **marcare la ID in testa alla coda**
  - **generare**, una alla volta, **le successive ID di quella in testa alla coda** e copiarle in fondo alla coda.

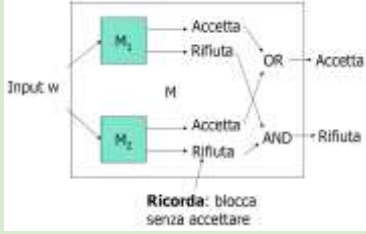
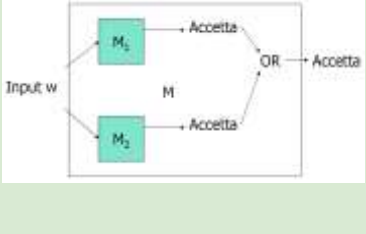
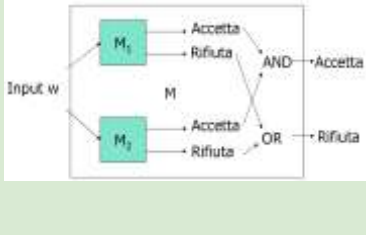
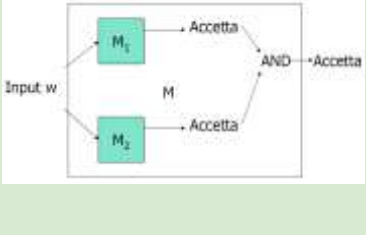
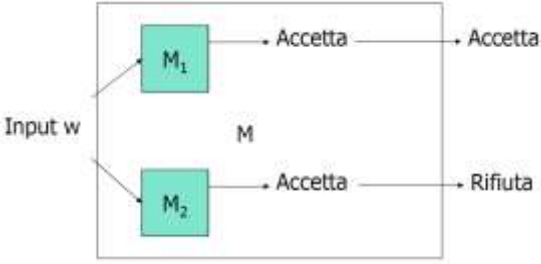


- la DTM cerca la ID in testa alla coda
- cerca lo stato della ID per determinare le possibili mosse
- crea una nuova ID per ogni mossa e la copia in fondo alla coda
- dopo aver effettuata la copia di tutte le ID in fondo alla coda, si sposta la marcatura in testa alla coda
- se si crea un ID con stato accettante la ID si ferma

## Vantaggi delle estensioni

Si è visto che tutte le Turing Machine con le diverse tipologie di estensioni sono realizzabili con Turing Machine semplici ed in ogni caso il linguaggio riconosciuto è lo stesso, per questo motivo si possono utilizzare le diverse tipologie per ottenere scopi simili.

## Proprietà di chiusura per Linguaggi Ricorsivi e ricorsivamente enumerabili

	Ricorsivi	Ricorsivamente enumerabili
<b>Chiusura rispetto ad Unione</b> Siano: <ul style="list-style-type: none"> <li>★ <math>L_1 = L(M_1)</math> e <math>L_2 = L(M_2)</math></li> <li>★ <math>M_1</math> e <math>M_2</math> due Turing Machine con un solo nastro semi-infinito</li> </ul> Si costruisce una nuova Turing Machine $M$ con 2 nastri che: <ol style="list-style-type: none"> <li>1. copia l'input sul secondo nastro</li> <li>2. simula in parallelo le due Turing Machine                             <ol style="list-style-type: none"> <li>a. <math>M_1</math> sul primo nastro</li> <li>b. <math>M_2</math> sul secondo nastro</li> </ol> </li> </ol>	Se $M_1$ e $M_2$ sono algoritmi allora $M$ sicuramente termina entrambe le simulazioni: è un algoritmo.  $M$ accetta se $M_1$ o $M_2$ accettano. 	Potrebbe accadere che una, o entrambe, $M_1$ e $M_2$ non terminino mai (senza bloccarsi né accettare)  $M$ accetta se $M_1$ o $M_2$ accettano. 
<b>Chiusura rispetto ad Intersezione</b> Siano: <ul style="list-style-type: none"> <li>★ <math>L_1 = L(M_1)</math> e <math>L_2 = L(M_2)</math></li> <li>★ <math>M_1</math> e <math>M_2</math> due Turing Machine con un solo nastro semi-infinito</li> </ul> Si costruisce una nuova Turing Machine $M$ con 2 nastri che: <ol style="list-style-type: none"> <li>3. copia l'input sul secondo nastro</li> <li>4. simula in parallelo le due Turing Machine                             <ol style="list-style-type: none"> <li>a. <math>M_1</math> sul primo nastro</li> <li>b. <math>M_2</math> sul secondo nastro</li> </ol> </li> </ol>	$M$ accetta se $M_1$ e $M_2$ accettano. 	$M$ accetta se $M_1$ e $M_2$ accettano. 
<b>Chiusura rispetto a Complemento</b> 	La Turing Machine prima o poi si ferma. Se non accetta la stringa, la sua complementare accetta.	La Turing Machine potrebbe non fermarsi mai e non è pertanto possibile determinare se accetta o meno. In generale il complemento può non essere Ricorsivamente Enumerabile. Teorema di Post Se $L$ è RE ed il complemento di $L$ è RE allora $L$ è ricorsivo.
<b>Chiusura rispetto a Differenza</b> La differenza tra $L_1$ e $L_2$ è l'intersezione di $L_1$ con il complemento di $L_2$ . $L_1 \setminus L_2 = L_1 \cap L_2^{-1}$	I linguaggi ricorsivi sono chiusi rispetto a intersezione e complemento $\Rightarrow$ sono chiusi anche rispetto alla differenza.	Se fossero chiusi rispetto alla differenza dovrebbero essere chiusi anche rispetto al complemento.
	Ricorsivi	Ricorsivamente enumerabili
<b>Chiusura rispetto a concatenazione</b> Siano: <ul style="list-style-type: none"> <li>★ <math>L_1 = L(M_1)</math> e <math>L_2 = L(M_2)</math></li> <li>★ <math>M_1</math> e <math>M_2</math> due Turing Machine con un solo nastro semi-infinito</li> </ul> Si costruisce una nuova Turing Machine non deterministica $M$ con 2 nastri che: <ol style="list-style-type: none"> <li>1. prova una suddivisione dell'input <math>w = xy</math></li> <li>2. muove <math>y</math> sul secondo nastro</li> </ol>	Si provano sistematicamente tutte le possibili suddivisioni (sono finite) dell'input $w = xy$ : <ul style="list-style-type: none"> <li>• se <math>M_1</math> e <math>M_2</math> accettano entrambe una qualche suddivisione allora <math>M</math> accetta</li> <li>• rifiuta se non trova suddivisioni che</li> </ul>	Tramite non determinismo, se $M_1$ e $M_2$ accettano, allora $M$ accetta.

3. simula $M_1$ su $x$ e $M_2$ su $y$ 4. accetta se entrambe accettano	accettano.	
<b>Chiusura rispetto a Stella di Kleene</b>	Come per la concatenazione.	Come per la concatenazione.
<b>Chiusura rispetto al Reverse</b>	Si inverte l'input e si simula una Turing Machine per $L$ sull'input invertito.	Si inverte l'input e si simula una Turing Machine per $L$ sull'input invertito.

## Decidibilità

### Macchine di Turing codificate come numeri

E' possibile definire un **ordinamento** tra le infinite stringhe di un alfabeto, esiste pertanto una **relazione biunivoca da numeri naturali a stringhe** e ha senso parlare di stringa  $j$ -esima (stringa di indice  $j$ ).

Allo stesso modo è possibile definire un **ordinamento tra tutte le Turing Machine** (es. ordine lessicografico della loro descrizione finita), anche le Turing Machine sono infinite ed esiste allora una **funzione biunivoca da numeri naturali a Turing Machine** (e ha senso parlare di Turing Machine  $i$ -esima, di indice  $i$ ).

## Diagonalizzazione

E' possibile costruire una **tabella di accettazione** che indichi, per ogni Turing Machine di indice  $i$ , se questa riconosce la stringa di indice  $j$ .

La cella  $(i,j)$  della tabella assume valori:

- 0 se la Turing Machine  $i$ -esima **non riconosce** la stringa  $j$ -esima
- 1 se la Turing Machine  $i$ -esima **riconosce** la stringa  $j$ -esima.

	Stringa $j$ -esima →							
		1	2	3	4	5	6	...
Turing Machine $i$ -esima ↓	1							
	2							
	3							
	4							
	5							
	6							
	...							

I valori presenti nella diagonale costituiscono una sequenza infinita di 0 e 1.

Invertendo tale sequenza (1 al posto degli 0 e viceversa) si ottiene una sequenza diversa, per almeno un valore, da tutte le righe della tabella.

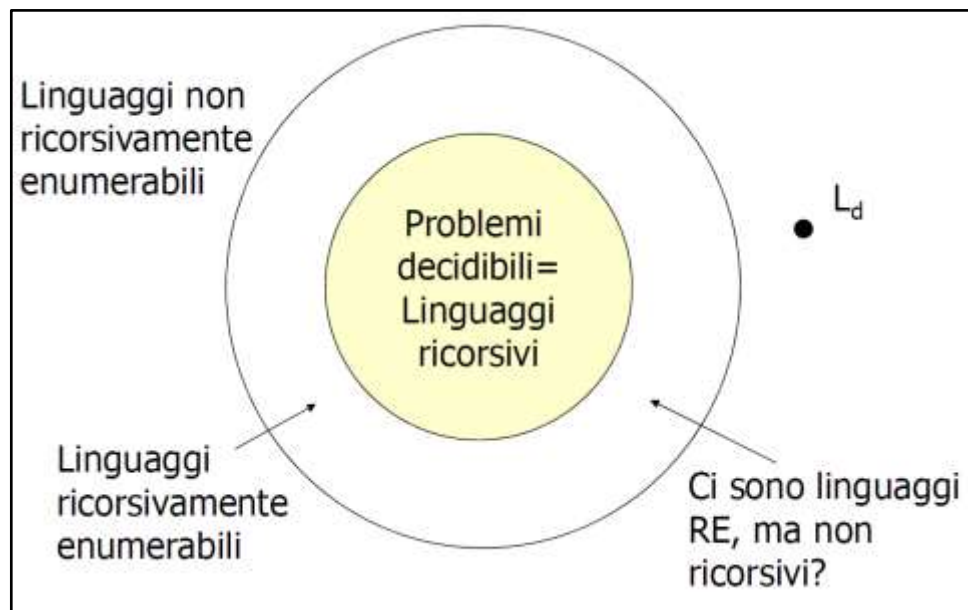
Formalmente, la sequenza  $D = a_1a_2...$  è definita come:  $D = a_i$  con  $a_i = 1 - Ta(i,i)$ , con  $Ta$  = Tabella di accettazione.

$D$  non può essere una riga della tabella e rappresenta un linguaggio (sequenza infinita di stringhe) che nessuna Turing Machine riconosce.

Si definisce allora il linguaggio di diagonalizzazione:

$$L_d = \{ w \mid w \text{ è la stringa } i\text{-esima e la } i\text{-esima Turing Machine non riconosce } w \}$$

$L_d$  non è un linguaggio ricorsivamente enumerabile perché non esiste nessuna Turing Machine in grado di riconoscerlo. Si tratta di un risultato teorico interessante.



## Problemi indecidibili

Esempi di problemi indecidibili sono:

1. Può una certa linea di codice essere eseguita durante l'esecuzione di un programma?
2. Può una variabile contenere un certo valore durante l'esecuzione di un programma?

## Linguaggio Universale

Un esempio di linguaggio **ricorsivamente enumerabile** ma **non ricorsivo** è il linguaggio  $L_u$  della **Macchina di Turing Universale (UTM)**.

La UTM ha come input l'indice  $i$  di una certa Turing Machine  $M$  e l'indice  $j$  di una stringa  $w$  e accetta se e solo se  $M_i$  accetta  $w_j$ .

In pratica UTM (realizzabile con più nastri) simula le Turing Machine.

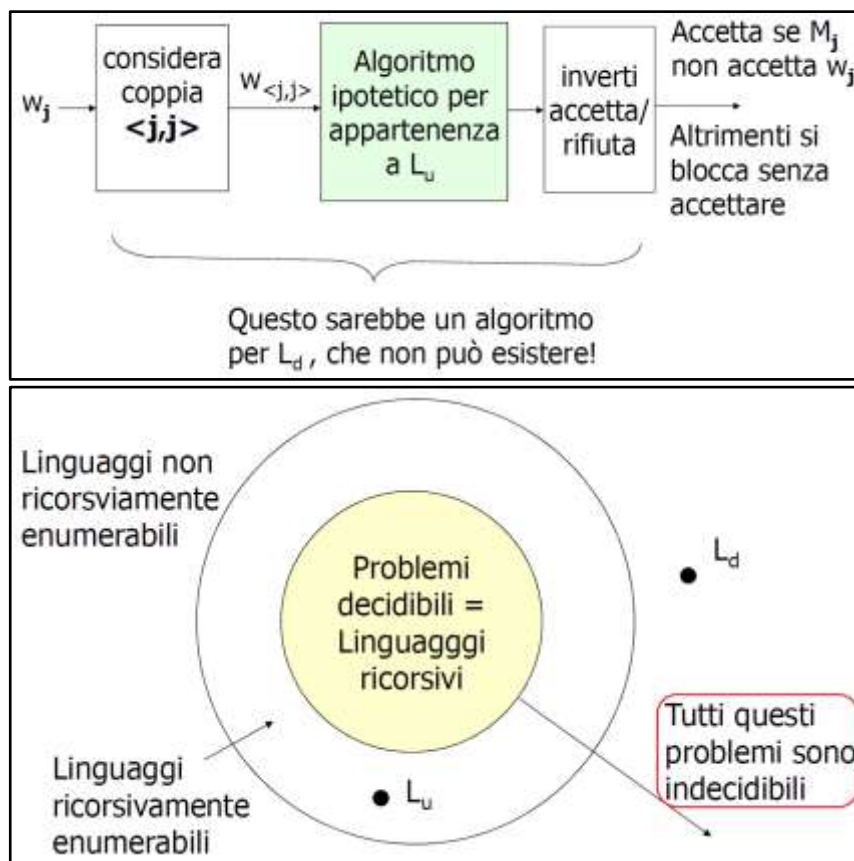
Formalmente, supponendo di **enumerare le coppie**  $\langle i, j \rangle$  e di rappresentarle con corrispondenti **stringhe**  $w_{\langle i, j \rangle}$ , allora  $L_u = \{ w_{\langle i, j \rangle} \mid M_i \text{ accetta } w_j \}$ .

UTM garantisce che  $L_u$  è sicuramente **ricorsivamente enumerabile**.

$L_u$  non può essere ricorsivo infatti, supponendo per assurdo che  $L_u$  sia ricorsivo, data una stringa  $w$  si può decidere se essa si trovi in  $L_d$ :

- si calcola a quale indice  $j$  corrisponde la stringa  $w \Rightarrow w = w_j$
- si verifica se  $w_{\langle j, j \rangle}$  **non appartiene a**  $L_u$  e in tal caso si conclude che  $w$  è in  $L_d \Rightarrow$  IMPOSSIBILE.

Conclusione:  $L_u$  è ricorsivamente enumerabile ma non ricorsivo.



## Indecidibilità di $L_u$ (problema della fermata)

Alcuni esempi:

- stabilire se, dato il codice di una funzione (es. in C) ed il valore dei parametri, l'esecuzione del codice terminerà è un problema **indecidibile**
- allo stesso modo, dato un programma, il nome  $x$  di una variabile un valore  $v$ , capire se  $x$  può assumere il valore  $v$  durante l'esecuzione è un problema **indecidibile**
- stabilire se, data una formula della logica dei predicati, essa è valida o meno, è ancora un problema **indecidibile**.

Si osservi che:

- **la programmazione logica** (es. Prolog) è **Turing equivalente**
- **un programma logico è una formula** nella logica dei predicati
- **la terminazione con successo** dell'esecuzione, per un goal, **si ha quando la formula totale è valida**.

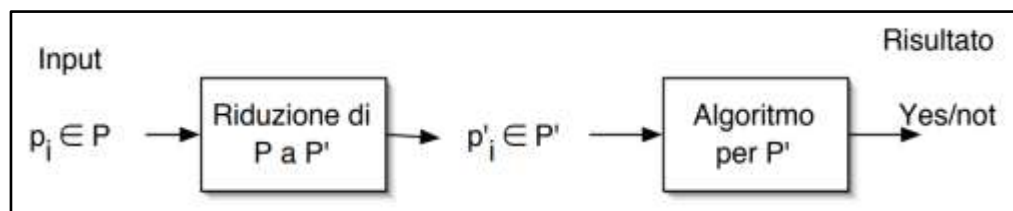
Ora, considerando un ipotetico **linguaggio di programmazione** con il quale è possibile scrivere **solli algoritmi**, cioè un linguaggio di programmazione in cui si ha sempre garanzia della terminazione di un programma (accettazione o rifiuto). Tale programma accetta tutti i linguaggi ricorsivi (cioè tutti i problemi risolvibili con algoritmi), ma tale linguaggio può esistere? Utilizzando la tabella di diagonalizzazione è possibile dimostrare che tale linguaggio non esiste, infatti la diagonale definisce un linguaggio non riconosciuto.

	<b>Ricorsivamente enumerabile</b>	<b>Ricorsivo = Decidibile</b>
<b><math>L_u</math></b>	<b>Si</b>	<b>No</b>
<b><math>L_d</math></b>	<b>No</b>	<b>No</b>

## Riduzioni

In generale un algoritmo che converte istanze di un problema  $P_1$  in istanze di un problema  $P_2$  che ha le stesse risposte/soluzioni, è detto **riduzione**.

In altre parole un problema  $P_1$  si dice **Turing riducibile** ad un problema  $P_2$  se esiste una macchina di Turing che, presa una qualsiasi istanza  $p \in P_1$  come input, produce un istanza associata  $p' \in P_2$  in cui la soluzione di  $p$  può essere ricavata dalla soluzione di  $p'$ .



$P_2$  è complesso almeno quanto  $P_1$ , pertanto (**teorema**):

se esiste una riduzione da  $P_1$  a  $P_2$  allora:

- se  $P_1$  è indecidibile lo è anche  $P_2$
- se  $P_1$  è non ricorsivamente enumerabile, lo è anche  $P_2$ .

## Teorema di Rice

### Decidibilità di linguaggi ricorsivamente enumerabili (riconosciuti da Turing Machine)

Se, data una Turing Machine  $M$ ,  $L(M)$  soddisfa una proprietà  $P$  (ad esempio  $P$  = essere vuoto), allora il problema è decidibile.

Alcuni esempi di problemi su linguaggi ricorsivamente enumerabili, quindi riconosciuti da una Turing Machine  $M$ :

- $L(M)$  è un linguaggio regolare?
- $L(M)$  è un CFL?
- $L(M)$  include stringhe palindromo?
- $L(M)$  è vuoto?
- $L(M)$  contiene più di 1000 stringhe?
- $L(M) = L(M')$ ?
- ...

Chiedersi se  $L(M)$  soddisfa una certa proprietà  $P$  equivale a valutare se  $L_P$  è **ricorsivo**, considerando  $L_P = \{ w_i \mid L(M_i) \text{ soddisfa } P \}$ .

Scelta una proprietà  $P$  per un linguaggio ricorsivo, si possono osservare due casi **banali**:

- la proprietà  $P$  è sempre soddisfatta ( $L_P = \emptyset$ )
- la proprietà  $P$  non è mai soddisfatta ( $L_P =$  tutte le stringhe)

Si osserva che i due linguaggi sono complementari e, per il teorema di Post, sono ricorsivi.

**Teorema di Rice: ogni proprietà non banale dei linguaggi ricorsivamente enumerabili è indecidibile.**

In altre parole: è impossibile riconoscere con una Turing Machine le stringhe binarie che rappresentano codici di una Turing Machine il cui linguaggio soddisfa una proprietà non banale.

Esempio: “ $L$  è un linguaggio CFL”.

Una proprietà si dice non banale se appartiene a tutti i linguaggi ricorsivamente enumerabili, oppure a nessuno.

**Il teorema di Rice non si applica a problemi che riguardano caratteristiche di una Turing Machine non esclusivamente riguardanti il linguaggio**, ad esempio:

- $M$  riconosce  $w$  in 10 passi?
- $M$  è la  $i$ -esima Turing Machine?
- $M_i$  riconosce  $w_i$ ?



## Calcolabilità di funzioni

La decidibilità di un problema può essere anche espressa come **calcolabilità della funzione**  $f: \mathbb{N} \rightarrow \{0,1\}$ , per cui  $f(i) = 1$  se  $w_i \in L$ .

- **f si dice calcolabile quando il problema è decidibile** (calcolato da una Turing Machine che termina sempre)
- **f si dice parzialmente calcolabile quando il problema è semi-decidibile** (calcolato tramite Turing Machine).

Il concetto di calcolabilità può essere esteso a **calcolabilità di funzioni**:  $f: \mathbb{N} \rightarrow \mathbb{N}$  per cui:

- **f si dice calcolabile quando esiste una Turing Machine che termina sempre e che, su un input i, da risultato f(i)**
- anziché valutare se le Turing machine terminano o meno, **si valuta se le Turing Machine danno un risultato.**

## Esercizio

Si consideri il seguente linguaggio

$L = \{ w_{\langle x,y \rangle} \mid x \text{ e } y \text{ sono tali che } L(M_x) \text{ e } L(M_y) \text{ hanno almeno una stringa in comune} \}$

dove  $M_x$  e  $M_y$  sono la  $x$ -esima e la  $y$ -esima macchina di Turing.

Dire se  $L$  è ricorsivo, ricorsivamente enumerabile o nemmeno ricorsivamente enumerabile.

Giustificare la risposta.

## Soluzione

**L e' ricorsivamente enumerabile**, infatti, considerando una TM NON-DET  $M$  con multipli nastri che opera come segue (inizialmente l'input " $w$ " si trova sul primo nastro):

1. decodifico l'input calcolando, su due nastri dedicati,  $x$  e  $y$  tali che  $w = w_{\langle x,y \rangle}$
2. decodifico  $x \rightarrow M_x$  ed  $y \rightarrow M_y$
3. creo non-deterministicamente una stringa qualsiasi  $w'$  in un nastro non ancora usato
4. copio  $w'$  in modo da averla in due nastri
5. uso i due nastri per simulare in parallelo  $M_x$  con input  $w'$  ed  $M_y$  con input  $w'$  (come per le costruzioni di chiusura rispetto a unione e intersezione)
6. se entrambe le simulazioni hanno raggiunto l'accettazione allora accetto.

**L non e' ricorsivo:** Mi chiedo se ci sia un linguaggio che so non essere ricorsivo che si riduce a questo (cioè ipotizzando per assurdo che ci sia un algoritmo per questo allora otterrei un algoritmo anche per il linguaggio noto).

In questo caso basta considerare  $L_u = \{ w_{\langle x,z \rangle} \mid w_z \text{ appartiene a } L(M_x) \}$

La riduzione da  $L_u$  ad  $L$  è la seguente.

Per stabilire se  $w_{\langle x,z \rangle}$  sta in  $L_u$  pongo:

- $y =$  indice di una qualsiasi TM tale che  $L(M_y) = \{ w_z \}$

e poi uso un ipotetico algoritmo per stabilire se " $w_{\langle x,y \rangle}$ " sta in  $L$

Ho che:

$w_{\langle x,y \rangle}$  appartiene a  $L$  se e solo se  $L(M_x)$  e  $L(M_y) = \{ w_z \}$  hanno una stringa in comune, cioè  $w_z$  in  $L(M_x)$ .

Quindi ipotizzando di avere un algoritmo per  $L$  avrei un algoritmo per  $L_u$ : assurdo!

**L non e' ricorsivo (alternativa più semplice)**

Mi chiedo se ci sia un linguaggio che so non essere ricorsivo che si riduce a questo (cioè ipotizzando per assurdo che ci sia un algoritmo per questo allora otterrei un algoritmo anche per il linguaggio noto).

Considero il linguaggio

$$L' = \{w_x \mid L(M_x) \text{ diverso dal vuoto}\}$$

(per il teorema di Rice è non ricorsivo, basta prendere  $L_P$  con proprietà  $P = \text{"non essere vuoto"}$ )

La riduzione da  $L'$  ad  $L$  è la seguente.

Per stabilire se  $w_x$  sta in  $L'$  considero:

- $y =$  indice di una qualsiasi TM tale che  $L(M_y) =$  tutte le stringhe  
e poi uso un ipotetico algoritmo per stabilire se " $w_{\langle x,y \rangle}$ " sta in  $L$

Ho che:

$w_{\langle x,y \rangle}$  appartiene a  $L$  se e solo se  $L(M_x)$  e  $L(M_y)$  hanno una stringa in comune, cioè  $L(M_x)$  diverso dal vuoto.

Quindi ipotizzando di avere un algoritmo per  $L$  avrei un algoritmo per  $L'$ : assurdo!

## Esercizio

Si consideri il seguente linguaggio

$$L = \{ w_{\langle x,y \rangle} \mid x \text{ e } y \text{ sono tali che } L(M_x) \text{ e } L(M_y) \text{ non hanno almeno una stringa in comune} \}$$

dove  $M_x$  e  $M_y$  sono la  $x$ -esima e la  $y$ -esima macchina di Turing.

Dire se  $L$  è ricorsivo, ricorsivamente enumerabile o nemmeno ricorsivamente enumerabile.

Giustificare la risposta.

## Soluzione

Osservazione preliminare:

$L$  non è Ricorsivo poichè abbiamo già dimostrato (esercizio precedente) che il complementare di  $L$  non è Ricorsivo (per assurdo se  $L$  fosse ricorsivo allora, siccome i ricorsivi sono chiusi rispetto alla complementazione, dovrebbe esserlo anche il suo complementare).

### **$L$ non e' Ricorsivamente Enumerabile**

Per il teorema di Post se riusciamo a mostrare che:

- 1) il complementare di  $L$  è RE
- 2)  $L$  non è ricorsivo

allora possiamo concludere che  $L$  non è RE (per assurdo se  $L$  fosse RE allora, siccome il suo complementare è RE,  $L$  dovrebbe essere ricorsivo).

Dimostrazione di 1)

Il complementare di  $L$  è il linguaggio dell'esercizio precedente che abbiamo già dimostrato essere Ricorsivamente Enumerabile.

### **$L$ non e' Ricorsivamente Enumerabile (alternativa: dimostrazione diretta)**

Mi chiedo se ci sia un linguaggio che so essere non ricorsivamente enumerabile che si riduce a  $L$ : cioè ipotizzando per assurdo che ci sia una procedura (TM) che riconosce  $L$  allora otterrei una procedura (TM) che riconosce il linguaggio noto.

In questo caso basta considerare  $L_d = \{w_x \mid w_x \text{ non appartiene a } L(M_x)\}$

La riduzione da  $L_d$  ad  $L$  è la seguente.

Per stabilire se  $w_x$  sta in  $L_d$  pongo:

- $y =$  indice di una qualsiasi TM tale che  $L(M_y) = \{w_x\}$

e poi uso una ipotetica procedura (TM) che riconosce  $L$  dandogli in input  $w_{\langle x,y \rangle}$  (qualora stia in  $L$  la procedura termina accettando altrimenti la procedura può non terminare o bloccarsi)

Ho che:

$w_{\langle x,y \rangle}$  appartiene a  $L$  se e solo se  $L(M_x)$  e  $L(M_y) = \{w_x\}$  non hanno stringhe in comune, cioè  $w_x$  non appartiene a  $L(M_x)$ .

Quindi ipotizzando di avere una procedura (TM) che riconosce  $L$  avrei una procedura (TM) che riconosce  $L_d$ : assurdo!

### Altro esercizio (tipico da esame)

Si consideri il linguaggio delle stringhe  $a^n b^n c^n$ .

Classificare il linguaggio dicendo se è un linguaggio regolare, libero, ricorsivo, ricorsivamente enumerabile o nemmeno ricorsivamente enumerabile. Giustificare la risposta.

E' ricorsivo:

- 1) bisogna dimostrare (con il pumping lemma) che non è libero
- 2) mostrare che esiste una TM che può individuare la stringa:
  - a) considero input  $w$  nel primo nastro
  - b) cerco le  $a$  all'inizio della stringa  $w$  e le copio su un secondo nastro
  - c) proseguo cercando le  $b$  in  $w$  e le copio su un terzo nastro
    - i) se incontro  $a$  fallisco
  - d) proseguo cercando le  $c$  in  $w$  e le copio su un quarto nastro
    - i) se incontro  $a$  o  $b$  fallisco
  - e) controllo che le stringhe sul secondo e terzo nastro abbiano lunghezza uguale
  - f) controllo che le stringhe sul terzo e quarto nastro abbiano lunghezza uguale.