

Linguaggi di programmazione

Appunti

Giovanni Palma e Alex Basta

Contents

Chapter	Topic	Page
Chapter 1	Macchine Astratte	
1.1	Macchine Astratte e Linguaggi Macchine Astratte Imperative — • Linguaggio Macchina —	
1.2	Gerarchie di Macchine Astratte e Linguaggi Macchina Hardware — • Macchina Microprogrammata — • Macchina Software —	
1.3	Implementare un linguaggio: Interpreti e Compilatori Definizione di programma — • Implementazione interpretativa pura — • Implementazione compilativa pura — • Differenze — • Generazione Compilatori —	
Chapter 2	Introduzione ai linguaggi normali: grammatiche	
2.1	Linguaggi (naturali o artificiali) Sintassi — • Semantica — • Pragmatica — • Implementazione —	
2.2	Lessico e frasi di un linguaggio	
2.3	Notazioni e definizioni ausiliarie Lunghezza — • Concatenazione — • Sottostringa — • Suffisso — • Prefisso — • Potenza n-esima — • Linguaggio —	
2.4	Operazione sui linguaggi Complemento — • Unione e intersezione — • Concatenazione — • Potenza di un linguaggio — • Stella di kleene —	
2.5	Definire finitamente un linguaggio esempio 1: frasi palindrome — • Esempio 2 —	
2.6	Grammatiche	
2.7	Derivazioni	
2.8	Linguaggio Generato Algoritmo di Naif —	
2.9	Alberi di derivazione Ambiguità — • Rimuovere l'ambiguità —	
Chapter 3	Struttura di un compilatore, semantica statica, semantica dinamica	
3.1	Vincoli contestuali	
3.2	Semantica statica	
3.3	semantica dinamica Utilità della semantica dinamica — • definire la semantica —	
3.4	Pragmatica nella descrizione di un linguaggio	
3.5	implementazione Correttezza dell'implementazione — • Struttura di un compilatore —	

3.6	fasi principali della compilazione analisi lessicale (scanner) — • analisi sintattica (parser) — • Analisi semantica — • Generazione della forma intermedia — • Ottimizzazione — • Generazione del codice — • Tabella dei simboli —
3.7	semantica operazionale strutturata Definizione di un linguaggio a cui dare semantica —
3.8	Dare semantica ad un linguaggio Semantica delle espressioni aritmetiche — • Semantica delle espressioni booleane —

Chapter 4 Analisi lessicale: espressioni regolari, DFA, NFA _____ Page _____

4.1	analisi lessicale token — • espressioni regolari — • equivalenza tra espressioni regolari —
4.2	Automi a stati finiti diagrammi di transizione — • Automi a stati finiti non deterministico — • Linguaggio riconosciuto/accettato — • Automi a stati finiti deterministici — • Da espressioni regolari a NFA equivalenti — • Da NFA a espressione regolare — • Chiusura dei linguaggi regolari rispetto alle operazioni regolari —
4.3	Grammatiche regolari da grammatiche regolari a NFA equivalenti — • Da DFA a grammatiche regolari — • Grammatiche regolari ed espressioni regolari — • Per riassumere le relazioni tra espressioni regolari, linguaggi regolari e automi NFA e DFA — • minimizzazione — • automa minimo —
4.4	Espressività dei linguaggi regolari e pumping lemma Pumping lemma —

Chapter 5 Linguaggi liberi nondeterministici _____ Page _____

5.1	Grammatiche libere Semplificazione e forme normali —
5.2	PDA Definizione formale — • Transizioni — • Linguaggio Accettato —
5.3	Equivalenza fra grammatiche libere e PDA Da grammatica libera a PDA — • Da PDA a grammatica libera — • Relazione fra linguaggi regolari e linguaggi liberi —
5.4	Proprietà Chiusura — • Pumping Theorem —

Chapter 6 linguaggi liberi deterministici _____ Page _____

6.1	PDA e linguaggi deterministici PDA deterministici — • Definizione di linguaggi liberi deterministici — • Analizzatori sintattici: parser —
6.2	Semplificazione delle grammatiche Eliminare le produzioni ϵ — • Eliminazione delle produzioni unitarie — • Rimuovere i simboli inutili — • mettere insieme le cose — • forme normali — • Eliminare la ricorsione a sinistra — • Ricorsione sx non-immediata — • Fattorizzazione a sinistra —

Chapter 7 Parser Top-Down _____ Page _____

7.1	Parser a discesa ricorsiva
7.2	Parser predittivo First — • Follow — • Parser per linguaggi $LL(1)$ — • Parser per linguaggi $LL(K)$ —

Chapter 8	Bottom up parser	Page
8.1	Parser shift-reduce nondeterministico	
8.2	Parser LR Funzionamento del parsing LR — • DFA a prefissi variabili — • Item LR(0) — • Costruire l'NFA dei prefissi variabili —	
8.3	Automa canonico $LR(0)$ Costruzione diretta dell'automa canonico $LR(0)$ —	
8.4	tabella di parsing LR Riempire la tabella di parsing —	
8.5	Algoritmo del parser	
8.6	SLR(1), LR(1), LALR(1) Tabella di parsing SLR(1) — • Parser LR(1) — • Parser LALR(1) —	
8.7	Grammatiche $LR(k)$ Grammatiche $SLR(k)$ — • Grammatiche $LALR(k)$ — • relazione tra le varie grammatiche —	
8.8	Linguaggi generati dalle varie grammatiche	
Chapter 9	Nomi e Ambiente	Page
9.1	Nomi e Oggetti denotabili	
9.2	Ambienti e Blocchi Blocchi — • Tipi di Ambiente — • Operazioni sull'ambiente — • Vita di un oggetto —	
9.3	Regole di scope Scope statico — • Scope dinamico —	
9.4	determinare l'ambiente	
Chapter 10	Gestione Memoria	Page
10.1	Allocazione Statica	
10.2	Allocazione Dinamica Allocazione Dinamica con Pila — • Allocazione Dinamica con Heap —	
10.3	Implementazione delle Regole di Scope Scope Statico —	
Chapter 11	Strutturare il controllo - espressioni, comandi, iterazione, ricorsione	Page
11.1	Espressioni sintassi delle espressioni — • Semantica delle espressioni —	
Chapter 12	Astrazione sul controllo: sottoprogrammi ed eccezioni	Page
12.1	parametri Modalità di passaggio dei parametri —	
12.2	Funzioni di ordine superiore Funzione come argomento —	
Chapter 13	Esercitazioni	Page
13.1	Scope Esercizio 1 — • Esercizio 2 — • Esercizio 3 — • Esercizio 4 — • Esercizio 5 — • Esercizio 6 — • Esercizio 7 —	

13.2 sottoprogrammi

Es. 1 — • Es. 2 — • Es. 3 — • Es. 4 — • Es. 5 —

Chapter 1

Macchine Astratte

1.1 Macchine Astratte e Linguaggi

Per analizzare la computabilita' e la complessita' di algoritmi e per implementare nuovi linguaggi di programmazione, risulta necessario avere un modello teorico (realizzato in hardware o software) in grado di eseguire operazioni e memorizzare gli output, che chiamiamo **macchina astratta**.

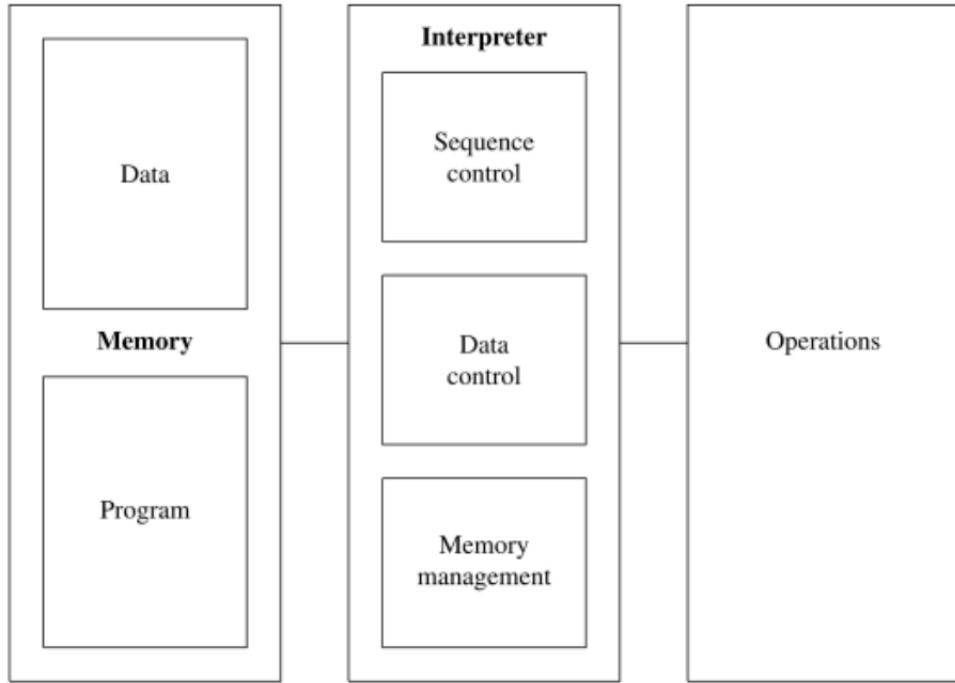
Dare una definizione dettagliata e allo stesso tempo valida per tutti i tipi di macchina astratta e' pressoché impossibile, dato che esistono molti modelli di computazione che differiscono tra loro, come per esempio il modello logico, funzionale, orientato agli oggetti e altri. Dato che, per motivi puramente tecnologici, la maggior parte delle macchine reali si basano sull'**architettura di von Neumann**, noi ci soffermeremo sul modello di computazione *imperativo* (le cui caratteristiche essenziali sono strettamente legate a tale architettura).

1.1.1 Macchine Astratte Imperative

Definition 1.1.1: Macchina Astratta (imperativa)

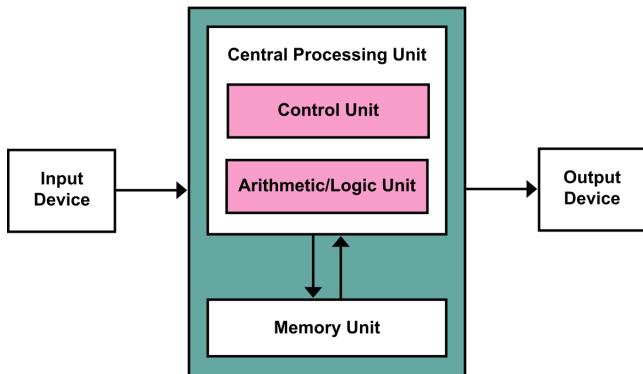
Una **macchina astratta (imperativa)** e' un insieme di **strutture dati** e **algoritmi** in grado di memorizzare ed eseguire programmi. Le componenti principali sono:

- un **interprete**
- una **memoria**, contenente sia dati che il programma
- un insieme di operazioni primitive
- un controllo di sequenza, che ha il compito di acquisire l'istruzione corretta seguendo il flusso dell'algoritmo
- un'unita' per il controllo e il trasferimento dei dati in memoria



Macchina di von Neumann

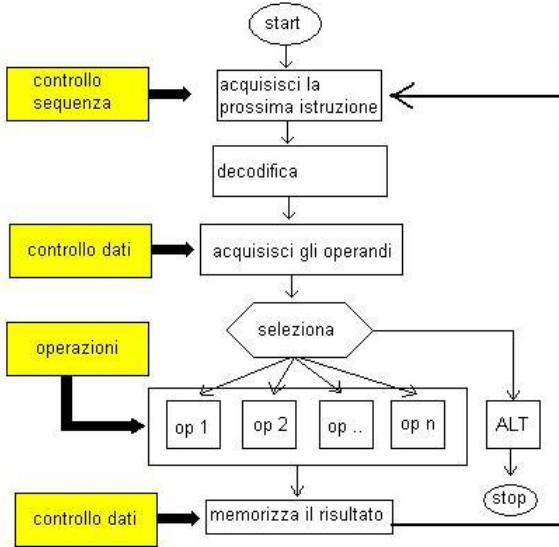
Come accennato prima, l'architettura di von Neumann e' un modello semplice ma estremamente flessibile per la realizzazione di macchine.



E' un modello ben adatto alla programmazione imperativa, ed e' spesso usata come base concettuale per le macchine astratte imperative. Cio' e' dovuto al concetto di "istruzione" (intrinseca al modello di von Neumann), ovvero operazioni che vengono eseguite in modo sequenziale dall' **interprete**.

Interprete

L'**interprete** e' la componente fondamentale che da' alla macchina la capacita' di eseguire programmi. E' un semplice algoritmo, anche chiamato ciclo *fetch-decode-execute*, che permette la corretta esecuzione delle istruzioni, coordinando le componenti della macchina astratta finche' non viene eseguita l'istruzione primitiva di arresto, che provoca l'uscita dal ciclo e l'arresto della macchina.



Come si vede dall'immagine, l'interprete e' costituito da operazioni e strutture dati che servono per: (implementazione nelle macchine fisiche)

- Elaborare dati primitivi (ALU)
- Controllo sequenza di esecuzione (PC e salti)
- Controllo dati (Metodi di indirizzamento e trasferimento di blocchi ecc.)

La struttura dell'interprete **rimane invariata per qualsiasi macchina astratta imperativa**, cambiano le altre componenti.

1.1.2 Linguaggio Macchina

Data una macchina astratta M , chiamiamo L_M il **Linguaggio Macchina di M**, ovvero l'insieme delle istruzioni che la macchina M e' in grado di "comprendere". Per chi programma, il linguaggio che si usa per creare programmi eseguibili da una macchina astratta e' semplicemente una sequenza di caratteri. Mentre all'interno della macchina stessa il programma viene trasformato da algoritmi, che possono sfruttare diverse strutture dati, per diventare un linguaggio comprensibile da una macchina astratta di livello piu' basso, o direttamente dalla macchina fisica (se M e' realizzata da hardware).

1.2 Gerarchie di Macchine Astratte e Linguaggi

Proviamo a costruire una macchina astratta $M_{\mathcal{L}}$ che e' in grado di comprendere ed eseguire istruzioni complesse di un linguaggio \mathcal{L} di alto livello, partendo da zero. Prima di tutto, vediamo in generale come si forma la gerarchia di macchine astratte:

1.2.1 Macchina Hardware

Per prima cosa dobbiamo per forza partire dal **livello fisico**, dato che ci serve dell'hardware per eseguire il nostro programma automaticamente. Rimanendo sulla strada di macchine imperative, possiamo utilizzare l'architettura di von Neumann per progettare il sistema. Dobbiamo pero' pensare al linguaggio che l'interprete fisico deve riuscire ad eseguire: teoricamente, sarebbe possibile architettare direttamente un interprete fisico che riesca a comprendere il nostro linguaggio di alto livello, ma questo sarebbe davvero molto costoso e' difficile da implementare, quindi evitiamo. Scegliamo allora una serie di istruzioni basiche, del tipo:

- **Operazioni primitive:** operazioni aritmetico logiche, manipolazione bit, I/O

- **Controllo sequenza:** salti (condizionali), chiamate di subroutine e strutture dati per punti di ritorno di sottoprogrammi
- **Controllo dati:** acquisizione di operandi e memorizzazione dei risultati
 - Architettura a registri → registri indice e indirizzamento indiretto
 - Architettura a pila → gestione pila
- **Gestione memoria:**
 - Architettura a registri → nessuna (memoria statica)
 - Architettura a pila → allocazione e recupero dati sulla pila

A seconda del livello di complessità delle singole istruzioni, possiamo dire che la nostra macchina fisica sia:

- **RISC** (Reduced Instruction Set Computers): istruzioni semplici e solitamente poche
- **CISC** (Complex Instruction Set Computers): istruzioni complesse e solitamente tante

L'insieme di queste istruzioni si chiama **linguaggio macchina**.

1.2.2 Macchina Microprogrammata

A volte, nelle macchine hardware di tipo CISC, è più conveniente creare col **firmware** una macchina astratta che sta sopra il livello hardware e che interpreti le istruzioni macchina. La **macchina microprogrammata** traduce queste istruzioni in istruzioni più semplici (μ istruzioni) che può eseguire l'interprete della macchina hardware (μ interprete).

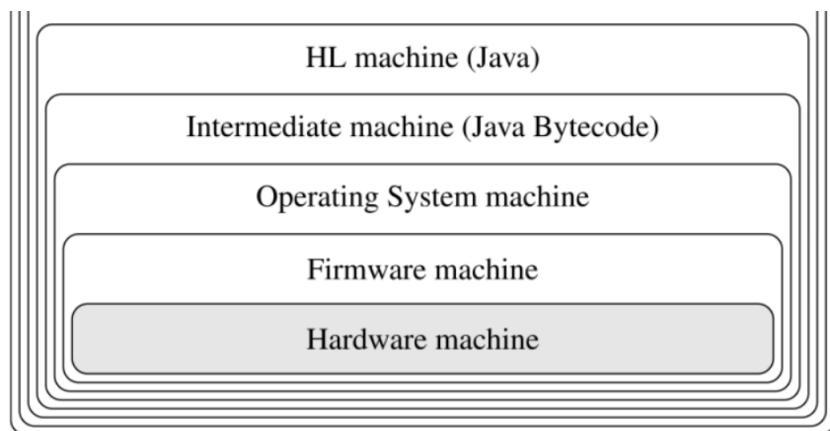
Notare che la macchina microprogrammata deve essere per forza scritta in un linguaggio compreso dal livello inferiore.

1.2.3 Macchina Software

A questo punto possiamo scrivere una macchina astratta utilizzando il linguaggio fornito dalla macchina sottostante. Facciamo ciò per implementare algoritmi e strutture dati più complesse, che possono servire anche ai livelli superiori.

Soltanente, dopo il livello hardware (o firmware se c'è), si trova il livello del Sistema Operativo. La macchina astratta di questo livello è scritta in linguaggio macchina ed estende le capacità del livello sottostante implementando primitive di livello più alto non presenti al livello hardware (come la gestione file). Tale macchina è conosciuta come la **host machine** (macchina ospite o MO).

Partendo dalla macchina ospite possiamo implementare finalmente il nostro linguaggio di alto livello, creando direttamente una macchina astratta che lo riconosce (che sia eseguibile dal Sistema Operativo) o utilizzando una macchina intermedia, possiamo anche tradurre il linguaggio in uno di livello più basso usando un compilatore.



1.3 Implementare un linguaggio: Interpreti e Compilatori

Ora che abbiamo capito come funziona la gerarchia di macchine astratte che ci permette di creare linguaggi di alto livello, definiamo in modo formale il processo:

1.3.1 Definizione di programma

Vogliamo implementare un linguaggio \mathcal{L} (cioè realizzare una macchina astratta $M_{\mathcal{L}}$) partendo da una macchina ospite $M_{\mathcal{L}_O}$. Prima di tutto, definiamo cosa sia un programma scritto in un linguaggio:

Definition 1.3.1: Programma

$\mathcal{P}_r^{\mathcal{L}}$ indica un programma \mathcal{P}_r scritto in un linguaggio \mathcal{L} .
A $\mathcal{P}_r^{\mathcal{L}}$ è associata una funzione parziale $\mathcal{P}^{\mathcal{L}}$:

$$\mathcal{P}^{\mathcal{L}} : D \multimap D$$

Dove D è l'insieme di dati (se $\mathcal{P}^{\mathcal{L}}$ non è definita per un certo $d \in D$, allora il programma non termina per quell'input). Tale funzione descrive la **semantica** del programma $\mathcal{P}_r^{\mathcal{L}}$.

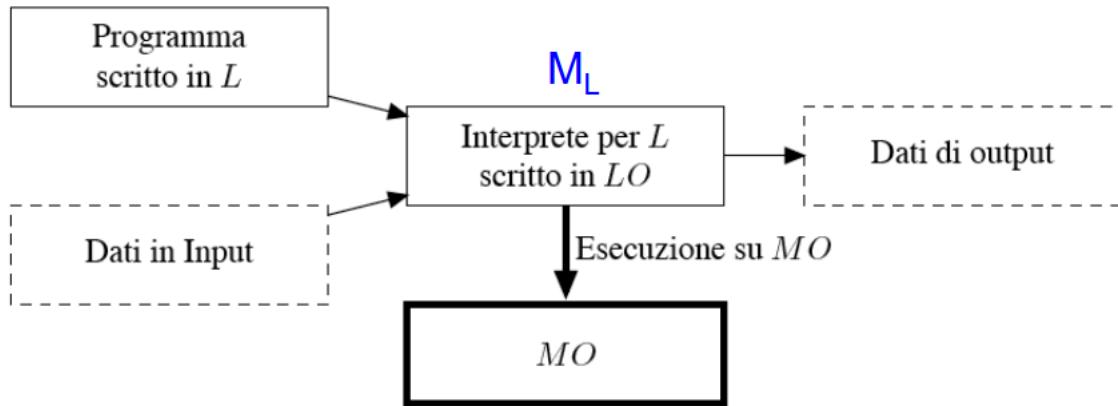
Note:

Per programmi concorrenti la questione è più delicata.

Esistono due modi radicalmente diversi per implementare un programma:

1.3.2 Implementazione interpretativa pura

Come abbiamo visto all'inizio del capitolo (1.1.1), l'interprete è la parte fondamentale delle macchine astratte. Quindi, se costruiamo un interprete per un linguaggio \mathcal{L} usando software scritto con il linguaggio \mathcal{L}_O della macchina ospite, abbiamo essenzialmente creato una nuova macchina astratta $M_{\mathcal{L}}$ sopra a M_O .



Diamo ora una definizione più formale di interprete come funzione (senza interessarci degli aspetti implementativi):

Definition 1.3.2: Interprete

Un interprete nel linguaggio \mathcal{L} , scritto nel linguaggio \mathcal{L}_O , è un programma che realizza la funzione parziale:

$$I_{\mathcal{L}}^{\mathcal{L}_O} : (\text{Prog}_{\mathcal{L}} \times D) \multimap D$$

tale che:

$$I_{\mathcal{L}}^{\mathcal{L}_O}(\mathcal{P}_r^{\mathcal{L}}, \text{Input}) = \mathcal{P}^{\mathcal{L}}(\text{Input})$$

Ovvero l'interprete calcola la corretta semantica del programma scritto nel nostro linguaggio \mathcal{L} .

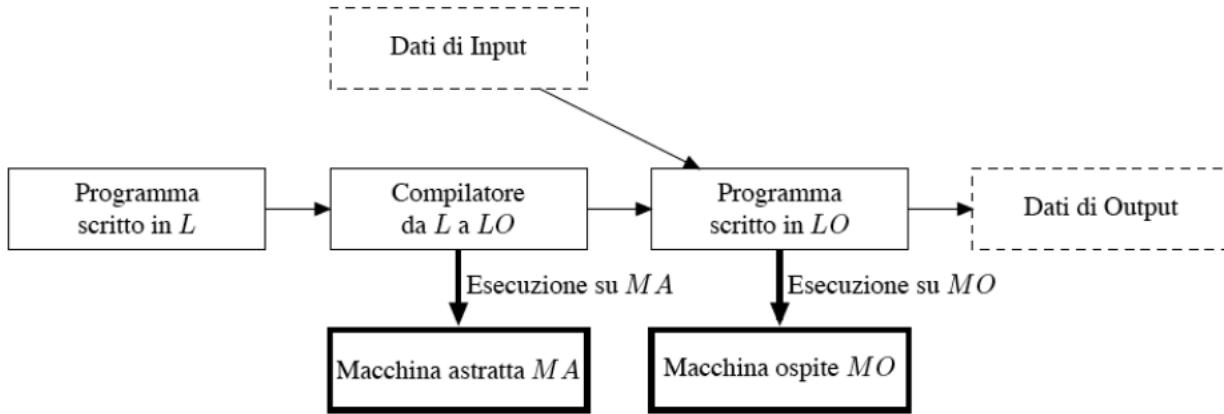


Figure 1.1: Nota: M_A puo' benissimo coincidere con M_O

1.3.3 Implementazione compilativa pura

Oltre all'implementazione interpretativa, esiste un altro modo per rendere un nuovo linguaggio eseguibile su una macchina ospite, che non ha bisogno di creare una nuova macchina astratta (e quindi un nuovo interprete). L'implementazione compilativa si basa sulla **traduzione** del linguaggio nuovo \mathcal{L} in un linguaggio riconosciuto da una macchina di livello piu' basso.

Per eseguire tale traduzione, viene usato un programma chiamato **compilatore**, che puo' essere scritto in un qualunque linguaggio riconosciuto dalle macchine astratte del calcolatore. Il compilatore, quindi, traduce programmi scritti nel linguaggio \mathcal{L} in nuovi programmi **equivalenti** scritti in \mathcal{L}_O :

Definition 1.3.3: Compilatore

Un compilatore da \mathcal{L} a \mathcal{L}_O scritto in un linguaggio \mathcal{L}_A e' un programma che realizza una funzione:

$$C_{\mathcal{L}, \mathcal{L}_O}^{\mathcal{L}_A} : \text{Prog}_{\mathcal{L}} \rightarrow \text{Prog}_{\mathcal{L}_O}$$

tale che, dato

$$C_{\mathcal{L}, \mathcal{L}_O}^{\mathcal{L}_A}(P_r^{\mathcal{L}}) = P_c_r^{\mathcal{L}_O}$$

allora, $\forall \text{Input} \in D$:

$$P^{\mathcal{L}}(\text{Input}) = P_c_r^{\mathcal{L}_O}(\text{Input})$$

Ovvero il compilatore **preserva la semantica** del programma: il programma originale e quello tradotto calcolano la stessa funzione: $P^{\mathcal{L}} = P_c^{\mathcal{L}_O}$.

1.3.4 Differenze

Vediamo i pro e i contro dei due metodi:

- **Interprete**

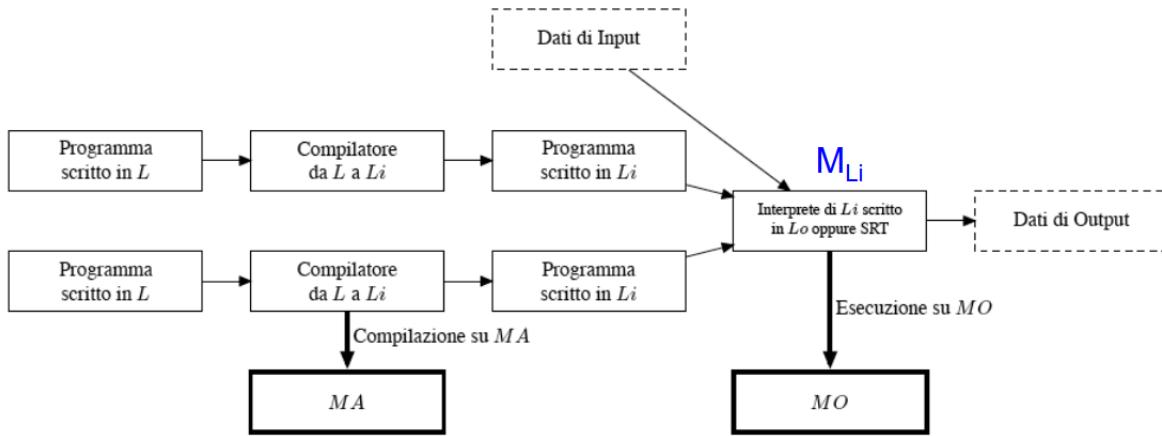
Pro	Contro
<ul style="list-style-type: none"> - Flessibile (debugging facile) - Più facile da realizzare - Occupa meno memoria 	<ul style="list-style-type: none"> - Scarsa efficienza (decodifica per ogni esecuzione)

- **Compilatore**

Pro	Contro
<ul style="list-style-type: none"> - Buona efficienza - Più facile da realizzare 	<ul style="list-style-type: none"> - Scarsa flessibilità - Perdita di info (astrazione) del programma sorgente (debugging difficile) - Occupa più memoria (non molto rilevante oggi)

Per riuscire a combinare l'efficienza dei compilatori con la portabilità dei linguaggi interpretati, nella realtà si utilizza spesso una via di mezzo. Solitamente:

- Alcune istruzioni (es. ingresso/uscita) sono sempre **simulate** (interpretate)
- I programmi vengono **tradotti** in un codice intermedio



Come si vede nell'immagine, il linguaggio L viene prima tradotto nel linguaggio intermedio L_i , che poi viene interpretato dall'interprete di M_{L_i} scritto in L_O . In base a quanto è diverso l'interprete L_i da L_O , abbiamo due implementazioni diverse:

- Interpretativa: l'interprete della macchina intermedia M_{L_i} è sostanzialmente diverso dall'interprete della macchina ospite M_{L_O} . (Es. Java)
- Compilativa: $M_{L_i} = M_{L_O} + \text{opportuni meccanismi, come I/O, gestione memoria, ecc.}$ (Es. C)

Cosa interpretare e cosa compilare

Abbiamo visto che le soluzioni di tipo compilative sono più efficienti, mentre quelle di tipo interpretativo privilegiano flessibilità e portabilità. In linea di principio:

- Traduzione per i costrutti di L che corrispondono da vicino a costrutti di L_O
- Simulazione per gli altri

1.3.5 Generazione Compilatori

Ora che abbiamo esplorato i modi per eseguire programmi scritti in un linguaggio nuovo usando compilatori e interpreti, vediamo come costruire questi ultimi strumenti.

Si possono sempre realizzare?

Immaginiamo un interprete e un compilatore scritti nello stesso linguaggio L_{osp} e che ricevono in input lo stesso linguaggio L_{sorg} . L'interprete è in grado di calcolare la semantica del programma in input usando L_{osp} e il compilatore preserva la semantica del programma tradotto, quindi tutte le funzioni che L_{sorg} può calcolare possono essere simulate o tradotte con L_{osp} . Quindi:

$$L_{osp} \text{ è non meno espressivo di } L_{sorg}$$

Dato che, come vedremo più avanti, tutti i linguaggi di programmazione "veri" **sequenziali** sono ugualmente espressivi (Turing-completi, se forniti di memoria "illimitata"), allora sappiamo che questa condizione vale sempre (se L_{osp} è un "vero" linguaggio sequenziale).

Note:

Esistono formalismi/linguaggi molto meno espressivi ma che sono utilissimi per la loro efficienza, come ad esempio:

- Automi finiti (analizzatori lessicali)
- Automi a pila (analizzatori sintattici)

Come viene generato un compilatore?

Esistono diversi modi per creare un compilatore/interprete dato un linguaggio L :

- **Strumenti automatici**

- **Lex**: generatore di analizzatori **lessicali**
- **Yacc**: generatore di analizzatori **sintattici**

Data una descrizione formale della sintassi di L , questi strumenti sono in grado di generare automaticamente un programma che riesce a riconoscere stringhe sintatticamente legali.

- **Implementazione via Kernel**

Si sceglie $H \subset L$, ovvero un sottoinsieme proprio di L che forma un nucleo (kernel) con il quale possiamo costruire:

- C_{L,L_O}^H , ovvero un compilatore scritto in H che traduce L in L_O , oppure
- I_L^H , ovvero un interprete per L scritto in H

Manca poi scrivere un interprete/compilatore per H , in modo da poter eseguire uno di questi due sulla macchina ospite, che se H e' stato scelto bene non dovrebbe essere troppo difficile.

Implementare prima un sottoinsieme di L porta a dei vantaggi:

- Semplifica l'implementazione, dato che H e' piu' vicina a L rispetto a L_O
- Facilita la portabilita, perche' basta re-implementare H nel nuovo linguaggio macchina

- **Bootstrapping**

Immaginiamo di aver creato i seguenti compilatori e interprete, dove M_i e' una macchina astratta intermedia:

- $C_{L,L_i}^{L_i}$
- $I_{L_i}^{L_O}$
- C_{L,L_O}^L

Otteniamo un compilatore per L scritto in L_O in questo modo:

$$I_{L_i}^{L_O}(C_{L,L_i}^{L_i}, C_{L,L_O}^L) = C_{L,L_O}^{L_i}$$

$$I_{L_i}^{L_O}(C_{L,L_O}^L, C_{L,L_O}^L) = C_{L,L_O}^{L_O}$$

Piu' efficiente rispetto al kernel, dato che abbiamo un compilatore che compila direttamente al linguaggio di M_O senza usare un linguaggio intermedio.

Chapter 2

Introduzione ai linguaggi normali: grammatiche

2.1 Linguaggi (naturali o artificiali)

La descrizione di un linguaggio avviene su 3 dimensioni:

- **Sintassi:** regole di formazione, ovvero la relazione tra segni
- **Semantica:** attribuzione di significato
- **Pragmatica:** in quale modo frasi corrette e sensate sono usate
- **Implementazione:** come eseguire una frase corretta rispettandone la semantica

Partiamo con il descrivere questi elementi

2.1.1 Sintassi

Definition 2.1.1: sintassi

La **sintassi** è la parte della grammatica che studia la struttura delle frasi e il modo in cui le parole si combinano per formare enunciati corretti e significativi

Si dirama in diversi aspetti:

- **Aspetto lessicale** che riguarda le parole che si possono usare, quindi:
 - descrizione del lessico, intuitivamente i dizionari per i linguaggi naturali assolvono a questo compito
 - errori dovuti a vocaboli inesistenti
- **Aspetto grammaticale** che si riferisce nel modo in cui è possibile costruire frasi corrette è possibile costruire con il lessico. Le frasi grammaticalmente corrette possono essere costruite grazie all'uso delle regole grammaticali, che sono in numero finito, mentre il numero di frasi generabili è infinito.

Un errore grammaticale è una frase scorretta, anche se il lessico utilizzato è corretto. Es. "La cane abbaiano"

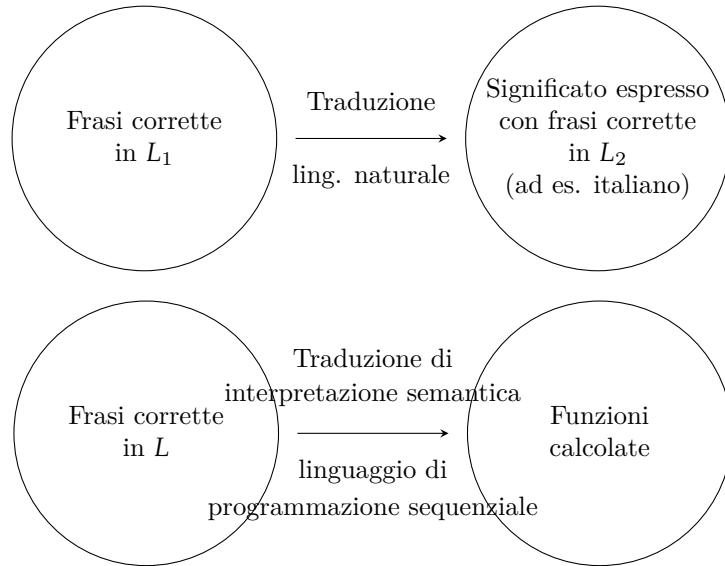
2.1.2 Semantica

Definition 2.1.2: Semantica

La **semantica** è la branca della linguistica che studia il significato delle parole, delle frasi e degli enunciati

- Per il **lessico** (quindi lo studio e il significato delle parole) bastano i **dizionari**
- Per le **frasi** è più complicato, devo sapere infatti:

1. a quale linguaggio appartiene la frase
2. su quale linguaggio basarmi per dare significato



2.1.3 Pragmatica

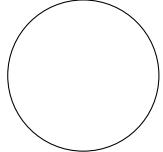
Definition 2.1.3: Pragmatica

La **pragmatica** è un insieme di regole che guidano l'uso e come i contesti influiscono sull'interpretazione di frasi sensate e corrette

Ad esempio quando e a chi dare del "tu" o del "lei" quando ci rivolgiamo a delle persone

2.1.4 Implementazione

L'implementazione è l'esecuzione di una frase sintatticamente corretta rispettandone la semantica



La semantica di P , è la funzione f che è pure la semantica del programma compilato Q , quindi l'implementazione Q di P preseva la semantica di P !

2.2 Lessico e frasi di un linguaggio

Innanzitutto diamo tre definizioni

Definition 2.2.1: alfabeto

Un **alfabeto** è un insieme (tipicamente) finito i cui elementi sono detti simboli

Definire l'alfabeto ci porta alla definizione di lessico:

Definition 2.2.2: Lessico

Il **lessico** è un insieme di sequenze finite costituite con caratteri o simboli dell'alfabeto

Il quale ci porta alla fenizione di frase:

Definition 2.2.3: frase

Una **frase** è un insieme sequenze finite contruite con parole del lessico.

È, quindi, facile notare che il lessico è un alfabeto per le frasi

Si ci si può ora astrarre e definire un linguaggio formale:

Definition 2.2.4: linguaggio formale

Un **linguaggio formale** L su alfabeto A è un sottoinsieme di A^* ($L \subseteq A^*$), dove:

$$A^* = \bigcup_{n \geq 0} A^n \quad \text{dove } A^0 = \{\epsilon\}$$

e

$$A^{n+1} = A \cdot A^n \quad n \geq 0$$

$$\text{con } A \cdot A^n = \{aw \mid a \in A \wedge w \in A^n\}$$

Si osservi che A^* è un insieme infinito contabile dato un ordinamento $<$ sui simboli di A , possiamo elencare tutte le parole come segue:

1. elenco la parola vuota ϵ (A^0)
2. poi elenco le parole di lunghezza 1 (A^1) secondo l'ordinamento $<$
Es. a, b, c, \dots
3. poi elenco le parole in A^2 secondo $<$
Es. $aa, ab, ac, \dots, ba, bb, bc, \dots$
4. così via

Anche se l'alfabeto A fosse infinito (quindi $A = \{a_0, a_1, \dots\}$) A^* sarebbe ancora contabile, ovvero esisterebbe la possibilità di elencare tutte le possibili parole in A^* . Infatti, esiste una biezione tra A e \mathbb{N} e riguardo ai numeri naturali si sa che:

- $\mathbb{N} \times \mathbb{N}$ (prodotto cartesiano) è numerabile. La dimostrazione viene fatta attraverso il *dove-tailing*, una tecnica comune per dimostrare la numerabilità di coppie di numeri naturali. Questa dimostrazione introduce la cosiddetta **funzione di decodifica**

$$f^2(x_1, x_2) = \frac{(x_1 + x_2)(x_1 + x_2 + 1)}{2} + x_2$$

che presi due numeri in \mathbb{N} completa la seguente tabella ordinando i numeri naturali in "diagonale"

x_1	0	1	2	3	4	5	-
x_2	0	1	3	6	10	15	
	1	2	4	7	11	16	
	2	5	8	12	17		
	3	9	13	18			
	4	14	19				
	5	20					

è pertanto vero, quindi, che

$$f^2 : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

è biunivoca

- \mathbb{N}^k è numerabile. Infatti si può dimostrare attraverso questo algoritmo:

$$f^k(x_1, x_2, \dots, x_k) = \begin{cases} \text{if } (k = 2) \text{ then } f^2(x_1, x_2) & \text{else } f^2(x_1, f^{k-1}(x_2, \dots, x_k)) \end{cases}$$

Ovvero riduce una funzione con k variabili nel dominio ad una serie di funzioni matrioska per ricondurla alla forma $f^2 : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

- $\mathbb{N}^* = \bigcup_{k \geq 0} \mathbb{N}^k$ è numerabile. Infatti:

$$f(x_1, \dots, x_k) = f^2(k, f^k(x_1, \dots, x_k))$$

2.3 Notazioni e definizioni ausiliarie

Vengono riportate qui alcune definizioni/notazioni

2.3.1 Lunghezza

Definition 2.3.1: lunghezza

La **lunghezza** di una parola o stringa è definita per induzione così:

- Caso $|\epsilon|: 0$
- Caso $|aw|: 1 + |w|$

Es: $|abc| = 2$

2.3.2 Concatenazione

Definition 2.3.2: Concatenazione

La **concatenazione** xy tra una stringa x e y , è la parola ottenuta giustapponendo x e y . Formalmente:

$$w = xy \iff \begin{cases} |w| = |x| + |y| \\ w(j) = x(j) & \text{per } 1 \leq j \leq |x| \\ w(|x| + j) = x(j) & \text{per } 1 \leq j \leq |y| \end{cases}$$

Dove $w(j)$ indica il j -esimo simbolo di w

Questi sono le leggi della concatenazione

- **Associatività:** $x(yz) = (xy)z$
- **Elemento neutro (ϵ):** $x\epsilon = x = \epsilon x$

2.3.3 Sottostringa

Definition 2.3.3: Sottostringa

La stringa v si dice **sottostringa** di $w \iff \exists x, y \in A^*$ t.c. $w = xvy$ dove x e y possono essere ϵ

Si osservi, quindi, che:

- Ogni stringa è sottostringa di se stessa
- ϵ è sottostringa di ogni stringa

2.3.4 Suffisso

Definition 2.3.4: suffisso

v si dice **suffisso** di $w \iff \exists x \in A^x. w = xv$

2.3.5 Prefisso

Definition 2.3.5: prefisso

v si dice **prefisso** di $w \iff \exists x \in A^x. w = vx$

2.3.6 Potenza n-esima

Definition 2.3.6: potenza n-esima

Si dice **potenza n-esima** di una stringa w il valore $n \geq 0$ il cui significato è definito per induzione:

- Caso 0: $w^0 = \epsilon$
- Caso $n + 1$: $w^{n+1} = ww^n$

2.3.7 Linguaggio

Definition 2.3.7: linguaggio

Si dice **linguaggio L su alfabeto A** un sottoinsieme $L \subseteq A^*$

Vengono riportati qui alcuni esempi

Example 2.3.1

Se $A = \{a\}$, si possono avere:

- $\emptyset, \{\epsilon\}, \{a, aaa\}$ sono linguaggi finiti
- $L_1 = \{a^n | n \geq 1\} = \{a, aa, aaa, \dots\} = A^* \setminus \{\epsilon\}$
- $L_2 = \{a^{2n} | n \geq 0\} = \{\epsilon, aa, aaaa\}$

2.4 Operazione sui linguaggi

Qui sono elencati le varie operazioni

2.4.1 Complemento

Definition 2.4.1: complemento

È definito **complemento** il linguaggio completare ad un linguaggio L , ovvero:

$$\bar{L} = \{w \in A^* | w \notin L\} = A^* \setminus L$$

2.4.2 Unione e intersezione

Definition 2.4.2: unione e intersezione

Ovvi:

$$\begin{aligned} L_1 \cup L_2 &= \{w | w \in L_1 \vee w \in L_2\} \\ L_1 \cap L_2 &= \{w | w \in L_1 \wedge w \in L_2\} \end{aligned}$$

2.4.3 Concatenazione

Definition 2.4.3: concatenazione

È definita **concatenazione** tale operazione:

$$L_1 \cdot L_2 = \{w_1 w_2 | w_1 \in L_1 \wedge w_2 \in L_2\}$$

Ecco alcuni esempi:

Example 2.4.1

- $L_1 = \{a^n | n > 0\} \quad L_2 = \{b\}$
 $L_1 \cdot L_2 = \{a^n b | n > 0\}$
- $L_1 = \{a^{2n} | n > 0\} \quad L_2 = \{b^m | m > 0\}$
 $L_1 \cdot L_2 = \{a^{2n} b^m | n, m > 0\}$
- $L_1 = \{a^m b^n | n > 0\} \quad L_2 = \{b^m | n > 0\}$
 $L_1 \cdot L_2 = \{a^m b^{n+m} | n, m > 0\}$
 $= \{a^m b^n | m \geq n > 0\}$
- $L_1 = \{a^m | m > 1\} \quad L_2 = \{a^m b^m | n > 0\}$
 $L_1 \cdot L_2 = \{a^{m+n} b^m | m > 1, m > 0\}$
 $= \{a^m b^m | m > m > 0\}$

- $L_1 = \{ab^m \mid n \geq 1\}$ $L_2 = \{a, c\} \cup \{b^n \mid n \geq 1\}$
- $L_1 \cdot L_2 = \{ab^m a \mid m \geq 1\} \cup \{ab^m c \mid n \geq 1\}$
 $\cup \{ab^n \mid n \geq 2\}$
- $A = \{0, 1\}$
 $L_1 = \{w \in A^* \mid w \text{ contiene un numero pari di "0"}\}$
 $L_2 = \{w \in A^* \mid w = 0y \text{ e } y \in \{1^*\}\}$
 $L_1 \cdot L_2 = \{w \in A^* \mid w \text{ ha un numero dispari di "0"}\}$

2.4.4 Potenza di un linguaggio

Definition 2.4.4: Potenza di un linguaggio

La **potenza di un linguaggio** viene definita per induzione:

- Caso 0: $L^0 = \{\epsilon\}$
- Caso $n + 1$: $L \cdot L^n \quad \forall n \geq 0$

2.4.5 Stella di kleene

Definition 2.4.5: stella di kleene

Si dice **stella di kleene**:

$$L^* \bigcup_{n \geq 0} L^n$$

Oppure

$$L^+ = \bigcup_{n \geq 1} L^n$$

Quest'ultima detta **chiusura positiva**

2.5 Definire finitamente un linguaggio

2.5.1 esempio 1: frasi palindrome

Una frase palindroma è una parola che letta da sx a dx è uguale a se stessa letta da dx a sx
Es. "I topi non avevano nipoti"

- $A = \{a, b\}$ $L = \{\epsilon, a, b, aa, bb, aba, bab, \dots\}$. Come si nota è piuttosto scomodo
- Una palindroma può essere:
 - o è la stringa ϵ
 - oppure a
 - oppure b
 - oppure a "palindroma" a
 - oppure b "palindroma" b
- rappresentazione tramite **Backus-naur form (BNF)**.

$$\langle P \rangle := \epsilon \mid a \mid b \mid a \langle P \rangle a \mid b \langle P \rangle b$$

- Come **grammatica**:

$$P \rightarrow \epsilon \mid a \mid b \mid aPa \mid bPb$$

- definizione ricorsiva in cui:
 - P è detto *simbolo non terminale*
 - a, b sono "simboli terminali"

2.5.2 Esempio 2

espressioni aritmetiche formate a partire dalle variabili a e b con gli operatori $\times, +$ e le parentesi $(,)$
Una $expr$ può essere:

- – la variabile a
 - la variabile b
 - $expr \times expr$
 - $expr + expr$
 - $(expr)$
 - **bnf:**
- $$\langle E \rangle ::= a \mid b \mid \langle E \rangle \times \langle E \rangle \mid \langle E \rangle + \langle E \rangle \mid (\langle E \rangle)$$
- Grammatica:
- $$E \rightarrow a \mid b \mid E \times E \mid E + E \mid (E)$$

2.6 Grammatiche

Una grammatica è un insieme di regole che descrivono come le parole e le frasi possono essere combinate per formare espressioni valide. Queste regole determinano la struttura sintattica di un linguaggio, specificando come le unità di base (come le parole o i simboli) si connettono per formare frasi o espressioni più complesse. Ogni grammatica **segue lo stesso pattern definito** differenziandosi solo per come sono caratterizzate le produzioni.
Quelle più utili sono le cosiddette grammatiche libere dal contesto (in rapporto tra facilità di analisi ed espressività)

Definition 2.6.1: grammatiche libere (dal contesto)

Una **grammatica libera** da contesto è una quadrupla (NT, T, R, S) dove:

- NT è un insieme finito di simboli non terminali
- T è un insieme finito di simboli terminali
- $S \in NT$ è detto simbolo iniziale
- R è un insieme finito di produzione (o regole) della forma:

$$V \rightarrow w \text{ dove } V \in NT \wedge w \in (T \cup NT)^*$$

Alcuni esempi:

Example 2.6.1

$$G = (\{S\}, \{a, b, +, \times\}, S, R)$$

Con

$$R = \{S \rightarrow a, S \rightarrow b, S \rightarrow S + S, S \rightarrow S \times S\}$$

2.7 Derivazioni

Definition 2.7.1: derivazione immediata

Data $G = (NT, T, R, S)$ libera dal contesto, diciamo che da v si **deriva immediatamente** w , e lo denotiamo con $v \Rightarrow w$, se:

$$\frac{v = xAy \quad (A \rightarrow z) \in R \quad w = xzy}{v \Rightarrow w}$$

Definition 2.7.2: derivazione

Diciamo che da v si **deriva** w (o anche "v si riscrive in w"), e lo denotiamo con $v \Rightarrow^* w$, se esiste una sequenza finita (eventualmente vuota) di derivazione immediate

$$v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w$$

Cioè:

$$\frac{}{v \Rightarrow^* v} \quad \frac{v \Rightarrow^* w \quad w \Rightarrow z}{v \Rightarrow^* z}$$

Dove \Rightarrow^* è la chiusa riflessiva e transitiva della relazione \Rightarrow

2.8 Linguaggio Generato

Definition 2.8.1: Linguaggio Generato

Il **linguaggio generato** da una grammatica $< G = (NT, T, R, S) >$ è l'insieme

$$L(G) = \{w \in T^* | S \xrightarrow{*} w\}$$

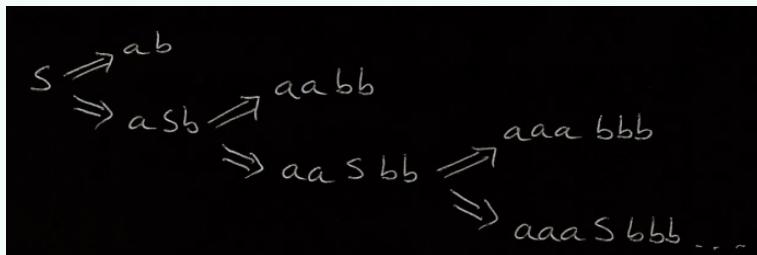
2.8.1 Algoritmo di Naif

Data una grammatica G è opportuno chiedersi come si fa a determinare un linguaggio $L(G)$ e a verificare se $w \in L(G)$. La domanda può essere complessa, tuttavia in casi semplici ci viene in aiuto l'**algoritmo di Naif** che consiste nel partire da S e provare ad applicare in tutti i modi possibili le produzioni (regole) per trovare una derivazione che genera w .

In certi casi questa verifica è semplice

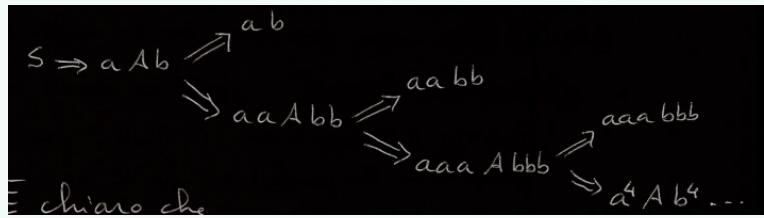
Example 2.8.1

- Sia G_3 con $S \rightarrow aSb|ab$



In questo esempio è facile determinare che $L(G_3) = \{a^n b^n | n \geq 1\}$

- Sia $G_1 \rightarrow aAb$ e $A \rightarrow aAb \mid \epsilon$



Quindi $L(G_1) = \{a^n b^n \mid n \geq 1\}$

Si può notare che G_1 e G_3 sono grammatiche equivalenti perché $L(G_1) = L(G_3)$
In generale esistono grammatiche diverse che generano lo stesso linguaggio

2.9 Alberi di derivazione

Per rappresentare graficamente e semplicemente una certa grammatica esiste uno strumento utilissimo, ovvero l'albero di derivazione

Definition 2.9.1: albero di derivazione

Data una grammatica libera $G = (NT, T, S, R)$, un albero di derivazione (o di parsing) è un albero ordinato in cui:

- Ogni nodo è etichettato con un simbolo in $NT \cup \{\epsilon\} \cup T$
- la radice è etichettata con S
- ogni nodo interno è etichettato con un simbolo in NT
- se il nodo n
 - ha etichetta $A \in NT$
 - i suoi figli sono nell'ordine m_1, \dots, m_k con etichetta x_1, \dots, x_k (in $NT \cup T$), allora

$$A \rightarrow x_1, \dots, x_k \text{ è una produzione in } R$$

- se il nodo n ha etichetta ϵ , allora n è una foglia, è figlio unico e, dato A suo padre, $A \rightarrow \epsilon$ è una produzione di R
- se inoltre ogni nodo foglia è etichettato su $T \cup \{\epsilon\}$ è una produzione di R
- se inoltre ogni nodo foglia è etichettato su $T \cup \{\epsilon\}$, allora l'alberello di derivazione corrisponde ad una derivazione completa

Un albero di derivazione, quindi, riassume tante derivazioni diverse ma tutte equivalenti (ovvero generano lo stesso albero)

Example 2.9.1

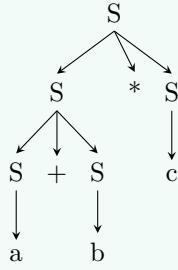
Consideriamo la grammatica

$$S \rightarrow a|b|c|S + S|S \times S$$

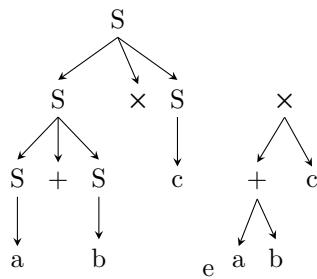
Inoltre si consideri la derivazione

$$S \Rightarrow \underline{S} + S \Rightarrow \underline{S} \times S + S \Rightarrow a \times \underline{S} + S \Rightarrow a \times b + \underline{S} \Rightarrow a \times b + c$$

Allora il suo albero di derivazione è:



Si osservi inoltre che l'albero di derivazione fornisce informazioni semantiche: "quali operandi per quali operatori" e possono essere riassunti nei cosiddetti **alberi sintattici**, tipo



Theorem 2.9.1

Una stringa $w \in T^*$ appartiene a $L(G)$ sse ammette un albero di derivazione completo (le cui foglie, lette da sx a dx diano la stringa w) cioè visita in ordine anticipato tralasciando i nonterminali

2.9.1 Ambiguità

Per generare una stringa w , una grammatica libera G puo' avere diversi *alberi di derivazione*, che quindi attribuiscono un significato diverso allo stesso input, vediamo un esempio:

Example 2.9.2

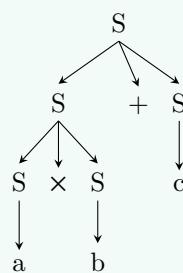
Si consideri la seguente grammatica

$$S \rightarrow a|b|c|S + S|S \times S$$

Si hanno due diverse derivazioni per $a \times b + c$:

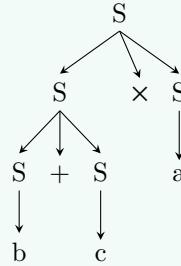
$$1. S \Rightarrow \underline{S} + S \Rightarrow \underline{S} \times S + S \Rightarrow a \times \underline{S} + S \Rightarrow a \times b + \underline{S} \Rightarrow a \times b + c$$

con l'albero:



$$2. S \Rightarrow \underline{S} \times S \Rightarrow a \times \underline{S} \Rightarrow a \times \underline{S} + s \Rightarrow a \times b + \underline{S} \Rightarrow a \times b + c$$

Con l'albero:



Nella grammatica dell'esempio 2.9.1, la stringa $a \times b + c$ ha più di un albero di derivazione. In questi casi si dice che la grammatica è quindi **inutilizzabile per dare semantica** a $a \times b + c$ e viene definita **ambigua**. Bisogna utilizzare grammatiche non ambigue, o manipolare grammatiche ambigue per disambiguarle.

Definition 2.9.2: Grammatica ambigua

Una grammatica libera G è **ambigua** se $\exists w \in L(G)$ che ammette più alberi di derivazione.

E di conseguenza:

Definition 2.9.3: Linguaggio ambiguo

Un linguaggio L è **ambiguo** se tutte le grammatiche G , tali che $L(G) = L$, sono ambigue

Attenzione! Abbiamo detto che una grammatica e' ambigua se, per lo stesso input, ha *alberi* di derivazione diversi, **NON** *derivazioni* diverse. Questa distinzione e' necessaria perche' due derivazioni possono differire semplicemente nell'ordine in cui sostituiscono i nonterminali, generando pero' lo stesso albero di derivazione e quindi attribuendo lo stesso significato all'input.

Derivazioni rightmost e leftmost

Per poter studiare l'ambiguità di una grammatica guardando le sue derivazioni, ci e' utile fissare un'unica strategia per decidere quale nonterminale sostituire ad ogni passaggio di derivazione. Una convenzione semplice e' quella di prendere il nonterminale piu' a sinistra (o a destra) della stringa intermedia di derivazione. Queste due strategie si chiamano derivazioni *leftmost* e *rightmost*:

Definition 2.9.4: Derivazione leftmost (e rightmost)

Sia $G = (NT, T, R, S)$ una grammatica libera. Diciamo che $u \Rightarrow_l v$ se $u = xXy, v = xzy$, dove $x \in T^*, y \in (T \cup NT)^*$ e $X \rightarrow z \in R$. Se, invece, $x \in (T \cup NT)^*$ e $y \in T^*$, allora $u \Rightarrow_r v$.

Quindi, scriviamo $u \Rightarrow_l v$ quando sostituiamo il nonterminale di u che si trova piu' a sinistra, e $u \Rightarrow_r v$ quando sostituiamo quello piu' a destra. Detto cio', una **derivazione leftmost** e' una derivazione $u_1 \Rightarrow_l^* u_n$ tale che:

$$u_1 \Rightarrow_l u_2 \Rightarrow_l \dots \Rightarrow_l u_n$$

Quindi ad ogni passo sostituiamo il nonterminale piu' a sx (in modo analogo per rightmost).

Si puo' dimostrare che, se per una stringa w esiste un'albero di derivazione per una grammatica G , allora esistono sicuramente derivazioni leftmost e rightmost di w . In piu', possiamo dimostrare che i seguenti numeri sono uguali:

- Il numero di derivazioni leftmost di w
- Il numero di derivazioni rightmost di w
- Il numero di alberi di derivazione distinti di w

Detto cio', possiamo utilizzare la definizione di grammatica ambigua per arrivare alla seguente proposizione:

Proposition 2.9.1 Derivazioni leftmost/rightmost e ambiguità'

Data una grammatica libera G , questa' e **ambigua** sse $\exists w \in L(G)$ tale che:

w ha piu' di una derivazione leftmost/rightmost

2.9.2 Rimuovere l'ambiguità

Alcune grammatiche possono essere manipolate di modo da

- rimuovere l'ambiguità
- generare lo stesso linguaggio

In altre, invece, ti tieni l'ambiguità (non è possibile rimuoverla (cazzo)). Questa CAZZATA viene spiegata con solo degli esempi:

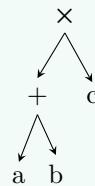
Example 2.9.3

Sia S la grammatica

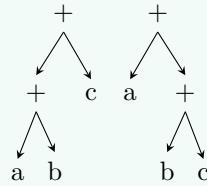
$$S \rightarrow a \mid b \mid c \mid S + S \mid S \times S$$

Problemi:

- Precedenza del $*$ rispetto al $+$ in modo che $a \times b + c$ sia interpretato come:



- associatività del $+$ e del \times



$a + b + c$ ovvero bisogna scegliere l'associatività a dx o sx

Un modo per eliminare questa ambiguità è definire la grammatica così:

$$e \rightarrow E + T \mid T \quad T \rightarrow A \times T \mid A \quad A \rightarrow a \mid b \mid c \mid (E)$$

Chapter 3

Struttura di un compilatore, semantica statica, semantica dinamica

3.1 Vincoli contestuali

Definition 3.1.1: vincoli sintattici contestuali

I vincoli sintattici contestuali sono termini o parole riservate non esprimibili per mezzo di grammatiche libere (perché non possono descrivere vincoli che dipendono dal contesto) che bisogna evitare di considerare quando si esegue il codice

Tradizionalmente i vincoli sintattici contestuali appartengono alla sintassi, ma nel gergo dei LP, si intende:

- **Sintassi**: quello che si **scrive** per mezzo di Grammatiche Libere
- **semantica** tutto il resto ...

Pertanto i **vincoli contestuali** sono dunque vincoli semantici, detti di **semantica statica** cioè vincoli che possono essere verificati ispezionando il codice **senza mandare il programma in esecuzione**

Il compilatore delega questi controlli di semantica statica alla cosiddetta **analisi semantica**

3.2 Semantica statica

Definition 3.2.1: Semantica statica

Per **semantica statica** si intende l'insieme di quei controlli che possono essere fatti sul testo del programma senza eseguirlo

Example 3.2.1

```
int A;  
bool B  
A := B (errore di tipo)
```

3.3 semantica dinamica

Per **semantica dinamica** si intende una rappresentazione formale dell'esecuzione del programma, la quale può mostrare errori durante l'esecuzione

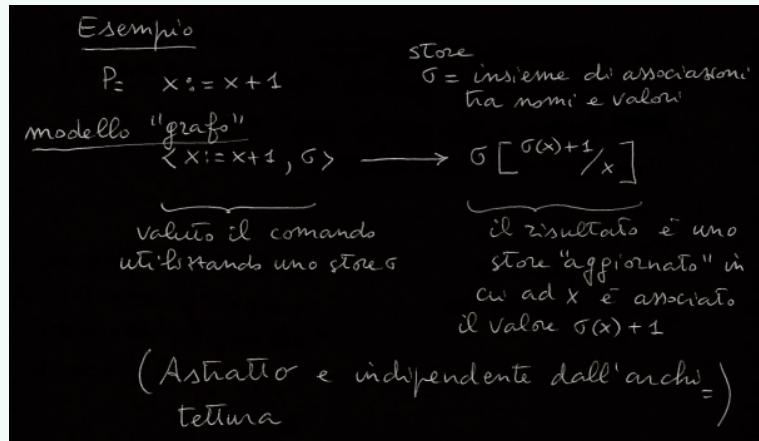
Example 3.3.1

`read(A);
B := $\frac{10}{A}$` (Se $A = 0$ si da un errore in esecuzione. F)

Staticamente non si può sapere l'errore perché la sua occorrenza **dipende dall'input dell'utente che fornirà durante l'esecuzione del programma**

Per implementare una semantica dinamica occorre fornire un modello matematico che descriva indipendentemente dall'architettura su cui il programma viene eseguito, il "comportamento del programma"

Example 3.3.2



3.3.1 Utilità della semantica dinamica

A chi serve la semantica dinamica?

- **Al programmatore:** *ANALISI DEL PROGRAMMA*
 - deve sapere esattamente cosa debba fare il suo programma
 - deve poter dimostrare proprietà del suo programma (ad es.: "termina sempre per ogni possibile input?")
- **Al progettista del linguaggio:**
 - strumento di specifica del linguaggio
 - deve poter dimostrare proprietà del linguaggio (ad es.: "è Turing-completo?")
- **All'implementatore del linguaggio:**
 - riferimento per dimostrare la correttezza dell'implementazione

Infatti **un compilatore è corretto quando preserva la semantica dinamica**, quindi per dimostrare che un compilatore è corretto serve avere una semantica per il linguaggio sorgente e per il linguaggio oggetto

3.3.2 definire la semantica

Per definire la semantica si utilizzano due tecniche principali:

- **operazionale:** (macchina astratta a stati e transizioni)
Ovvero si costruisce una specie di automa che, passo a passo, mostra l'effetto dell'esecuzione delle varie istruzioni. **vi è una maggiore enfasi su COME si calcola**
- **Denotazionale:** si associa ad ogni programma sequenziale una funzione da input ad output (incluse strutture ausiliarie e memoria). **vi è una maggiore enfasi su COSA si calcola**

3.4 Pragmatica nella descrizione di un linguaggio

Definition 3.4.1: Pragmatica nella descrizione di un linguaggio

si definisce **pragmatica nella descrizione di un linguaggio** insieme di regole sul modo in cui è meglio usare le istruzioni a disposizione

Esempio:

Example 3.4.1

- evitare le istruzioni di salto quando possibile
- usare le variabili di controllo del `for` solo a quello scopo
- scelta della modalità più appropriata di passaggio di parametri ad una funzione
- scelta tra iterazione determinata (`for`) e indeterminata (`while`)

3.5 implementazione

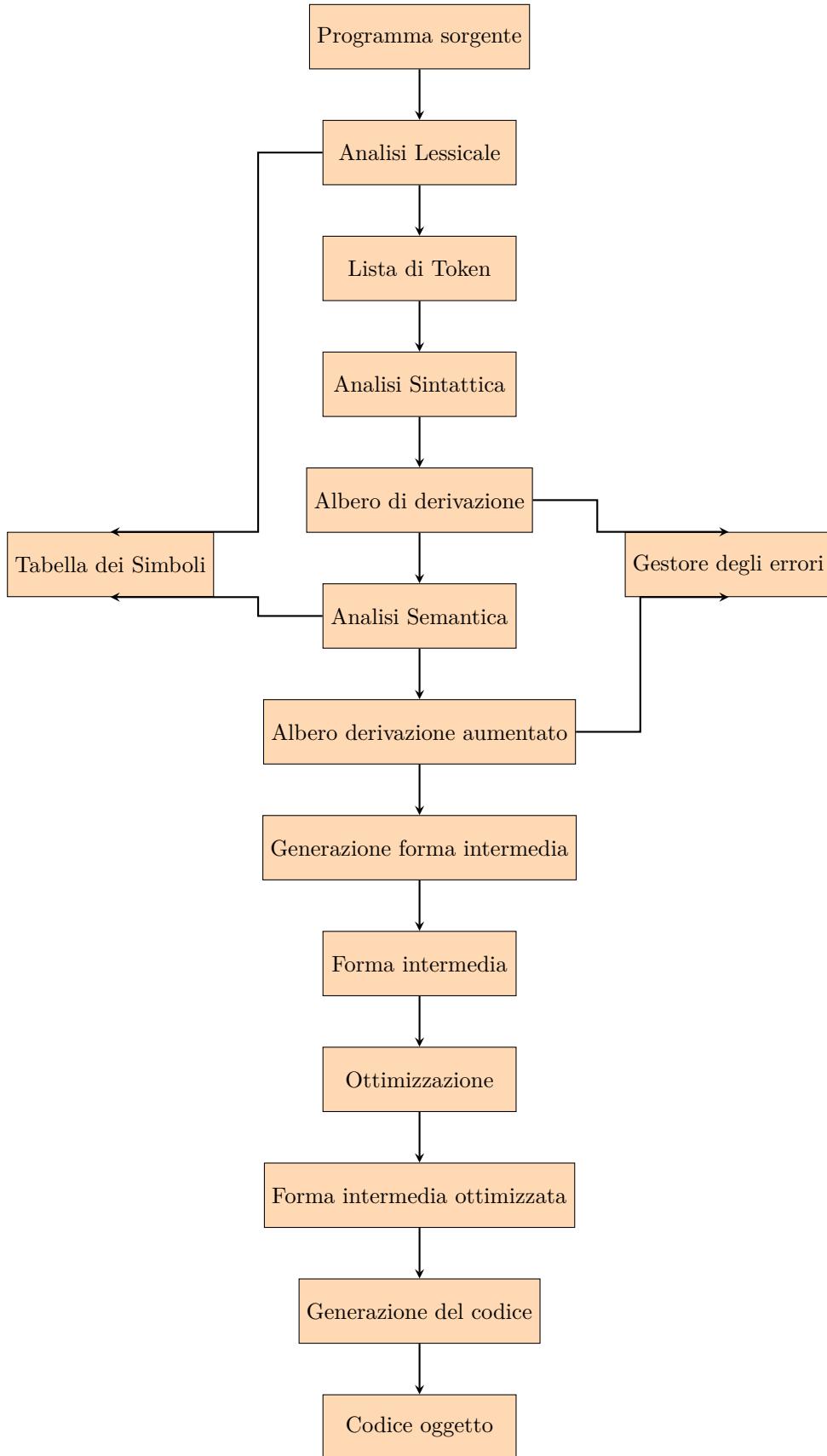
Definition 3.5.1: implementazione

Per **implementazione** si intende la scrittura di un compilatore per una macchina ospite già realizzata, costruendo così una macchina astratta per il linguaggio

3.5.1 Correttezza dell'implementazione

Per far sì che un compilatore sia corretto occorre **dimostrare che il programma preservi la semantica**, ovvero il programma sorgente e quello oggetto calcolino la stessa funzione

3.5.2 Struttura di un compilatore



3.6 fasi principali della compilazione

3.6.1 analisi lessicale (scanner)

L'analisi lessicale spezza il programma sorgente nei componenti sintattici primitivi chiamati "tokens" (identificatori, numeri, operatori, parametri, parole riservate)

- controlla solo che il lessico sia ammissibile
- riempie parzialmente la tabella dei simboli per gli identificatori di variabili, procedure funzioni . . .

Per realizzare uno scanner avremo bisogno di studiare:

- **grammatiche regolari**
- **espressioni regolari:** un formalismo usato per descrivere i linguaggi generati da grammatiche regolari
- **automi a stati finiti:** uno strumento che permette di riconoscere i linguaggi regolari

3.6.2 analisi sintattica (parser)

A partire dalla lista di tokens, generata dallo scanner, il parser produce l'albero di derivazione del programma, riconoscendo se le frasi sono sintatticamente corrette

Ad esempio controlla che:

- le parentesi siano bilanciate: ((a)+b))
- che i comandi siano composti secondo le regole grammaticali `if(x=5) then then x:=3`

Per realizzare un Parser, avremo bisogno di:

- grammatiche libere dal contesto
- automi a pila

3.6.3 Analisi semantica

l'analisi semantica **esegue dei controlli di semantica statica** (ovvero sintattici contestuali) per rilevare eventuali errori semantici

Arricchisce l'albero di derivazione generato dal Parser con informazioni sui tipi, verifica i tipi negli assegnamenti, parametri attuali vs. formali, dichiarazione e uso di variabili e genera eventuali errori

3.6.4 Generazione della forma intermedia

Genera codice scritto in un **linguaggio intermedio** indipendente dall'architettura, facilmente traducibile nel linguaggio macchina di varie macchine diverse. Nel generare questo codice intermedio si esegue la struttura dell'albero sintattico, ricavato dall'albero di derivazione

3.6.5 Ottimizzazione

Si effettuano ottimizzazioni nel codice intermedio per renderlo più efficiente

- rimozione di codice inutile (dead code)
- espansione in linea di chiamate di funzioni
- fattorizzazione di sottoespressioni
- mettere fuori dai cicli sottoespressioni che non variano

Alla fine si ottiene un codice intermedio **ottimizzato**

3.6.6 Generazione del codice

Viene generato codice per una specifica architettura (include anche l'assegnazione dei registri e ottimizzazioni specifiche macchine)

3.6.7 Tabella dei simboli

Memorizza le informazioni sui nomi presenti nel programma (identificatori di variabili, funzioni, procedure)
Es: per le matrice mette, come attributo la dimensione e il tipo dei suoi elementi

3.7 semantica operazionale strutturata

3.7.1 Definizione di un linguaggio a cui dare semantica

La **semantica operazionale strutturata** È utilizzata per descrivere come ogni singola istruzione o espressione in un linguaggio modifica lo stato di un sistema in termini di transizioni di stato

Il suo **linguaggio** viene **definito tramite sintassi astratta semplice ed intuitiva, ma ambigua** ed una stringa viene sempre accoppiata ad un albero sintattico (non ambiguo)

Alcuni elementi fondamentali del linguaggio vengono definiti attraverso **insiemi di base**:

- **Booleani**: l'insieme dei valori booleani è composto da due valori: $\{\text{tt}, \text{ff}\}$. Le metavariabili sono $t, t_1, t' \in \mathbb{T}$
- **numeri naturali**: $\{0, 1, 2, \dots\} \quad n, m, p \in \mathbb{N}$
- **variabili**: $a, b, c, \dots, z \quad v \in Var$

Per descrivere espressioni più complesse, vengono definiti alcuni **insiemi derivati** utilizzando la notazione BNF (Backus-Naur Form)

- **espressioni aritmetiche (exp)**:

$$e ::= m|v|e + e|e - e|e * e$$

- **espressioni booleane (Bexp)**:

$$b ::= t|e = e|b \text{ or } b|\neg b$$

- **Comandi Com**:

$$c ::= \text{skip}|v := e|c;c| \text{while } b \text{ do } c|\text{if } b \text{ then } c \text{ else } c$$

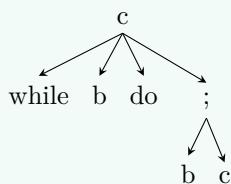
Questo tipo di sintassi è piuttosto semplice ma è ambigua, **per una sintassi non ambigua ne dovrei costruire una completa, ma molto più complicata** (dovrei gestire le precedenze, le parentesi ecc...) ma non serve nel dare una semantica in un linguaggio di programmazione perché un **parser (analizzatore sintattico)** prende in input un **programma scritto in sintassi concreta (non ambigua)** e restituisce un albero sintattico di sintassi astratta (quella che stiamo appena definendo), pertanto, nel dare semantica possiamo partire dagli alberi di sintassi astratta (ambigua) e ignorare la parte di anali del parser

Example 3.7.1

Riportiamo qui un esempio di sintassi astratta.

Che tipo di albero sintattico vogliamo intendere con la seguente espressione?

while b do $c_1; c_2$



3.8 Dare semantica ad un linguaggio

Entriamo nel vivo del discorso, ma prima definiamo, per ogni categoria sintattica (cioè **Exp**, **Bexp**, **Com**) un modello detto **sistema di transizione** che è fondamentalmente un "grafo" di stati

Definition 3.8.1: sistema di transizione

Un **sistema di transizione** è una tripla $\langle \Gamma, T, \rightarrow \rangle$ dove

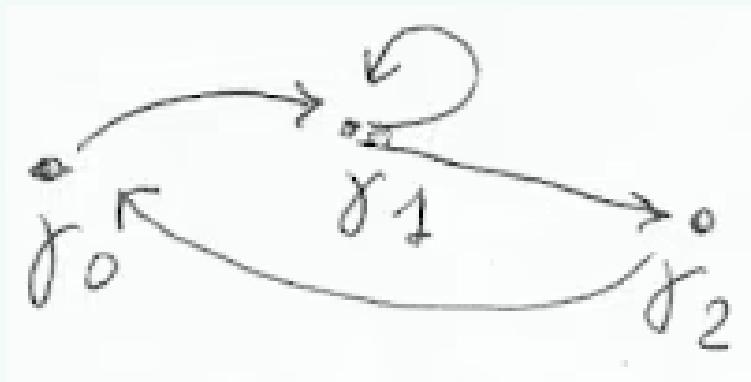
- Γ è l'insieme di stati (o configurazione)
- $T \subseteq \Gamma$ è l'insieme degli stati terminali (ovvero tutti quegli stati in cui il calcolo è stato terminato con successo)
- $\rightarrow \subseteq \Gamma \times \Gamma$ è la relazione di transazione che prende in input uno stato $\in \Gamma$ e restituisce un'altro stato $\in \Gamma$

Una computazione a partire dallo stato γ_0 è una sequenza $\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \dots$ che può essere finita o infinita, invece con \rightarrow^* si indica la chiusura riflessiva e transitiva di \rightarrow , ovvero:

$$\frac{}{\gamma \rightarrow^* \gamma} \quad \frac{\gamma \rightarrow^* \gamma' \quad \gamma' \rightarrow \gamma''}{\gamma \rightarrow^* \gamma''}$$

ovvero si può raggiungere da uno stato γ uno stato γ'' in più passi

Example 3.8.1



Questa è una rappresentazione grafica di un grafo in cui i nodi sono gli stati e gli archi le transizioni

Se voglio definire la semantica (se voglio usare questo tipo di struttura) del linguaggio con la sintassi definita prima occorre definire uno stato di transazione specifico per **Exp**, per **Bexp** e per **Com**

Vi sono tuttavia **diversi problemucci**, del tipo:

1. Γ è di solito un insieme infinito contabile, allora vi è la **necessità di trovare una rappresentazione finita ed implicita attraverso grammatiche**. Questo vuol dire che Γ coincide con uno dei linguaggi delle 3 categorie sintattiche (ovvero **Exp**, **Bexp**, **Com**)

Example 3.8.2

$$\Gamma_e = \{ \langle e, \sigma \rangle | e \in Exp, \sigma \in Store \}$$

Dove σ è una funzione che associa ad ogni variabile un numero naturale, perché lo stato del mio sistema è una coppia in cui la prima parte indica l'espressione che devo valutare, la seconda componente è lo store che indica il valore dell'espressione

2. $\rightarrow \subseteq \Gamma \times \Gamma$ è una relazione costituita da infinite coppie $\gamma \rightarrow \gamma'$, anche qui vi è la necessità di trovare una rappresentazione finita ed implicita come minima relazione che soddisfa **un certo insieme finito di assiomi e regole di inferenza**, quindi la semantica non è che un insieme di regole di inferenza che mi indicano in modo calcolare le transizioni che mi portano ad eseguire un certo comando
3. per dare significato alle variabili (che posono solo assumere valore su \mathbb{N}) è necessario introdurre uno **store** $\sigma : Var \rightarrow \mathbb{N}$, come funzione che associa ad ogni variabile un valore

$$\sigma = \{x_1/n_1, x_2/n_2, \dots, x_k/n_k\}$$

Se supponiamo che $var = \{x_1, x_2, \dots, x_k\}$

3.8.1 Semantica delle espressioni aritmetiche

Adesso introduciamo la **Semantica delle espressioni aritmetiche**, un tipo di semantica operazionale. Deve ovviamente avere un sistema di transizione $\langle \Gamma_e, T_e, \rightarrow_e \rangle$ dove:

- $\Gamma_e = \{\langle e, \sigma \rangle | e \in Exp, \sigma \in Store\}$
- $T_e = \{\langle n, \sigma \rangle | n \in \mathbb{N}, \sigma \in Store\}$
- La relazione \rightarrow_e è definita come la minima relazione che soddisfa gli assiomi e le regole di inferenza qui sotto:

1. Variabile:

$$\overline{\langle v, \sigma \rangle \rightarrow_e \langle \sigma(v), \sigma \rangle}$$

Ovvero il tuo stato terminale sarà il numero della variabile v indicato dallo Store (inoltre σ rimane inalterato). Quindi valuto ciò che v vale in σ

2. Somma 1:

$$\frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e'_0, \sigma' \rangle}{\langle e_0 + e_1, \sigma \rangle \rightarrow_e \langle e'_0 + e_1, \sigma' \rangle}$$

Nel momento in cui riesco a fare un passo di valutazione da e_0 a e'_0 alterando anche lo stato dello store da σ a σ' questa trasformazione si anche applicare durante una somma, in altre parole l'espressione si semplifica o riduce (ad esempio, una variabile viene sostituita con il suo valore), e nel contempo lo stato della memoria potrebbe essere aggiornato se l'espressione stessa comporta una modifica ai valori delle variabili

3. Somma 2:

$$\frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e'_1, \sigma' \rangle}{\langle m + e_1, \sigma \rangle \rightarrow_e \langle m + e'_1, \sigma' \rangle}$$

Stessa roba ma con un numero m

4. Somma 3:

$$\overline{\langle m + m', \sigma \rangle \rightarrow_e \langle P, \sigma \rangle} \quad \text{dove } P = m + m'$$

5. Sottrazione 1:

$$\frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e'_0, \sigma' \rangle}{\langle e_0 - e_1, \sigma \rangle \rightarrow_e \langle e'_0 - e_1, \sigma' \rangle}$$

6. Sottrazione 2:

$$\frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e'_1, \sigma' \rangle}{\langle m - e_1, \sigma \rangle \rightarrow_e \langle m - e'_1, \sigma' \rangle}$$

7. Sottrazione 3:

$$\overline{\langle m - m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle}$$

Si noti come la somma e la sottrazione prima valutano la sottoespressione di sinistra (e_0) con somma/sottrazione 1 poi, se questa s'è mutata in numero, valutano la sottoespressione di destra (e_1) con somma/-sottrazione 2 ed infine, se questa s'è mutata in un numero, viene fatta la somma/sottrazione finale con somma/sottrazione 3

Example 3.8.3

Esempietto per valutare $\langle (x+2) - y, \{x/5, y/3\} \rangle$:

$$\begin{array}{l}
 (\text{Var}) \quad \overline{\langle x, \{x/5, y/3\} \rangle} \longrightarrow \langle 5, \{x/5, y/3\} \rangle \\
 (\text{Sum}_1) \quad \overline{\langle x+2, \{x/5, y/3\} \rangle} \longrightarrow \langle 5+2, \{x/5, y/3\} \rangle \\
 (\text{Sub}_1) \quad \overline{\langle (x+2)-y, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_0} \overline{\langle (5+2)-y, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_1} \\
 (\text{Sum}_3) \quad \overline{\langle 5+2, \{x/5, y/3\} \rangle} \longrightarrow \langle 7, \{x/5, y/3\} \rangle \\
 (\text{Sub}_1) \quad \overline{\langle (5+2)-y, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_1} \overline{\langle 7-y, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_2} \\
 (\text{Var}) \quad \overline{\langle y, \{x/5, y/3\} \rangle} \longrightarrow \langle 3, \{x/5, y/3\} \rangle \\
 (\text{Sub}_2) \quad \overline{\langle 7-y, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_2} \overline{\langle 7-3, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_3} \\
 (\text{Sub}_3) \quad \overline{\langle 7-3, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_3} \overline{\langle 4, \{x/5, y/3\} \rangle} \xrightarrow{\gamma_4}
 \end{array}$$

Si ha, quindi, che $\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \in T_e$ cioè $\langle (x+2) - y, \{x/5, y/3\} \rangle \rightarrow^* \langle 4, \{x/5, y/3\} \rangle$

Theorem 3.8.1

Vogliamo dimostrare che \rightarrow_e è deterministico, ovvero:

$$\gamma \rightarrow_e \gamma' \text{ e } \gamma \rightarrow_e \gamma'', \text{ allora } \gamma' = \gamma'' \quad \forall \gamma, \gamma', \gamma''$$

In altre parole significa che **da ogni transizione esce al più una transizione**, mai più di una

dimostrazione: mi riduco a dimostrare che $(\langle e, \sigma \rangle \rightarrow_e \gamma' \wedge \langle e, \sigma \rangle \rightarrow_e \gamma'') \Rightarrow \gamma' = \gamma''$
Procedo per induzione strutturale, con HP $(\langle e, \sigma \rangle \rightarrow_e \gamma' \wedge \langle e, \sigma \rangle \rightarrow_e \gamma'') \Rightarrow \gamma' = \gamma''$.

1. $e = m \in \mathbb{N}$: se $\langle e, \sigma \rangle \not\rightarrow_e e$ allora la conclusione è vera perché la premessa è falsa
2. $e = v \in \text{Var}$: Per la regola (Var), l'unica transizione derivabile per $\langle v, \sigma \rangle$ è $\langle v, \sigma \rangle \rightarrow_e \langle \sigma(v), \sigma \rangle$ poiché σ è una funzione (cioè $\sigma(v)$ è univoco) e la sola regola (Var) è applicabile allora per forza $\langle \sigma(v), \sigma \rangle = \sigma'$ e $\langle \sigma(v), \sigma \rangle = \sigma''$ quindi $\gamma' = \gamma''$
3. $e = e_0 + e_1$: Supponiamo che $\langle e_0 + e_1, \sigma \rangle \rightarrow \gamma'$ e $\langle e_0 + e_1, \sigma \rangle \rightarrow \gamma''$

Ci sono 3 sottocasi da esaminare in accordo nel modo in cui derivo $\langle e_0 + e_1, \sigma \rangle \rightarrow \gamma'$:

- (a) $\langle e_0, \sigma \rangle \rightarrow \langle e'_0, \sigma' \rangle$ e $\gamma' = \langle e'_0 + e_1, \sigma' \rangle$ in questo caso ho che $e_0 \notin \mathbb{N}$ e la regola che ho applicato è **Somma**
1. Allora se $\langle e_0 + e_1, \sigma \rangle \rightarrow \gamma''$ è necessario che $\langle e_0, \sigma \rangle \rightarrow \langle e''_0, \sigma'' \rangle$ e che $\gamma'' = \langle e''_0 + e_1, \sigma'' \rangle$.
Tuttavia per (HP) si ha che $\langle e'_0, \sigma' \rangle = \langle e''_0, \sigma'' \rangle$ pertanto deve essere che $e'_0 = e''_0$ e $\sigma' = \sigma''$, da cui discende $\gamma' = \gamma''$

(b) $e_0 = m \in \mathbb{N}$ ed $\langle e_1, \sigma \rangle \rightarrow \langle e'_1, \sigma' \rangle$ Caso analogo al precedente, dato che ho che $e_1 \notin \mathbb{N}$ e la regola che ho applicato è **Somma 2**. Allora se $\langle e_0 + e_1, \sigma \rangle \rightarrow \gamma''$, è necessario che $\langle e_1, \sigma \rangle \rightarrow \langle e''_1, \sigma'' \rangle$ e $\gamma'' = \langle e_0 + e''_1, \sigma'' \rangle$. Tuttavia per (HP) si ha che $\langle e'_1, \sigma' \rangle = \langle e''_1, \sigma'' \rangle$ pertanto deve essere che $e'_1 = e''_1$ e $\sigma' = \sigma''$, da cui discende $\gamma' = \gamma''$

(c) $e_0 \in \mathbb{N}$ ed $e_1 \in \mathbb{N}$ In questo caso, solo **Somma 3** è applicabile, ottenendo una sola passibile transizione:

$$\langle e_0 + e_1, \sigma \rangle \rightarrow \langle P, \sigma \rangle \text{ dove } P = e_0 + e_1$$

Quindi la tesi segue:

$$\langle e_0 + e_1, \sigma \rangle \rightarrow \gamma' \wedge \langle e_0 + e_1, \sigma \rangle \rightarrow \gamma'' \implies \gamma' = \gamma''$$

4. $e = e_1 - e_2$: DEL TUTTO ANALOGO AL CASO PRECEDENTE

Q.e.d.



Questo teorema ci porta ad un dio boia di corollario:

Corollary 3.8.1

poiché \rightarrow_e è deterministica, a partire da $\langle e, \sigma \rangle$ arriveremo su una sola configurazione terminale $\langle n, \sigma \rangle$: "n è il valore di e in σ "

È possibile perciò definire una funzione

$$eval : Expr \times Store \rightarrow \mathbb{N}$$

che da semantica alle espressione

$$eval(e, \sigma) = \begin{cases} m & \text{se } \langle e, \sigma \rangle \rightarrow^* \langle m, \sigma \rangle \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Esempi:

Example 3.8.4

- $eval((x + 2) - y, \{x/5, y/3\}) = 4$ dato che

$$\langle (x + 2) - y, \{x/5, y/3\} \rangle \rightarrow^* \langle 4, \{x/5, y/3\} \rangle$$

- $eval((x + 2) - y, \{x/2, y/7\}) = \text{indefinito}$ dato che

$$\langle (x + 2) - y, \{x/2, y/7\} \rangle \rightarrow^* \langle 4 - 7, \{x/2, y/7\} \rangle \rightarrow$$

Inoltre sia introdotta la definizione di equivalenza:

Definition 3.8.2: Equivalenza tra espressioni

Siano e ed e' due espressioni, allora si dicono **equivalenti** sse $\forall \sigma \in Store \quad eval(e, \sigma) = eval(e', \sigma)$
E si denota con $e \equiv e'$

Esempietto:

Example 3.8.5

$$v_1 + (v_2 + v_3) \equiv (v_1 + v_2) + v_3$$

Si osservi come **Eval** è definita rispetto alla disciplina di valutazione **IS** (interno destro), pertanto, rigorosamente, **Eval** è denotato come $Eval_{is}$. Si può, inoltre, dimostrare che anche per **ID** (interno destro), il risultato della valutazione è lo stesso:

$$Eval_{is} = Eval_{id}$$

Dove $Eval_{id}(e, \sigma) = \begin{cases} m & \text{se } \langle e, \sigma \rangle \xrightarrow{id}^* \langle m, \sigma \rangle \\ \text{indefinita} & \text{altrimenti} \end{cases}$

Come vedremo, è possibile definire anche altre discipline di valutazione come Esterna Sinistra, Esterne Destra, Esterna parallela.

3.8.2 Semantica delle espressioni booleane

Arriviamo alle espressioni booleane con la seguente grammatica:

$$b ::= t|e = e|b \text{ or } b|\neg b$$

(ricordo che t è una metavariabile con un valore di verità true o false) E il seguente sistema di transazione:

$$\langle \Gamma_b, T_b, \rightarrow_b \rangle \text{ dove } \Gamma_b = \{\langle b, \sigma \rangle | b \in Bexp, \sigma \in Store\} \text{ e } T_b = \{\langle tt, \sigma \rangle, \langle ff, \sigma \rangle | \sigma \in Store\}$$

e \rightarrow_b è la minima relazione generata dai seguenti assiomi e regole di inferenza:

- **Eq1**

$$\frac{\langle e_0 = e_1, \sigma \rangle \rightarrow_b \langle e'_1, \sigma' \rangle}{\langle m = e_1, \sigma \rangle \rightarrow_b \langle m = e'_1, \sigma' \rangle}$$

- **Eq2**

$$\frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e'_1, \sigma' \rangle}{\langle m = e_1, \sigma \rangle \rightarrow_b \langle m = e'_1, \sigma' \rangle}$$

- **Eq3**

$$\frac{}{\langle m = m, \sigma \rangle \rightarrow_b \langle t, \sigma \rangle} \text{ dove } t = \begin{cases} tt & \text{se } m = n \\ ff & \text{se } m \neq n \end{cases}$$

- **Or1**

$$\frac{\langle b_0, \sigma \rangle \rightarrow_b \langle b'_0, \sigma' \rangle}{\langle b_0 \text{ or } b_1, \sigma \rangle \rightarrow_b \langle b'_0 \text{ or } b_1, \sigma' \rangle}$$

- **Or2**

$$\overline{\langle tt \text{ or } b_1, \sigma \rangle \rightarrow_b \langle tt, \sigma \rangle}$$

- **Or3**

$$\overline{\langle ff \text{ or } b_1, \sigma \rangle \rightarrow_b \langle b_1, \sigma \rangle}$$

- **Neg1**

$$\frac{\langle b, \sigma \rangle \rightarrow_b \langle b', \sigma' \rangle}{\langle \neg b, \sigma \rangle \rightarrow_b \langle \neg b', \sigma' \rangle}$$

- **Neg2**

$$\overline{\langle \neg b, \sigma \rangle \rightarrow_b \langle t', \sigma \rangle} \text{ dove } t' = \begin{cases} tt & \text{se } t = ffff \\ ff & \text{se } t = tt \end{cases}$$

Si tenga presente che Eq1, Eq2 e Eq3 sono cosiddette **interne sinistre** perché inizio a valutare la sottoespressione di sinistra per poi restituire un valore di verità t sse ho ottenuto numeri in tutte e due le sottoespressioni mentre Or1, Or2 e Or3 sono **esterne sinistre** perché inizio a valutare la sottoespressione di sinistra per poi restituire un valore di verità t sse ho ottenuto numeri almeno in una sottoespressione. Quindi se nelle interne dovevo avere dei numeri in tutte le sottoespressioni per poi eseguire la valutazione finale nelle esterne per eseguire la valutazione finale mi basta avere una quantità sufficiente

Anche per i booleani si ha questo teorema:

Theorem 3.8.2

\rightarrow_b è deterministica, ovvero

$$(\gamma \rightarrow_b \gamma' \wedge \gamma \rightarrow_b \gamma'') \implies \gamma' = \gamma''$$

Che porta al seguente corollario:

Corollary 3.8.2

si può, quindi, definire:

$$\text{eval}_b(b, \sigma) = \begin{cases} t & \text{se } \langle b, \sigma \rangle \xrightarrow{*} \langle t, \sigma \rangle \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

E si ha anche la seguente definizione:

Definition 3.8.3: Equivalenza booleani

Siano b ed b' due booleani, allora si dicono **equivalenti** sse $\forall \sigma \in \text{Store} \quad \text{eval}_b(b, \sigma) = \text{eval}_b(b', \sigma)$
E si denota con $b \equiv b'$

Example 3.8.6

$$\neg((3 = v) \vee (3 = 4)) = \neg(v = 3)$$

Si possono definire per b_0 or b_1 regole di valutazioni diverse da ES. Ad esempio ED o IS, ma non sono tutte equivalenti, si provi, ad esempio, con ED:

- **Or1':**

$$\frac{\langle b_1, \sigma \rangle \rightarrow_b \langle b'_1, \sigma' \rangle}{\langle b_0 \text{ or } b_1, \sigma \rangle \rightarrow_b \langle b_0 \text{ or } b'_1, \sigma' \rangle}$$

- **Or2':**

$$\overline{\langle b_0 \text{ or } tt, \sigma \rangle \rightarrow_b \langle tt, \sigma \rangle}$$

- **Or3':**

$$\overline{\langle b_0 \text{ or } ff, \sigma \rangle \rightarrow_b \langle b_0, \sigma \rangle}$$

Example 3.8.7

$$\gamma = \langle \rangle$$

Chapter 4

Analisi lessicale: espressioni regolari, DFA, NFA

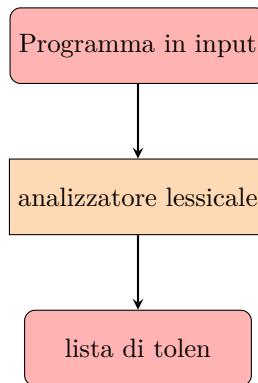
4.1 analisi lessicale

partiamo dalla definizione

Definition 4.1.1: Analisi lessicale

Riconoscere nella stringa in ingresso gruppi/sequenze di simboli che corrispondono a specifiche categorie sintattiche

La stringa in input, poi, è trasformata in una sequenza di simboli astratti, detti **token**, si analizzi la figura:



4.1.1 token

Definition 4.1.2: Token

un token è una coppia (nome, valore), dove:

- il *nome* è un simbolo astratto che rappresenta una categoria semantica
- il *valore* una sequenza di simboli del testo in ingresso

Esempietto:

Example 4.1.1

Un esempio di token è $\langle \text{Ide}, x1 \rangle$, dove:

- *Ide*: è l'informazione che identifica una classe di token

- x_1 : è l'informazione che identifica lo specifico token
-

Siano inoltre tali definizioni

Definition 4.1.3: Pattern

è la descrizione generale della forma dei valori di una classe di token

Example 4.1.2

Sia $(x \mid y)(x \mid y \mid 0 \mid 1)^*$ un'espressione regolare per rappresentare un Pattern

Definition 4.1.4: lessema

si definisce **lessema** una stringa istanza di un pattern

Example 4.1.3

nel nostro esempio $x1$ è un'istanza di un pattern

vedremo che ad ogni nome di categoria sintattica è associato un pattern che specifica i possibili valori che possono essere presi per quel nome, come lessemi

Example 4.1.4

dalla strinfa C si ha:

```
if(x == 0) printf("zero")
```

un analizzatore lessicale potrebbe produrre la seguente sequenza di token:

- $\langle \text{if} \rangle$
- $\langle () \rangle$
- $\langle \text{ide}, x \rangle$
- $\langle \text{Operel}, == \rangle$
- $\langle \text{const - num}, 0 \rangle$
- $\langle () \rangle$
- $\langle \text{Ide}, \text{printf} \rangle$
- $\langle () \rangle$
- $\langle \text{const - string}, \text{zero} \rangle$
- $\langle () \rangle$

In realtà, normalmente lo scanner associa agli identificatori un indirizzo della tabella dei simboli, quindi $\langle \text{ide}, x \rangle$ è in realtà $\langle \text{ide}, \text{puntatore alla tabella dei simboli} \rangle$

4.1.2 espressioni regolari

Nello stesso modo in cui possiamo scrivere espressioni matematiche utilizzando operatori matematici ($+, \times, \dots$) e' possibile, utilizzando appositi operatori regolari, definire **espressioni regolari** che descrivono un linguaggio. Ad

esempio:

$$(a \mid b)c^*$$

dove 'a', 'b' e 'c' vengono intesi come i linguaggi $\{a\}$, $\{b\}$ e $\{c\}$, ' \mid ' e' l'operazione di unione, quindi $\{a\} \cup \{b\} = \{a, b\}$. Fra le parentesi e la 'c' e' sottintesa l'operazione di congiunzione ' \cdot ', ma ha priorita' l'operatore '*' che ha lo stesso effetto della stella di Kleene. Quindi diventa:

$$\{a, b\} \cdot \{c\}^* = \{a, b, ac, bc, acc, bcc, \dots\}$$

Definiamo formalmente questo tipo di espressioni e i linguaggi che definiscono:

Definition 4.1.5: espressioni regolari

fissato un alfabeto $A = \{a_1, a_2, \dots, a_n\}$, definiamo le espressioni regolari su A con la seguente BNF

$$r ::= \emptyset \mid \varepsilon \mid a \mid r \cdot r \mid r|r \mid r^*$$

Note:

La definizione non e' ciclica perche' le espressioni sono definite da altre espressioni piu' piccole.

Note:

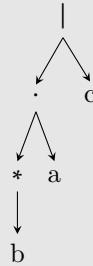
Non confondere le ER \emptyset e ε : la prima identifica il linguaggio vuoto, che non accetta nessuna stringa, mentre il linguaggio associato alla seconda accetta la stringa vuota.

Note:

Si tenga presente che questa è una sintassi astratta ambigua, ci vorrebbero le parentesi per disambiguare, tuttavia noi assumiamo che:

- la concatenazione, disgiunzione e ripetizione associano a sx
- la precedenza tra gli operatori sia: $* > \cdot > |$
- la concatenazione \cdot è di solito omessa

Per cui, ad esempio, $b * a | c$ corrisponde all'albero sintattico:



Quindi secondo una sintassi non ambigua: $((b)^*) \cdot (a)) | (c)$

linguaggio denotato da una espressione regolare

Definition 4.1.6

dato l'alfabeto A , definiamo la funzione:

$$\mathcal{L} : \text{Exp-Reg} \rightarrow \mathcal{P}(A^*)$$

Come segue:

$$\begin{aligned}\mathcal{L}[\emptyset] &= \emptyset \text{ (linguaggio vuoto)} \\ \mathcal{L}[\varepsilon] &= \{\varepsilon\} \text{ (linguaggio che contiene solo la stringa vuota)} \\ \mathcal{L}[a] &= \{a\} \\ \mathcal{L}[r_1 \cdot r_2] &= \mathcal{L}[r_1] \cdot \mathcal{L}[r_2] \\ \mathcal{L}[r_1|r_2] &= \mathcal{L}[r_1] \cup \mathcal{L}[r_2] \\ \mathcal{L}[r^*] &= (\mathcal{L}[r])^*\end{aligned}$$

Note:

Si ricordi che:

$$\begin{aligned}L_1 \cdot L_2 &= \{xy \mid x \in L_1, y \in L_2\} \\ L_1 \cup L_2 &= \{x \mid x \in L_1 \text{ or } x \in L_2\} \\ L^\circ &= \{\varepsilon\} \\ L^{n+1} &= L \cdot L^n \\ L^* &= \bigcup_{n \geq 0} L^n\end{aligned}$$

linguaggio regolare

Definition 4.1.7: Linguaggio regolare

un linguaggio $L \subseteq A^*$ è definito regolare sse \exists una espressione regolare r tale che:

$$L = \mathcal{L}[r]$$

Proposition 4.1.1

ogni linguaggio finito è regolare

Example 4.1.5

Sia $L = \{a, bc\}$ con $r = a \mid bc$ si ha che:

$$\mathcal{L}[a \mid bc] = \mathcal{L}[a] \cup \mathcal{L}[bc] = \{a\} \cup \mathcal{L}[b] \cdot \mathcal{L}[c] = \{a\} \cup \{b\} \cdot \{c\} = \{a, bc\} = L$$

Si osservi che esistono anche linguaggi regolari infiniti:

$$\begin{aligned}\mathcal{L}[a^*b] &= \mathcal{L}[a^*] \cdot \mathcal{L}[b] = (\mathcal{L}[a])^* \cdot \mathcal{L}[b] \\ &= \{a\}^* \cdot \{b\} = \bigcup_{m \geq 0} \{a^m\} \cdot \{b\} \\ &= \{\varepsilon, a, aa, \dots\} \cdot \{b\} = \{a^m b \mid n \geq 0\}.\end{aligned}$$

$$\mathcal{L}[a \mid a^*b] = \mathcal{L}[a] \cup \mathcal{L}[a^*b] = \{a\} \cup \{a^m b \mid n \geq 0\}.$$

$$\begin{aligned}\mathcal{L}[(a \mid b) \cdot b^*] &= \mathcal{L}[a \mid b] \cdot \mathcal{L}[b^*] = (\mathcal{L}[a] \cup \mathcal{L}[b]) \cdot (\mathcal{L}[b])^* \\ &= \{a, b\} \cdot \{b\}^* = \{ab^m \mid n \geq 0\} \cup \{b^n \mid n \geq 1\}.\end{aligned}$$

Example 4.1.6 (espressioni regolari)

$$A = \{0, 1\}$$

- 0^*10^*
Con $L_2\{w \in A^* \mid A^* \mid w \text{ contiene un solo } 1\}$
- $(0 \mid 1)^*001(0 \mid 1)^*$
Con $L_1\{w \in A^* \mid A^* \mid w \text{ contiene un solo } 1\}$
- $1^*(011^*)^*$
Con $L_3 = \{w \in A^* \mid w \text{ contiene 001 come sottostringa}\}$

altri operatori ausiliari

Definition 4.1.8

- Ripetizione positiva: $r^+ = rr^*$
- Possibilità: $r? = r \mid \epsilon$
- Elenco: $[a_1, \dots, a_n] = a_1 \mid \dots \mid a_n$

4.1.3 equivalenza tra espressioni regolari

Definition 4.1.9: equivalenza

Due espressioni regolari r ed s sono **equivalenti** sse $\mathcal{L}[r] = \mathcal{L}[s]$ (cioè demotano lo stesso linguaggio) e lo denotiamo con $r \equiv s$

Esistono molte leggi per \equiv , alcune sono le seguenti

$$\begin{aligned}r|s &\equiv s|r \quad (\text{l è commutativa}) \\ r|(s|t) &\equiv (r|s)|t \quad (\text{l è associativa}) \\ z|z &\equiv z \quad (\text{l è idempotente}) \\ z \cdot (s \cdot t) &\equiv (z \cdot s) \cdot t \quad (\cdot \text{ è associativa}) \\ \epsilon \cdot z &\equiv z \cdot \epsilon \equiv z \quad (\epsilon \text{ è l'elemento neutro per } \cdot) \\ (r^*)^* &\equiv r^* \quad (*) \text{ è idempotente} \\ r(s|t) &\equiv rs|rt \quad (\text{distribuisce a sinistra su } |) \\ (r|s)t &\equiv rt|st \quad (\text{distribuisce a destra su } |)\end{aligned}$$

In alcuni casi è facile dimostrare queste leggi:

$$\begin{aligned}\mathcal{L}[r|s] &= \mathcal{L}[r] \cup \mathcal{L}[s] \\ &= \mathcal{L}[s] \cup \mathcal{L}[r] \\ &= \mathcal{L}[s|r]\end{aligned}$$

$$\begin{aligned}\mathcal{L}[r|r] &= \mathcal{L}[r] \cup \mathcal{L}[r] \\ &= \mathcal{L}[r]\end{aligned}$$

$$\begin{aligned}\mathcal{L}[r\varepsilon] &= \mathcal{L}[r] \cdot \{\varepsilon\} \\ &= \mathcal{L}[r]\end{aligned}$$

$$\begin{aligned}\mathcal{L}[\emptyset^*] &= (\mathcal{L}[\emptyset])^* \\ &= \emptyset^* \\ &= \emptyset^0 \cup \emptyset^1 \cup \emptyset^2 \cup \dots \\ &= \{\varepsilon\} \cup \emptyset \cup \emptyset \\ &= \{\varepsilon\} \\ &= \mathcal{L}[\varepsilon]\end{aligned}$$

$$\begin{aligned}\mathcal{L}[r \cdot \emptyset] &= \mathcal{L}[r] \cdot \mathcal{L}[\emptyset] \\ &= \mathcal{L}[r] \cdot \emptyset \\ &= \emptyset \\ &= \mathcal{L}[\emptyset]\end{aligned}$$

Le espressioni regolari servono per specificare il pattern di una categoria sintattica, ovvero la forma dei possibili lessemi, tuttavia occorre riconoscere se una certa sequenza in ingresso è un lessema per una certa categoria sintattica. A questo ci vengono in aiuto gli automi a stati finiti

4.2 Automi a stati finiti

Un **automa a stati finiti** (o DFA, deterministic finite automaton, per la versione deterministica) è un modello computazionale utilizzato per rappresentare e analizzare linguaggi regolari. È un dispositivo astratto che processa stringhe di simboli e decide se appartengono o meno a un linguaggio

Si può immaginare il DFA come un automa ideale dotato di una testina di lettura, una sequenza di simboli in input da leggere e un sistema di stati, del tipo

Dove inizialmente

- la testina di lettura è posizionata sul primo carattere dell'input
- e vi è un controllo su sullo stato iniziale q_0

E funziona ciclicamente nel modo seguente:

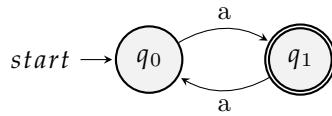
- leggi il carattere in input e in baso allo stato in cui si trova decide:
 - di cambiare di stato
 - di spostare la testina sull'input successivo

FINO A CHE:

- ha finito di leggere l'input (e riconosce la stringa)
- non ha riconosciuto la stringa

4.2.1 diagrammi di transazione

il funzionamento di un automa finito è ben descritto dai cosiddetti **diagrammi di transizione**

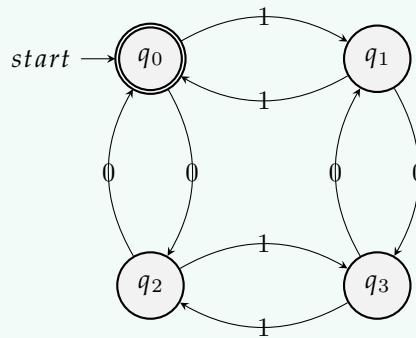


Riconoscere una stringa w significa trovare un cammino etichettato w sul grafo a partire dallo stato iniziale che finisce su uno stato finale

$$L = \{a^{2n+1} \mid n \geq 0\} = \{a^n \mid n \text{ è dispari}\} = \mathcal{L}[a(aa)^*]$$

Example 4.2.1

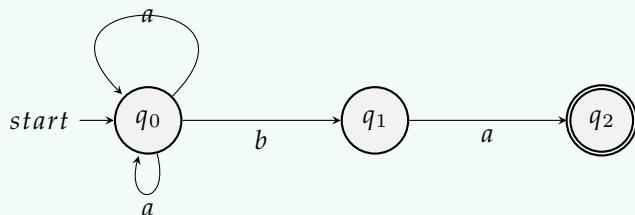
- Primo esempio: Sia $L = \{w \in \{0, 1\}^* \mid \text{in } w \text{ il numero di } 0 \text{ e } 1 \text{ è sempre pari}\}$
Il suo automa è:



- secondo esempio

Sia $L = \mathcal{L}[(a|b)^*ba]$

il suo automa è:



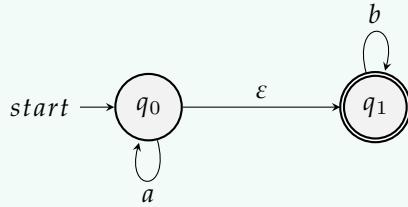
Si noti che $ba \in L[M]$ (ovvero ba appartiene al linguaggio riconosciuto dall'automa M) perché esiste un cammino da q_0 a q_2 etichettato ba

Questo linguaggio è **non deterministico**:

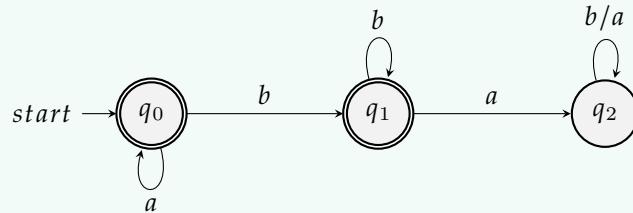
- (q_0, b) offre 2 mosse o su q_0 o su q_1
- (q_1, b) non offre mosse
- $(q_2, a/b)$ non offre mosse
- terzo esempio:

Sia $L[M] = \mathcal{L}[a^*b^*]$

Riconosciuto dal seguente automa:



Che **nondeterministico** perché è possibile spostarsi dallo stato q_0 allo stato q_1 senza leggere l'input.
Se vogliamo un automa deterministico:



Dove q_2 è uno stato pozzo d'errore. Inoltre da ogni stato per ognuno dei due simboli (a e b), esce una e una sola transizione e non vi sono transizioni ϵ

4.2.2 Automi a stati finiti non deterministico

Adesso formalizziamo la definizione

Definition 4.2.1: Automi a stati finiti non deterministici

Si definisce **NFA** o **automa a stati finiti non deterministico** una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove:

- Σ è un alfabeto finito di simboli in input
- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali
- δ è la funzione di transizione con tipo

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$$

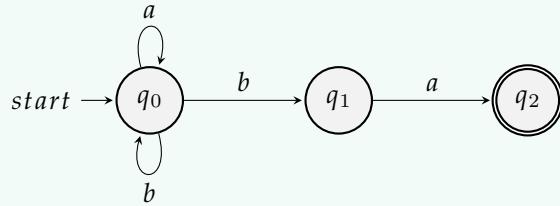
Esempietto:

Example 4.2.2

$$\Sigma = \{a, b\} \quad Q = \{q_0, q_1, q_2\} \quad q_0 \text{ iniziale} \quad F = \{q_2\}$$

δ	a	b	ϵ
q_0	$\{q_0\}$	$\{q_0, q_1\}$	\emptyset
q_1	$\{q_2\}$	\emptyset	\emptyset
q_2	\emptyset	\emptyset	\emptyset

Si può vedere la seguente tabella mutarsi in automa



4.2.3 Linguaggio riconosciuto/accettato

Fornisco prima, formalmente, prima alcune definizioni:

Definition 4.2.2: mossa

Si definisce **mossa** da uno stato q ad uno stato q' leggendo un simbolo σ dall'input e la si denota con \vdash_n tale derivazione (attraverso regole di inferenza logica):

$$\frac{q' \in \delta(q, \sigma)}{(q, \sigma w) \vdash_n (q', w)} \text{ con } \begin{array}{l} \sigma \in \Sigma \cup \{\varepsilon\} \\ w \in \Sigma^* \end{array}$$

Da cui discende la definizione di cammino

Definition 4.2.3: cammino (chiusura riflessiva e transitiva di \vdash_n)

Si definisce cammino da uno stato q ad uno stato q'' tale derivazione:

$$\frac{(q, w) \vdash_n^* (q', w') \quad (q', w') \vdash_n (q'', w'')}{(q, w) \vdash_n^* (q'', w'')}$$

Inoltre si ha:

$$\overline{(q, w) \vdash_n^* (q, w)}$$

da cui discenda la definizione di riconoscimento

Definition 4.2.4: riconoscimento

Una stringa w si definisce **riconosciuta** se è vera tale proposizione:

$$w \in L[N] \iff \exists q \in F. (q_0, w) \vdash_n^* (q, \varepsilon)$$

e da cui discende la definizione di linguaggio accettato

Definition 4.2.5: linguaggio accettato

Un linguaggio L si definisce **accettato da un automa** N , indicato con $L[N]$, è

$$L[N] = \{w \in \Sigma^* \mid \exists q \in F. (q_0, w) \vdash_n^* (q, \varepsilon)\}$$

Note:

due NFA N_1 e N_2 si dicono **equivalenti** sse accettano lo stesso linguaggio, cioè se $L[N_1] = L[N_2]$

Gli NFA sono comodi, ovvero facili da costruire, tuttavia sono **inefficienti**, infatti accettare w significa cercare un cammino su un grafo nondeterministico, il che porta a tante potenziali strade alternative.
In alternativa si possono costruire dei DFA ovvero automi deterministici a stati finiti

4.2.4 Automi a stati finiti deterministici

Un DFA a differenza degli NFA ha le seguenti caratteristiche:

- $\delta(q, \sigma)$ è sempre un singoletto (solo una mossa possibile)
- non ci sono mosse ε

E questo implica

- una scansione completa dell'input garantita
- in un tempo $O(|w|)$ sappiamo se w è accettata o meno
- difficile da definire

Introduciamo la definizione formalmente

Definition 4.2.6: Automi deterministici a stati finiti

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

- Σ è un alfabeto finito di simboli in input
- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali
- δ è la funzione di transazione con tipo

$$\delta : Q \times \Sigma \rightarrow Q$$

e si ha che $(q, \sigma) = q'$

Si osservi che

Claim 4.2.1

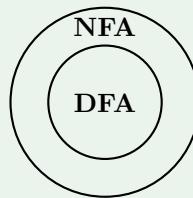
Un DFA è un particolare tipo di NFA tale che:

- $\forall q \in Q. \delta(q, \varepsilon) = \emptyset$

Ovvero **non ci sono transizioni ε**

- $\forall \sigma \in \Sigma. \forall q \in Q. \exists q' \in Q. \delta(q, \sigma) = \{q'\}$

Ovvero l'insieme delle mosse possibile è sempre un singoletto



Si vuole ora dimostrare che i **DFA sono tanto espressivi quanto gli NFA**, sebbene siano un loro sottoinsieme proprio

Proposition 4.2.1

Per ogni NFA, è possibile costruire un DFA ad esso equivalente

Dimostrazione: Occorre seguire contemporaneamente tutti i possibili cammini alternativi dell'NFA di modo che gli stati del DFA che andranno a costruire sono costituiti da insiemi di stati dell'NFA



Per dimostrare questa cosa occorre prima introdurre diversi concetti

ε -closure

Definition 4.2.7: ε -closure

L' ε -closure di uno stato $q \in Q$, denotata come $\varepsilon\text{-closure}(q)$, è definita come l'insieme degli stati q' tali che esiste un cammino da q a q' usando solo transizioni ε , inclusivamente q stesso.

In simboli:

$$\varepsilon\text{-closure}(q) = \{q' \in Q \mid q \xrightarrow{\varepsilon^*} q'\},$$

In altre parole si può definire come il minimo insieme che rispetta le seguenti regole:

$$\overline{\{q\} \subseteq \varepsilon\text{-closure}(q)} \quad \frac{p \in \varepsilon\text{-closure}(q)}{\delta(p, \varepsilon) \subseteq \varepsilon\text{-closure}(q)}$$

nel caso abbiamo un insieme P di nodi allarghiamo la definizione di ε -closure a quella di:

$$\varepsilon\text{-closure}(P) = \bigcup_{p \in P} \varepsilon\text{-closure}(p)$$

Example 4.2.3

Considera il seguente NFA:

- Stati: $Q = \{q_0, q_1, q_2\}$
- Transizioni:
 - $\delta(q_0, \varepsilon) = \{q_1\}$
 - $\delta(q_1, \varepsilon) = \{q_2\}$
 - $\delta(q_2, a) = \{q_2\}$

Il calcolo dell' ε -closure è presto fatto:

- $\varepsilon\text{-closure}(q_0)$:
 - Da q_0 , puoi raggiungere q_1 attraverso una transizione ε
 - Da q_1 , puoi raggiungere q_2 attraverso un'altra ε

Quindi: $\varepsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$

- $\varepsilon\text{-closure}(q_1)$:
 - Da q_1 , puoi raggiungere q_2 attraverso una transizione ε

Quindi: $\varepsilon\text{-closure}(q_1) = \{q_1, q_2\}$

- $\varepsilon\text{-closure}(q_2)$:
 - q_2 non ha transizioni ε in uscita

Quindi: $\varepsilon\text{-closure}(q_2) = \{q_2\}$

Qui è presentato l'algoritmo per calcolare la ε -closure:

Claim 4.2.2

usando le ε -closure, si può definire il linguaggio riconosciuto da un NFA in modo elegante.

Definiamo la seguente funzione

$$\hat{\delta} : Q \times \Sigma^* Q(P)$$

Algorithm 1: ϵ -closure

Input: Stato p
Output: insieme di stati raggiungibili da p con mosse ϵ

```
1  $T \leftarrow P;$                                 // inizializzazione
2  $\epsilon\text{-closure}(p) = P;$                   // inizializzazione
3 while  $T \neq \emptyset$  do
4   scegli un  $r \in T$  e rimuovilo da  $T$ ;
5   foreach  $s \in \delta(r, \epsilon)$  do
6     if  $s \neq \epsilon\text{-closure}(p)$  then
7       add  $s$  to  $\epsilon\text{-closure}(p)$ ;
8       add  $s$  to  $T$ ;
```

per in induzione

$$\hat{\delta}(q, \epsilon) = \epsilon\text{-closure}(q)$$

$$\hat{\delta}(q, xa) = \epsilon\text{-closure}(q) \text{ dove } P = \{p \in Q \mid \exists r \in \hat{\delta}(q, x) \wedge p \in (r, a)\}$$

Pertanto si può dimostrare (dimostrazione difficile) che:

$$w \in L[N] \iff \exists p \in Ft.c.p \in \hat{\delta}(q_0, w)$$

funzione mossia

Definition 4.2.8: funzione mossia

Si definisce la **funzione mossia** come estensione della funzione di transizione δ di un NFA come:

$$\begin{aligned} mossia : \mathcal{P}(Q) \times \Sigma &\rightarrow \mathcal{P}(Q) \\ mossia(P, a) &= \bigcup_{p \in P} \delta(p, a) \end{aligned}$$

Cioè l'insieme delle mosse a da un insieme di nodi P è l'unione di tutte le mosse a di ogni nodo dell'insieme.

funzione transazione

Definition 4.2.9: funzione transizione

Si definisce **transizione** e la si denota con Δ tale funzione:

$$\Delta(A, b) = \epsilon\text{-closure}(mossa(A, b))$$

Con A un insieme di stati e b una transizione

costruzione per sottoinsiemi del DFA

Qua di seguito l'algoritmo per la costruzione di sottoinsiemi, che serve per passare da un NFA a un DFA equivalente. Definiamo quindi il DFA equivalente:

Algorithm 2: costruzionePerSottoinsiemi()

Input: NFA $N = (\Sigma, Q, \delta, q_0, F)$
Output: DFA $M_N = (\Sigma, T, \Delta, \varepsilon\text{-closure}(q_0), \mathcal{F})$

```

1  $S \leftarrow \varepsilon\text{-closure}(q_0);$                                 // stato iniziale del DFA
2  $T = \{S\};$                                                  //  $T$  è il l'insieme degli stati del DFA
3 while c'è un  $P \in T$  non marcato do
4   marca  $P;$ 
5   foreach  $a \in \Sigma$  do
6      $R \leftarrow \varepsilon\text{-closure}(\text{mossa}(P, a));$ 
7     if  $R \notin T$  then
8       add  $R$  to  $T;$                                          //  $R$  non ha marca
9     definisci  $\Delta(P, a) = R;$ 

```

Definition 4.2.10: DFA equivalente

Si definisce il DFA equivalente tale automa:

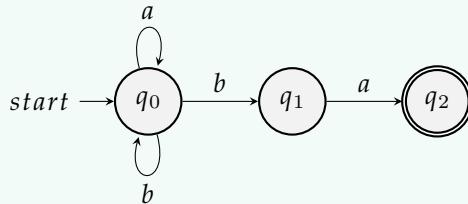
$$M_n = (\Sigma, T, \Delta, \varepsilon\text{-closure}(q_0), \mathcal{F})$$

dove $R \in \mathcal{F} \iff \exists q \in R \text{ con } q \in F$

Si può notare come l'algoritmo che definisce la funzione di transizione (vista poch'anzì) rispetta le limitazioni dei DFA: non ci sono mosse epsilon e per ogni caraterè dell'alfabeto esiste una ed una sola mossa in qualunque stato

Example 4.2.4

Consideriamo l'NFA seguente (relativo all'espressione regolare $((a|b)^*ab)$):



Vogliamo ora trovare un DFA equivalente ad esso, applichiamo l'algoritmo di costruzione per sottoinsiemi:

- calcoliamo lo stato iniziale $S = \varepsilon\text{-closure}(q_0)$, che è $S = q_0$.
- Creiamo T e gli inseriamo S marcandolo
- Per ogni simbolo dobbiamo calcolare $\varepsilon\text{-closure}(\text{mossa}(P, a))$, ma visto che l'alfabeto è $\{a, b\}$ lo facciamo 2 volte:
 - per il carattere a calcoliamo con $P = S = \{q_0\}$:

$$\begin{aligned} R &= \varepsilon\text{-closure}(\text{mossa}(P, a)) = \varepsilon\text{-closure}\left(\bigcup_{p \in \{q_0\}} \delta(p, a)\right) \\ &= \varepsilon\text{-closure}(\delta(q_0, a)) = \varepsilon\text{-closure}(\{q_0\}) = \{q_0\} \end{aligned}$$

Si definisca $\delta(S, a) = \{q_0\}$ e visto che $\{q_0\} \in T$ non lo andiamo a riaggiungere

- per il carattere a calcoliamo con $P = S = \{q_0\}$:

$$\begin{aligned} R &= \varepsilon\text{-closure}(\text{mossa}(P, b)) = \varepsilon\text{-closure}\left(\bigcup_{p \in \{q_0\}} \delta(p, b)\right) \\ &= \varepsilon\text{-closure}(\delta(q_0, b)) = \varepsilon\text{-closure}(\{q_0, q_1\}) = \{q_0, q_1\} \end{aligned}$$

visto che $\{q_0, q_1\} \notin T$ lo andiamo ad aggiungere

- si calcoli ora con $P = \{q_0, q_1\}$ andando a calcolare la stessa cosa:

– per a si ha:

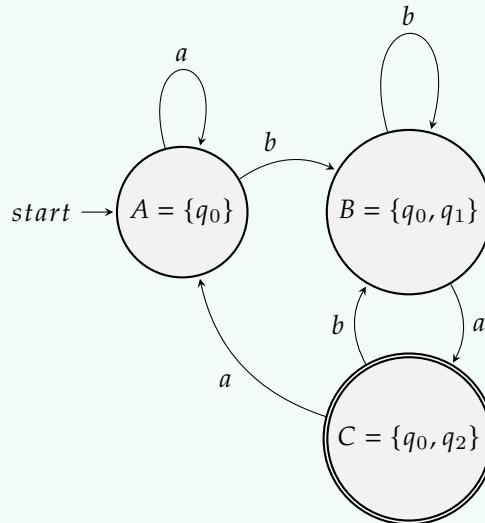
$$\begin{aligned} R &= \varepsilon\text{-closure}(mossa(P, a)) = \varepsilon\text{-closure}\left(\bigcup_{p \in \{q_0, q_1\}} \delta(p, a)\right) \\ &= \varepsilon\text{-closure}(\{q_0\} \cup \{q_2\}) = \{q_0, q_2\} \end{aligned}$$

quindi $\Delta(\{q_0, q_1\}, a) = \{q_0, q_2\}$. Visto $\{q_0, q_2\} \notin T$ lo aggiungiamo

– per b si ha $R = \{q_0, q_1\}$. Quindi $\Delta(\{q_0, q_1\}, b) = \{q_0, q_1\}$ e non lo si raggiunge

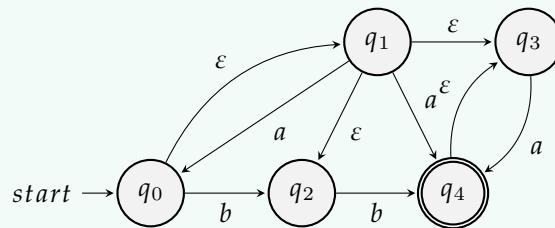
- ripetiamo per $P = \{q_0, q_2\}$ (salto i calcoli)

alla fine di sto ambaradam si ottiene DFA $N' = (\Sigma, \{\{q_0\}, \{q_0, q_1\}, \{q_0, q_2\}\}, \Delta, \{q_0\}, \{q_0, q_2\})$ che in forma di diagramma di transizione e:



esempio bello corposo:

Example 4.2.5

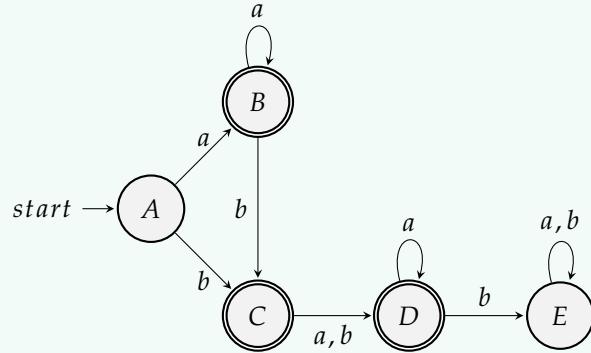


si noti i seguenti calcoli

$$\begin{aligned} A &= \varepsilon\text{-closure}(q_0) = \{q_0, q_1, q_2, q_3\} \\ \Delta(A, a) &= \varepsilon\text{-closure}(\text{move}(A, a)) = \varepsilon\text{-closure}(\{q_0, q_4\}) = \{q_0, q_1, q_2, q_3, q_4\} = B \\ \Delta(A, b) &= \varepsilon\text{-closure}(\text{move}(A, b)) = \varepsilon\text{-closure}(\{q_2, q_3, q_4\}) = \{q_2, q_3, q_4\} = C \\ \Delta(B, a) &= \varepsilon\text{-closure}(\text{move}(B, a)) = \varepsilon\text{-closure}(\{q_0, q_4\}) = B \\ \Delta(B, b) &= \varepsilon\text{-closure}(\text{move}(B, b)) = \varepsilon\text{-closure}(\{q_2, q_4\}) = C \\ \Delta(C, a) &= \varepsilon\text{-closure}(\text{move}(C, a)) = \varepsilon\text{-closure}(\{q_4\}) = \{q_3, q_4\} = D \\ \Delta(C, b) &= \varepsilon\text{-closure}(\text{move}(C, b)) = \varepsilon\text{-closure}(\{q_4\}) = D \\ \Delta(D, a) &= \varepsilon\text{-closure}(\text{move}(D, a)) = \varepsilon\text{-closure}(\{q_4\}) = \emptyset = D \\ \Delta(D, b) &= \varepsilon\text{-closure}(\text{move}(D, b)) = \varepsilon\text{-closure}(\emptyset) = \emptyset = E \\ \Delta(E, a) &= \varepsilon\text{-closure}(\text{move}(E, a)) = \varepsilon\text{-closure}(\emptyset) = \emptyset = E \end{aligned}$$

$$\Delta(E, b) = \epsilon\text{-closure}(\text{move}(E, b)) = \epsilon\text{-closure}(\emptyset) = \emptyset = E$$

il cui DFA M_N è:

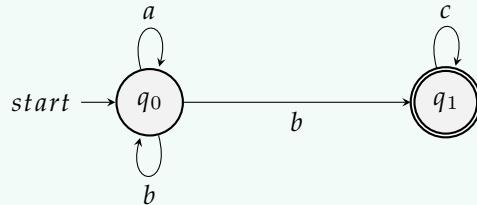


casi pessimi

Claim 4.2.3

Example 4.2.6

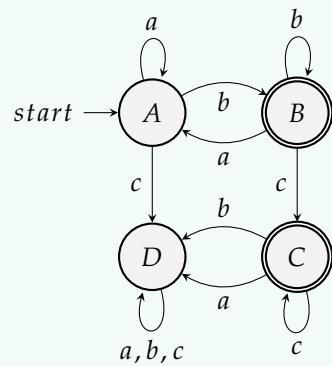
Sia N il seguente NFA con 2 stati:



per costruire il suo M_N si hanno i seguenti nodi

- $A = \{q_0\}$
- $B = \{q_0, q_1\}$
- $C = \{q_1\}$
- $D = \emptyset$

in diagramma di transazione diventa:



Adesso si può dimostrare il teorema dell'equivalenza

teorema d'equivalenza tra NFA e DFA

Theorem 4.2.1 Bonzo-GioLaPalma

Sia $N = (\Sigma, Q, \delta, q_0, F)$ un NFA e sia M_n l'automa ottenuto con la costruzione per sottoinsiemi. allora M_n è un DFA e si ha che

$$L[N] = L[M_n]$$

In altre parole, N e M_n sono equivalenti.

Dimostrazione: Sia $N = (Z, \Sigma, \delta, q_0, F)$ un NFA e sia $M_N = (\Gamma, \Sigma, \Delta, A, F')$ l'automa ottenuto con l'algoritmo.

- M_N è **deterministico**: Infatti $\Delta(A, a)$ con $A \in T \wedge a \in \Sigma$ è definita in modo univoco
- Quindi mi riduco a dimostrare che $L[N] = L[M_n]$
 - si osservi che per un DFA, $\varepsilon\text{-closure}(R) = R$
 - chiamiamo $i_m = \varepsilon\text{-closure}(q_0)$ lo stato iniziale di M_n

Vogliamo dimostrare che $\forall w \in \Sigma^*$

$$\delta(q_0, w) = \Delta(A, w)$$

per induzione sulla lunghezza di w

- **Caso base:** $|w| = 0$ cioè $w = \varepsilon$:

$$\hat{\delta}(A, \varepsilon) = \varepsilon\text{-closure}(q_0)$$

$$\hat{\Delta}(i_m, \varepsilon) = \varepsilon\text{-closure}(i_m) = i_M = < \varepsilon\text{-closure}(q_0)$$

- **Caso induttivo:** $w = xa$ con $x \in \Sigma^*, a \in \Sigma$ Per ipotesi induttiva, sappiamo che:

$$\hat{\delta}(q_0, x) = \Delta(\{q_0\}, x) = \{p_1, \dots, p_k\}$$

Per definizione di $\hat{\delta}$:

$$\hat{\delta}(q_0, xa) = \Delta(\varepsilon\text{-closure}(\bigcup_{i=1}^k \delta(p_i, a)), a)$$

Similmente:

$$\Delta(\{q_0\}, xa) = \Delta(\{p_1, \dots, p_k\}, a)$$

In base all'algoritmo, la definizione di Δ ci dice che:

$$\Delta(\{p_1, \dots, p_k\}, a) = \varepsilon\text{-closure}(\text{mossa}(\{p_1, \dots, p_k\}, a)) = \varepsilon\text{-closure}(\bigcup_{i=1}^k \delta(p_i, a)) = \hat{\delta}(q_0, xa)$$

OK.

—

Infine,abbiamo che:

$$w \in L[N] \iff \exists p \in \hat{\delta}(q_0, w) \text{ con } p \in F$$

$$\iff \exists p \in \Delta(i_M, w) \text{ con } p \in F$$

$$\iff w \in L[M_N] \quad \forall w \in \Sigma^*$$

Quindi:

$$L[N] = L[M_N] \quad \text{c.v.d.}$$



4.2.5 Da espressioni regolari a NFA equivalenti

Theorem 4.2.2 Basta - Dario è sVenuto

Data un'espressione regolare S , possiamo costruire un NFA $N[S]$ tale che:

$$\mathcal{L}[S] = L[N[S]]$$

ovvero un linguaggio individuato da un linguaggio regolare è equivalente ad un linguaggio riconosciuto da un automa non deterministico a stati finiti costruito a partire da S , questo significa che **gli NFA riconoscono TUTTI i linguaggi regolari**, vedremo poi che riconoscono solo i linguaggi regolari

Dimostrazione: Dimostreremo il teorema per induzione sulla sintassi astratta della espressione regolare S . Si costruisca un possibile NFA associato all'espressione regolare S , in modo da mantenere i seguenti due invarianti:

- lo stato iniziale non ha archi entranti
- $N[S]$ ha un solo stato finale senza archi uscenti

Procedo ad esaminare i vari casi

- Sia $s = \emptyset$

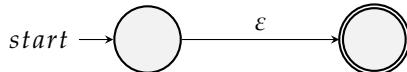
In questo caso sarà possibile costruire un NFA con due stati (iniziale e finale) non connessi, quindi $N[s]$:



Si osservi che $\mathcal{L}[\emptyset] = \emptyset = L[N[S]]$

- Sia $s = \epsilon$

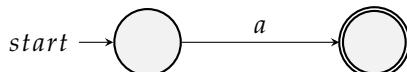
In questo caso sarà possibile costruire un NFA con due stati (iniziale e finale) connessi da un ϵ , quindi $N[s]$:



Si osservi che, in questo caso, $\mathcal{L}[\epsilon] = \{\epsilon\} = L[N[S]]$

- Sia $s = a$

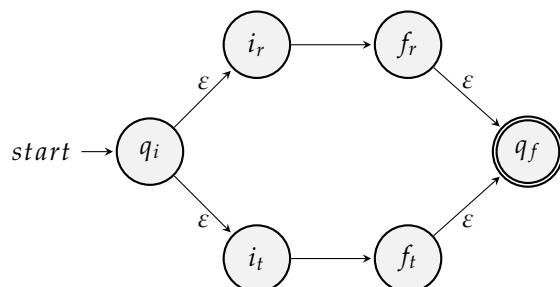
In questo caso sarà possibile costruire un NFA con due stati (iniziale e finale) connessi da una transizione a , quindi $N[S]$:



Si osservi che, in questo caso, $\mathcal{L}[a] = \{a\} = L[N[S]]$

- caso $s = r|t$

Ipotizziamo di avere già costruito gli automati $N[t]$ ed $N[r]$, per costruire "l'or" è possibile partire da uno stato iniziale i e collegarlo con transizioni ϵ ai due automati, da entrambi si confluisce in uno stato finale f con transizioni ϵ

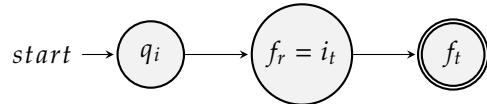


Algebricamente si può dimostrare che $\mathcal{L}[r|t] = \mathcal{L}[t] \cup \mathcal{L}[r]$
e per ipotesi induttiva si ha:

$$\mathcal{L}[t] \cup \mathcal{L}[r] = L[N[r]] \cup L[N[t]] = L[N[r|t]]$$

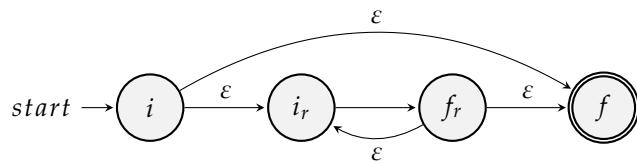
- Sia $s = r \cdot t$

Ipotizziamo di avere già costruito gli automi $N[t]$ ed $N[r]$, per costruire la concatenazione è possibile fondere lo stato finale del primo automa con quello del secondo automa, si ha, quindi, che $N[s]$:



- Sia $s = r^*$

Ipotizzando di aver già costruito $N[r]$ è possibile costruire l'automa creando un ciclo ε dalla fine di $N[r]$ al suo inizio



Si osservi che:

$$\mathcal{L}[r^*] = (\mathcal{L}[r])^*$$

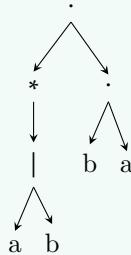
per ipotesi induttiva

$$(\mathcal{L}[r])^* = (L[N[r]])^* = L[N[r^*]]$$

©

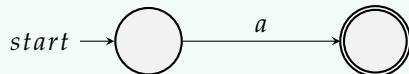
Example 4.2.7

Sia $S = (a|b^*)ba$ un'espressione regolare e costruiamnno l'albero sintattico

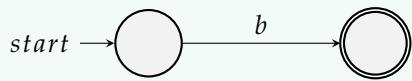


Partiamo dalle foglie e risaliamo alla radice (bottom-up) Si ha:

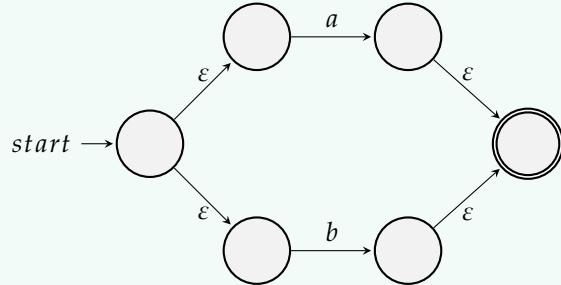
- $N[a]$:



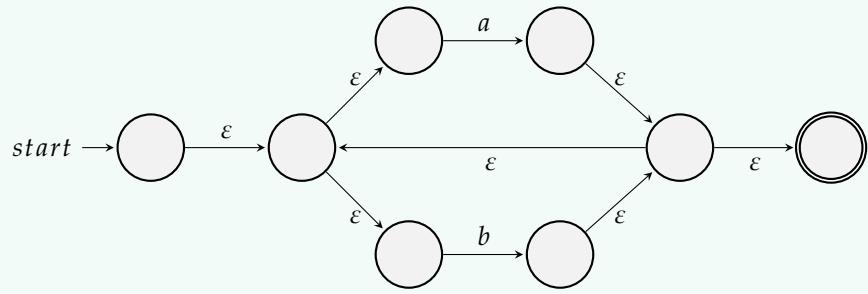
- $N[b]$:



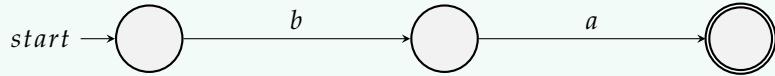
- $N[a|b]$:



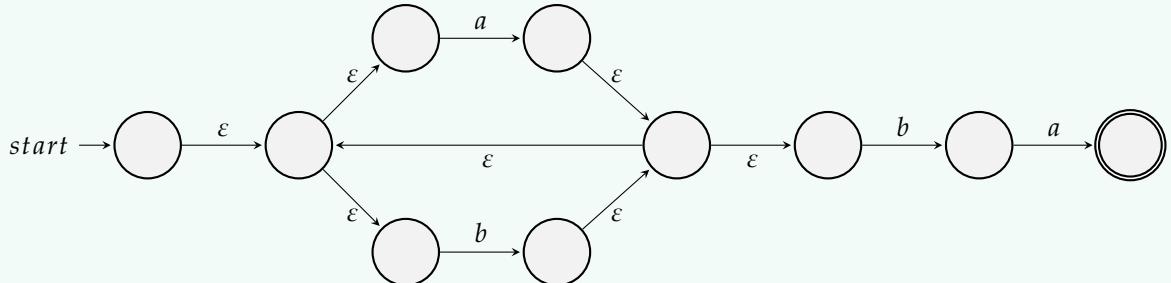
- $N[(a|b)^*]$:



- $N[ba]$:



- $N[s]$:



Altro esempio:

Example 4.2.8

TODO da fare l'automa diretto qui

Costruiamo il DFA associato a $N[s]$:

$$A = \varepsilon\text{-closure}(0) = \{0\}$$

$$\Delta(A, a) = \varepsilon\text{-closure}(\{1\}) = \{1, 2, 3\} = B$$

$$\Delta(A, b) = \varepsilon\text{-closure}(\emptyset) = \emptyset = C$$

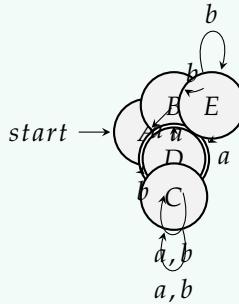
$$\Delta(B, a) = \varepsilon\text{-closure}(\{5\}) = \{5\} = D$$

$$\Delta(B, b) = \varepsilon\text{-closure}(\{3\}) = \{3, 4, 2\} = E$$

$$\Delta(D, a) = \emptyset = C, \quad \Delta(D, b) = \emptyset = C$$

$$\Delta(E, a) = \varepsilon\text{-closure}(\{5\}) = D, \quad \Delta(E, b) = \varepsilon\text{-closure}(\{3\}) = E$$

$$\Delta(C, a) = \emptyset = C, \quad \Delta(C, b) = \emptyset = C$$



DFA associato a $N[a \ b^* \ a]$

4.2.6 Da NFA a espressione regolare

Aggiungo una dimostrazione presa dal libro di Michael Sipser, "Introduction to the Theory of Computation", che descrive come creare un'espressione regolare partendo direttamente da un NFA. Nella sezione successiva introdurremo le grammatiche regolari con le quali e' possibile dimostrare lo stesso teorema, quindi questa parte non e' necessaria ma solo un'idea interessante e piu' diretta.

Come anticipato precedentemente, e' possibile anche dimostrare la direzione inversa del teorema, ovvero che dato un NFA che riconosce un linguaggio $L[N]$, e' sempre possibile generare un'espressione regolare che riconosca lo stesso linguaggio. Quindi vale il teorema generale:

Theorem 4.2.3 Teorema Piccolomini

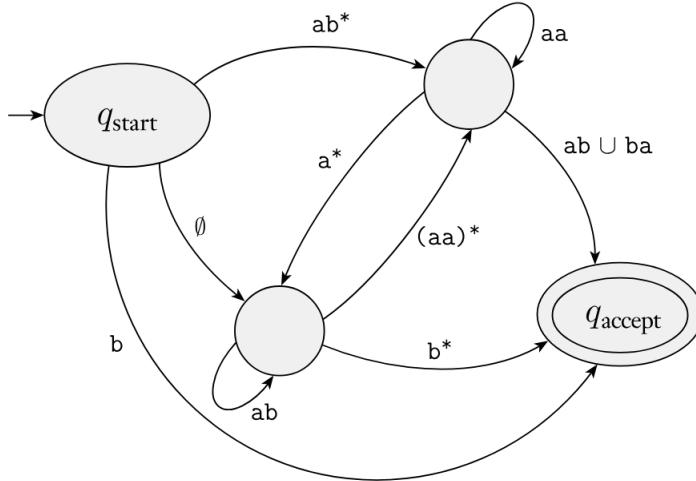
Un linguaggio e regolare sse esiste un automa a stati finiti che lo riconosce.

Sopra abbiamo gia dimostrato che preso un linguaggio regolare (ovvero descritto da un'espressione regolare) possiamo sempre costruire un NFA equivalente. Ci tocca dimostrare la direzione inversa, ovvero che preso un qualsiasi NFA, il linguaggio che riconosce e' regolare e quindi esiste un'espressione regolare che lo descrive.

Dato che abbiamo dimostrato col teorema Bonzo-GiolaPalma che un NFA e' sempre convertibile a un DFA, possiamo partire proprio da un DFA e definire un procedimento per ottenere un'espressione regolare equivalente. Dividiamo in due parti questo processo: prima convertiamo il DFA in un GNFA (General Nondeterministic Finite Automata), che ora definiremo, poi convertiamo il GNFA in un'ER.

Automi finiti nondeterministici generali

I GNFA sono praticamente dei NFA con funzione di transizione che funziona utilizzando le ER. Possono leggere blocchi alla volta dall'input, non solo caratteri, e se tale blocco soddisfa l'ER di una transizione che parte dallo stato corrente, allora si puo' seguire l'arco e cambiare stato. Ovviamente e' fortemente nondeterministico e possono esserci molti modi per elaborare la stessa sequenza di caratteri.



Formalmente:

Definition 4.2.11: GNFA

Un **automa finito nondeterministico generale** e' una quintupla $(Q, \Sigma, \delta, q_{start}, q_{accept})$ tale che:

- Q e' un insieme finito di stati
- Σ e' l'alfabeto dell'input
- $\delta : (Q - q_{accept}) \times (Q - q_{start}) \rightarrow \mathcal{R}_\Sigma$ e' la funzione di transizione

Dove \mathcal{R}_Σ e' l'insieme di tutte le ER sull'alfabeto Σ .

Un GNFA riconosce una stringa $w \in \Sigma^*$ se esistono una suddivisione $w = w_1w_2...w_k$ con $w_i \in \Sigma^*$ e una serie di stati q_0, q_1, \dots, q_k tali che:

1. $q_0 = q_{start}$
2. $q_k = q_{accept}$
3. $\forall i > 0. w_i \in [R_i]$, dove $R_i = \delta(q_{i-1}, q_i)$

Per i fini della dimostrazione, ci interessiamo solo di GNFA specifici che rispettano le seguenti condizioni:

- Lo stato iniziale ha archi che vanno verso tutti gli altri stati, ma nessuno entrante.
- Lo stato finale e' unico e ha archi entranti da tutti gli altri stati, ma nessuno uscente. Inoltre e' distinto dallo stato iniziale.
- Senza contare lo stato iniziale e lo stato finale, gli stati hanno archi che vanno verso tutti gli altri stati e anche verso loro stessi.

Da DFA a GNFA

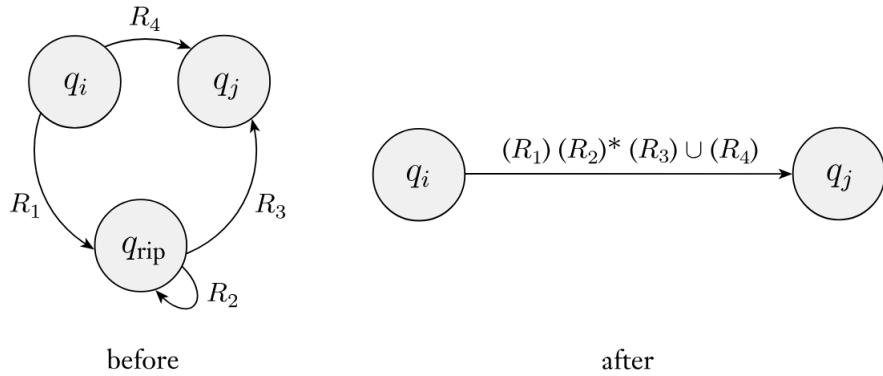
Convertire un DFA in un GNFA in forma speciale e' semplice: basta aggiungere uno stato iniziale con arco ε verso lo stato iniziale vecchio, aggiungere uno stato finale con archi ε entranti da tutti i vecchi stati finali e se ce piu' di un arco nella stessa direzione fra due nodi, fonderli insieme creando un unico arco la cui ER corrispondente e' l'unione delle ER dei vecchi archi. Infine, se mancano degli archi, aggiungere transizioni segnati dall'ER \emptyset , che non puo' essere mai usata.

Da GNFA a ER

Per ottenere l'ER a partire dal GNFA, l'idea principale e' la seguente:

Diciamo che il GNFA ha k stati. Sappiamo che $k \geq 2$ perche ci devono essere uno stato finale e uno iniziale distinti. Se riusciamo a rimuovere uno stato alla volta, che non sia finale o iniziale, mantenendo il nuovo GNFA appena ottenuto equivalente a quello prima, arriviamo ad un punto dove $k = 2$ e l'automa e' costituito semplicemente da un arco che applica l'ER che ci porta dallo stato iniziale direttamente a quello finale, e che descrive quindi il linguaggio riconosciuto dal DFA iniziale.

Dobbiamo fare in modo che quando uno stato q_{rip} viene rimosso, le ER sugli archi vengano modificate in modo da simulare tutti i percorsi che sono stati rimossi. Quindi le nuove ER devono considerare sia le stringhe che portano direttamente dallo stato q_i allo stato q_j , che le stringhe che portavano da q_i a q_j passando da q_{rip} .



Quindi, se nel vecchio GNFA:

1. q_i va da q_{rip} per l'ER R_1
2. q_{rip} va a se stesso per R_2
3. q_{rip} va a q_j per R_3
4. q_i va a q_j per R_4

allora nel nuovo GNFA dobbiamo etichettare l'arco da q_i a q_j con l'ER:

$$(R_1)(R_2)^*(R_3) \cup (R_4)$$

Ad ogni iterazione applichiamo questa trasformazione per ogni arco fra una coppia $q_i q_j$ (anche quando $i = j$). Formalizziamo tutto cio' (zio perone):

Dimostrazione formale: Sia M il DFA equivalente all'NFA del teorema. Usando la procedura descritta sopra trasformiamo M in un GNFA G . Usiamo la procedura ricorsiva $\text{CONVERT}(G)$ per ottenere l'ER equivalente:

1. Sia k il numero di stati di G
2. Se $k = 2$, ritorna l'espressione regolare associata ai due nodi
3. Se $k > 2$, seleziona uno stato $q_{\text{rip}} \in Q$ che sia diverso dallo stato iniziale e finale e sia $G' = (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ tale che:

- $Q' = Q \setminus \{q_{\text{rip}}\}$
- $\forall q_i \in Q' \setminus \{q_{\text{start}}\}, \forall q_j \in Q' \setminus \{q_{\text{accept}}\} :$

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$$

$$\text{con } R_1 = \delta(q_i, q_{\text{rip}}), R_2 = \delta(q_{\text{rip}}, q_{\text{rip}}), R_3 = \delta(q_{\text{rip}}, q_j), R_4 = \delta(q_i, q_j)$$

Ora dimostriamo che $\text{CONVERT}(G)$ ritorni il valore corretto.

Dimostriamo per induzione su k , il numero di stati del GNFA G :

- **Caso base:**

Se $k = 2$, per la definizione di GNFA specifico abbiamo uno stato iniziale e un arco che lo collega allo stato finale. L'ER dell'arco descrive tutte le stringhe che portano G allo stato finale, quindi è equivalente a G .

- **Caso induttivo su k :**

Assumiamo che $\text{CONVERT}(G')$ equivale a G' , dove G' è il caso con $k - 1$ stati. Dimostriamo che G' e G sono equivalenti. Prendiamo una parola $w \in \Sigma^*$ accettata da G . Quindi siano

$$q_{\text{start}}, q_1, q_2, q_3, \dots, q_{\text{accept}}$$

la sequenza di stati di G di un percorso che accetta w . Se q_{rip} (lo stato rimosso da G per creare G') non appartiene a questa sequenza, allora chiaramente w è riconosciuto anche da G' . Questo è perché i nuovi archi fra tutti gli stati mantengono l'ER originale tramite un'unione. Altrimenti, se q_{rip} fa parte del percorso, basta rimuovere tutte le evenienze dalla sequenza per trovare un percorso per cui G' accetta w . Questo è perché nella definizione di CONVERT , facciamo in modo che le nuove ER fra due nodi considerino descrivono anche tutte le stringhe che portavano fra i due stati passando da q_{rip} .

Al contrario, se G' riconosce w , ad ogni arco è associata un'ER che riconosce stringhe che in G fanno passare da q_i a q_j sia direttamente che tramite q_{rip} . Quindi ovviamente anche G riconosce w . Avendo dimostrato che G e G' sono equivalenti, possiamo usare l'ipotesi induttiva per dire che $\text{CONVERT}(G)$ equivale a G .



4.2.7 Chiusura dei linguaggi regolari rispetto alle operazioni regolari

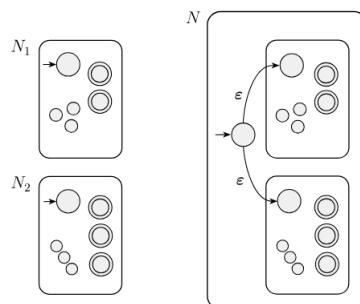
Per aiutarci ad architettare automi finiti o per distinguere linguaggi regolari da altri tipi di linguaggi che vedremo più avanti, puoi tornare utile sapere se sono chiusi rispetto alle operazioni regolari (concatenazione, unione, stella di Kleene). Generalmente, diciamo che un'insieme di oggetti è chiuso rispetto a un'operazione se applicando quest'ultima a qualunque elemento dell'insieme, il risultato appartiene sempre allo stesso insieme. Quindi se consideriamo come macro-insieme l'insieme di tutti i linguaggi regolari e dimostriamo la chiusura rispetto a una certa operazione, allora siamo sicuri che il risultato di questa operazione applicata a linguaggi regolari sarà sempre un linguaggio regolare. Partiamo con l'operazione di unione:

Theorem 4.2.4 Chiusura di linguaggi regolari rispetto all'unione

La classe dei linguaggi regolari è chiusa sotto l'unione.

In altre parole, se A_1 e A_2 sono linguaggi regolari, lo è anche $A_1 \cup A_2$.

Dato che abbiamo già dimostrato l'equivalenza fra DFA e NFA, possiamo usare il nondeterminismo per dimostrare il teorema sopra (è possibile anche usare solo DFA, ma è più lunga). L'idea della dimostrazione è quella di creare due NFA, N_1, N_2 che riconoscono rispettivamente i linguaggi A_1, A_2 , e poi di unirli per creare un nuovo NFA N che riconosca $A_1 \cup A_2$. Quindi, N deve riconoscere sia le stringhe di A_1 che di A_2 . Creando un nuovo stato iniziale e collegandolo ai due stati iniziali di N_1 e N_2 con mosse ϵ , quindi usando il nondeterminismo per accettare sia i casi in cui l'input appartiene ad A_1 , sia quando appartiene ad A_2 .



Dimostrazione formale: Siano $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ e $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$ NFA che riconoscono rispettivamente A_1 e A_2 .

Costruisco un nuovo NFA $N = \{Q, \Sigma, \delta, q_0, F\}$ per riconoscere $A_1 \cup A_2$ in tale modo:

$$1. Q = \{q_0\} \cup Q_1 \cup Q_2$$

$$2. F = F_1 \cup F_2$$

$$3. \forall q \in Q. \forall a \in \Sigma. \delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \wedge a = \epsilon \\ \emptyset & q = q_0 \wedge a \neq \epsilon \end{cases}$$

Quindi per il teorema Basta-Dario e' sVenuto, $A_1 \cup A_2$ e' un linguaggio regolare. \circlearrowright

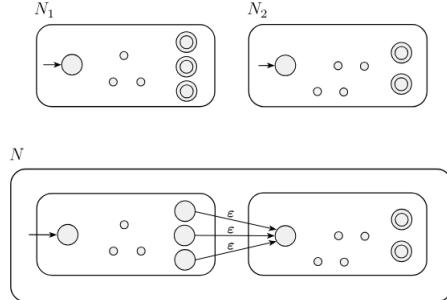
Ora passiamo alla concatenazione:

Theorem 4.2.5 Chiusura dei linguaggi regolari rispetto alla concatenazione

I linguaggi regolari sono chiusi sotto la concatenazione.

In altre parole, se A_1 e A_2 sono linguaggi regolari, anche $A_1 \cdot A_2$ e' regolare.

Il punto chiave di questa dimostrazione e' come sapere quando finisce la parola di A_1 e inizia una stringa di A_2 . Infatti, con i DFA risulta molto piu' complicato, ma dato che possiamo usare i NFA risulta abbastanza banale. Come prima, vogliamo costruire un NFA N utilizzando altri due NFA N_1 e N_2 che riconoscono A_1 e A_2 . Possiamo fare cio' collegando tutti gli stati finali di N_1 allo stato iniziale di N_2 , usando transizioni ϵ per accettare tutti i possibili casi. E' possibile, infatti, che una sotto-stringa di una parola in A_1 appartenga anch'essa ad A_1 , quindi senza il nondeterminismo sarebbe impossibile sapere quando passare a N_2 . Se alla fine dell'input N_2 e' su uno stato finale, allora la parola appartiene a $A_1 \cdot A_2$.



Dimostrazione formale: Siano $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$, $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$ NFA che riconoscono rispettivamente A_1, A_2 .

Costruisco il NFA $N = \{Q, \Sigma, \delta, q_1, F_2\}$ che riconosce $A_1 \cdot A_2$ in tale modo:

$$1. Q = Q_1 \cup Q_2$$

$$2. \forall q \in Q. \forall a \in \Sigma : \delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \setminus F_1 \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \wedge a = \epsilon \\ \delta_1(q, a) & q \in F_1 \wedge a \neq \epsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

Sempre per il teorema Basta-Dario e' sVenuto, $A_1 \cdot A_2$ e' regolare. \circlearrowright

Finiamo con la stella di Kleene:

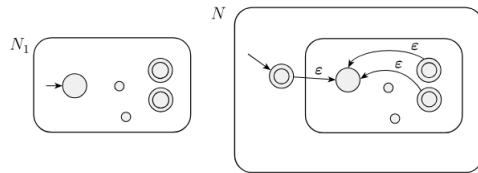
Theorem 4.2.6 Chiusura dei linguaggi regolari rispetto alla stella di Kleene

I linguaggi regolari sono chiusi rispetto alla stella di Kleene.

In altre parole, se A_1 e' un linguaggio regolare, allora lo e' anche A_1^* .

Come nei primi due casi, vogliamo dimostrare per costruzione che esiste un NFA N che riconosce A_1^* . Considero l' automa N_1 che riconosce A_1 . Noi vogliamo che N riconosca una concatenazione di qualunque lunghezza di

parole in A_1 , quindi possiamo usare l'idea della dimostrazione precedente e collegare tutti gli stati finali di N_1 questa volta al proprio stato iniziale, sempre con transizioni ϵ . Non possiamo dimenticarci però che anche $\epsilon \in A_1^*$, quindi basta aggiungere un nuovo stato iniziale che sia pure finale e collegarlo al vecchio stato iniziale con un'altra transizione ϵ .



Dimostrazione formale: Sia $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ un NFA che riconosce il linguaggio regolare A_1 . Costruisco $N = \{Q, \Sigma, \delta, q_0, F\}$ che riconosce A_1^* nel seguente modo:

1. $Q = \{q_0\} \cup Q_1$
2. $F = \{q_0\} \cup F_1$

$$3. \forall q \in Q, \forall a \in \Sigma : \delta(q, a) = \begin{cases} \delta(q, a) & q \in Q_1 \setminus F_1 \\ \delta(q, a) & q \in F_1 \wedge a \neq \epsilon \\ \delta(q, a) \cup \{q_1\} & q \in F_1 \wedge a = \epsilon \\ \{q_1\} & q = q_0 \wedge a = \epsilon \\ \emptyset & q = q_0 \wedge a \neq \epsilon \end{cases}$$

Gia lo sapete cosa ci dicono il Basta e il Darione, A_1^* è regolare. ∅

TODO: intersezione

4.3 Grammatiche regolari

Definition 4.3.1: grammatiche regolari

Una grammatica libera si definisce **regolare** sse ogni produzione è della forma $V \rightarrow aW$ oppure $V \rightarrow a$ dove $V, W \in NT \wedge a \in T$. Per il simbolo iniziale S è ammessa anche la produzione $S \rightarrow \epsilon$

Note:

A volte si userà la definizione più lasca che permette produzione $V \rightarrow \epsilon$ anche per nonterminali diversi da S

Di seguito riportati degli esempi

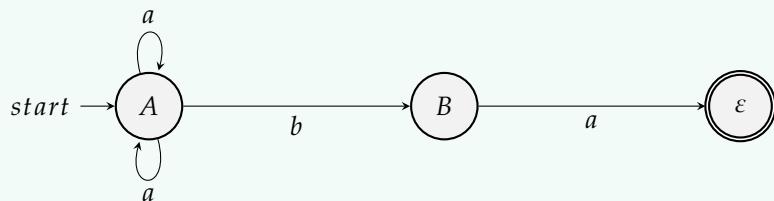
Example 4.3.1

Sia G la seguente grammatica regolare:

$$\begin{aligned} A &\rightarrow aA \mid bB \mid ba \\ B &\rightarrow a \end{aligned}$$

Si ha che $L(G) = (A|b)^*ba!!$

Adesso si costruisca un NFA a G :



si ha che:

- abbiamo associato a ogni nonterminale uno stato dell'NFA
- abbiamo aggiunto stato finale ϵ

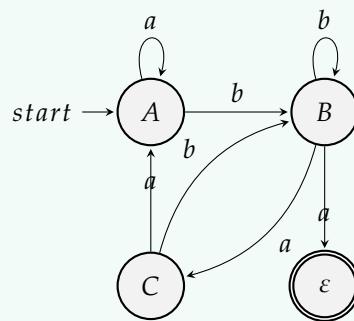
Esempio 2:

Example 4.3.2

Sia G la seguente grammatica regolare:

$$\begin{aligned} A &\rightarrow aA \mid bB \\ B &\rightarrow bB \mid aC \mid a \\ C &\rightarrow aA \mid bB \end{aligned}$$

l'NFA è:



4.3.1 da grammatiche regolari a NFA equivalenti

Theorem 4.3.1

Data una grammatica regolare G si può costruire un NFA N_G equivalente

Dimostrazione: Sia $G = (NT, T, R, S)$, definiamo inoltre $N_G = (T, Q, \delta, S, \{\epsilon\})$ come segue:

- $Q = NT \cup \{\epsilon\}$
- δ definita come segue:
 - $(V \rightarrow aZ \in R) \implies (Z \in \delta(V, a))$
 - $(V \rightarrow a \in R) \implies (\epsilon \in \delta(V, a))$
 - $(V \rightarrow \epsilon \in R) \implies (\epsilon \in \delta(V, \epsilon))$

Si può dimostrare che

$$S \implies_G^* w \text{ con la grammatica } G \iff (S, w) \vdash_{N_G}^* \text{ con l'automa } N_G$$

non verrà dimostrata quest'ultima parte cazzo



4.3.2 Da DFA a grammatiche regolari

Theorem 4.3.2

Da un DFA M possiamo definire una grammatica regolare G_M tale che:

$$L[M] = L(G_M)$$

Dimostrazione: Sia $\mathcal{M} = (\Sigma, Q, \delta, q_0, F)$ il DFA. La grammatica $G_{\mathcal{M}} = (Q, \Sigma, R, q_0)$ ha:

- **Per non terminali:** gli stati di \mathcal{M} .
- **Per terminali:** l'alfabeto di \mathcal{M} .
- **Per simbolo iniziale:** lo stato iniziale di \mathcal{M} .
- **Per produzioni R :**
 - Per ogni $\delta(q_i, a) = q_j$, la produzione $q_i \rightarrow aq_j \in R$.
 - Inoltre, se $q_j \in F$, anche $q_i \rightarrow a \in R$.
 - Se $q_0 \in F$, allora $q_0 \rightarrow \varepsilon \in R$.

Versione alternativa che usa la definizione di grammatica regolare più lasca:

- Per ogni $\delta(q_i, a) = q_j$, la produzione $q_i \rightarrow aq_j \in R$.
- Se $q \in F$, allora $q \rightarrow \varepsilon \in R$.

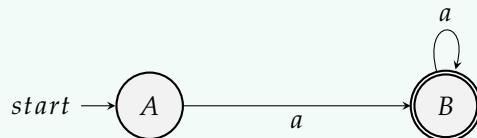
Si può dimostrare che:

$$w \in L[\mathcal{M}] \iff w \in L(G_{\mathcal{M}})$$



Example 4.3.3

Sia M :



Si ha che aa^* è l'espressione regolare che descrive $L[M]$

Secondo la costruzione appena descritta G_M è:

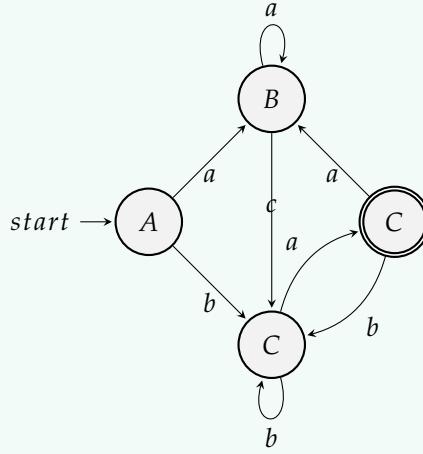
$$\begin{aligned} A &\rightarrow aB \mid a \\ B &\rightarrow aB \mid a \end{aligned}$$

secondo la variante, invece:

$$\begin{aligned} A &\rightarrow aB \\ B &\rightarrow aB \mid \varepsilon \end{aligned}$$

Example 4.3.4

sia il seguente automa M :



secondo il metodo descritto si ha che G_M è:

$$\begin{aligned} A &\rightarrow aB \mid bC \\ B &\rightarrow aB \mid bC \\ C &\rightarrow bC \mid aD \mid a \\ D &\rightarrow aB \mid bC \end{aligned}$$

alternativamente:

$$A \rightarrow aB \mid bCB \rightarrow aBmidbCC \rightarrow bC \mid aDD \rightarrow aB \mid bC \mid \epsilon$$

4.3.3 Grammatiche regolari ed espressioni regolari

Theorem 4.3.3

Il linguaggio definito da una grammatica regolare G è un linguaggio regolare, cioè è possibile costruire una espressione regolare S_G tale che:

$$L(G) = \mathcal{L}[S_G]$$

sketch della dimostrazione: idea della prova:

- **caso semplice:** un solo non terminale:

$$A \rightarrow aA \mid b \mid \epsilon$$

è intuitivo vedere che $a^*(b|\epsilon)$ è la espressione regolare associata

- **caso medio:** due non terminali

$$\begin{aligned} A &\rightarrow aA \mid bB \mid c \\ B &\rightarrow cA \mid aB \mid d \end{aligned}$$

ricaviamo B dalla seconda equazione

$$B \approx a^*(cA|d)$$

dove A compare nella espressione regolare

ora sostituiamo B nella prima "equazione"

$$A \approx aA \mid ba^*(cA \mid d) \mid c$$

con opportune manipolazioni su espressioni regolari, usando leggi che abbiamo visto, possiamo scrivere

$$A \approx aA \mid ba^*cA \mid ba^*d \mid c$$

e quindi

$$a \approx (a \mid ba^*c)A \mid ba^*d \mid c$$

ora siamo nella forma "semplice"-

A ha associata la espressione regolare

$$(a \mid ba^*c)^*(ba^*d \mid c)$$

- in generale

$$A_1 \simeq a_{11}A_1 \mid \cdots \mid a_{1m}A_m \mid b_1 \mid \cdots \mid b_{1p_1}$$

$$A_2 \simeq a_{21}A_1 \mid \cdots \mid a_{2n}A_n \mid b_{21} \mid \cdots \mid b_{2p_2}$$

⋮

$$A_m \simeq a_{m1}A_1 \mid \cdots \mid a_{mn}A_n \mid b_{n1} \mid \cdots \mid b_{np_n}$$

Si parte con:

$$A_m \simeq S_m[A_1, \dots, A_{n-1}]$$

cioè si costruisce una espressione regolare per A_m che usa A_1, \dots, A_{n-1} .

Poi si procede sostituendo A_n (o meglio $S_n[A_1, \dots, A_{n-1}]$) al posto di A_n nell'equazione per A_{n-1} , cioè:

$$A_{n-1} \simeq S_{n-1}[A_1, \dots, A_{n-2}]$$

e così via fino ad arrivare ad A_1 (che è il simbolo iniziale).



Example 4.3.5

Sia G la seguente grammatica

$$\begin{aligned} A &\rightarrow aB \mid \epsilon \\ B &\rightarrow bA \mid \epsilon \end{aligned}$$

Per B la espressione regolare associata è $(bA \mid \epsilon)$, sostituisco questa al posto di B nella prima equazione

$$A \approx a(bA \mid \epsilon) \mid \epsilon$$

manipolo la espressione regolare

$$A \approx abA \mid a \mid \epsilon$$

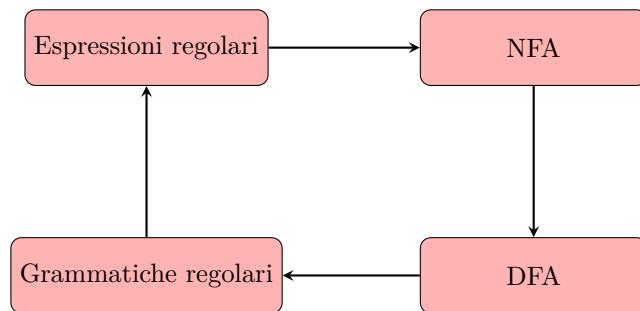
siamo ora nel caso semplice

$$A \approx (ab)^*(a \mid \epsilon)_{S_G}$$

Note:

la grammatica regolare G con unica prodizione $A \rightarrow aA$ definisce il linguaggio vuoto, non a^* quindi $S_G = \emptyset$

4.3.4 Per riassumere le relazioni tra espressioni regolari, linguaggi regolari e automi NFA e DFA



Claim 4.3.1

Si osservi che tutti questi formalismi sono equivalenti

Tutti generano/riconoscono/descrivono la stessa classe di linguaggi, ovvero i linguaggi regolari

Tuttavia vi è un serio problema, infatti, per costruire uno scanner si parte dalla specifica dei pattern associati alle categorie sintattiche del linguaggio, mediante espressioni regolari. Per poi

$$\text{regexp} \rightarrow \text{NFA} \rightarrow \text{DFA}$$

ma se l'NFA ha n stati il DFA potrebbe averne al più 2^n , pertanto serve trovare un DFA equivalente più piccolo possibile

$$\min(\text{DFA}) \text{ minimizzato}$$

4.3.5 minimizzazione

Alle volte per minimizzare degli automi occorre individuare stati equivalenti nello stesso automa, eliminarli e fonderli insieme costruendo, così, l'automa minimo. Prima della definizione formale occorre definire la seguente notazione:

Definition 4.3.2: notazione $\hat{\delta}$

Per un DFA $N = (\Sigma, Q, \delta, q_0, F)$ si ha che

$$\begin{aligned}\hat{\delta} : Q \times \Sigma^* &\rightarrow Q \text{ è definita come} \\ \hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x)a)\end{aligned}$$

Si ha quindi

$$w \in L[N] \iff \hat{\delta}(q_0, w) \in F$$

equivalenza/indistinguibilità

Definition 4.3.3: stati equivalenti/indistinguibili

due stati si dicono q_1 e q_2 di un DFA N si definiscono **equivalenti (o indistinguibili)** se

$$\forall x \in \Sigma^* (\hat{\delta}(q_1, x) \in F \iff \hat{\delta}(q_2, x) \in F)$$

Simmetricamente si può definire la non equivalenza

Definition 4.3.4: non equivalenza/indistinguibilità

due stati q_1 e q_2 **non sono equivalenti (o indistinguibili)** se

$$\exists x \in \Sigma^* ((\hat{\delta}(q_1, x) \in F \wedge \hat{\delta}(q_2, x) \notin F) \vee (\hat{\delta}(q_1, x) \notin F \wedge \hat{\delta}(q_2, x) \in F))$$

q_1 e q_2 pertanto sono distinguibili

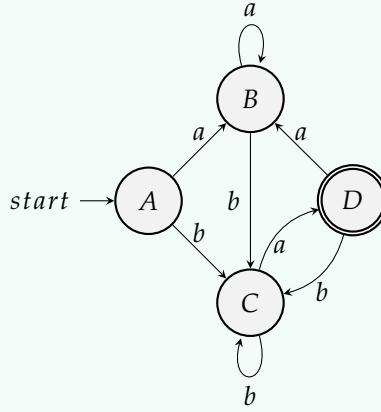
Note:

Una strategia per individuare la distinguibilità tra due stati è cercare di distinguere due stati a partire da $x \in \Sigma^*$ partendo dalla più corta ε

Si noti questo esempio

Example 4.3.6

Sia N il seguente DFA:



Inizialmente cercavo di vedere quali coppie di stati NON sono equivalenti, a partire dalla stringa ϵ :

- ϵ distingue ogni stato in F da ogni stato in $Q \setminus F$, dato che $\delta(q, \epsilon) = q$, sicuramente non sono equivalenti:

$$\{A, D\} \quad \{B, D\} \quad \{C, D\}$$

Infatti $A \notin F \wedge D \in F$, $B \notin F \wedge D \in F$ e $C \notin F \wedge D \in F$

- adesso si consideri le stringhe di lunghezza q , ovvero a e b

– si parti con la lettera a :

* a distingue B e C perché

$$\delta(B, a) = A \text{ e } \delta(C, a) = D$$

E (B, D) l'avevo già cancellata del primo punto

* a distingue anche A e C perché

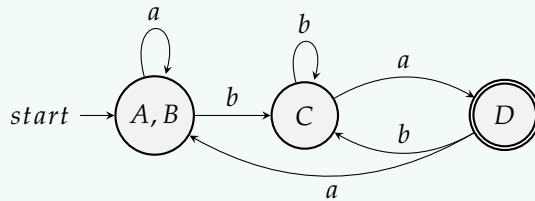
$$\delta(A, a) = B \text{ e } \delta(C, a) = D$$

E (B, D) l'avevo già cancellata del primo punto

– b non permette di fare ulteriori distinzioni

- procedo con le stringhe di lunghezza lunghe 2, ma non riesco a fare nessuna ulteriore distinzione. quindi ho finito

Non sono riuscito a distinguere solo A e B , quindi sono equivalenti e li fondo. Il nostro automa finale sarà



Famiglia di relazioni

Definition 4.3.5: famiglia di relazioni

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, defininiamo una famiglia relazione $\sim_i \subseteq Q \times Q$ nel seguente modo:

- $\sim_0 = F \times F \cup (Q \setminus F) \times (Q \setminus F)$
ovvero stati che non possono essere distinti da ϵ
- $q_1 \sim_{i+1} q_2 \iff \forall a \in \Sigma \quad \delta(q_1, a) \sim_i \delta(q_2, a)$
ovvero q_1 e q_2 sono in relazione \sim_{i+1} se $\forall x \in \Sigma^*$ cin $|x| \leq i + 1$

$$\hat{\delta}(q_1, x) \in F \iff \hat{\delta}(q_2, x) \in F$$

Note:

Si osservi che:

- La relazione $Id = \{(q, q) \mid q \in Q\}$ è tale che $Id \subseteq \sim_i \forall i$. Si ha in fatti che uno stato è sempre equivalente a se stesso
- \sim_i è una relazione d'equivalenza $\forall i$
 - \sim_0 ha solo 2 classi d'equivalenza, ovvero F e $Q \setminus F$
 - Per \sim_i , riflessività e simmetria sono ovvie, mentre la transitività è meno banale:

$$q_1 \sim_i q_2 \text{ e } q_2 \sim_i q_3 \implies q_1 \sim_i q_3$$

- Ad ogni passo, rimuovo qualche coppia!
Si verifica che:

$$\sim_0 \supseteq \sim_1 \supseteq \sim_2 \supseteq \sim_3 \supseteq \dots$$

Ossia una **catena decrescente** (o non crescente) di relazioni di equivalenza.

- Se esiste un K tale che $\sim_K = \sim_{K+1}$, allora per ogni $j > K$ vale che $\sim_j = \sim_K$.
In altre parole: Non appena la relazione non viene modificata in un passo, ho trovato la soluzione
- Un tale K esiste sicuramente ed è minore di:

$$|\sim_0| = |F \times F| + |(Q \setminus F) \times (Q \setminus F)| = |F|^2 + |Q \setminus F|^2$$

Questo perché, nella peggiore delle ipotesi, ad ogni passo iterativo si rimuove solo una (o al massimo due) coppia.

In realtà si può dimostrare che $K < |Q| - 1$, poiché $|Q|$ è la lunghezza del massimo cammino aciclico.
Infatti, negli esempi che abbiamo visto, ho considerato solo stringhe che non portavano a cicli.

Theorem 4.3.4

Siano \sim_2 , \sim_3 e \sim_4 le relazioni d'equivalenza mostrate poc'anzi si ha che

$$\sim_2 = \sim_1 \implies \sim_3 = \sim_2$$

Dimostrazione con un esempio: Supponiamo, per assurdo, che $r_2 = r_1$ ma che $r_3 \neq r_2$, cioè esistano A, B tali che:

$$A \sim_2 B \quad \text{ma} \quad A \not\sim_3 B,$$

ossia:

$$\sim_3 \subset r_2 = r_1.$$

Allora, poiché $A \not\sim_3 B$, deve essere:

$$\begin{array}{ccccccc} A & \xrightarrow{a_1} & A_1 & \xrightarrow{a_2} & A_2 & \xrightarrow{a_3} & A_3 \\ & & & & & & \\ B & \xrightarrow{a_1} & B_1 & \xrightarrow{a_2} & B_2 & \xrightarrow{a_3} & B_3 \end{array}$$

con $A_3 \in F$ e $B_3 \notin F$ (ovviamente).

Ma allora $A_1 \not\sim_2 B_1$ e poiché $r_2 = r_1$, deve essere:

$$A_1 \not\sim_1 B_1.$$

Ma allora:

$$A_1 \xrightarrow{b} A_u \quad B_1 \xrightarrow{b} B_u$$

con $A_u \in F$ e $B_u \notin F$ (ovviamente).

Ma allora $A \not\sim_2 B$ perché:

$$A \xrightarrow{a_1} A_1 \xrightarrow{b} A_u \quad B \xrightarrow{a_1} B_1 \xrightarrow{b} B_u,$$

contraddicendo l'ipotesi iniziale.

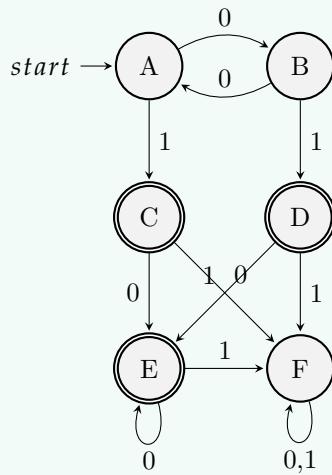
$$\implies \text{non è possibile che } r_1 = r_2 \text{ e } r_3 \subset r_2.$$

Quindi, se $\sim_{K+1} = \sim_K$, allora per ogni $j > K$ vale $\sim_j = \sim_K$

QED

Example 4.3.7

Proviamo a minimizzare il DFA seguente:



Costruiamo ora l'insieme \sim_0 come per definizione:

$$\sim_0 = \{(A, A), (B, B), (A, F), (B, A), (B, F), (F, F), (F, A), (F, B)\}$$

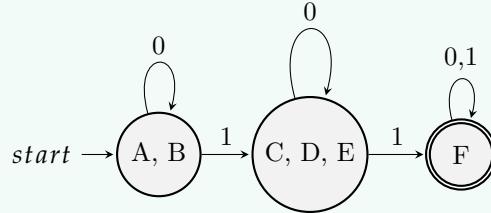
\cup

$$\{(C, C), (C, D), (C, E), (D, C), (D, D), (D, E), (E, C), (E, D), (E, E)\}.$$

Procediamo ora per costruire l'insieme \sim_1 . Per velocizzare controlliamo solo le coppie in \sim_0 e ovviamente non quelle identiche (tipo (A, A)). Per ora consideriamo solo stringhe di lunghezza 1 ("0" e "1"). Otteniamo:

$$\begin{aligned} \sim_1 = & \{(A, A), (A, B), (B, B), (A, F), (B, A), (B, F), (F, F)\} \cup \\ & \{(C, C), (C, D), (C, E), (D, C), (D, D), (D, E), (E, C), (E, D), (E, E)\}. \end{aligned}$$

Si ha che $\sim_2 = \sim_1$, quindi possiamo terminare e ridisegnare il DFA. Gli stati equivalenti sono quindi quelli che non compaiono nelle coppie $(A, B), (F), (C, D, E)$. Otteniamo quindi:



È semplice anche dire che la regex associata è 0^*10^* . Inoltre, lo stato F rappresenta uno stato di errore a cui si arriva con stringhe del tipo $0^*10^*10^+$.

tabella d'equivalenza

Le tabella di equivalenza sono un modo per memorizzare l'equivalenza ,di modo, da costruire un algoritmo per la minimizzazione poi, l'idea è la seguente

- **Tabella con solo coppie "vere".**
- Al round 0 dell'algoritmo iterativo, metto una marca X_0 per indicare che la coppia è distinta: (finale, non finale) oppure (non finale, finale).
- Al round 1, metto la marca X_1 per distinguere le coppie (q_1, q_2) non ancora marcate che, per qualche $a \in \Sigma$, hanno $(\delta(q_1, a), \delta(q_2, a))$ già marcata.
- Al round 2, metto la marca X_2 e così via per le coppie successive.
- **Condizione di arresto:** se non riesco a mettere nessuna nuova marca in un round, l'algoritmo termina.

Example 4.3.8

Prendiamo in esame il DFA dell'esempio precedente

Prima di tutto costruiamo una tabella con solo delle coppie che non sono identiche per far ciò la costruiamo a scaletta. Si ha che:

	A	B	C	D	E
B	X_0				
C	X_0	X_0			
D	X_0	X_0	X_0		
E	X_0	X_0	X_0	X_0	
F	X_1	X_1	X_0	X_0	X_0

Per calcolarla è possibile utilizzare il seguente algoritmo

Theorem 4.3.5

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ l'algoritmo della tabella a scala termina, inoltre due stati p e q sono distinguibili se e solo se la casella (p, q) (o la casella (q, p)) è marcata

Dimostrazione: Procedo a dimostrare le due parti

- Poiché abbiamo visto che $\exists k. \sim_k = \sim_{k+1}$ e quando l'algoritmo iterativo termina entro k iterazioni
- procedo a dimostrare la distinguibilità:
 - \Rightarrow Se p e q sono distinguibili, allora $\exists x \in \Sigma^*. \hat{\delta}(p, x) \in F$ e $\hat{\delta}(q, x \notin F)$ (o viceversa). Se prendo $k = |x|$, allora di sicuro $(p, q) \notin \sim_k$ cioè (p, q) viene marcata entro il round k

Algorithm 3: Costruzione di una tabella d'equivalenza

Input: DFA N
Output: tabella d'equivalenza

- 1 Sia $T^{n \times n}$ dove T è il numero di stati in n ;
- 2 marca x_0 ogni coppia (q_1, q_2) tale che $q_1 \in F$ e $q_2 \in Q \setminus F$ o viceversa;
- 3 $b \leftarrow \text{true}$;
- 4 $i \leftarrow 1$;
- 5 **while** b **do**
- 6 $b \leftarrow \text{false}$;
- 7 **foreach** (q_1, q_2) non marcata **do**
- 8 **if** $\exists a \in \Sigma$ con $(\delta(q_1, a), \delta(q_2, a))$ già marcate **then**
- 9 marca (q_1, q_2) con x_i ;
- 10 $b \leftarrow \text{true}$;
- 11 $i \leftarrow i + 1$;

– \Leftarrow

Se (p, q) sono marcati, allora sicuramente (p, q) sono distinguibili:

occorre prendere la catena di coppie che portano ad una coppia non presente in \sim_0 . ad esempio

$$(p, q) \rightarrow (p', q') \rightarrow (p'', q'')$$

Allora ab è la stringa che distingua p e q



4.3.6 automa minimo

Definition 4.3.6: Automa minimo

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'automa minimo equivalente $M_{min} = (\Sigma, Q_{min}, \delta_{min}, [q_0], F_{min})$ è dato da:

- $Q_{min} = \{[q] \mid q \in Q\}$ con $[q] = [q' \in Q \mid q \sim q']$
Gli stati di M_{min} sono classi di equivalenza di stati di M
- $\delta_{min}([q], a) = [\delta(q, a)]$
- $F_{min} = \{[q] \mid q \in F\}$

Note:

non esistono 2 stati distinti in M_{min} che siano tra loro equivalenti:

$$[q] \neq [q'] \implies q \not\sim q'$$

Adesso estendiamo la notazione di $\hat{\delta}$ per i DFA M_{min}

$$\begin{aligned}\hat{\delta}_{min} : Q_{min} \times \Sigma^* &\rightarrow Q_{min} \\ \hat{\delta}_{min}([q], \varepsilon) &= [q] \\ \hat{\delta}_{min}([q], xa) &= \hat{\delta}_{min}(\hat{\delta}_{min}([q], x), a) \\ w \in L[M_{min}] &\iff \hat{\delta}_{min}([q_0], w) \in F_{min}\end{aligned}$$

Theorem 4.3.6 Morbidelli - Morigi (innamorato italiano, innamorata giapponese)

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ l'automa $M_{min} = (\Sigma, Q_{min}, \delta_{min}, [q_0], F_{min})$ riconosce lo stesso linguaggio

di M , ed ha il minimo numero di stati tra tutti gli automi deterministici per questo linguaggio

Dimostrazione: Dimostriamo per gradi:

- dimostro che M_{min} è ben definito, cioè δ_{min} non dipende dallo specifico stato scelto per rappresentare la classe di equivalenza.

Si ha se $q \sim q'$, allora:

- $[q] = [q']$
- $\delta_{min}([q], a) = [\delta] = [\delta(q', a)] = \delta_{min}([q', a])$

Se non fosse vero allora $\delta(q, a)$ e $\delta(q', a)$ sarebbero distinguibili, quindi pure q e q'

- Per dimostrare che $L[M] = L[M_{min}]$, dimostrare che

$$\delta(q_0, w) = r \iff \hat{\delta}([q_0], w) = [r]$$

ovvero che

$$\hat{\delta}([q_0], w) = [\hat{\delta}, w]$$

si fa la dimostrazione per induzione su $|w|$

- Caso base: $|w| = 0$ cioè $w = \epsilon$

$$\hat{\delta}(q_0, \epsilon) = q_0 \quad \hat{\delta}([q_0], \epsilon) = [q_0] = [\hat{\delta}(q_0, \epsilon)]$$

Dimostrato

- Passo induttivo: $w = xa$

$$\hat{\delta}(q_0, xa) = \delta(\hat{\delta}(q_0, x), a)$$

e, per definizione di $\hat{\delta}_{min}$

$$\hat{\delta}([q_0], xa) = \delta_{min}(\hat{\delta}([q_0], x), a)$$

Per ipotesi induttiva

$$\delta_{min}(\hat{\delta}([q_0], x), a) = \delta_{min}([\hat{\delta}(q_0, x)], a)$$

per definizione di δ_{min}

$$\delta_{min}([\hat{\delta}(q_0, x)], a) = [\delta(\hat{\delta}(q_0, x), a)]$$

per definizione di $\hat{\delta}$

$$[\delta(\hat{\delta}(q_0, x), a)] = [\hat{\delta}(q_0, xa)]$$

Quindi

$$w \in L[M] \iff \delta(q_0, w) = r \in F \iff \hat{\delta}_{min}([q_0], w) = [r] \in F_{min} \iff w \in L[M_{min}]$$

- rimane da dimostrare che è minimo, ovvero che un qualunque altro automa deterministico N non può avere meno stati Supponiamo esista un DFA N tale che $L[N] = L[M_{min}]$, ma con un numero di stati inferiore a quelli di M_{min}

- Gli stati iniziali di N e M_{min} devono essere equivalenti (nell'automa $N \cup M_{min}$), poiché $L[N] = L[M_{min}]$.

* Infatti, si suppone che N e M_{min} abbiano lo stesso linguaggio.

- Se più $q \in M_{min}$ e $q' \in N$ sono equivalenti, nell'automa $N \cup M_{min}$, allora sono equivalenti anche i loro successori per ogni $a \in \Sigma$.

- N non ha stati inaccessibili dal suo iniziale (altrimenti si potrebbe costruire un automa N' con ancora meno stati).

* M_{min} non ha stati inaccessibili per costruzione.

Di conseguenza, ogni stato di M_{min} è equivalente ad almeno uno stato di N (per il punto precedente).

- Poiché N ha meno stati di M_{\min} , due stati p e p' di M_{\min} devono essere equivalenti ad uno stesso stato q di N .
 - * Ma la relazione di equivalenza \sim è transitiva, quindi $p \sim p'$ e dunque p e p' devono essere equivalenti!
 - * Ma questo è impossibile!

Per costruzione di M_{\min} , non ci sono due stati diversi in M_{\min} equivalenti tra loro.



4.4 Espressività dei linguaggi regolari e pumping lemma

Finora abbiamo parlato solo di linguaggi regolari, senza dimostrare l'esistenza di altri tipi di linguaggi più espressivi. In questa sezione, faremo vedere che esistono linguaggi che non possono essere riconosciuti da automi finiti.

Prendiamo come esempio il linguaggio $B = \{0^n 1^n \mid n \geq 0\}$. Per costruire un automa che lo riconosca, dobbiamo ricordarci il numero di 0 inseriti all'inizio per controllare che il numero di 1 sia uguale. Ma, dato che il valore di n non ha un limite superiore, dobbiamo rappresentare un numero infinito di valori di n con un numero finito di stati, che dimostreremo essere impossibile in questo caso.

Potrebbe sembrare che qualunque linguaggio simile a B con n illimitato sia non regolare, ma questo non è corretto. Anche se a prima apparenza puo' sembrare che un linguaggio necessiti di memoria infinita, ci possono essere modi intelligenti per circonvenire questa apparente necessità. Prendiamo, ad esempio, il linguaggio $C = \{w \mid w \text{ ha un ugual numero di sottostringhe } 01 \text{ e } 10\}$, si puo' dimostrare che questo linguaggio sia regolare! Quindi la nostra intuizione puo' spesso sbagliare, vediamo ora un modo per dimostrare per certo che certi linguaggi non sono regolari:

4.4.1 Pumping lemma

Il pumping lemma è un teorema che attribuisce una proprietà speciale a tutti i linguaggi regolari. Se un linguaggio non ha questa proprietà, allora sappiamo per certo che non è regolare. In breve, la proprietà dice che ogni stringa di un linguaggio regolare puo' essere "pompata" se è almeno lunga quanto una specifica lunghezza chiamata **lunghezza di pompaggio HHahaHAahaha**. Ovvero, ognuna di queste stringhe ha una sezione che puo' essere concatenata a se stessa un numero illimitato di volte e la stringa risultante fa sempre parte dello stesso linguaggio.

Theorem 4.4.1 Pumping lemma

Sia A un linguaggio regolare, allora $\exists p$ tale che $\forall s \in A. s \geq p$, s puo' essere suddivisa in tre parti $s = xyz$ tali che:

1. $\forall i \geq 0. xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

In questa formalizzazione, y è la parte della stringa che puo' essere ripetuta (condizione 1), che deve essere non nulla per non rendere trivialmente vero il teorema (condizione 2). La terza condizione sarà utile per dimostrare che certi linguaggi non sono regolari, e ci dirà che prendiamo la **prima** porzione della stringa che puo' essere ripetuta. Vediamo perché:

Dimostrazione

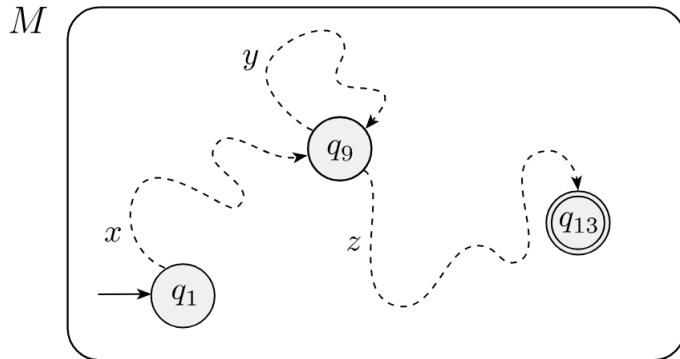
Prima della dimostrazione formale, proviamo a capire l'idea di base. Sia $M = (Q, \Sigma, \delta, q_1, F)$ un DFA che riconosce A . Poniamo che la lunghezza di pompaggio sia uguale al numero di stati di M ($p = |Q|$). Se in A non ci sono stringhe di tale lunghezza, allora il teorema è ovviamente vero perché vale per tutte le zero stringhe a cui puo' essere applicato.

Altrimenti, $\exists w \in A. |w| = n \geq p$ che viene accettato da M passando da una serie di stati $q_1, q_3, q_{20}, q_9, \dots, q_{13}$ (dove q_1 è lo stato iniziale e $q_{13} \in F$), che avrà lunghezza uguale a $n + 1$. Dato che $n \geq p$, siamo sicuri che

$n + 1 > p$, quindi ci sono più stati nella sequenza di quanti stati distinti esistano. Allora, ci deve essere per forza uno stato che si ripete (per il **pigeonhole principle**):

$$s = |_{q_1} s_1 |_{q_3} s_2 |_{q_{20}} s_3 |_{q_9} s_4 |_{q_{17}} s_5 |_{q_9} s_6 |_{q_6} \dots |_{q_{35}} s_n |_{q_{13}}$$

in questo caso, lo stato q_9 è il primo che si ripete. Possiamo ora dividere s in tre parti x, y e z : x è la parte di s prima di q_9 , y è la parte fra le due ripetizioni di q_9 e z è la parte rimanente.



Vediamo perché questa suddivisione rispetta le tre condizioni. Consideriamo l'input $xyyz$ passato a M : x ci porta da q_1 a q_9 , la prima y ci porta da q_9 a q_9 , anche la seconda ci porta da q_9 a q_9 , infine z ci porta da q_9 a q_{13} (lo stato finale). Quindi $xyyz$ è accettato da M , ma anche $xy^i z$ per lo stesso motivo. Nel caso $xy^0 z$ vediamo che viene riconosciuto anche lui, quindi la prima condizione è verificata. Dato che y è la parte della stringa che ci porta da q_9 a q_9 , deve essere per forza non nulla perché M è deterministico. Infine, la terza condizione è rispettata dato che, per la **pigeonhole principle** la prima ripetizione deve avvenire per forza entro i primi $p + 1$ stati, quindi $|xy| \leq p$.

Dimostrazione formale: Sia $M = (Q, \Sigma, \delta, q_1, F)$ un DFA che riconosce il linguaggio regolare A e $p = |Q|$. Sia $s = s_1 s_2 \dots s_n$ una stringa in A tale che $|s| = n \geq p$ e siano r_1, r_2, \dots, r_{n+1} la sequenza di stati che attraversa M quando viene dato s come input, quindi $r_{i+1} = \delta(r_i, s_i)$. Dato che $n + 1 \geq p + 1$, per il pigeonhole principle fra i primi $p + 1$ stati ce ne deve essere uno che si ripete. Chiamiamo r_j il primo e r_l la ripetizione, quindi abbiamo che $l \leq p + 1$.

Ora siano $x = s_1 \dots s_{j-1}$, $y = s_j \dots s_{l-1}$ e $z = s_l \dots s_n$. Dato che x porta M da r_1 a r_j , y porta da r_j a r_l e z porta da r_l a r_{n+1} e siccome r_j e r_l sono lo stesso stato, M deve accettare $xy^i z \forall i \geq 0$. Siccome $j \neq l$, $|y| > 0$; e $l \leq p + 1$, quindi $|xy| \leq p$. Abbiamo quindi soddisfatto tutte le condizioni del pumping lemma. \square

Utilizzo ed esempi

Per dimostrare che B non è un linguaggio regolare, prima assumiamo che B sia regolare e poi dimostriamo che non vale la proprietà P del pumping lemma:

$$\neg P \implies B \text{ non e' regolare}$$

Dato che proprio per il lemma, B è regolare $\implies P$. Quindi, bisogna assumere che p sia la lunghezza garantita dal pumping lemma e prendere una stringa $s \in B$, $|s| \geq p$ tale che noi sappiamo che z non può essere pompata. Dimostriamo questo mostrando che per qualunque suddivisione xyz che rispetti le ultime due condizioni, la prima condizione non vale mai. Solitamente possiamo raggruppare i tipi di suddivisione in diverse categorie che possono essere analizzate individualmente, in modo da non dover controllare davvero ogni suddivisione. Trovando s , abbiamo dimostrato $\neg P$ e quindi che B non è regolare.

Per trovare s , solitamente ci vuole un po' di creatività. Bisogna provare a trovare una stringa che raccolga l'"essenza" dell'irregolarità del linguaggio, vediamo un paio di esempi per chiarire:

Example 4.4.1

$$B = \{0^n 1^n \mid n \geq 0\}$$

Example 4.4.2
$$C = \{w \mid w \text{ ha lo stesso numero di } 0 \text{ e } 1\}$$
Example 4.4.3
$$F = \{ww \mid w \in \{0, 1\}^*\}$$

TODO: fai esercizi (che sono risolti sul libro)

Chapter 5

Linguaggi liberi nondeterministici

Fino ad ora abbiamo studiato due modi equivalenti per descrivere linguaggi: gli **automi finiti** e le **espressioni regolari**. Abbiamo anche visto le limitazioni di questi metodi, che non riescono ad accettare linguaggi, anche alcuni semplici come $\{0^n 1^n \mid n \geq 0\}$.

Per espandere la quantita' di linguaggi che possiamo studiare, introduciamo le cosidette **grammatiche libere (dal contesto)**, un metodo piu' potente che permette di descrivere certe proprietà di linguaggi che hanno una struttura ricorsiva. Inizialmente queste grammatiche sono state studiate per analizzare linguaggi umani, ma piu' recentemente hanno visto svariate applicazioni, come nel caso dei **parser**, utilizzati dalla maggior parte di compilatori e interpreti per estrarre il significato dal codice.

L'insieme dei linguaggi che possono essere riconosciuti da grammatiche libere sono i **linguaggi liberi (dal contesto)**, che includono propriamente i linguaggi regolari. Per riconoscere tali linguaggi, introdurremo un nuovo tipo di automa: l'**automa a pila** (PDA - pushdown automata).

5.1 Grammatiche libere

Per qualche motivo, le grammatiche libere le abbiamo viste all'inizio, io le metterei qua ma va beh andate qui per capire cosa sono: [2.6](#)

5.1.1 Semplificazione e forme normali

Questa roba qua invece e' stata messa nei linguaggi liberi deterministici, secondo me ha piu' senso metterla qua' o fare proprio un capitolo solo sulle grammatiche: [6.2.1](#)

5.2 PDA

In che modo possiamo modificare gli automi visti fin'ora (DFA/NFA) per far si che riescano a riconoscere linguaggi non-regionalari?

Vediamo ad esempio il linguaggio $L = \{ww^R \mid w \in \{a,b\}^*\}$. Si puo' dimostrare (fallo per esercizio) che non e' un linguaggio regolare, dato che per riconoscerlo si dovrebbe **memorizzare** tutta la prima parte della parola palindroma (la cui lunghezza non e' limitata). Quindi serve una forma di memoria ausiliaria, che nei PDA e' implementata con una **pila** con memoria illimitata.

Essendo una pila (LIFO), ci sono certe restrizioni per come puo' essere gestita la memoria:

- Si puo' leggere solo il primo elemento in cima alla pila (top)
- Si puo' rimuovere solo l'elemento top
- Si puo' aggiungere elementi solo in cima alla pila (in modo che diventino il nuovo top)

I PDA possono essere *nondeterministici*, e come tali hanno una potenza espressiva maggiore rispetto ai PDA deterministici (DPDA). Dimostreremo alla fine della sezione che i PDA (nondeterministici) hanno lo stesso potere espressivo delle grammatiche libere.

5.2.1 Definizione formale

La definizione formale di un PDA e' simile a quella dei NFA, con l'aggiunta della presenza della pila. Questa pila contiene simboli di un alfabeto che puo' essere diverso rispetto a quello di input, quindi dobbiamo definire sia un alfabeto Σ per l'input, sia l'alfabeto Γ per la pila. L'operazione di transizione, che ci dice come si comporta l'automa, ha come dominio $Q \times \Sigma_\epsilon \times \Gamma$ (dove $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$), quindi la mossa del PDA e' determinata dallo stato corrente, dal carattere in input e dal carattere in cima alla pila. Di questi, il secondo puo' essere anche ϵ , quindi la macchina puo' fare una mossa senza leggere l'input, ma il simbolo in cima alla stack viene sempre letto (e consumato).

Dato un input, l'automa puo' spostarsi su un nuovo stato e aggiungere un numero arbitrario di elementi alla pila (anche nessuno). Inoltre, dato che il nondeterminismo ha come conseguenza la possibilita' di avere diverse transizioni possibili dato lo stesso input, dobbiamo considerare come codominio l'insieme di tutti i possibili insiemi che hanno come elementi coppie del tipo $Q \times \Gamma^*$, ovvero l'insieme potenza $\mathcal{P}(Q \times \Gamma^*)$.

Definition 5.2.1: PDA

Un automa a pila nondeterministico e' una 7-pla $(Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ dove:

- Q e' un insieme finito di stati
- Σ e' un alfabeto di input finito
- Γ e' un alfabeto per la pila finito
- $\delta : Q \times \Sigma_\epsilon \times \Gamma \rightarrow Q \times \Gamma^*$ e' la funzione di transizione
- q_0 e' lo stato iniziale
- $\perp \in \Gamma$ e' il primo simbolo sulla pila
- $F \subset Q$ e' l'insieme degli stati finali

5.2.2 Transizioni

A ogni passo di computazione, possiamo attribuire a un PDA una **descrizione instantanea** (o configurazione) che ci dice lo stato corrente $q \in Q$, l'input rimasto $w \in \Sigma^*$ e la stringa sulla pila $\beta \in \Gamma^*$ (dove l'elemento top e' il "carattere piu' a sinistra"):

$$(q, w, \beta)$$

Le mosse possono essere definite come una derivazione da una configurazione a quella successiva (indicata col simbolo \vdash_N , come con gli altri automi). Mostriamo, usando l'inferenza logica, i due tipi di mossa:

1. Lettura di un carattere in input ($a \in \Sigma$):

$$\frac{(q', a) \in \delta(q, a, X)}{(q, aw, X\beta) \vdash_N (q', w, a\beta)}$$

2. Senza lettura dell'input:

$$\frac{(q', \epsilon) \in \delta(q, \epsilon, X)}{(q, w, X\beta) \vdash_N (q', w, \alpha\beta)}$$

Come per gli automi finiti, introduciamo anche \vdash_N^* , la chiusura riflessiva e transitiva di \vdash_N :

$$\frac{(q, w, \beta) \vdash_N^* (q', w', \beta') \vdash_N (q'', w'', \beta'')}{(q, w, \beta) \vdash_N^* (q, w, \beta)}$$

5.2.3 Linguaggio Accettato

Ci sono diversi modi in cui possiamo determinare se una stringa e' accettata o meno da un PDA. Oltre al classico riconoscimento per stato finale, possiamo dire che una stringa e' accettata se alla fine dell'input la pila e' vuota:

- $L[N] = \{w \in \Sigma^* \mid (q_0, w, \perp) \vdash_N^* (q, \epsilon, \alpha). q \in F\}$
- $P[N] = \{w \in \Sigma^* \mid (q_0, w, \perp) \vdash_N^* (q, \epsilon, \epsilon)\}$

Osserviamo che non per forza un PDA deve svuotare la pila alla fine di un input accettato, quindi in generale:

$$L[N] \neq P[N]$$

Dimostriamo pero' che questi due metodi di riconoscimento hanno la stessa espressivita':

Theorem 5.2.1

Dato un PDA N si ha che:

- Se $L = L[N]$ e' il linguaggio riconosciuto da N per stato finale, allora $\exists N'$ PDA tale che $L = P[N']$, dove $P[N']$ e' il linguaggio riconosciuto da N' per pila vuota.
- Viceversa, se $L = P[N]$ allora $\exists N''$ PDA tale che $L = L[N'']$.

Quindi i PDA che riconoscono per stato finale e quelli che riconoscono per pila vuota sono equivalenti.

Vediamo prima un'osservazione che ci servira' per dimostrare questo teorema:

Note:

Si osservi che, se la pila viene completamente svuotata, il PDA che abbiamo definito non puo' avere altre transizioni. Questo e' perche' $\epsilon \notin \Gamma$, e quindi non e' neanche presente nel dominio della funzione di transizione. Quindi se la pila di un PDA e' vuota, allora si blocca.

Detto questo, passiamo a una dimostrazione per costruzione di entrambe gli enunciati: TODO aggiungi figure

Dimostrazione: Dimostriamo (in ordine inverso) i due versi:

- Sia N un PDA che accetta un linguaggio L per pila vuota ($L = P[N]$). Costruiamo un PDA N' che riconosca lo stesso linguaggio per stato finale. L'idea e' quella di aggiungere un elemento in fondo alla pila, in modo tale da poter riconoscere quando la pila di N si sarebbe svuotata, e di aggiungere transizioni da tutti gli stati ad un nuovo stato finale. Per aggiungere l'elemento sulla pila, basta aggiungere un nuovo stato iniziale con una transizione verso il vecchio stato iniziale che non legga l'input e che aggiunga un elemento $Z \notin \Gamma$ sotto a \perp .

Le transizioni aggiuntive non leggono caratteri in input (sono nondeterministiche) e guardano se in cima alla pila e' presente l'elemento finale aggiunto inizialmente. In questo modo, se la pila non arriva mai all'ultimo elemento allora e' impossibile arrivare allo stato finale, mentre se la pila viene svuotata e' possibile spostarsi sullo stato finale. Se N' si trova nell'ultimo caso ma non ha finito di leggere la stringa, il PDA si blocca e la parola non viene accettata. Altrimenti viene riconosciuta, sse appartiene al linguaggio L .

- Sia N un PDA che accetta un linguaggio L per stato finale ($L = L[N]$). Costruiamo un PDA N'' che riconosca lo stesso linguaggio per pila vuota. L'idea e' simile a quella sopra: aggiungiamo un elemento $Z \notin \Gamma$ in fondo alla pila (sempre per evitare che la pila si svuoti, rendendo impossibili ulteriori transizioni), e aggiungiamo delle transizioni nondeterministiche, questa volta solo dagli stati finali di N , che portano a uno stato da cui parte un'unica transizione verso se stesso. Questa transizione non fa altro che rimuovere elementi dalla pila senza leggere input, in modo che se N arriva ad uno stato finale dopo aver consumato tutto l'input, allora N'' ha una possibile strada che svuota completamente la pila. Altrimenti, o la pila non viene mai svuotata, oppure viene svuotata ma senza consumare tutto l'input (e quindi non viene riconosciuta). Osservare che in tutte le altre transizioni non si puo' leggere Z dalla pila (dato che non appartiene all'alfabeto della pila), quindi non potra' accadere che la pila si svuoti consumando tutto l'input senza raggiungere lo stato aggiunto, riconoscendo erroneamente l'input.

5.3 Equivalenza fra grammatiche libere e PDA

In questa sezione dimostreremo che i PDA e le grammatiche libere hanno lo stesso potere espressivo, ovvero che riconoscono lo stesso insieme di linguaggi. Quindi sia le grammatiche libere che i PDA riconoscono linguaggi liberi e faremo vedere come costruire un PDA equivalente partendo da una CFG (context free grammar) e viceversa. Dato che per definizione un linguaggio e' libero sse esiste una grammatica libera che lo descrive, ci riduciamo a dimostrare il seguente teorema:

Theorem 5.3.1

Un linguaggio e' libero sse esiste un PDA che lo riconosce

Dobbiamo dimostrare le due direzioni. Dato che sono un po' corpose, separiamole in due lemmi:

5.3.1 Da grammatica libera a PDA

Lemma 5.3.1

Dato un linguaggio libero L , esiste un PDA N tale che

$$L = P[N]$$

Notare che potevamo anche scrivere $L = L[N]$, dato che abbiamo dimostrato che i due metodi di riconoscimento hanno la stessa espressività. Per la dimostrazione ci e' piu' utile costruire un PDA che riconosca per pila vuota, andiamo a vedere come costruirlo:

Idea della dimostrazione: Data la grammatica libera G che descrive L , vogliamo costruire un PDA N che accetta un input w se esiste una derivazione di G che porta a w . Ricordiamo che una derivazione e' la serie di sostituzioni che fa una grammatica, partendo dalla variabile iniziale, per generare una stringa. Ad ogni passo viene generata una **stringa intermedia** formata da terminali e non-terminali. Il compito di N e' quello di decidere se esistono una serie di sostituzioni (prese dalle regole di G) che generano w .

La parte difficile e' sapere quale sostituzione e' quella giusta, dato che possono esserci piu' regole di sostituzione per una sola variabile. Fortunatamente, utilizzando il nondeterminismo possiamo semplicemente considerare tutte le opzioni come percorsi validi.

Allo stato iniziale, quindi, il PDA si ritrova solo il nonterminale S sulla pila. Poi passa per una serie di stringhe intermedie prima di avere solo terminali, che se combaciano con l'input significa che esiste una derivazione di G che genera w e quindi N lo accetta. Se in tutte le possibili diramazioni nondeterministiche l'input non combacia con la stringa nella pila, allora G non puo' generarlo e non viene riconosciuto da N .

L'unico problema e' che un PDA puo' solo accedere al primo elemento della pila. Questo significa che se un non-terminale non e' top, non possiamo sostituirlo senza rimuovere tutti i terminali sopra, quindi non possiamo tenere in memoria l'intera stringa intermedia. Possiamo risolvere questo problema consumando il top ogni volta che appare un terminale che combacia con il carattere in input, passando quindi al carattere successivo. Se non combaciano, il PDA deve fare *backtracking* e provare una nuova combinazione di sostituzioni. Se per tutte le combinazioni c'e' un terminale in top che non combacia con l'input, la stringa non viene accettata.

Notare che in questo modo, il PDA svolge una **derivazione leftmost** (2.9.1) dal simbolo iniziale all'input. ☐

Dimostrazione formale: Sia L un linguaggio libero. Allora per definizione esiste una grammatica $G = (NT, T, R, S)$ che lo descrive. Vogliamo costruire un PDA N tale che $L = P[N]$. Sia $N = (\{q\}, T, NT \cup T, \delta, q, S, \emptyset)$. Aggiungiamo le transizioni $\forall a \in T, \forall A \in NT$:

$$\begin{aligned}\delta(q, a, a) &= \{(q, \epsilon)\} \\ \delta(q, \epsilon, A) &= \{(q, \beta) \mid A \rightarrow \beta \in R\}\end{aligned}$$

Si puo' dimostrare per induzione sulla lunghezza dell'input che:

$$S \Rightarrow_l^* w \iff (q, w, s) \vdash_N^* (q, \epsilon, \epsilon)$$

5.3.2 Da PDA a grammatica libera

Lemma 5.3.2

Dato un PDA N tale che $L = P[N]$, possiamo costruire una grammatica libera G tale che:

$$L = L[G]$$

Questa direzione e' molto piu' lunga e complessa rispetto alla prima, dato che "programmare" un automa e' molto piu' semplice rispetto a "programmare" una grammatica. E piu' semplice dimostrare il lemma se consideriamo un PDA con un solo stato, quindi ci torna utile il seguente lemma:

Lemma 5.3.3

Per ogni PDA N , esiste un PDA N' con un solo stato tale che:

$$P[N] = P[N']$$

Questo lemma ci dice che ogni PDA puo' essere trasformato in un PDA equivalente che ha un solo stato. Non lo dimostriamo perche sbattezz.

Ora ci possiamo ricondurre a dimostrare il lemma con l'ipotesi aggiuntiva che N ha un solo stato:

Idea della dimostrazione: Dato che la dimostrazione formale e' davvero lunghissima, tale da mandare in tilt anche il dio degli audiolibri detto "Il Guerriero", ne parleremo solo schematicamente.

L'idea e' la seguente: dato un PDA $N = (\{q\}, \Sigma, \Gamma, \delta, q, \emptyset)$ a uno stato, generiamo una grammatica libera $G = (T, NT, R, S)$ equivalente tale che:

$$\begin{aligned} \forall a \in \Sigma_\epsilon, \forall A \in \Gamma. (q, B_1 B_2 \dots B_k) \in \delta(q, a, A) \text{ con } k \geq 0 : \\ A \rightarrow a B_1 B_2 \dots B_k \in R \end{aligned}$$

Si puo' dimostrare che $S \Rightarrow^* w \iff (q, w, s) \vdash_N^* (q, \epsilon, \epsilon)$. □

5.3.3 Relazione fra linguaggi regolari e linguaggi liberi

Abbiamo quindi dimostrato che gli automi a pila riconoscono la classe di linguaggi liberi. Dato che gli automi finiti riconoscono i linguaggi regolari e possono essere convertiti in PDA che ignorano la pila, possiamo dire che:

Proposition 5.3.1 Relazione fra linguaggi regolari e linguaggi liberi

I linguaggi regolari sono un sottoinsieme proprio dei linguaggi liberi.

TODO: figura

5.4 Proprietà

5.4.1 Chiusura

Come nei linguaggi regolari, anche i linguaggi liberi sono chiusi rispetto alle operazioni regolari:

Theorem 5.4.1

I linguaggi liberi sono chiusi per:

1. Unione
2. Concatenazione
3. Ripetizione (stella di Kleene)

Dimostrazione: Siano L_1, L_2 i linguaggi generati da due grammatiche libere $G_1 = (NT_1, T_1, R_1, S_1)$, $G_2 = (NT_2, T_2, R_2, S_2)$ e assumiamo che $NT_1 \cap NT_2 = \emptyset$, allora:

1. Unione:

Costruisco la grammatica libera $G = (NT_1 \cup NT_2 \cup \{S\}, T_1 \cup T_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 \mid S_2\}, S)$. Si puo' dimostrare che $L(G) = L_1 \cup L_2$.

2. Concatenazione:

Costruisco la grammatica libera $G = (NT_1 \cup NT_2 \cup \{S\}, T_1 \cup T_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$. Si puo' dimostrare che $L(G) = L_1 \cdot L_2$.

3. Ripetizione:

Costruisco la grammatica libera $G = (NT_1 \cup \{S\}, T_1, R_1 \cup \{S \rightarrow \epsilon \mid S_1 S\}, S)$. Si puo' dimostrare che $L(G) = L_1^*$.



Rispetto all'intersezione:

Theorem 5.4.2

Dati un linguaggio libero L_1 e un linguaggio regolare L_2 , la loro intersezione $L_1 \cap L_2$ e' un linguaggio libero.

Notare che questo teorema **NON** dice che i linguaggi liberi sono chiusi rispetto all'intersezione (infatti vedremo dopo che non lo sono). Dimostriamolo:

Idea della dimostrazione: Vogliamo in qualche modo eseguire contemporaneamente sia il PDA che riconosce il linguaggio libero (per stato finale), sia il DFA che riconosce il linguaggio regolare. Se alla fine dell'input sono entrambe in stato di riconoscimento, allora significa che la stringa fa parte sia del linguaggio regolare che di quello libero, quindi appartiene all'intersezione. 

Dimostrazione: Dati un PDA $N_1 = (Q_1, \Sigma, \Gamma, \delta_1, q_1, Z, F_1)$ e un DFA $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ che riconoscono rispettivamente il linguaggio libero $L_1 = L[N_1]$ e il linguaggio regolare $L_2 = L[N_2]$, costruiamo un PDA N che riconosca il linguaggio $L_1 \cap L_2$:

- $N = (Q_1 \times Q_2, \Sigma, \Gamma, \delta, \langle q_1, q_2 \rangle, Z, F_1 \times F_2)$
- $\forall q \in Q_1, \forall p \in Q_2, \forall a \in \Sigma, \forall X \in \Gamma :$

$$\delta(\langle q, p \rangle, a, X) = \{(\langle r, s \rangle, \gamma) \mid s = \delta_2(p, a) \wedge \langle r, \gamma \rangle \in \delta_1(q, a, X)\}$$

Si puo' dimostrare che $\forall w \in \Sigma^*$:

$$(q_1, w, Z) \vdash_{N_1}^* (q, \epsilon, \gamma) \wedge \hat{\delta}_2(q_2, w) = q'$$

sse

$$(\langle q_1, q_2 \rangle, w, Z) \vdash_N^* (\langle q, q' \rangle, \epsilon, \gamma)$$

Quindi, se $q \in F_1$ e $q' \in F_2$ e quindi $w \in L_1 \cap L_2$, anche N riconosce w dato che $\langle q, q' \rangle \in F_1 \times F_2$. Vale anche la direzione opposta, quindi $L_1 \cap L_2 = L[N]$. 

Possiamo usare questa proprietà per dimostrare che un linguaggio non è libero. Infatti, dato un linguaggio L_1 che vogliamo analizzare, basta trovare un linguaggio L_2 regolare per cui $L_1 \cap L_2$ non è libero. Se riusciamo a trovare tale linguaggio, allora siamo sicuri che L_1 non è regolare:

$$\forall L_1,$$

$$\exists L_2 \text{ regolare } L_1 \cap L_2 \text{ non e' libero} \implies L_1 \text{ non e' libero}$$

I linguaggi liberi, come anticipato, non sono chiusi per intersezione. Per dimostrarlo, basta trovare una controprova:

Controprova: Consideriamo i due linguaggi liberi $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ e $L_2 = \{a^n b^m c^n \mid n, m \geq 0\}$. Le uniche stringhe che appartengono sia a L_1 che a L_2 sono quelle che hanno lo stesso numero di tutti e tre i caratteri, ovvero: $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$. Vediamo ora il **pumping theorem** per dimostrare che questo linguaggio non è libero. 

5.4.2 Pumping Theorem

Il **pumping theorem** ci serve per dimostrare quando linguaggio non fa parte della classe di linguaggi liberi. Si ricordi il pumping lemma (TODO: aggiungi ref), introdotto con i linguaggi regolari. Il pumping theorem è abbastanza simile, dato che anche questo dimostra l'esistenza di una **lunghezza di pompaggio** per i linguaggi liberi, tale che ogni stringa di lunghezza maggiore o uguale può essere pompata. In questo caso, però, non c'è una sola sezione che può essere ripetuta, ma due. Vediamo in dettaglio:

Theorem 5.4.3

Se L è un linguaggio libero, allora esiste un numero p tale che, sia $s \in L$ con $|s| \geq p$, allora s può essere divisa in cinque sezioni $s = uvxyz$ che soddisfano tali condizioni:

1. $\forall i \geq 0, uv^i xy^i z \in L$
2. $|vy| > 0$
3. $|vxy| \leq p$

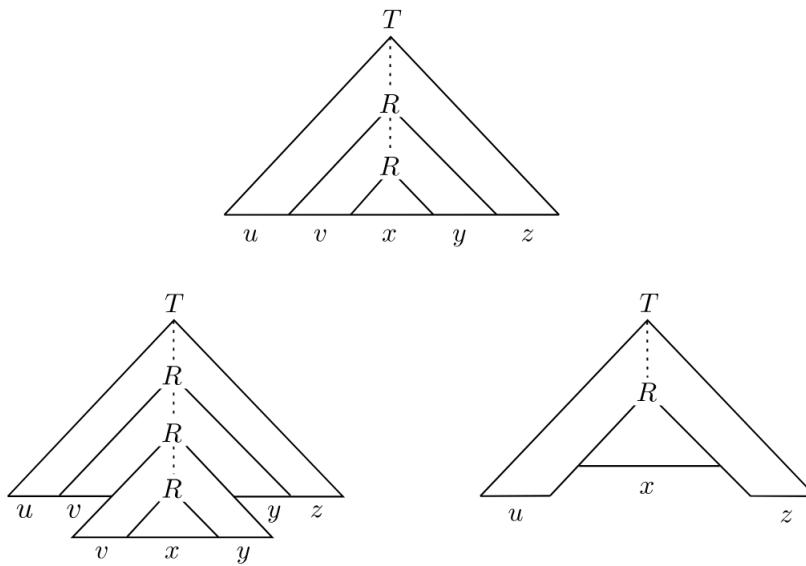
La seconda condizione ci dice che v o y devono essere non nulli, se no il teorema diventa ovviamente vero per ogni linguaggio. La terza condizione ci può aiutare in alcune dimostrazioni.

Idea della dimostrazione: Per capire questa dimostrazione, è necessario avere ben presente le grammatiche libere e i relativi alberi di derivazione (2.9). Infatti dato che L è un linguaggio libero, avrà anche una grammatica che lo genera.

Consideriamo una parola $s \in L$ "abbastanza lunga" (defineremo dopo cosa significa): questa avrà almeno un'**albero di derivazione** che la genera. Avendo scelto s abbastanza lunga, allora la distanza massima dalla radice S a una foglia (ovvero l'**altezza dell'albero**) è tale da garantire, per il **pigeonhole principle**, il ripetersi di uno dei nonterminali fra i nodi del percorso.

Chiamiamo A il nonterminale ripetuto. Come si vede dalla figura, possiamo rimpiazzare il sottoalbero radicato nella seconda ripetizione di A con il sottoalbero radicato nella prima ripetizione di A , ottenendo un albero di derivazione valido e quindi una stringa appartenente a L . Possiamo ripetere questa procedura all'infinito, e vale anche se facciamo al contrario.

Quindi, se divido s in cinque sezioni $uvxyz$ come indica la figura, è possibile ripetere v e y e ottenere sempre parole che appartengono a L .



Entriamo ora nei dettagli e proviamo a dimostrare in modo più formale tutte e tre le condizioni, oltre a vedere come calcolare la lunghezza di pompaggio p :

Dimostrazione: Sia $G = (T, NT, R, S)$ una grammatica libera che genera L . Sia b il massimo numero di simboli (sia T che NT) che si trovano nella parte destra di una produzione in R ($b = \max\{|\alpha| \mid A \rightarrow \alpha \in R\}$). Possiamo assumere che $b \geq 2$, dato che altrimenti la grammatica sarebbe banale. In qualsiasi albero di derivazione generato da G , il massimo numero di figli che puo' avere un nodo e' b . In altre parole, ci saranno un massimo di b nodi che distano 1 dalla radice S , al massimo b^2 che distano 2 e in generale un massimo di b^h nodi che distano esattamente h dalla radice. Quindi, se un albero e' alto h la stringa che genera e' lunga al massimo b^h . Viceversa, se una stringa e' lunga almeno $b^h + 1$, allora il suo albero di derivazione sara' alto almeno $h + 1$.

$$|w| \geq b^h + 1 \implies \text{height}(\text{parseTree}(w)) \geq h + 1$$

Sia $|NT|$ il numero di nonterminali in G , impostiamo la lunghezza di pompaggio nel seguente modo: $p = b^{|NT|+1}$. Dato che $b \geq 2$, sappiamo che $p > b^{|NT|}$, e quindi $p \geq b^{|NT|} + 1$ (teoricamente si puo' mettere maggiore stretto, ma per la dimostrazione non serve). Cosi' facendo, se s e' una stringa di L tale che $|s| \geq p$, sappiamo che il suo albero di derivazione e' alto almeno $|NT| + 1$:

$$|w| \geq p = b^{|NT|+1} \geq b^{|NT|} + 1 \implies \text{height}(\text{parseTree}(w)) \geq |NT| + 1$$

Notare che l'implicazione vale anche se poniamo $b^{|NT|} + 1$ come pumping length, ma per garantire la terza condizione dobbiamo scegliere la lunghezza maggiore che puo' generare un albero alto $|NT| + 1$, vedremo dopo perche'.

Per vedere come pompare una stringa $w \in L$, $|w| \geq p$, chiamiamo τ il suo albero di derivazione che ha il minor numero di nodi (questa condizione serve per garantire la seconda condizione). Sappiamo che $\text{height}(\tau) \geq |NT| + 1$, ovvero che esiste un percorso dalla radice a una foglia lungo almeno $|NT| + 1$. Contando la radice, ci sono $|NT| + 2$ nodi lungo questo percorso, di cui uno e' la foglia che e' quindi un terminale. Quindi i restanti $|NT| + 1$ nodi sono tutti nonterminali, e per il *pigeonhole principle* almeno uno si deve ripetere. Chiamiamo A il nonterminale che si ripete entro i $|NT| + 1$ nonterminali "piu' in basso", ovvero piu' vicini alla foglia, lungo il percorso (questa scelta serve per garantire la terza condizione).

Dividiamo w in cinque parti $uvxyz$ come nella figura. L'occorrenza superiore di A ha un sottoalbero piu' grande che genera vxy , mentre quella piu' in basso ha un sottoalbero piu' piccolo e genera solo x . Entrambe le sottostringhe sono generate dalla stessa variabile, quindi possiamo sostituire l'una con l'altra e ottenere un nuovo albero di derivazione valido. Sostituendo il minore con il maggiore ripetutamente ci da tutti gli alberi di derivazione per le stringhe $uv^i xy^i z$ con $i > 1$. Sostituendo il maggiore con il minore otteniamo la stringa uxz (il caso in cui $i = 0$), quindi abbiamo stabilito che questa divisione soddisfa la prima condizione, vediamo le altre due:

Per la condizione 2, dobbiamo accertarci che u e y non siano entrambe ϵ . Usando l'eliminazione del not, assumiamo che siano entrambe ϵ e dimostriamo il falso. Se fosse cosi', potremmo sostituire il sottoalbero minore a quello maggiore ottenendo comunque la stringa iniziale ($uxz = uv^i xy^i z$), pero' con un albero piu' piccolo. Ma questo e' assurdo, perche' come ipotesi avevamo stabilito che τ fosse l'albero di derivazione di w con meno nodi.

La terza condizione dice che $|vxy| \leq p$. Questa e' la sottostringa generata dalla prima occorrenza di A (dalla radice verso le foglie), il cui sottoalbero e' alto al massimo $|NT| + 1$, dato che le due ripetizioni di A devono essere entro i $|NT| + 1$ nonterminali piu' vicini alla foglia. La stringa piu' lunga che puo' generare un albero di quell'altezza e' $b^{|NT|+1} = p$, quindi $|vxy| \leq p$ come volevamo dimostrare. \square

Chapter 6

linguaggi liberi deterministici

6.1 PDA e linguaggi deterministici

6.1.1 PDA deterministici

Definition 6.1.1: PDA deterministico

Un PDA $N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ si dice **deterministico** sse:

1. $\forall q \in Q, \forall z \in \Gamma, (\forall a \in \Sigma, (\delta(q, \epsilon, z) \neq \emptyset \implies \delta(q, a, z) = \emptyset))$
2. $\forall q \in Q, \forall z \in \Gamma, \forall a \in (\Sigma \cup \{\epsilon\}), (|\delta(q, a, z)| \leq 1)$

Ovvero un PDA è libero deterministico sse in ogni configurazione, il PDA ha al massimo una transizione possibile per un dato stato, simbolo di input, e simbolo in cima alla pila e se ha una transizione ϵ disponibile allora non ha altri tipi di transizioni.

Quindi un PDA:

- ha al massimo una transizione
- non ha conflitti tra transizioni ϵ e transizioni che leggono un simbolo

6.1.2 Definizione di linguaggi liberi deterministici

Definition 6.1.2: Linguaggio libero deterministico

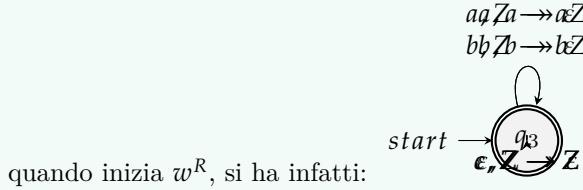
Un linguaggio è **libero deterministico** se è accettato per stato finale da un DPDA

Theorem 6.1.1

la classe dei linguaggi liberi deterministici è inclusa propriamente nella classe dei linguaggi libri :

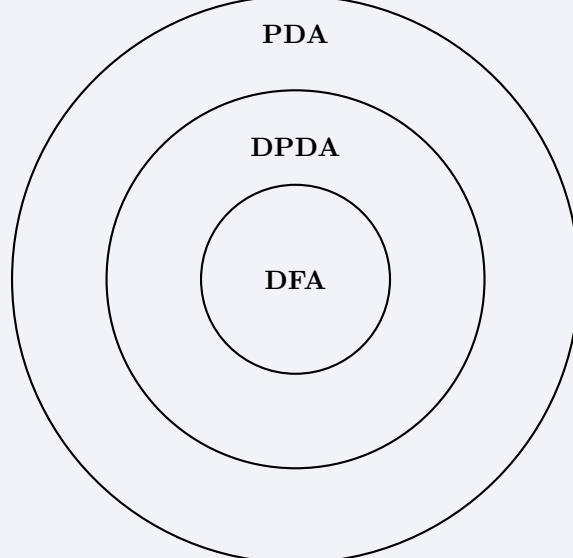
Example 6.1.1

- Sia $L_1 = \{ww^R \mid w \in \{a, b\}^*\}$ è libero, am si può dimostrare che non esiste un DPDA che lo riconosca, infatti con un DPDA non esiste un modo deterministico per riconoscere quando finisce w e inizia w^R
- $L_2 = \{wcw^R \mid w \in \{a, b\}^*\}$ è libero deterministico grazie al segnaposto c è possibile riconoscere



Theorem 6.1.2

Se L è regolare, allora \exists DPDA N tale che $L = L[N]$ per stato finale



a^*b^*

wcw^R

ww^R

dimostrazione: Se L è regolare, allora \exists DFA M tale che $L = L[M]$. A partire da M , posso costruire un DPDA N si compone come M senza mai manipolare lo stack, allora si che $L = L[N]$ per stato finale \square

Prefix property

Si giunge così al seguente fatto:

Claim 6.1.1

Un linguaggio libero deterministico L è riconosciuto da un DPDA per pila vuota sse L gode della "prefix property", ovvero

$$\nexists x, y \in L : x \text{ è prefisso di } y$$

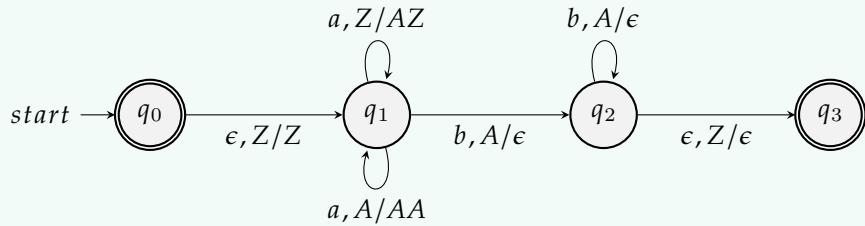
Pertanto si ha che:

- Se L è non gode della prefix property non può essere riconosciuto da un PDPA per pila vuota
- Se L è libero deterministico gode della prefix property, allora può essere riconosciuto da un PDPA per pila vuota
- Se L è libero deterministico, allora $L\$ = \{w\$ \mid w \in L\}$ gode della prefix property, infatti $L\$$ può essere riconosciuto da un PDPA per pila vuota.

Dove $\$ \notin \Sigma$ ovvero $\$$ non è un simbolo dell'alfabeto di L , ma grazie ad esso alla fine di ogni linguaggio regolare vale la prefix property

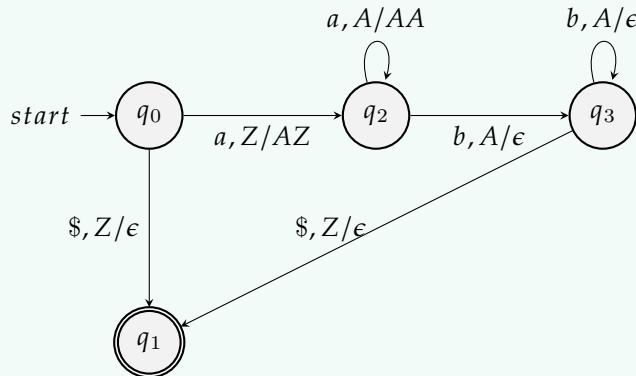
Example 6.1.2

Sia $L_1 = \{a^n b^n \mid n \geq 0\}$ il seguente linguaggio che non gode della prefix propriety, in quanto $\epsilon \in L_1$ e ϵ è prefisso di ab



Si ha quindi che il seguente linguaggio è riconosciuto da un DPDA *per stato finale* e non *per pila vuota*. Tuttavia il seguente linguaggio con il $\$ \notin \Sigma$ è riconosciuto da un DPDA per *per pila vuota*

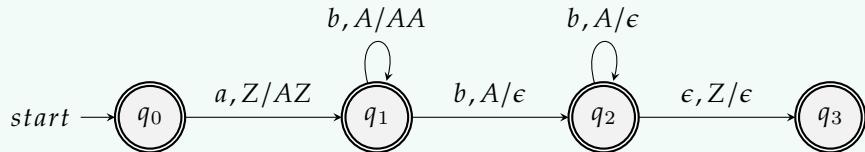
Sia $L_1\$$:



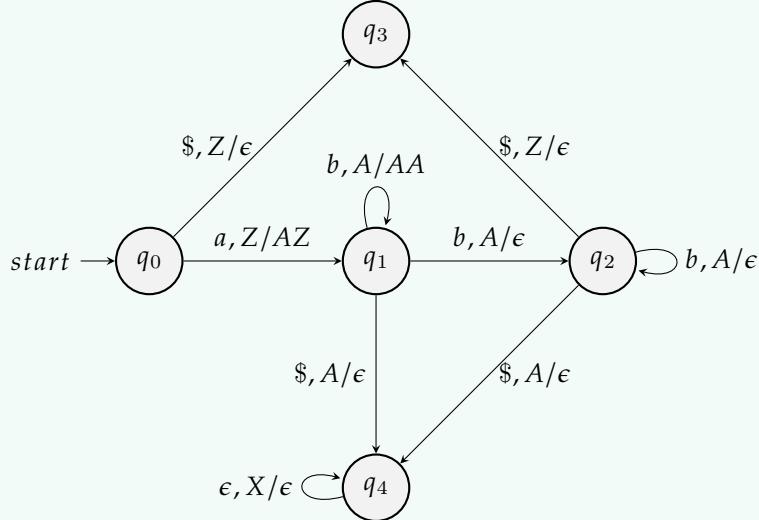
Si verifichi, infatti, come egli venga riconosciuto per pila vuota

Example 6.1.3

Sia $L_3 = \{a^n b^m \mid n \geq m \geq 0\}$ e dato che $\epsilon \in L_3$ tale linguaggio non gode della prefix propriety



e sia $L_3\$$:



non ambiguità dei linguaggi liberi deterministici

Proposition 6.1.1

Se L è libero deterministico, ovvero riconosciuto da un DPDA per *stato finale*, allora L è generabile da una **grammatica libera non ambigua**.

Si ha quindi che i **linguaggi liberi deterministici non sono ambigui**

Proprietà dei linguaggi libri deterministici

I linguaggi libri deterministici presentano le seguenti proprietà:

Proposition 6.1.2 chiusura solo per complementazione

Sia L un linguaggio libero deterministico, allora questo è chiuso per complementazione, ovvero:

$$(\exists \text{DPDA } N : L = L(N)) \implies (\exists \text{DPDA } N' : \bar{L} = L(N')) \text{ dove } \bar{L} = \Sigma^* \setminus L$$

Dimostrazione: Bisogna rendere totale la δ di N , eventualmente aggiungendo stati non finali, e poi N' si ottiene da questo N "aumentato", semplicemente scambiando **finali e non finali** ⊗

Proposition 6.1.3 Non chiusura per intersezione

Un linguaggio libero deterministico non è chiuso per intersezione

Example 6.1.4

$L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ è libero deterministico

$L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$ è libero deterministico

ma

$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ non è libero!

Proposition 6.1.4 non chiusura per unione

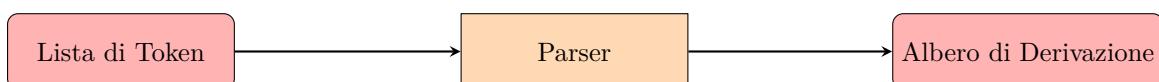
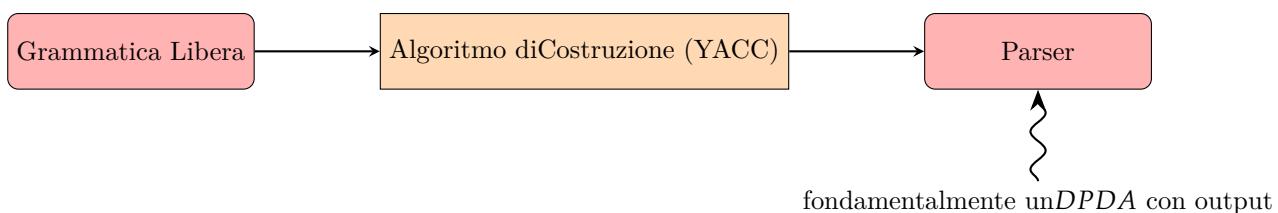
Un linguaggio libero deterministico non è chiuso per unione

Dimostrazione: Assumiamo per assurdo che un linguaggio libero deterministico sia chiuso per unione, allora:

$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$$

Per questo fatto e la proposizione precedente si è verificato un assurdo ⊗

6.1.3 Analizzatori sintattici: parser



I parser possono essere:

- **nondeterministici**: se, durante la ricerca di una derivazione, si scopre che una scelta è improduttiva e non porta a riconoscere l'input, il parser torna indietro (*backtracking*), disfa parte della derivazione appena costruita e scegli un'altra produzione, **tornando a leggere parte dell'input**
- **deterministici**: leggono l'input una sola volta ed **ogni loro decisione è definitiva**

entrambi cercano di sfruttare informazioni dall'input per guidare la ricerca della derivazione

introduzione al top-down parsing

Definition 6.1.3

Data $G = (NT, T, S, R)$ lebrea, costruiamo il *PDA* $M = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$, che riconosce per pila vuota, dove $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^* \times \Gamma^*$ è definita:

- **espandi**: $(q, \beta) \in \delta(q, \epsilon, A)$ se $A \rightarrow \beta \in R$
- **consuma**: $\forall a \in T ((q, \epsilon) \in \delta(q, a, a))$

Tale che $L(G) = P[M]$ (riconoscimento per pila vuota)

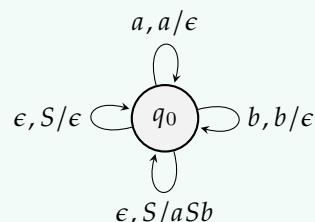
Quindi, grazie alla prima regola se il simbolo in cima alla pila è un non terminale A , l'automa può espanderlo sostituendolo con la produzione β , come descritto nelle regole R della grammatica, mentre secondo la regola di consumazione si ha che se il simbolo sulla cima della pila e il simbolo corrente della stringa in input sono entrambi a , allora l'automa può eliminare quel simbolo dalla pila e procedere nella lettura dell'input.

Example 6.1.5

Sia G la seguente grammatica:

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

Si ha che $S \implies aSb \implies aaSbb \implies aabb$

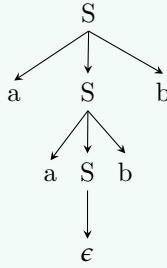


Si osservi che il seguente automa che riconosce il linguaggio della grammatica S non è un PDPA

Nella maggior parte delle configurazioni standard di un Pushdown Automaton (PDA) utilizzato per il parsing top-down, la pila viene inizializzata con il simbolo iniziale della grammatica, nel nostro caso, quindi, la serie di passaggi che porteranno a riconoscere il linguaggio sarà:

$$\begin{aligned}
 & (q, aabb, S) \vdash (q, aabb, aSb) \\
 & \vdash (q, abb, Sb) \vdash (q, abb, aSbb) \\
 & \vdash (q, bb, Sbb) \vdash (q, bb, bb) \\
 & \vdash (q, b, b) \vdash (q, \epsilon, \epsilon)
 \end{aligned}$$

che costruisce il seguente albero di derivazione



Si ha, quindi:

- derivazione canonica a sx (leftmost)
- costruzione dell'albero dall'alto in basso

Tuttavia in questo esempio è facile verificare che il parser è nondeterministico, infatti l'automa è un *PDA*, tuttavia questo nondeterminismo può essere risolto scegliendo la produzione in base al simbolo di lettura nell'input (**look-ahead**), ad esempio:

- se leggo *a*, espando $S \rightarrow aSb$
- se leggo *b*, espando $S \rightarrow \epsilon$

input	stack	azione
<u><i>aabb\$</i></u>	<u><i>S</i></u>	leggo <i>a</i> e espando in aSb
	<u><i>aSb</i></u>	consumo
<u><i>abb\$</i></u>	<u><i>Sb</i></u>	leggo <i>a</i> e espando in aSb
	<u><i>aSbb</i></u>	consumo
<u><i>bb\$</i></u>	<u><i>Sbb</i></u>	leggo <i>b</i> e espando in ϵ
	<u><i>bb</i></u>	consumo
<i>b\$</i>	<i>b</i>	leggo <i>b</i> e espando in ϵ
<i>\$</i>	ϵ	fin

Note:

Non tutte le grammatiche sono adatte per il top down-parser

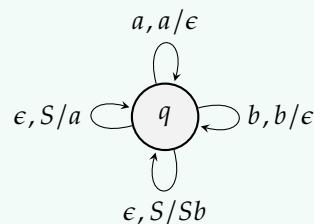
Example 6.1.6

Sia G la seguente grammatica:

$$S \rightarrow Sb \mid a$$

e $L(G) = ab^*$ (linguaggio regolare semplice)

L'automa che riconosce il linguaggio è:



Con i seguenti passaggi:

$$(q, ab, S) \vdash (q, ab, Sb) \vdash (q, ab, Sbb) \vdash \dots$$

Qui il determinismo non funziona, infatti se vogliamo espandere quando leggiamo *a* in input non possiamo anche consumare, pertanto l'automa espanderà all'infinito

Occorre manipolare la grammatica affinche non vi siano ricorsioni sinistre

Introduzione al botto-up parsing

Claim 6.1.2

Data una grammatica libera $G = (NT, T, R, S)$, costruiamo un PDPA $M = (T, \{q\}, T \cup NT \cup \{Z\}, \delta, q, Z, \emptyset)$ che riconosce $L(G).$ \$ dove:

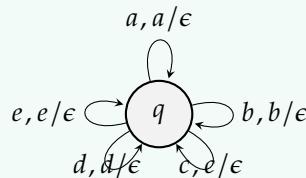
- **shift:** $\forall a \in T, (\forall X \in T \cup NT \cup Z, ((q, aX) \in \delta(q, a, Z)))$
- **reduce:** $(A \rightarrow \alpha R) \implies (q, A) \in \delta(q, \epsilon, \alpha^R)$
Con α^R una generalizzazione dei PDA in cui si consuma una stringa sulla pila anziché solo il top
- **accept:** $(q, \epsilon) \in \delta(q, \$, SZ)$
Con S che deve essere alla fine sulla pila e $\$$ simbolo di fine input

Cominciamo subito con un esempio

Example 6.1.7

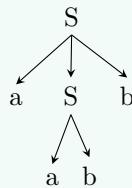
Sia G la seguente grammatica:

$$S \rightarrow aSb \mid ab$$



stack	input	azione
Z	$aabb\$$	shift
Za	$abb\$$	shift
Zaa	$bb\$$	shift
$Zaab$	$b\$$	reduce $S \rightarrow ab$
ZaS	$b\$$	shift
$ZaSb$	$\$$	reduce $S \rightarrow aSb$
ZS	$\$$	accept
ϵ	ϵ	

In passaggi, $S \implies aSb \implies aabb$. Viene prodotto il seguente albero di derivazione:



Si può verificare come la costruzione dell'input sia:

- **left-to-right** (come leggo l'input)
- **right-derivation**
-

Tuttavia è facile verificare che c'è molto nondeterminismo:

- Conflitti: Shift-Reduce

1. $zaab \ b\$$ (può fare shift)
2. $zaabb \ \$$ (ma è un percorso infruttuoso)

- Conflitti: Reduce-Reduce

Più riduzioni possibili (non ci sono in quest'esempio, ma con grammatiche più complesse è possibile)

Per ottenere un DPDA, serve introdurre informazioni aggiuntive per risolvere i conflitti:

- **Più stati:** (o strutture particolari di supporto alle decisioni, come un DFA dei prefissi validi).
- **Look-ahead:** (guardare l'input in avanti).

Vediamo un esempio più corposo relativo alle grammatiche delle espressioni aritmetiche:

Example 6.1.8

Sia G tale grammatica:

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow T \times A \mid A \\ A &\rightarrow a \mid b \mid (E) \end{aligned}$$

Si ha:

G	M
(R1) $E \rightarrow T + E$	$t_0 : \delta(q, a, X) = (q, aX) \quad \forall aX \text{ SHIFT}$
(R2) $E \rightarrow T$	$t_1 : \delta(q, \epsilon, E + T) = (q, E)$
(R3) $T \rightarrow T * A$	$t_2 : \delta(q, \epsilon, T) = (q, E)$
(R4) $T \rightarrow A$	$t_3 : \delta(q, \epsilon, A * T) = (q, T) \quad \text{REDUCE}$
(R5) $A \rightarrow (E)$	$t_4 : \delta(q, \epsilon, A) = (q, T)$
(R6) $A \rightarrow a$	$t_5 : \delta(q, \epsilon, (E)) = (q, A)$
(R7) $A \rightarrow b$	$t_6 : \delta(q, \epsilon, a) = (q, A)$
	$t_7 : \delta(q, \epsilon, b) = (q, A)$
	$t_8 : \delta(q, \$, EZ) = (q, \epsilon) \quad \text{ACCEPT}$

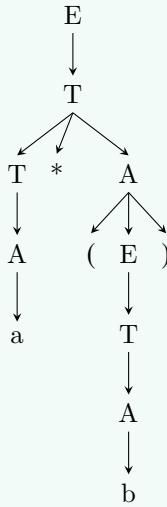
La cui sequenza è:

$$a * (b) \Leftarrow A * (b) \Leftarrow T * (b) \Leftarrow T * (A) \Leftarrow T * (T) \Leftarrow T * (E) \Leftarrow T * A \Leftarrow T \Leftarrow E$$

La cui pila-stack-input è:

Stack	Input	Action
Z	$a * (b)\$$	shift
Za	$*(b)\$$	reduce R_6
ZA	$*(b)\$$	reduce R_4
ZT	$*(b)\$$	shift
ZT*	$(b)\$$	shift
ZT*(b)	$b)\$$	shift
ZT*(b)	$)\$$	shift
ZT*(b)	$\$$	reduce R_7
ZT*(A)	$)\$$	reduce R_4
ZT*(T)	$)\$$	reduce R_2
ZT*(E)	$)\$$	shift
ZT*(E)	$\$$	reduce R_5
ZT*A	$\$$	reduce R_3
ZT	$\$$	reduce R_2
ZE	$\$$	accept
ϵ	ϵ	

Il cui albero di derivazione:



L'automa è non deterministico perché:

- Chi ha precedenza tra *shift* e *reduce*?

Ad esempio:

- $Za * (b) \$$ *shift*
- $Za * (b) \$$ non buono! Qui *reduce* è in conflitto con *shift*!

Altro esempio:

- $ZT * (b) \$$ *reduce* R_2
- $ZE * (b) \$$ non buono! Qui *shift* è in conflitto con *reduce*!

- Chi ha precedenza tra due diverse *reduce*?

Ad esempio:

- $ZT * A \$$ *reduce* R_3
- $ZT \$$
- $ZT * T \$$ se faccio, *reduce* R_4
Qui *reduce* R_3 è in conflitto con *reduce* R_4 !

È buona norma scegliere in modo tale che ciò che si trova nella pila sia un prefisso (a rovescio) di una parte destra di una produzione della grammatica.

Ad esempio:

- Za
- ZA con la “norma” produco chi coincide con ciò che è un prefisso di una parte destra che comincia con A .

problema con le pruzioni ϵ

Sia la produzione $S \rightarrow \epsilon$

TODO! NON HO CAPITO

6.2 Semplificazione delle grammatiche

Per avere **PDA** efficienti e con minor non determinismo è necessario semplificare le grammatiche. Ad esempio:

- Eliminare le produzioni ϵ (del tipo $A \rightarrow \epsilon$) inadatte al bottom up parsing
- Eliminare le **produzione unitarie** (del tipo $A \rightarrow B$ che possono creare dei cicli $A \implies^+ A$)
- Eliminare **simboli inutili**, cioè quei terminali e non terminali che non sono raggiungibili/generabili a partire dal simbolo iniziale S
es. gli stati d'errore
- Eliminare la **ricorsione sinistra** (del tipo $A \rightarrow A\alpha$), perché inadatte al top - down parsing
- **fattorizzare** le grammatiche, per ottenere grammatiche con meno non determinismo nel top-down parsing

6.2.1 Eliminare le produzioni ϵ

Per fare ciò si usa un algoritmo che ha:

- in **input**: una G libera con produzione ϵ
- in **output**: una G' libera senza produzione ϵ tale che $L(G') = L(G) \setminus \{\epsilon\}$

Note:

Se $\epsilon \in L(G)$ e si vuole ottenere una G'' t.c. $L(G) = L(G'')$, basta considerare $G' = (NT, T, S, R')$ e definire $G'' = G' \cup \{S' \rightarrow \epsilon | S\}$ t.c.

$$G'' = (NT \cup \{S'\}, T, S', R' \cup \{S' \rightarrow \epsilon | S\})$$

simboli annullabili

Per l'algoritmo occorre innanzi tutto definire i **simboli annullabili**

Definition 6.2.1: simboli annullabili

I **simboli annullabili** sono quei non terminali tale che possono riscriversi in uno o più passi in ϵ , ovvero:

$$N(G) = \{A \in NT | A \implies^+ \epsilon\}$$

Dove $N(G)$ è l'insieme dei simboli annullabili e viene calcolato induttivamente come segue:

- $N_0(G) = \{A \in NT | A \rightarrow \epsilon\}$, questo è il caso in cui un non terminale A viene riscritto direttamente in ϵ tramite una produzione
- $N_{i+1}(G) = N_i(G) \cup \{B \in NT | B \rightarrow c_1, \dots, c_k \in \Sigma \text{ e } c_1, \dots, c_k \in N_i(G)\}$ questo è il caso in cui un non terminale B possa essere ricondotto a ϵ in più passi

Ovviamente $\exists i_c$ tale che $N_{i_c}(G) = N_{i_c+1}(G)$, cioè che ad un certo punto non aggiungo nessun altro B all'insieme (NT è finito)

Theorem 6.2.1

L'insieme $N(G) = N_{i_c}(G)$ è esattamente l'insieme di tutti i simboli annullabili

algoritmo per il calcolo della grammatica

Una volta calcolato $N(G)$ per $G = (N, T, S, R)$, costruiamo la grammatica $G' = (N, T, S, R')$ dove per ogni produzione $A \rightarrow \alpha \in R$ con $\epsilon \notin \alpha$, in cui occorrono simboli annullabili Z_1, \dots, Z_k , mettiamo in R' tutte le produzioni del tipo $A \rightarrow \alpha'$ dove α' si ottiene da α cancellando tutti i possibili sottoinsiemi di Z_1, \dots, Z_k (incluso \emptyset), ad eccezione del caso in cui α' risulta ϵ , in altre parole si creano tutte le possibili combinazioni di α eliminando uno o più di questi simboli Z_1, \dots, Z_n :

- in G' non mettiamo produzioni $A \rightarrow \epsilon \in R$,
- in G' non introduciamo mai produzioni del tipo $A \rightarrow \epsilon$.

Theorem 6.2.2

Data una grammatica libera G , la grammatica G' determinata dall'algoritmo sopra non ha ϵ -produzioni, e $L(G') = L(G) \setminus \{\epsilon\}$

Example 6.2.1

Sia G una grammatica tale che:

$$G = \begin{cases} S \rightarrow AB \\ A \rightarrow aAA \mid \epsilon \\ B \rightarrow bBB \mid \epsilon \end{cases} \quad N_0(G) = \{A, B\} \text{ e } N_1(G) = \{A, B, S\} = N(G)$$

Quindi tutti sono simboli annullabili

Adesso procedo con l'algoritmo, procedo per ogni simbolo annullabili

- $S \rightarrow AB$: secondo l'algoritmo devo cancellare in tutti i modi possibili i simboli non terminali che compaiono nella parte destra della produzione. In questo caso dobbiamo considerare 4 casi:
 - $\emptyset \implies S \rightarrow AB$, rinuncio a cancellare
 - $\{B\} \implies S \rightarrow A$, se cancello B rimane A nella parte destra
 - $\{A\} \implies S \rightarrow B$
 - $\{A, B\} \implies S \rightarrow \epsilon$ dato che non devo mai introdurre produzioni del tipo $A \rightarrow \epsilon$ **non posso cancellare A e B**

Unisco i vari sottoinsiemi e si ha:

$$S \rightarrow AB|B|A$$

- $A \rightarrow aAA \in R$. dobbiamo considerare 4 casi:

- $\emptyset \implies A \rightarrow aAA$
- $\{A\} \implies A \rightarrow aA$
- $\{A, A\} \implies A \rightarrow a$

Quindi si ha:

$$A \rightarrow aAA|aA|a$$

- si ha la stessa cosa con B , quindi cancellarehe verrà trasformato in

$$B \rightarrow bBB|bB|b$$

Così la nuova grammatica sarà:

$$G' = \begin{cases} S \rightarrow AB|A|B \\ A \rightarrow aAA|aA|a \\ B \rightarrow bBB|bB|b \end{cases}$$

6.2.2 Eliminazione delle produzioni unitarie

Definition 6.2.2: Produzione unitaria

Una produzione si dice **unitaria** quando $A \rightarrow B$ si ha che $A, B \in NT$

coppie unitarie

Per eliminare queste produzioni unitarie si deve però calcolare quelle che sono definite le "coppie unitarie"

Definition 6.2.3: Coppia unitaria

Una coppia (A, B) si dice **unitaria** quando $A \implies {}^*B$ (quindi quando A può riscriversi in 0 o più passi nel non terminale B) usando solo produzioni unitarie.

Vi è qui ripostata la definizione induttiva:

- $U_0(G) = \{(A, A) | A \in NT\}$, quindi ogni non terminale fa coppia con se stesso
- $U_{i+1}(G) = U_i(G) \cup \{(A, C) | (A, B) \in U_i(G)\} \text{ e } B \rightarrow C \in R$, quindi è l'insieme delle coppie al passo i unito alle coppie alle coppie (A, C) tali che (A, B) sono coppie presenti nell'insieme dell'iterazione precedente e $B \rightarrow C \in R$

Anche in questo caso $\exists i_c$ t.c. $U_{i_c}(G) = U_{i_c+1}(G)$ dato che NT è finito. Pertanto per definizione si ha che $U(G) = U_{i_c}(G)$, detto insieme di tutte le coppie unitarie

algoritmo per il calcolo dell'eliminazione delle produzioni unitarie

Data $G = (N, T, R, S)$ libera, si definisce una $G' = (N, T, R', S)$ dove, per ogni $(A, B) \in U(G)$, R' contiene tutte le produzioni $A \rightarrow \alpha$, dove $B \rightarrow \alpha \in R$ e non è unitaria

Note:

Poiché, per ogni $A \in N$, la coppia $(A, A) \in U(G)$, R' contiene tutte le produzioni non unitarie di R e in aggiunta un po' di altre.

Theorem 6.2.3

Sia $G = (NT, T, R, S)$ libera e sia $U(G)$ l'insieme delle sue coppie unitarie. Sia $G' = (NT, T, R', S)$ la grammatica ottenuta dall'algoritmo G' non ha produzione unitarie e $L(G) = L(G')$

Esempio:

Example 6.2.2

Prediamo con esempio la grammatica non ambigua E delle espressioni aritmetiche:

$$E = \begin{cases} E \rightarrow E + T | T \\ T \rightarrow T * A | A \\ A \rightarrow a | b | (E) \end{cases}$$

Si noti subito che ha 2 produzioni unitarie: $E \rightarrow T$ e $T \rightarrow A$. Iniziamo a calcolare l'insieme delle coppie unitarie:

- $U_0(G) = \{(E, E), (T, T), (A, A)\}$ e grazie al cuzzo
- $U_1(G) = U_0(G) \cup \{(E, T), (T, A)\}$
- $U_2(G) = U_1(G) \cup \{(E, A)\} = U_3(G) = U(G)$ dato che da E si arriva ad T e si arriva A

Per calcolare la grammatica G' devo prendere tutte le produzioni non unitarie della grammatica originale, ovvero:

$$G' = \begin{cases} E \rightarrow E + T \\ T \rightarrow T * A \\ A \rightarrow a | b | (E) \end{cases}$$

In aggiunta:

$$\begin{cases} E \rightarrow E + T & \text{perché } (E, T) \in U_1(G) \\ T \rightarrow a|b|(E) & \text{perché } (T, A) \in U_1(G) \\ E \rightarrow a|b|(E) & \text{perché } (E, A) \in U_2(G) \end{cases}$$

Pertanto G' sarà:

$$G' = \begin{cases} E \rightarrow E + T | T \times A | a|b|(E) \\ T \rightarrow T \times A | a|b|(E) \\ A \rightarrow a|b|(E) \end{cases}$$

Sia ha che non contiene produzioni unitarie ed è equivalente a G

6.2.3 Rimuovere i simboli inutili

Definition 6.2.4: Simboli generatori, raggiungibili e utili

Un simbolo $X \in T \cup NT$ è

- Un **generatore** $\iff \exists w \in T^* \text{ con } x \implies {}^*w$

Quindi un generatore è o un terminale (un simbolo può risciversi in se stesso) oppure un non terminale che in uno o più passi. è definito induttivamente come segue:

- $G_0(G) = T$ se $a \in T, a \implies {}^*a$ (quindi tutti i terminali sono generatori)
- $G_{i+1}(G) = G_i(G) \cup \{B \in NT | B \rightarrow C_1, \dots, C_k \in R \wedge C_1, \dots, C_k \in G_i(G)\}$

- Un **raggiungibile** $\iff (\exists \alpha, \beta \in (T \cup NT)^*. (S \implies {}^*\alpha X \beta))$. Sono definiti induttivamente:

- $R_0(G) = \{S\}$
- $R_{i+1}(G) = R_i(G) \cup \{x_1, \dots, x_k\} \forall B \in R_i(G), B \rightarrow x_i, \dots, x_k \in R$

- **utile** sse è sia un generatore e sia raggiungibile, ovvero se $S \implies {}^*\alpha X \beta \implies {}^*x \in L(G)$ cioè X compare in almeno una derivazione di una stringa $z \in L(G)$

algoritmo per l'eliminazione dei simboli inutili

1. Prima di tutto elimino tutti i non-generatori (e tutte le produzione che usano almeno uno di questi)
2. Poi dalla nuova grammatica, elimino tutti i non - raggiungibili (E tutte le produzioni che li usano)

Theorem 6.2.4

Sia $G = (NT, T, R, S)$ una grammatica libera t.c. $L(G) \neq \emptyset$

- Sia G_1 la grammatica che si ottiene da G eliminando tutti i simboli che non appartengono a $G(G)$ (insieme dei generatori), e tutte le produzioni che fanno uso di algoritmo di tali simboli
- Sia G_2 la grammatica che si ottiene da G_1 eliminando tutti i simboli che non appartengono a $R(G)$, e tutte le produzioni che fanno uso di almeno uno di tali simboli

Allora G_2 non ha simboli inutili e $L(G_2) = L(G)$

Dimostrazione: La dimostrazione si divide nelle due parti dell'enunciato:

- $L(G_2) \subseteq L(G)$ è ovvio, dato che G_2 contiene meno produzioni di G

- $L(G) \subseteq L(G_2)$: dobbiamo dimostrare che $S \implies^* G w$ (ovvero se S deriva w usando le produzioni di G) allora $S \implies^* G_2 w$

Si ha che ogni simbolo usato in $S \implies^* G w$ è, ovviamente, sia raggiungibile sia generatore

Quindi quelle derivazioni è anche una derivazione per G_2

Q.e.d.



Note:

L'ordine dei due generatori è importante!

- prima elimino i non-generatori
- poi i non - raggiungibili

ma se inverti l'ordine, allora può capitare che non elimino tutti i simboli inutili

Esempio di eliminazione di tutti quei simboli non utili (inutili)

Example 6.2.3

Si parta da questa grammatica:

$$G = \begin{cases} S \rightarrow AB|a \\ B \rightarrow b \end{cases}$$

Poiché $a \implies^* a, b \implies^* b, S \implies^* a, B \implies^* b$ si ha che i generatori saranno $\{S, B, a, b\}$ (dove manca A). Possiamo così eliminare tutte le produzioni che includono A :

$$G' = \begin{cases} S \rightarrow a \\ B \rightarrow b \end{cases}$$

Adesso posso eliminare tutti i non raggiungibili da S , che in questo caso l'unico è solo B . Si ha che:

$$G'' = S \rightarrow a$$

Si ha che G'' è equivalente a G , ma non contiene simboli utili

Esempio secondo:

Example 6.2.4

$$G = \begin{cases} S \rightarrow aC \\ A \rightarrow a \\ B \rightarrow bB \\ C \rightarrow b \mid AC \\ D \rightarrow a \mid aS \end{cases}$$

Poiché $G(G) = \{S, a, C, b, A, D\}$ e solo B non è generatore, possiamo eliminare tutte le produzioni che includono B . Si ha quindi:

$$G_1 = \begin{cases} S \rightarrow aC \\ A \rightarrow a \\ C \rightarrow b \mid AC \\ D \rightarrow a \mid aS \end{cases}$$

A questo punto, notiamo che solo D non è raggiungibile, quindi possiamo eliminarlo. Si ottiene:

$$G_2 = \begin{cases} S \rightarrow aC \\ A \rightarrow a \\ C \rightarrow b \mid AC \end{cases}$$

G_2 è la grammatica semplificata, equivalente a G , senza simboli inutili.

In questo esempio si ha che $L(G_2) = \{ab, aab, aaab, \dots\} = a^+b$

Si osservi però che è possibile trovare una grammatica più semplice per il linguaggio a^+b :

$$S \rightarrow aS|ab$$

6.2.4 mettere insieme le cose

Se, nel semplificare la grammatica G , **seguiamo questo ordine**:

- Eliminare le ϵ -produzioni
- Eliminare le produzioni unitarie (ovvero i cicli)
- eliminare i simboli inutili

allora la grammatica risultante è garantita non avere né ϵ -produzioni, ne produzioni unitarie, ne simboli inutili ed è equivalente a quella di partenza.

Note:

Si presti attenzione all'ordine poiché alcune delle costruzioni possono interagire tra di loro durante la fase di eliminazione delle ϵ -produzioni, potremmo introdurre produzioni unitarie, pertanto le ϵ -produzioni vanno eliminate prima della fase di eliminazione delle produzioni unitarie

Esempietto:

Example 6.2.5

$$G = \begin{cases} S \rightarrow aAa \mid aa \\ A \rightarrow C \\ C \rightarrow S \mid \epsilon \end{cases}$$

1. **Togliere le ϵ -produzioni**

$$N(G) = \{S, C, A\} \Rightarrow G' = \begin{cases} S \rightarrow aAa \mid aa \\ A \rightarrow C \\ C \rightarrow S \end{cases}$$

2. **Togliere le produzioni unitarie**

$$U(G') = \{(A, A), (C, C), (S, S), (A, C), (C, S), (A, S)\}$$

$$G'' = \begin{cases} S \rightarrow aAa \mid aa & \text{perché } (S, S) \in U(G') \\ C \rightarrow aAa \mid aa & \text{perché } (C, S) \in U(G') \\ A \rightarrow aAa \mid aa & \text{perché } (A, S) \in U(G') \end{cases}$$

3. **Rimuovere i simboli inutili**

$$G(G'') = \{S, a, A, C\} \quad \text{tutti i generatori}$$

$$R(G'') = \{S, a, A\} \quad \text{ma non } C$$

$$G''' = \begin{cases} S \rightarrow aAa \mid aa \\ A \rightarrow aAa \mid aa \end{cases}$$

In questo esempio si ha che $L(G''') = \{aa, aaaa, \dots\} = (aa)^+$

Si osservi però che è possibile trovare una grammatica più semplice per il linguaggio $(aa)^+$:

$$S \rightarrow aSa \mid aa$$

o anche

$$S \rightarrow aaS \mid aa$$

6.2.5 forme normali

le **forme normali** sono particolari configurazioni di rappresentazione di un linguaggio formale o di un'espressione logica che rispettano determinate regole e strutture. Ne studieremo di due tipi:

- **Chomsky:** Una grammatica è in forma normale di Chomsky se ogni produzione ha la forma $A \rightarrow BC$ o $A \rightarrow a$, dove $A, B, C \in NT$ e $a \in T$. Ogni produzione deriva quindi o una coppia di variabili o un singolo terminale. Questa forma è utile, per esempio, negli algoritmi di parsing
- **Greibach:** Una grammatica è in forma normale di Greibach se ogni produzione ha la forma $A \rightarrow a\alpha$, dove $A \in NT$, $a \in T$ e α è (eventualmente) una stringa di variabili. La GNF è usata in particolare per costruire parser discendenti

Forma normale di Chomsky

Definition 6.2.5: Forma normale di Chomsky

Una grammatica si dice in **forma normale di Chomsky** se sono nella forma:

$$\begin{array}{l} A \rightarrow BC \\ A \rightarrow a \end{array}$$

Dove ϵ è trattato a parte $S \rightarrow \epsilon|BC$ e S non compare mai a destra in una produzione

Note:

se G è libera in forma normale di Chomsky, allora:

- non ha ϵ -produzioni
- non ha produzioni unitarie

Note:

ogni grammatica libera G può essere trasformata in una equivalente G' in forma normale di Chomsky

Forma normale di Greibach

Definition 6.2.6: forma normale di Greibach

Una grammatica si dice in **forma normale di Greibach** se sono nella forma:

$$\begin{array}{l} A \rightarrow aBC \\ A \rightarrow aB \\ A \rightarrow a \end{array}$$

Dove ϵ è trattato a parte $S \rightarrow \epsilon|BC$ e S non compare mai a destra in una produzione

Note:

se G è libera in forma normale di Greibach, allora:

- non ha ϵ -produzioni
- non ha produzioni unitarie
- non è ricorsiva a sinistra
- ogni produzione applicata in una derivazione allunga il prefisso di terminali \implies il parser costruito a partire della forma normale di Greibach sono meno non deterministici

Note:

ogni grammatica libera G può essere trasformata in una equivalente G' in forma normale di Greibach

6.2.6 Eliminare la ricorsione a sinistra

L'eliminazione della ricorsione a sinistra è un problema tipico dei parser top-down

Definition 6.2.7: produzione ricorsiva a sinistra

Si definisce **una produzione ricorsiva a sinistra** una produzione del tipo

$$A \rightarrow A\alpha \in R$$

Definition 6.2.8: grammatica ricorsiva a sinistra

Si definisce una **grammatica ricorsiva a sinistra** una grammatica G del tipo:

$$A \implies {}^+ A\alpha \text{ per qualche } A \in NT, \alpha \in (T \cup NT)^*$$

Una tipica ricorsione a sinistra è:

$$A \rightarrow A_{\alpha_1} | \dots | A_{\alpha_n} | \beta_1 | \dots | \beta_n$$

Dove le stringhe β_i non cominciano per A . Queste produzioni possono essere rimpiazzate da

$$\begin{aligned} A &\rightarrow \beta_1 A' | \dots | \beta_m A' \\ A' &\rightarrow \alpha_1 A' | \dots | \alpha_n A' | \epsilon \end{aligned}$$

Se nella grammatica originale avviamo la derivazione

$$A \implies A\alpha_{i_1} \implies A\alpha_{i_2}\alpha_{i_1} \implies \dots \implies A\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1} \implies \beta_i\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1}$$

Con la nuova grammatica si ha:

$$A \implies \beta_i A' \implies \beta_i\alpha_{i_k} A' \implies \dots \implies \beta_i\alpha_{i_k} \dots \alpha_{i_2} A' \implies \beta_i\alpha_{i_k} \dots \alpha_{i_1} A' \implies \beta_i\alpha_{i_k} \dots \alpha_{i_1}$$

Esempi concreti:

Example 6.2.6

$$\begin{aligned} A &\rightarrow Aa|b \\ &\Rightarrow \\ A &\rightarrow bA' \\ A' &\rightarrow aA'|\epsilon \end{aligned}$$

Poi

$$\begin{aligned} A &\rightarrow Ab|Ac|d \\ &\Rightarrow \\ A &\rightarrow dA' \\ A' &\rightarrow bA'|cA'|e \end{aligned}$$

Note:

Se $G = [A \rightarrow Aa]$, non si può applicare l'algoritmo perché mancano le produzioni di base da cui partire ($A \rightarrow \beta_1 | \dots | \beta_m$). Infatti, $L(G) = \emptyset$ e la grammatica corrispondente non ha produzioni

6.2.7 Ricorsione sx non-immediata

Consideriamo

$$G = \begin{cases} S \rightarrow Ba|b \\ B \rightarrow Bc|Sc|d \end{cases}$$

In G c'è ricorsione sx immediata ($B \rightarrow Bc$) ma anche non immediata ($S \implies Ba \implies Sca$)

algoritmo per il calcolo della ricorsione non immediata

Algorithm 4: ricorsione non immediata

Input: una G libera senza ϵ -prod, senza produzioni unitarie, ma con ricorsione sc non immediata

Output: una G libera senza ϵ -prod, senza produzioni unitarie e senza alcuna ricorsione a sx

- 1 Let $NT = \{A_1, A_2, \dots, A_n\}$ in un ordine fissato;
 - 2 **for** $i = 1$ to n **do**
 - 3 **for** $j = 1$ to $i - 1$ **do**
 - 4 Sostituisce ogni produzione della forma $A_i \rightarrow A_j \alpha$ con le produzioni $A_i \rightarrow \beta_1 \alpha | \dots | \beta_k \alpha$, dove $A_j \rightarrow \beta_1 | \dots | \beta_k$ sono produzioni correnti per A_j ;
 - 5 Elimina la ricorsione immediata su A_i ;
-

- L'obiettivo dell'algoritmo è che, alla fine, ogni produzione del tipo $A_i \rightarrow A_k \alpha$ sia tale che $i < k$, in modo che sia impossibile avere ricorsione sx non immediata.
- Quando $i = 1$, l'unica cosa che viene fatta è l'istruzione 2), che rimuove l'eventuale ricorsione sx immediata. Al termine, $A_1 \rightarrow A_k \alpha$ avremo $i < k$.
- Alla i -esima iterazione del **for** esterno, tutti i non-terminali A_m con $m < i$ hanno produzioni con la proprietà desiderata.
Ora il ciclo **for** interno (istruzione 1) aumenta progressivamente l'indice del non-terminali in prima posizione; finché, al termine del ciclo ($j = i - 1$), avremo che ogni produzione $A_i \rightarrow A_k \alpha$ è tale che $i < k$.
Ora l'istruzione 2) rimuove l'eventuale ricorsione sx immediata da A_i , sicché ogni produzione $A_i \rightarrow A_k \alpha$ è tale che $i < k$.
- Quindi al termine dell'algoritmo, avremo che ogni produzione $A_i \rightarrow A_k \alpha$ è tale che $i < k$, garantendo l'impossibilità di creare ricorsione sx non immediata.

Example 6.2.7

Come esempio si consideri la grammatica di prima, ovvero

$$G = \begin{cases} S \rightarrow Ba|b \\ B \rightarrow Bc|Sc|d \end{cases}$$

Si segua passo-passo l'algoritmo <3:

- **i=1** (ovvero S): il ciclo interno non viene eseguito e, siccome non c'è ricorsione immediata per S , non viene fatto nulla
- **i=2** (cioè $A_i = B$): il ciclo interno (j da 1 a 1) si esegue solo per $A_j = A_1 = S$.

Allora la produzione $B \rightarrow Sc$ viene rimpiazzata con:

$$B \rightarrow Bac|bc$$

Ora le produzioni complessive per B sono :

$$B \rightarrow Bc|Bac|bc|d$$

Dalla quale dobbiamo eliminare la ricorsione immediata, il risultato è:

$$\begin{aligned} B &\rightarrow bcB'|sB' \\ B' &\rightarrow cB'|acB'|\epsilon \end{aligned}$$

Pertanto la gigagrammatica risultante è:

$$\begin{aligned} S &\rightarrow Ba|B \\ B &\rightarrow bcB'|sB' \\ B' &\rightarrow cB'|acB'|\epsilon \end{aligned}$$

6.2.8 Fattorizzazione a sinistra

Si prendi in esempio la seguente grammatica:

$$A \rightarrow aBbC|aBd$$

Se, in un top-down parsing, sulla pila ha A e leggo in input a , non sono in grado di determinare quale produzione scegliere tipico del nondeterminismo, pertanto occorre **raccogliere la parte comune (aB) alle 2 produzioni e introdurre un nuovo nonterminale per rappresentare il resto delle produzione**, quindi:

$$\begin{aligned} A &\rightarrow aBA' \\ A' &\rightarrow bC|d \end{aligned}$$

algoritmo per il calcolo della fattorizzazione

Algorithm 5: Fattorizzazione LU

Input: Grammatica G non fattorizzata

Output: Grammatica G' fattorizzata

```
1 Let  $N$  be a new variable;  
2  $N \leftarrow NT;$   
3 while è possibile modificare a  $N$  o all'insieme delle produzione do  
4   foreach  $A \in N$  do  
5     Sia  $\alpha$  il prefissio più lungo comune alle parti destre di alcune produzione di  $A$ ;  
6     if  $\alpha \neq \epsilon$  then  
7       Sia  $A$  un nuovo non terminale ;  
8        $N \leftarrow N \cup \{A\};$   
9       rimpiazza tutte le produzione per  $A$  del tipo  
          
$$A \rightarrow \alpha\beta_1|\dots|\alpha\beta_k|\gamma_1|\dots|\gamma_h$$
  
          con le produzioni:  
          
$$A \rightarrow \alpha A'|\gamma_1|\dots|\gamma_h$$
  
          
$$A' \rightarrow \beta_1|\dots|\beta_k$$

```

Example 6.2.8

Riporto una grammatica da fattorizzare:

$$\begin{aligned} E &\rightarrow T|T+E|T-E \\ T &\rightarrow A|A*T \\ A &\rightarrow a|b|(E) \end{aligned}$$

Dove sia E che T si possono fattorizzare, perciò diventa:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow \epsilon|+E|-E \\ T &\rightarrow AT'| \\ T' &\rightarrow \epsilon|*T \\ A &\rightarrow a|b|(E) \end{aligned}$$

Chapter 7

Parser Top-Down

Un **parser Top-Down** è un tipo di analizzatore sintattico per analizzare strutture gerarchiche, come le frasi di una lingua o la struttura di un codice. Funziona esplorando e costruendo l'albero sintattico partendo dalla radice e procedendo verso le foglie, quindi "dall'alto verso il basso"

Adesso presentiamo un primo esempio di parser Top-Down **nondeterministico** che usa implicitamente una pila per gestire le chiamate ricorsive

7.1 Parser a discesa ricorsiva

Data una grammatica libera $G = (NT, T, S, R), \forall A \in NT$ con produzioni:

$$A \rightarrow X_1^1 \dots X_{n_1}^1 | \dots | X_k^1 \dots X_{n_k}^k$$

Definisce la funzione

Algorithm 6: A()

```
1 scegli non deterministicamente  $h$  tra 1 e  $k$ , ovvero una produzione  $X_k^h \dots X_{n_h}^h$ ;  
2 for  $i = 1$  to  $n_h$  do  
3   if  $X_i^h \in NT$  then  
4     Nothing();  
5   else if  $X_i^h = \text{ simbolo corrente dell'input}$  then  
6     avanza di un simbolo nell'input;  
7   else  
8     Fail(); // backtracking! si torna alla riga 2 e si sceglie un'altra produzione  
9   return;
```

Si comincia invocando la funzione per il simbolo iniziale S

Example 7.1.1

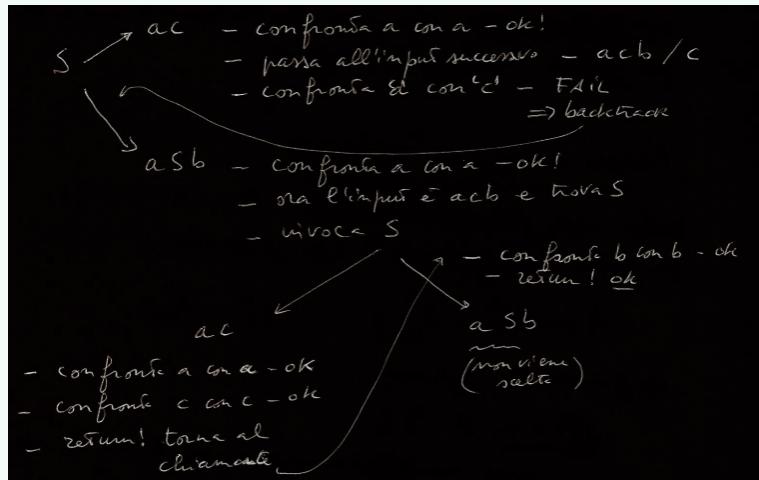
Sia G la grammatica:

$$S \rightarrow ac|aSb$$

Col linguaggio:

$$L = \{a^{n+1}cb^n | n \geq 0\}$$

e sia $aacb$ un input. Si ha



input Stack delle chiamate

<u>a</u> a c b	S
<u>a</u> c b	<u>a</u> c c fail
<u>a</u> a c b	S
<u>a</u> c b	<u>a</u> S b S b
<u>c</u> b	<u>a</u> c b <u>c</u> b
<u>b</u>	b ok

Tuttavia il parser a discesa ricorsiva è **parecchio inefficiente a causa della sua natura nondeterminista**, vi è infatti la necessità nel peggiore dei casi di esplorare tutte le alternative

Theorem 7.1.1

Sia w la lunghezza della stringa in inout, e sia b il massimo numero di produzioni per uno stesso nonterminale, allora la complessità computazionale di un parser a discesa riscorsiva nel caso peggiore è:

$$O(b^{|w|})$$

Per ovviare ovviare a questo problema di infecenza dobbiamo guidare la scelta della produzione per creare un parser top-down deterministico. Per farlo occorrono delle fuzioni ausiliarie

7.2 Parser predittivo

Il **parser predittivo** è un tipo parser deterministico (sotto alcune specifiche condizione che si vedranno più avanti), molto più efficiente in quanto non ha il backtracking, tuttavia per definirlo occorre prima definire delle funzioni ausiliarie

7.2.1 First

Definition 7.2.1: First

Data una grammatica libera G e $\alpha \in (T \cup NT)^*$, si definisce $\text{First}(\alpha)$ come l'insieme dei terminali che possono stare in prima posizione in una stringa che si deriva da α

- per $a \in T, a \in \text{First}(\alpha) \iff \alpha \Rightarrow^* a\beta$ per $\beta \in (T \cup NT)^*$
- inoltre $(\alpha \Rightarrow^* \epsilon) \Rightarrow \epsilon \in \text{First}(\alpha)$

Note:

Sia la grammatica

$$A \rightarrow \alpha_1 | \alpha_2$$

Se $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) = \emptyset$ la scelta della produzione è deterministica

Qui vi è riportato un esempio:

Example 7.2.1

$$A \rightarrow aB | bC$$

Si ha che:

$$\begin{aligned}\text{First}(aB) &= \{a\} \\ \text{First}(bC) &= \{b\}\end{aligned}$$

Pertanto abbiamo del determinismo con un solo carattere in lettura

algoritmo per calcolare il first

Algorithm 7: First()

```

Input: Una grammatica credo
Output: bho
1 for  $x \in T$  do
2    $\text{First}(x) \leftarrow \{x\};$                                 // un terminale è il primo elemento di se stesso
3 for  $X \in NT$  do
4    $\text{First}(X) \leftarrow \emptyset;$                             // per ogni x non terminale si inizializza il suo first a "0"
5 while almeno un  $\text{First}(X)$  può essere modificato in una iterazione do
6   foreach  $x \rightarrow Y_1, \dots, Y_k$  do
7     foreach  $i = 1$  to  $k$  do
8       // se ciascuno di questi simboli  $y_1, \dots, y_{i-1}$  può derivare la stringa vuota  $\epsilon$ 
9       if  $Y_1, \dots, Y_{i-1} \in N(G)$  then
10         $\text{First}(X) \leftarrow \text{First}(X) \cup (\text{First}(Y_i) \setminus \{\epsilon\});$       // allora è possibile aggiungere gli elementi di
11         $\text{FIRST}(Y_i)$  a  $\text{FIRST}(X)$  per la produzione  $y_1, \dots, y_k$ 
12        // Se invece uno dei simboli da  $Y_1$  a  $Y_{i-1}$  non è annullabile, si interrompe la ricerca per quella
           produzione, perché non possiamo "saltare" i simboli non annullabili per arrivare a  $Y_i$ 
13   foreach  $X \in N(G)$  do
14      $\text{First}(X) = \text{First}(X) \cup \{\epsilon\};$ 

```

In generale per una stringa α si ha che:

- Se $\alpha = \epsilon$, allora $\text{FIRST}(\alpha) = \{\epsilon\}$.
- Se $\alpha = X\beta$ e $X \notin N(G)$, allora $\text{FIRST}(X\beta) = \text{FIRST}(X)$.
- Se $\alpha = X\beta$ e $X \in N(G)$, allora $\text{FIRST}(X\beta) = (\text{FIRST}(X) \setminus \{\epsilon\}) \cup \text{FIRST}(\beta)$

In pratica se

$$A \rightarrow \alpha_1 | \dots | \alpha_k$$

si ha che

$$\text{First}(A) = \text{First}(\alpha_1) \cup \dots \cup \text{First}(\alpha_k)$$

Example 7.2.2

Si osservi la seguente grammatica:

$$\begin{aligned} S &\rightarrow Ab|c \\ A &\rightarrow aA|\epsilon \end{aligned}$$

$$\begin{aligned} \text{FIRST}(S) &= \text{FIRST}(Ab) \cup \text{FIRST}(c) \\ &= (\text{FIRST}(A) \setminus \{\epsilon\}) \cup \text{FIRST}(b) \cup \{\epsilon\} \\ &= \{a\} \cup \{b\} \cup \{c\} = \{a, b, c\} \end{aligned}$$

$$\begin{aligned} \text{FIRST}(A) &= \text{FIRST}(aA) \cup \text{FIRST}(\epsilon) \\ &= \{a\} \cup \{\epsilon\} = \{a, \epsilon\} \end{aligned}$$

7.2.2 Follow

Definition 7.2.2: Follow

Data una grammatica libera G e $A \in NT$, definiamo che $\text{Follow}(A)$ è l'insieme dei terminali che possono comparire immediatamente a destra di A in una forma sentenziale.

- Per ogni $a \in T$, $a \in \text{Follow}(A)$ se $S \Rightarrow^* \alpha A a \beta$ per qualche α e $\beta \in (T \cup NT)^*$.
- $\$ \in \text{Follow}(A)$ se $S \Rightarrow^* \alpha A$ (Poiché $S \Rightarrow^* S$, allora $\$ \in \text{Follow}(S)$!)

Riporto qui un esempio

Example 7.2.3

$$\begin{aligned} S &\rightarrow Ab \mid c \\ A &\rightarrow aA \mid \epsilon \end{aligned}$$

$$\text{Follow}(S) = \{\$\} \quad \text{Follow}(A) = \{b\}$$

- $S \Rightarrow^* S$
- $S \Rightarrow^* Ab$

algoritmo per calcolare il Follow

Algorithm 8: Follow()

Input: Una grammatica credo

Output: bho

1 foreach $X \in NT$ do

2 | $First(X) \leftarrow \emptyset;$

// per ogni X non terminale si inizializza il suo first a "0"

3 $Follow(S) \leftarrow \{\$\}$;

4 while almeno un $\text{Follow}(X)$ può essere modificato in una iterazione do

5 | foreach $X \rightarrow \alpha Y \beta$ do

6 | $Follow(Y) \leftarrow Follow(Y) \cup (First(\beta) \setminus \{\epsilon\});$

7 | foreach $X \Rightarrow \alpha Y$ do

| foreach $X \rightarrow \alpha\beta$, $\epsilon \in First(\beta)$ do

Follow(Y) \leftarrow Follow(Y) \cup Follow(X);

in pratica, occorre cercare tutte le produzioni in cui $Y \in NT$ appare e, per ognuna di esse, applicare la 1 o la 2 sopra

Example 7.2.4

	F_{1st}	$Follow$
E	a, b, (\$,)
E'	$\epsilon, +, -$	\$,)
T	a, b, (\$,), +, -
T'	$\epsilon, *$	\$,), +, -
A	a, b, (\$,), +, -, *

$$\text{Follow}(E) = \{ \} \quad \text{perché } E \text{ è il simbolo iniziale}$$

? $E' \rightarrow +E \mid -E$ richiedono che
 $\text{Follow}(E') \subseteq \text{Follow}(E)$

$$\text{Follow}(E') = \{ \text{prod} \mid E \rightarrow^* T E' \}, \text{ deve essere } \quad \Rightarrow \quad \begin{aligned} \text{Follow}(E) \\ = \text{Follow}(E') \end{aligned}$$

$$Follow(T) = \{ E \mid T E^* \Rightarrow \text{a word in } First(E^*) \} \\ \subseteq \{ E \mid T E^* \Rightarrow \text{a word in } First(E^*) \text{ and } E \in Follow(E) \}$$

$$T^t \rightarrow *T \Rightarrow \text{Follow}(T^t) \subset \text{Follow}(T)$$

$$Follow(T') = T \rightarrow AT' \Rightarrow Follow(T) \subseteq Follow(T') \Rightarrow Follow(T) = Follow(T')$$

$$\text{Follow}(A) = \text{--}^T \rightarrow A T^1 \Rightarrow \begin{aligned} & \text{include } \text{First}(T^1) \setminus \{\epsilon\} \\ & + \text{ include } \epsilon \in \text{First}(T^1) \Rightarrow \text{include } \text{Follow}(T) \end{aligned}$$

Adesso che abbiamo introdotto i le procedure First e Follow occorre fare un passo in più per definire i parser

7.2.3 Parser per linguaggi $LL(1)$

tabella di parsing $LL(1)$

La tabella di parsing $LL(1)$ è una struttura di dati usata nei parser sintattici molto utili per risolvere il non determinismo. Questi parser leggono l'input da sinistra a destra (da qui il primo "L" di "LL"), costruendo una derivazione sinistra, o leftmost (da qui il secondo "L") e usano un solo simbolo di lookahead (da cui il "(1)"). Questa tabella è formata da una **matrice bidimensionale** M che è formata da:

- **righe**: non-terminali
- **colonne**: terminali (incluso \$)
- **casella** (A, a) : $M[A, a]$ contiene le produzioni che possono essere scelte dal parser mentre tenta di espandere A e l'input corrente è a .

Se ogni casella contiene al più una produzione, allora il parser è **deterministico!**

Per riempire la tabella occorre procedere in questo modo:

Per ogni produzione $A \rightarrow \alpha$:

1. per ogni $a \in T$ e $a \in \text{First}(\alpha)$, inserisci $A \rightarrow \alpha$ nella casella $M[A, a]$
2. se $\epsilon \in \text{First}(\alpha)$, inserisci $A \rightarrow \alpha$ in tutte le caselle $M[A, x]$ per $x \in \text{Follow}(A)$ (x può essere \$)

Ogni casella vuota, dopo aver elaborato tutte le produzioni, è un errore (cioè la funzione ricorsiva chiama 'fail')

grammatica $LL(1)$

Definition 7.2.3: grammatica $LL(1)$

Una grammatica si definisce $LL(1)$ sse ogni casella della tabella di parsing $LL(1)$ contiene al più una produzione, ovvero non presenta conflitti

Si ha che se $G = LL(1)$ allora il parser è predittivo e deterministico, questo perché il parser ricostruisce l'albero di derivazione per l'input w , in modo top-down, predicendo quale produzione usare (tra le molte possibili) guardando il prossimo carattere dell'input

Theorem 7.2.1

G è $LL(1)$ sse per ogni coppia di produzioni distinte con la stessa testa

$$A \rightarrow \alpha | \beta$$

si ha che

1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
2. (a) $(\epsilon \in \text{First}(\alpha)) \implies (\text{First}(\beta) \cap \text{Follow}(A) = \emptyset)$
(b) $(\epsilon \in \text{First}(\beta)) \implies (\text{First}(\alpha) \cap \text{Follow}(A) = \emptyset)$

Dimostrazione: Se sono soddisfatte le condizione 1 e 2 per ogni coppia di produzioni distinte con medesima testa allora la tabella di parsing $LL(1)$ contiene al più una produzione in ogni cassella. Ma vale anche viceversa! \mathcal{Q}

Linguaggio $LL(1)$

Definition 7.2.4: Linguaggio $LL(1)$

Un linguaggio si definisce $LL(1) \iff \exists G'$ grammatica = $LL(1)$ che lo genera

Example 7.2.5

Sia G la seguente grammatica:

$$\begin{aligned} S &\rightarrow A|B \\ A &\rightarrow ab|cd \\ B &\rightarrow ad|cb \end{aligned}$$

Si può notare che G non è $LL(1)$ dato che $S \rightarrow A|B$ e

$$\begin{aligned} First(A) &= \{a, c\} \\ First(B) &= \{a, c\} \\ First(A) \cap First(B) &= \{a, c\} \end{aligned}$$

Dal teorema sopra fornito si può dimostrare che non è $LL(1)$

Tuttavia si può manipolarla per farla diventare $LL(1)$, quindi espando S :

$$\begin{aligned} S &\rightarrow ab|cd|ad|cb \\ S &\rightarrow aT|cT' \\ T &\rightarrow b|d \\ T' &\rightarrow b|d \end{aligned}$$

Poi osservo che T e T' sono identici, sia quindi G' la nuova grammatica:

$$\begin{aligned} S &\rightarrow aT|cT \\ T &\rightarrow b|d \end{aligned}$$

Si può dimostrare che è $LL(1)$, pertanto, per la definizione di linguaggio $LL(1)$ e nonostante G non sia $LL(1)$, si ha che $L(G) = \{ab, cd, ad, cb\}$ è un linguaggio $LL(1)$ perché G' che lo genera è una grammatica $LL(1)$

Theorem 7.2.2

Ogni linguaggio regolare è generabile da una grammatica G di classe $LL(1)$

Dimostrazione: Sia L un linguaggio regolare, allora \exists DFA $M = (Q, \Sigma, \delta, q_0, F) : L = [M]$.

A partire da M si può costruire una grammatica regolare $G = (NT, T, S, R)$ basata sul seguente automa M :

- $NT = \{[q] | q \in Q\}$, cioè un non terminale per ogni stato q
- $T = \Sigma$ cioè un terminale per ogni simbolo dell'alfabeto
- $S = [q_0]$ simbolo iniziale lo stato iniziale
- R (insieme delle produzioni) è definito come:

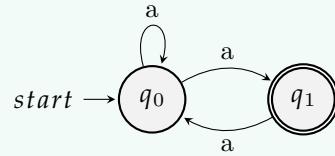
– se $\delta(q, a) = q'$, allora $[q] \rightarrow a[q'] \in R$

Infatti $\delta(q, a) = q'$ vuol dire che l'automa si trova allo stato q e legge in input a allora arriverà allo stato q' , che viene "tradotto" nella grammatica $[q] \rightarrow a[q']$ che corrisponde ad una produzione in cui il non terminale $[q]$ produce il terminale a e il non terminale $[q']$, per passare al non terminale $[q']$ occorre, infatti, fare match con a

– se $q \in F$, allora $[q] \rightarrow \epsilon \in R$

Poi che M è deterministico, $\forall q \in Q \forall a \in \Sigma \quad \exists! q'. q \xrightarrow{a} q'$, cioè $[q]$ avrà una sola produzione $[q] \rightarrow a[q']$ che "inizializza" per a e dato che se q è finale, allora $[q] \rightarrow \epsilon$ è applicabile solo per i $\text{Follow}([q]) = \{\$\}$ \implies nessun conflitto, dato che nessuna produzione genera $\$$, si ha che G è $LL(1)$ □

Example 7.2.6



Le produzioni corrispondenti sono:

$$\begin{array}{l} [q_0] \rightarrow a[q_1] \mid a[q_0] \\ [q_1] \rightarrow a[q_0] \mid \epsilon \end{array}$$

La grammatica G è $LL(1)$, perché:

$$\text{First}(a[q_0]) \cap \text{First}(\epsilon) = \emptyset, \quad \text{First}(a[q_1]) \cap \text{First}(\epsilon) = \emptyset$$

Riporto qui un esempio di tabella di parsing $LL(1)$:

Example 7.2.7

Sia G la seguente grammatica:

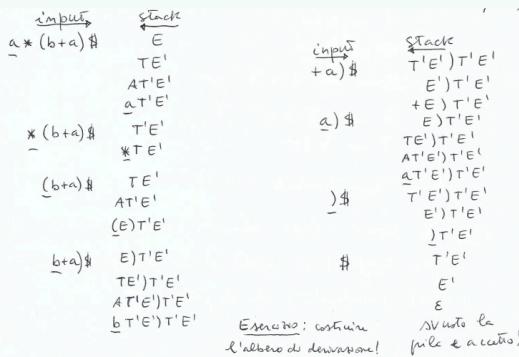
$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow \varepsilon \mid +TE' \mid -TE' \\
 T &\rightarrow AT' \\
 T' &\rightarrow \varepsilon \mid *T \\
 A &\rightarrow a \mid b \mid (E)
 \end{aligned}$$

Si può costruire la seguente tabella di First e Follow:

Produzione	First	Follow
E	$a, b, ($	$\$,)$
E'	$+, -, \varepsilon$	$\$,)$
T	$a, b, ($	$+, -, \$,)$
T'	$*, \varepsilon$	$+, -, \$,)$
A	$a, b, ($	$*, +, -, \$,)$

Ed ecco a voi la tabella di parsing:

	a	b	(+	*	\$
E	$E \rightarrow TE'$	$E \rightarrow TE'$	$E \rightarrow TE'$			
E'				$E' \rightarrow +TE'$	$E' \rightarrow -TE'$	$E' \rightarrow \varepsilon$
T	$T \rightarrow AT'$	$T \rightarrow AT'$	$T \rightarrow AT'$			
T'				$T' \rightarrow \varepsilon$	$T' \rightarrow *T$	$T' \rightarrow \varepsilon$
A	$A \rightarrow a$	$A \rightarrow b$	$A \rightarrow (E)$			



Algoritmo Per il calcolo di un parser $LL(1)$

Algorithm 9: Parser $LL(1)$

```

Input: Stringa  $w$ 
Output: Niente
1  $Pila \leftarrow S\$;$                                 // cima della pila a sinistra
2  $X \leftarrow S ;$                                   // top della pila
   // lettura in input
3  $input \leftarrow w\$;$ 
4  $i_c =$  primo carattere dell'input;
   // viene eseguito il ciclo While finché la pila non è vuota o l'input non è stato consumato
5 while  $X \neq \$$  do
6   if  $X$  è un terminale then
7     // si controlla se  $X$  fa "match" con  $i_c$ 
8     if  $X = i_c$  then
9       Pop  $X$  dalla pila;                          // rimuovo  $X$  dalla pila
10      avanza  $i_c$  sull'input;
11    else
12      Errore();                                 // no match
13    else
14      // se nella tabella di Parsing  $M$  esiste una regola  $X \rightarrow Y_1, \dots, Y_n$ 
15      if  $M[X, i_c] = X \rightarrow Y_1, \dots, Y_n$  then
16        Pop  $X$  dalla pila;
17        Push  $Y_1, \dots, Y_n$  sulla pila;           // mette sulla pila i simboli  $Y_1, \dots, Y_n$  con  $Y_1$  in cima
18        In output la produzione  $X \rightarrow Y_1, \dots, Y_n$ ;
19      else
20        Errore();                               // se non esiste una regola in  $M[X, i_c]$  si genera un errore, viene definito caso
21        "bianco"
22       $X \leftarrow$  top della pila;                // Aggiorna  $X$  con il nuovo simbolo in cima alla pila
23    if  $i_c \neq \$$  then
24      Errore();                                // pila svuotata ma vi è ancora dell'input da leggere

```

Un altro esempio:

Example 7.2.8

Sia G la seguente grammatica:

$$\begin{aligned} S &\rightarrow aAB \mid bS \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

Che genera il seguente linguaggio

$$L(G) = L(b^*aab)$$

Si ha che questa grammatica G è $LL(1)$, perché:

$$First(aAB) \cap First(bS) = \emptyset, \quad \{a\} \cap \{b\} = \emptyset$$

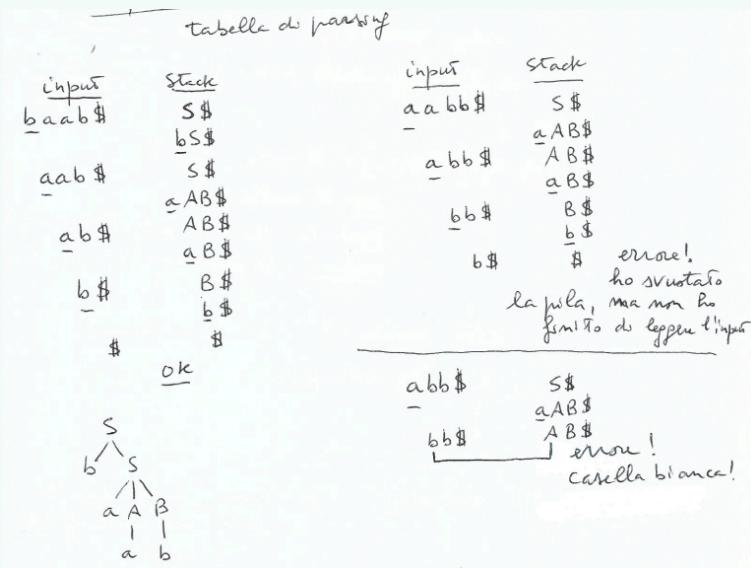
Si può proseguire con la tabella First e follow:

Simbolo	First	Follow
S	a, b	$\$$
A	a	b
B	b	$\$$

Da cui si può costruire la seguente tabella di parsing:

	a	b	\$
S	$S \rightarrow aAB$	$S \rightarrow bS$	
A	$A \rightarrow a$		
B		$B \rightarrow b$	

Viene qui descritto il funzionamento del parser con pila con diversi input:



7.2.4 Parser per linguaggi $LL(K)$

grammatiche $LL(K)$

Le grammatiche $LL(k)$ sono "un'estensione" del concetto di grammatiche $LL(1)$, dove il parser ha la capacità di guardare in avanti fino a k simboli per determinare le scelte di parsing

Per questi tipi di grammatica gli insiemi *First* e *Follow* assumo significati diversi rispetto a alle grammatiche $LL(K)$

Definition 7.2.5: First $LL(K)$

L'insieme $First_k(\alpha)$ contiene tutte le stringhe di lunghezza k o minore derivabili dall'inizio di una produzione con α , in particolare $w \in First_k(\alpha) \iff \alpha \Rightarrow^* w\beta$ con $|w| = k, w \in T^*, \beta \in (T \cup NT)^*$ oppure $\alpha \Rightarrow^* w$ con $|w| \leq k$ e $w \in T^*$

Definition 7.2.6: Follow $LL(K)$

L'insieme $Follow_k(A)$ definisce quali stringhe possono apparire immediatamente dopo un simbolo non terminale A in una derivazione a partire dal simbolo iniziale S . In particolare: $w \in Follow_k(A)$ se $S \Rightarrow^* \alpha Aw\beta$ con $|w| = k, w \in T^*$, e $\alpha, \beta \in (T \cup NT)^*$, oppure, $S \Rightarrow^* \alpha Aw$ con $|w| \leq k$ e $w \in T^*$.

Tabella di Parsing $LL(K)$

- Righe:** non terminali.
- Colonne:** $\{w \in T^* \mid |w| \leq k\}$ (solo quelle necessarie).

Per ogni produzione $A \rightarrow \alpha$, la tabella $M[A, w]$ contiene:

- $A \rightarrow \alpha$, per ogni $w \in \text{First}_k(\alpha)$ ($w \neq \varepsilon$);
- $w \in \text{Follow}_k(A)$ se $\varepsilon \in \text{First}_k(\alpha)$.

Ogni entrata/casella contiene al più una produzione. Se non esistono w_1 e w_2 tali che w_1 prefisso di w_2 con le due entrate corrispondenti su una riga entrambe riempite, allora G è una grammatica LL(k).

Nota Bene: Le colonne sono tante quante sono le stringhe w che appartengono a $\text{First}_k(\alpha)$ per $A \rightarrow \alpha$ o a $\text{Follow}_k(A)$ per $A \rightarrow \alpha$. Questo va verificato per tutte le produzioni:

$$w \in \text{First}_k(\alpha).$$

Example 7.2.9

Sia G la seguente grammatica

$$S \rightarrow aSb \mid ab \mid c$$

Si ha che $L(G) = \{a^n b^n \mid n \geq 1\} \cup \{a^n c b^n \mid n \geq 0\}$, inoltre:

- $\text{First}_2(aSb) = \{aa, ac\}$
- $\text{First}_2(ab) = \{ab\}$
- $\text{First}_2(c) = \{c\}$

Si può dimostrare che G è LL(2) dato che:

- $\text{First}_2(aSb) \cap \text{First}_2(ab) = \emptyset$
- $\text{First}_2(aSb) \cap \text{First}_2(ab) = \emptyset$
- $\text{First}_2(aSb) \cap \text{First}_2(ab) = \emptyset$

Da cui si può ricavare la tabella di parsing:

Simbolo	aa	ab	ac	c
S	$S \rightarrow aSb$	$S \rightarrow ab$	$S \rightarrow aSb$	$S \rightarrow c$

Theorem 7.2.3

- Una grammatica ricorsiva sinistra non è LL(K) per nessun K
- Una grammatica ambigua non è LL(K)
- Se G è LL(K) per qualche k , allora G non è ambigua
- Se G è LL(K), allora $L(G)$ è libero deterministico
- esiste L libero deterministico tale che non esiste G di classe LL(K) per nessun K , tale che $L = L(G)$

Viene qui riportato il funzionamento del parser con pila:

Linguaggio LL(K)

Definition 7.2.7: Linguaggio LL(K)

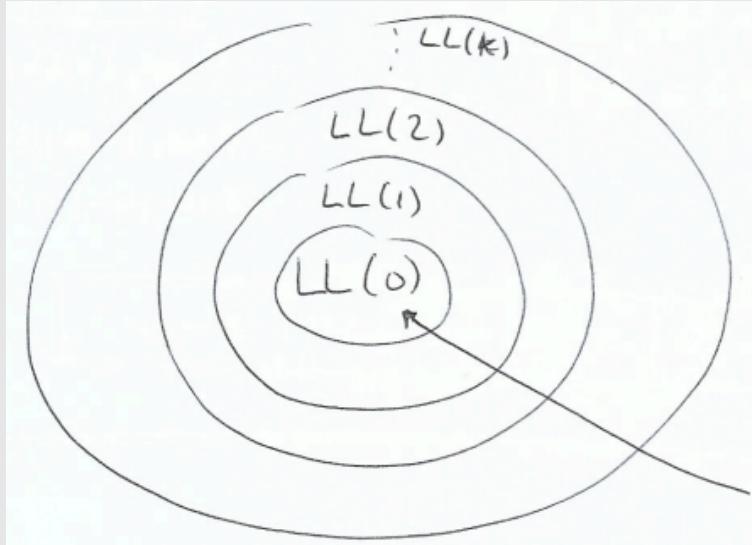
un linguaggio L è di classe LL(K) se G di classe LL(K) tale che $L = L(G)$

Theorem 7.2.4

$\forall k \geq 0$, la classe dei linguaggi $LL(K)$ contiene strettamente la classe dei linguaggi $LL(K)$

Note:

$\forall k \geq 0$, la classe dei linguaggi $LL(K+1)$ contiene strettamente la classe dei linguaggi $LL(K)$



Si ha che: $(\forall A \in NT, \exists! \alpha \in (T \cup NT)^*, A \rightarrow \alpha \implies G \text{ è } LL(0)) \implies L(G) = \{w\}$ ovvero una sola parola al massimo

Nella pratica tuttavia si usano solo $LL(1)$, spesso la si può manipolare trasformandola in $LL(1)$

Example 7.2.10

$$S \rightarrow Asb \mid ab \mid c$$

G è un $LL(2)$ ma non $LL(1)$. Si fattorizza

$$\begin{aligned} S &\rightarrow aT \mid c \\ T &\rightarrow Sb \mid b \end{aligned}$$

Ottenuta fattorizzando G è $LL(1)$ infatti:

- $First(aT) \cap First(c) = \emptyset$
- $First(Sb) \cap First(b) = \emptyset$

Si ha che $L = \{a^n b^n \mid n \geq 1\} \cup \{a^n c b^n \mid n \geq 0\}$ è un linguaggio di classe $LL(1)$ perché G' è $LL(1)$

Casi speciali

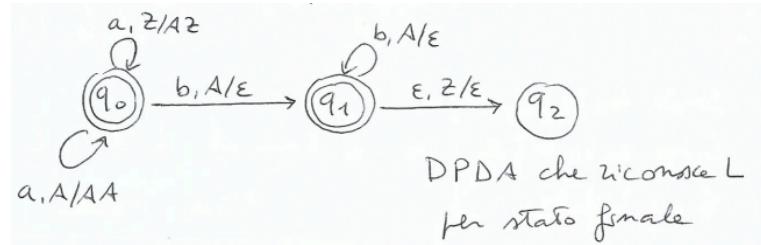
Sia

$$L = \{a^i b^j \mid i \geq j\}$$

Ma non è $LL(K)$ per nessun K !

Adesso, mostriamo una grammatica per L e dimostriamo che non è $LL(K)$ per nessun K . Sia G la seguente grammatica:

$$\begin{aligned} S &\rightarrow aS \mid B \\ B &\rightarrow aBb \mid \epsilon \end{aligned}$$



Sia G una grammatica libera per L

$$S \rightarrow aS \mid BB \rightarrow aBb \mid \epsilon$$

e poniamo $L = L(G)$

Per scegliere tra $S \rightarrow aS$ e $S \rightarrow B$ dovrei leggere fino in fondo l'input per sapere quante b in meno di a ci sono nella stringa! Allora G non può essere $LL(k)$ per nessun k . Infatti quanto possa essere grande il K posso trovare una stringa più lunga che richiede di leggere più di k simboli di lookahead

Non è tuttavia possibile alcuna G' e k tali che

$$L(G') = L \text{ e } G' \in L(K)$$

La dimostrazione non verrà illustrata

Chapter 8

Bottom up parser

Il **parser bottom up** è un tipo di analizzatore sintattico che costruisce l'albero di derivazione partendo dalle foglie, viene anche detto **Shift-Reduce** in quanto possiede due operazioni fondamentali:

- **Shift:** un simbolo terminale viene spostato dall'input alla pila
- **Reduce:** una serie di simboli (terminali e non terminali) sulla cima della pila corrisponde al "reverse" di una parte destra di una produzione, ovvero:

$$A \rightarrow \alpha \in R - \alpha^R \text{ sulla pila}$$

La stringa α^T viene rimossa dalla pila e sostituita con A , quindi α viene ridotta ad A

Possono essere di diversi tipi

8.1 Parser shift-reduce nondeterministico

Algorithm 10: Parser shift-reduce nondeterministico

Input: Una grammatica libera G con simbolo iniziale S e una stringa $w \in T^*$

Output: Una stringa $w \in T^*$

- 1 Inizializziamo la pila a $\$$;
- 2 Inizializziamo l'input con $w\$$;
- 3 usiamo il PDA seguente per trovare la derivazione per $w\$$:

$$M = (T, \{q\}, \underbrace{T \cup NT \cup \{\$\}}_{\Gamma: \text{ovvero i simboli sulla pila}}, \delta, \emptyset)$$

Dove:

1. $(q, aX) \in \delta(q, a, X) \forall A \in T \forall X \Gamma \quad \text{SHIFT}$
2. $(q, A) \in \delta(q, \epsilon, \alpha^R) \text{ se } A \in \alpha \in R \quad \text{REDUCE}$
3. $(q, \epsilon) \in \delta(q, \$, S\$) \quad \text{accept}$

;

ogni volta che facciamo "reduce", forniamo in output la produzione usata;
alla fine, $S\$$ sulla pila, con $\$$ in input \Rightarrow accettiamo;

Note:

generalizzazione della def. di PDA dove non si consuma solo il top della pila, ma una serie di caratteri contigui a cominciare dal top

Example 8.1.1

Sia la seguente grammatica

$$E \rightarrow T \mid T + E \mid T - E$$

$$T \rightarrow A \mid A * T$$

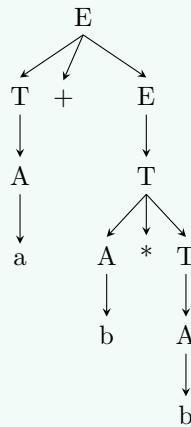
$$A \rightarrow a \mid b \mid (E)$$

N.	Pila	Input	Azione	Output
1	\$	$a + b * b \$$	Shift	
2	$\$a$	$+b * b \$$	Reduce	$A \rightarrow a$
3	$\$A$	$+b * b \$$	Reduce	$T \rightarrow A$
4	$\$T$	$+b * b \$$	Shift	
5	$\$T +$	$b * b \$$	Shift	
6	$\$T + b$	$*b \$$	Reduce	$A \rightarrow b$
7	$\$T + A$	$*b \$$	Shift	
8	$\$T + A *$	$b \$$	Shift	
9	$\$T + A * b$	$\$$	Reduce	$A \rightarrow b$
10	$\$T + A * B$	$\$$	Reduce	$T \rightarrow A * T$
11	$\$T + A * T$	$\$$	Reduce	$E \rightarrow T + T$
12	$\$E$	$\$$	Reduce	$E \rightarrow T + T$
13	$\$E$	$\$$	Accept	

Si ha che:

$$E \implies T + E \implies T + T \implies T + A * T \implies T + A * A \implies T + A * b \implies T + b * b \implies A + b * b \implies a + b * b$$

Il cui albero è:



Le cui proprietà sono:

Costruzione dell'albero di derivazione *bottom-up*

Derivazione canonica a destra a rovescio

forte non-determinismo:

- **confitti shift-reduce:**

- $\$a + b * b \$$ shift
- $\$a + b * b \$$

- **Conflitti reduce-reduce**

- $\$T + A * T \$$ reduce $E \rightarrow T$
- $\$T + A * E \$$

Note:

Si noti quindi che questo tipo parser genera moltissimi conflitti, in diverse situazioni era appunto possibile scegliere strade che non portano a riconoscere la stringa. F

Per "risolvere" tale nondeterminismo occorre fornire al PDA una tabella parsing (struttura di controllo) che "aiuti" a scegliere l'azione giusta, ed è grazie all'uso di questa che nascono i cosiddetti **parser LR**

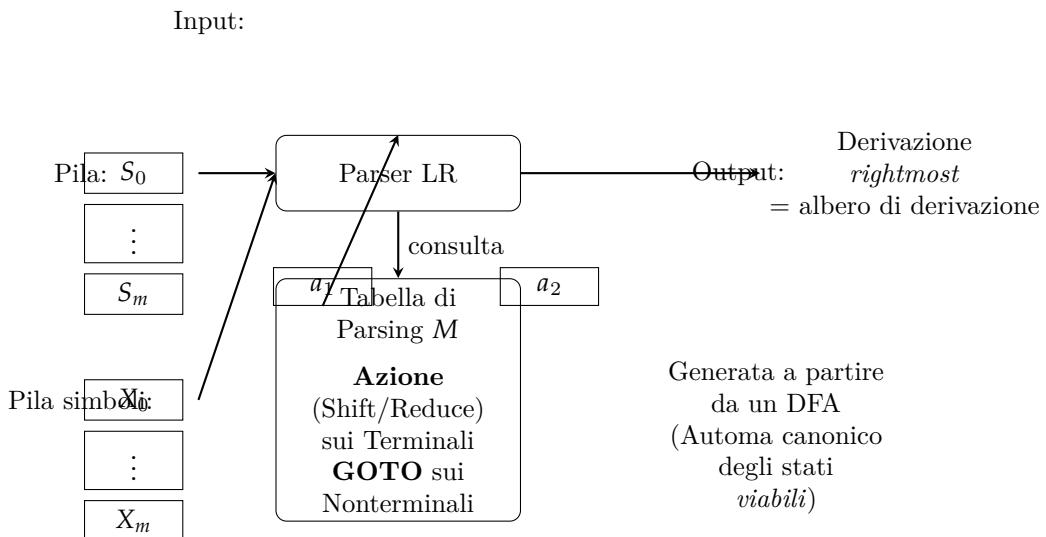
8.2 Parser LR

Definition 8.2.1: Parser LR

Il **parser LR** è un tipo di analizzatore sintattico che utilizza un approccio bottom-up per analizzare un input e verificare se appartiene al linguaggio generato da una grammatica libera. Il termine LR indica che:

- L: sta per *left-to-right*: l'analisi dell'input avviene da sinistra a destra
- R: sta per *Rightmost derivation*: l'analisi ricostruisce una derivazione più a destra in senso inverso

Presento, qui uno schema bellino bellino:



dove una configurazione di tale parser è:

$$(s_0, \dots, s_n, x_1 \dots x_m, a_1 \dots a_k \$)$$

Note:

$x_1 \dots x_m a_1 \dots a_k$ è una stringa intermedia della derivazione canonica destra

8.2.1 Funzionamento del parsing LR

il parser LR funziona nel seguente modo:

1. Legge lo stato nel top della pila s_n e il simbolo corrente dell'input a_i
2. consulta la tabella di parsing LR $M[s_n, a_i]$
 - Se $M[s_n, a_i] = \text{shift } s$, allora la nuova configurazione è:

$$(s_0 \dots s_n \underline{s}, x_1 \dots x_n \underline{a_i}, a_{i+1} \dots a_k \$)$$

- Se $M[s_n, a_i] = \text{reduce } A \rightarrow B$, allora la nuova configurazione è

$$(s_0 \dots s_{n-r} \underline{s}, x_1 \dots x_{m-r} A, a_i \dots a_k \$)$$

Dove $r = |\beta|$ e $M[s_{n-r}, A] = \text{goto } \underline{s}$

Cioè fa tre passi:

- faccio "pop" di r elementi dai 2 stack
- metto A in cima alla pila dei simboli
- calcolo il nuovo stato top, guardando

$M[s_{n-r}, A] = \text{goto } s$ e metto s in cima alla pila degli stati

- se $M[s_n, a_i] = \text{accept} \implies$ fine!
- se $M[s_n, a_i] = \text{"bianco"} \implies$ errore!

Example 8.2.1

Sia G la seguente grammatica:

$$\begin{aligned} S' &\rightarrow S \\ \text{Senza} &\rightarrow (S) \\ S &\rightarrow () \end{aligned}$$

Assumendo che siamo già forniti di questa **Tabella di Parsing**

Stato	Azione		Goto S
	()	\$	
0	$S2$		$g1$
1		Accept	$g3$
2	$S2$	$S5$	
3		$S4$	
4	$r2$	$r2$	$r2$
5	$r3$	$r3$	$r3$

Esecuzione del Parsing

Stato della pila	Stack simboli	Input	Azione	Output
(0	ϵ	(())\$	$S2$	-
(0, 2	((())\$	$S2$	-
(0, 2, 2	(()\$	$S5$	-
(0, 2, 2, 5	(())\$	$r3$	$S \rightarrow ()$
(0, 2, 3	(S)\$	$S4$	-
(0, 2, 3, 4	(S)	\$	$r2$	$S \rightarrow (S)$
(0, 1	S	\$	Accept	-

Tuttavia una domanda lecita è come si fa trovare questa tabella di parsing, bhe prima di presentarla occorre definire un automa dei prefissi variabili

8.2.2 DFA a prefissi variabili

Definition 8.2.2: prefisso variabile

un prefisso variabile è una stringa $\in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom-up per una computazione che accetta un input

Definition 8.2.3: prefisso viabile su una grammatica G libera

si definisce prefisso viabile su una grammatica G libera una stringa $\gamma \in (T \cup NT)^*$ sse esiste una derivazione rightmost

$$S \implies {}^*\delta A y \implies \delta \alpha \beta y = \gamma \beta y$$

Dove:

- $y \in T^*$
- $\delta \in (T \cup NT)^*$
- esiste una produzione $A \rightarrow \alpha\beta$

Inoltre S è un prefisso variabile per definizione

Definition 8.2.4: prefisso variabile completo e maniglia

un prefisso viabile si dice completo se $\beta = \epsilon$, in tal caso α è detta maniglia per γy

Dopo sta sbordolata di definizioni si possa al teorema che lega i prefissi variabili con i DFA:

Theorem 8.2.1

Data G libera, i prefissi viabili di G costituiscono un linguaggio regolare e può essere descritto con un DFA. detto **DFA a prefissi viabili o automa canonico LR(0)**

Pertanto si ha questo corollario

Corollary 8.1 |

parser può consultare il DFA dei prefissi viabili (ovvero la tabella di parsing) per decidere cosa fare:

- se la pila contiene un prefisso viabile completo il parser riduce
- se la pila contiene un prefisso viabile incompleto, allora il parser shifta
- se la pila non contiene un prefisso viabile, allora viene dato un errore

In base a come è fatto questo DFA (che eventualmente può usare informazioni di look-ahead e dei follow) il parser può risultare deterministico o meno

Si noti, inoltre la seguente osservazione

Note:

si può dimostrare che un prefisso di un prefisso viabile è un prefisso viabile pertanto si ha che non è necessario ripartire da capo quando viene modificata la pila (ovvero il parser)

infatti la pila viene modificata in due modi:

1. **shift**: la pila passa da $\$y$ a $\$yx$. Dato che si trova in uno stato s dopo aver elaborato $\$y$, occorre soltanto far ripartire il DFA da S con input x
2. la pila passa da $\$y\alpha$ a $\$yA$. Dato che si trova in uno stato s dopo aver elaborato $\$y\alpha$, non c'è bisogno di far ripartire il DFA dalla base della pila, ma basta ripristinare lo stato in cui si trovava subito prima di elaborare il primo simbolo di α e fornirgli il simbolo A in input

Pertanto occorre lo stack degli stati del DFA!

Adesso introduciamo degli elementi fondamentali per l'automa canonico LR(0)

8.2.3 Item LR(0)

Definition 8.2.5: Item LR(0)

Un item $LR(0)$ è una produzione con indicata, con un punto, una posizione sulla sua parte destra

Example 8.2.2

La produzione $A \rightarrow XYZ$ genera 4 item diversi:

$$\begin{aligned} A &\rightarrow .XYZ \\ A &\rightarrow X.YZ \\ A &\rightarrow XY.Z \\ A &\rightarrow XYZ. \end{aligned}$$

In questo caso l'item $A \rightarrow .XYZ$ è un item iniziale

La posizione del $.$ indica no a dove abbiamo già analizzato di questa produzione, per questo abbiamo che:

- Se l'item $A \rightarrow \alpha.\beta$ è nello stato attuale del DFA vuol dire che α è in cima sulla pila dei simboli e che stiamo attendo aspettando β
- Se invece abbiamo $A \rightarrow \alpha$. Vuol dire che abbiamo letto tutta la produzione e possiamo fare una reduce

8.2.4 Costruire l'NFA dei prefissi variabili

Per costruire il DFA dei prefissi variabili occorre prima costruire il suo corrispettivo NFA.

Data $G = (NT, T, S, R)$ libera, l'NFA dei prefissi variabili di G si costruisce partendo con un nuovo simbolo iniziale S' ed una produzione $S' \rightarrow S$, inoltre;

- $[S' \rightarrow .S]$ è lo stato iniziale
- dallo stato $[A \rightarrow \alpha.X\beta]$ c'è una transizione dello stato $[A \rightarrow \alpha X.\beta]$ etichettata X , per $X \in T \cup NT$
- dallo stato $[A \rightarrow \alpha X.\beta]$, per $X \in NT$ e per ogni produzione $X \rightarrow \gamma$, c'è una ϵ -transizione verso lo stato $[X \rightarrow .\gamma]$

Note:

non serve definire degli stati finali, perché l'NFA come ausilio al parser

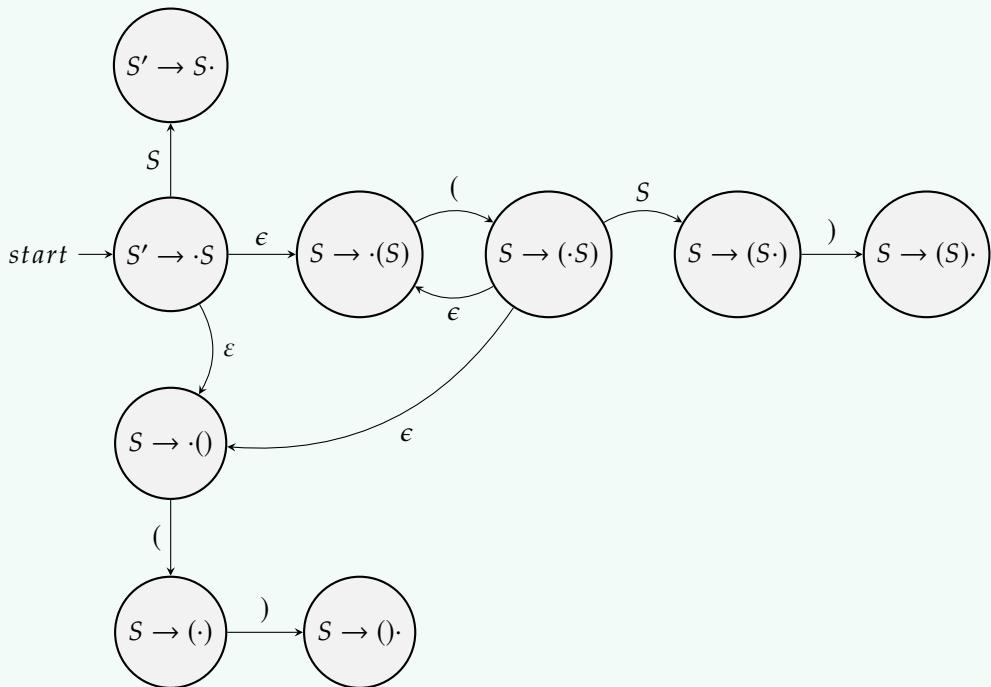
Example 8.2.3

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow (S) \\ S &\rightarrow () \end{aligned}$$

Tale grammatica mi diventa

$$\begin{aligned} S' &\rightarrow .S \mid S. \\ S &\rightarrow .(S) \mid (.S) \mid (S.) \mid (S). \\ S &\rightarrow .() \mid (.) \mid (). \end{aligned}$$

Ed il corrispettivo automa è:



8.3 Automa canonico $LR(0)$

l'automa canonico $LR(0)$ è il DFA DFA dei prefissi viabili. Per ottenerlo ci sono due metodi:

- Costruire prima lNFA e poi con la costruzione di sottoinsiemi oenere il DFA
- In un modo diretto usando due funzioni ausiliarie chiamate $clos(I)$ e $goto(I, X)$ dove I è un insieme di item e $X \in T \cup NT$

8.3.1 Costruzione diretta dell'automa canonico $LR(0)$

Andiamo prima a definire le due funzioni $clos(I)$ e $goto(I, X)$:

Clos()

Algorithm 11: Clos()

Input: I
Output: I

```

1 while  $I$  non è più modificato do
2   | foreach item  $A \rightarrow \alpha.X\beta \in I$  do
3     |   | foreach produzione  $X \rightarrow \gamma$  do
4       |   |   | aggiungi  $X \rightarrow \gamma$  ad  $I$ ;
5 return  $I$ ;

```

Si può notare come aggiungo ad I tutti gli item che sarebbero stati raggiungibili nel NFA con mosse $ε$

Goto()

algoritmo per il lacolo del DFA $LR(0)$

Date queste due funzioni adesso possiamo costruire lautoma DFA direttamente:

Algorithm 12: Goto

Input: Insieme di item I e $X \in (T \cup NT)$
Output: Insieme J di item completi

```
1  $J \leftarrow \emptyset;$ 
2 foreach item  $A \rightarrow \alpha.X\beta \in I$  do
3   aggiungi  $A \rightarrow \alpha X . \beta$  a  $J$ ;           // scorre tutti i punti, creando un nuovo nodo dove il punto si è mosso
4 return  $clos(j)$ ;                                // restituisce la closure del  $j$  appena creato
```

Algorithm 13: DFA LR(0)

```
1  $S \leftarrow \{clos(S' \rightarrow .S)\};$ 
2  $\delta \leftarrow \emptyset;$ 
3 while  $S$  o  $\delta$  non sono più modificati do
4   foreach  $I \in S$  do
5     foreach item  $A \rightarrow \alpha.X\beta \in I$  do
6        $J \leftarrow Goto(I, X);$ 
7       Aggiungi  $J$  a  $S$ ;
8       Aggiungi  $\delta(I, X) \leftarrow J$  a  $\delta$ ;
```

8.4 tabella di parsing LR

Definition 8.4.1: Tabella di parsing LR

Si definisce la tabella di parsign LR una matrice bidimensionale M tale che:

- le righe rappresentantano gli stati dell'automa canoninco LR(0)/LR(1)
- le colonne $T \cup \{\$\} \cup NT$

Inoltre si ha che:

- $M[s, X]$ contiene le azioni che può compiere un parser LR con S in cima alla pila degli stati e x simbolo in input (o nonterminale)
- Se $M[s, X]$ è "bianca" / vuota, allora ERRORE
- Se $M[s, X]$ contiene più automi, allora CONFLITTO

8.4.1 Riempire la tabella di parsing

La tabella di parsing si riempie in base all'automa canonico LR(0). Per ogni stato s va ripetuta la seguente cosa:

- se $x \in T$ e $S \rightarrow t$ nell'automa LR(0), inserisci shift in $M[s, x]$
- se $A \rightarrow \alpha \in s$ e $A \neq S'$, inserisci reduce $A \rightarrow \alpha$ in $M[s, x]$ per tutti gli $x \in T \cup \{\$\}$
- se $A \in NT$ e $S \rightarrow f$ nell'automa LR('), inserisci goto f

Definition 8.4.2: Grammatica di classe LR(0)

una grammatica viene definita di classe $LR(0)$ se ogni casella nella tabella di parsing $LR(0)$ contiene al più un elemento

8.5 Algoritmo del parser

Il parser data la tabella di parsing LR esegue questo algoritmo per calcolare l'albero di derivazione. questo algoritmo e per un generico parser LR (quindi funziona anche per $LR(k)$)

Algorithm 14: Parser LR

```

1 Inizializza la pila con  $\$s_0$ ;
2 inizializza  $i_c$  con il primo carattere in input;
3 while true do
4    $S \leftarrow Top(pila)$ ;
5   if  $M[S, i_c] == shift\ t$  then
6     push  $t$  sulla pila;
7     avanza  $i_c$  sull'input;
8   if  $M[S, i_c] == accept$  then
9     output("accept");
10  if  $M[S, i_c] == reduceA \rightarrow \alpha$  then
11    pop  $|\alpha|$  stati dalla pila;
12     $s_1 \leftarrow Top(pila)$ ;
13     $s_2 = M[s_1, A]$ ;
14    push  $s_2$  sulla pila ;
15    output( $A \rightarrow \alpha$ )
16  if  $M[S, i_c] == " "$  then
17    errore();

```

// s_2 è lo stato dato al goto

// casella vuota

8.6 SLR(1), LR(1), LALR(1)

Può accadere che data una grammatica non si riesce a creare un parser di tipo $LR(0)$ senza che ci siano conflitti, cioè ci possiamo trovare in una cella più di una azione. Questi conflitti sono spesso conflitti shift/reduce, cioè abbiamo una azione di shift e una di reduce, e accadono perché siamo stati poco restrittivi sulle azioni reduce e le abbiamo messo anche dove sappiamo che non potranno mai essere usate

8.6.1 Tabella di parsing SLR(1)

Per costruire una tabella di parsing un po' più "furba" di quella di $LR(0)$ occorre andare a limitare l'inserimento del reduce nelle caselle. Se infatti prima in $LR(0)$ si ha una reduce $\forall x \in T \cup \{\$\}$, in $SLR(1)$ si segue la seguente condizione:

$$(A \rightarrow \alpha. \in s \wedge A \neq S') \implies \forall x \in Follow(A). (M[s, x] = A \rightarrow \alpha)$$

Pertanto l'unica cosa che cambia è che la reduce viene messa solo in corrispondenza dei follow del non terminale in testa alla produzione pertanto grazie **al look-ahead** (il follow)

Example 8.6.1

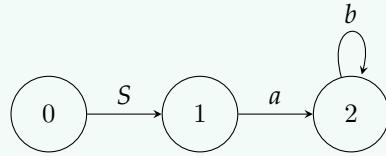
Consideriamo la grammatica G :

- (1) $S' \rightarrow S$
- (2) $S \rightarrow a$
- (3) $S \rightarrow ab$

L'insieme di linguaggio generato è:

$$L(G) = \{a, ab\}$$

il suo DFA shift-reduce è



La tabella LR(0) mostra un conflitto *shift-reduce*:

	<i>a</i>	<i>b</i>	\$	<i>S</i>
0	S1			
1	r2/S2	S2	r2	g3
2	r3	r3	acc	
3				

Come risolvere il conflitto? Usiamo il **lookahead**: guardiamo l'insieme *Follow(S)*. Se il simbolo attuale appartiene a *Follow(S)*, il conflitto è reale, altrimenti no!

Follow(S):

$$\text{Follow}(S) = \{\$\}$$

Risultato: Risolvendo il conflitto a favore dello shift, otteniamo la seguente tabella SLR(1):

	<i>a</i>	<i>b</i>	\$	<i>S</i>
0	S1			
1	S2	r2	r2	g3
2	r3	r3	acc	
3				

Conclusione: La grammatica G non è LR(0), ma è SLR(1) usando il lookahead per risolvere il conflitto

Il termine *SLR(1)* sta per "Single, left-to-right, rightmost derivation con un simbolo di lookahead

• **Note:**

Se G ha produzioni ε , allora G difficilmente è LR(0) (caso banale in cui uno stato con item $A \rightarrow \varepsilon$ non ha item del tipo $B \rightarrow a\alpha\beta$)

• **Note:**

Se L è libero deterministico e gode della **prefix-property** ("*L gode della prefix-property*"): se $\forall x, y \in L$ tale che x è prefisso di y , allora $L \in \text{LR}(0)$ Quindi, se L è libero deterministico ma non è LR(0), allora L non gode della prefix-property.

• **Note:**

Se $L \in \text{LR}(0)$ ed è **finito**, allora gode della prefix-property. Ovvero, se L è finito e non gode della prefix-property,

allora $L \notin \text{LR}(0)$.

• **Note:**

Se $L \in \text{LR}(0)$ ma è **infinito**, può $L = \{a, ab\}$ non godere della prefix-property.

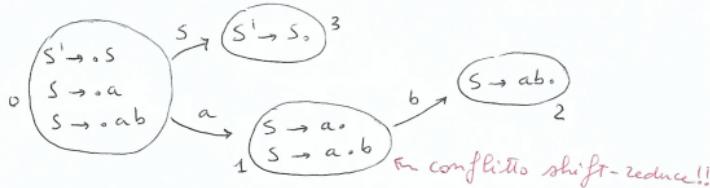
8.6.2 Parser LR(1)

Vi sono tuttavia delle grammatiche in cui neanche il look-ahead non basta, infatti il follow del non terminale in testa alla produzione è anch'esso troppo poco restrittivo

Example 8.6.2

Una grammatica libera G può non essere $LR(0)$!

$$(1) S^1 \rightarrow S \quad (2) S \rightarrow a \quad (3) S \rightarrow ab \quad L(G) = \{a, ab\}$$



	a	b	\$	S
0	s_1			g_3
1	r_2	r_2/r_2	r_2	
2	r_3	r_3	r_3	
3			acc	

Tabelle di parsing $LR(0)$

Come risolvere il conflitto? Usiamo il look-ahead!

Cioè guardiamo il $\text{Follow}(S)$: se $b \in \text{Follow}(S)$, allora il conflitto è reale! Altrimenti, no!
(può essere)

Ma $\text{Follow}(S) = \{\$\}$ \Rightarrow risolviamo il conflitto a favore della shift

	a	b	\$	S
0	s_1			g_3
1		s_2	r_2	
2			r_3	
3			acc	

Tabelle di parsing $SLR(1)$
↑
simple

- solo per i caratteri nel $\text{Follow}(S)$ mettiamo r_2

Ed ecco che entrano in gioco i cosiddetti **Parser LR(1)** che sfruttano gli $itemLR(1)$

Item LR(1)

Definition 8.6.1: Item LR(1)

Un item $LR(1)$ è una coppia formata da:

- un item $LR(0)$ che viene chiamato nucleo
- un simbolo di look-ahead in $T \cup \{\$\}$

introdurre questo simbolo di look-ahead è fondamentale, così che si può decidere che azione fare in base al simbolo letto in input

Example 8.6.3

Un possibile item $LR(1)$ è:

$$[S \rightarrow V. = E, \$]$$

Dove abbiamo che $S \rightarrow V. = E$ è il cuore e $\$$ è il simbolo di look ahead. Se avessi uno stato con due item dentro:

$$\begin{aligned} & [S \rightarrow V \cdot E, \$] \\ & [S \rightarrow V \cdot, \$] \end{aligned}$$

In questo caso se leggo il simbolo "=" si avrà uno shift per poi andare in un altro stato grazie al primo item, mentre se si legge \$ input avremo un reduce $E \rightarrow V$ proprio grazie al simbolo \$ di look-ahead

Intuizione : se l'automa canonico $LR(1)$ è in uno stato che contiene l'item $LR(1) [A \rightarrow \alpha.\beta, x]$:

- $[A \rightarrow \alpha.\beta, x]$:
 - Sta cercando di riconoscere la maniglia $\alpha\beta$
 - di essa, α è già sulla pila
 - Sull'input si aspetta una stringa derivabile da βx , ovvero si aspetta in input β (per fare uno shift) ed un x per la reduce
- $[A \rightarrow \alpha., x]$: fai la reduce $A \rightarrow \alpha$ se il prossimo input è proprio x

NFA LR(1)

Per creare gli NFA LR(1) si procede in maniera molto simile per NFA LR(0), ma tenendo traccia anche del simbolo di look ahead

- Stati: sono item $LR(1)$ della grammatica aumentata
- $[S' \rightarrow .S, \$]$ è lo stato iniziale
- dallo stato $[ato\alpha.X\beta, a]$ c'è una transizione allo stato $[A \rightarrow \alpha X.\beta, a]$ etichettata X per $X \in T \cup NT$
- dallo stato $[ato\alpha.X\beta, a]$ per $X \in T \cup NT$ e $\forall X \rightarrow \gamma$, c'è una ϵ -transizione verso lo stato $\forall b \in First(\beta a). [X \rightarrow \gamma, b]$, dove $First(\beta a) \subseteq T \cup \{\$\}$

Adesso si passa ai DFA LR(0)

Introduciamo le due funzioni ausiliarie: **Clos(I)** e **Goto(I,X)**

Algorithm 15: Clos

Input: I
Output: I

```

1 while I non è più modificato do
2   foreach item [A → α.Xβ] ∈ I do
3     foreach produzione X → γ do
4       foreach b ∈ First(βa) do
5         aggiungi [X → .γ, b] ad I;
6 return I;
```

Algorithm 16: Goto

Input: I
Output: I

```

1 Inizializza J = ∅;
2 foreach item [A → α.Xβ, a] ∈ I do
3   aggiungi [A → αX.β, a] a J;
4 return clos(J);
```

Viene presentato, qui, un esempio del grosso automa $LR(1)$

Example 8.6.4

Sia G la seguente grammatica

$$G = \begin{array}{l} 0) S' \rightarrow S \\ 1) S \rightarrow CC \\ 2) C \rightarrow cC \\ 3) C \rightarrow d \end{array}$$

Allora l'automa canonico LR(1) è:

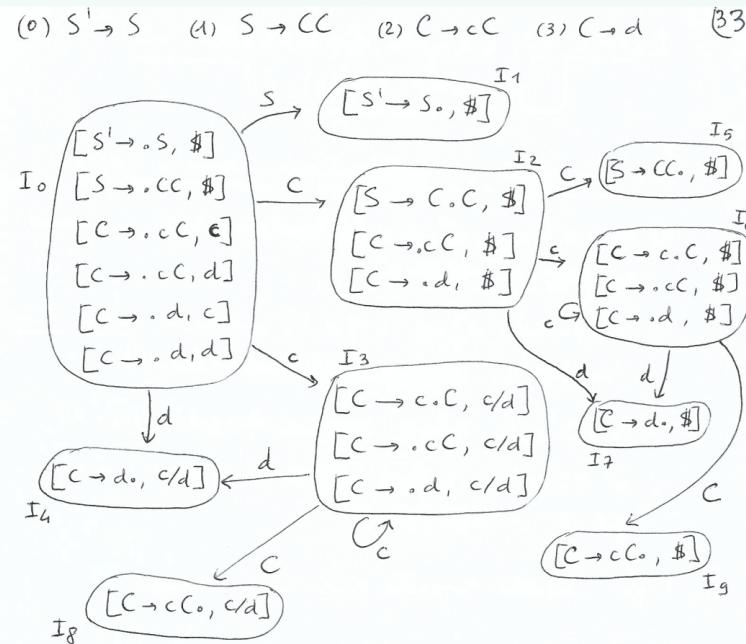


Tabella di parsing LR(0)

Uguale a quella $LR(0)$ e $SLR(1)$ solo che inserisco la reduce solo per il simbolo di look ahead. Riporto comunque la *minghia* di descrizione riportata dal buon Babbo Natale

1. Se $x \in T$ e $s \xrightarrow{x} t$ nell'automa $LR(1)$, inserisco shift t in $M[s, x]$.
2. Se $[A \rightarrow \alpha \cdot, x] \in s$ e $A3 \neq S'$, inserisco reduce $A \rightarrow \alpha$ in $M[s, x]$ (*solo per x del look-ahead!*).
3. Se $[S' \rightarrow S \cdot, \$] \in s$, inserisco Accept in $M[s, \$]$.
4. Se $A \in NT$ e $s \xrightarrow{A} t$ nell'automa $LR(1)$, inserisco goto t in $M[s, A]$.

Definition 8.6.2: Grammatica di classe $LR(1)$

Una grammatica libera G è di classe $LR(1)$ se ogni casella della sua tabella di parsing $LR(1)$ contiene al massimo una azione

Example 8.6.5

Si consideri l'automa dell'esempio precedente, allora si ha:

	c	d	\$	s	c
0	s3	s4		g1	g2
1			acc		
2	s6	s7			g5
3	s3	s4			g8
4	r3	r3			
5			r1		
6	s6	s7			g9
7			r3		
8	r2	r2			
9			r2		

si noti che per questa grammatica la tabella di parsing ha 7 stati anzichè 10, ed è pure senza conflitti

8.6.3 Parser LALR(1)

Il problema coi parser $LR(1)$ è che le tabelle sue tabelle di parsing sono molto grandi (centinaia di migliaia di stati per linguaggi di media grandezza), pertanto entrano in gioco i $LALR(1)$ che rappresentano un buon compromesso tra semplicità e selettività

Come si ottiene il parser $LALR(1)$

Si osservi che:

1. **Nucleo di uno stato $LR(1)$:**

L'insieme di item $LR(0)$ ottenuto dimenticando i *look-ahead* dagli item $LR(1)$.

2. **Nucleo di stato $LR(1)$:**

Ogni nucleo di uno stato $LR(1)$ corrisponde a uno stato dell'automa $LR(0)$.

3. **Transizioni dell'automa $LR(1)$:**

Le transizioni dell'automa $LR(1)$ dipendono solo dal nucleo. La funzione $goto(I, X)$ usa di I solo le parti "nucleo/ $LR(0)$ " degli item $LR(1)$.

Tabella di parsing LALR(1)

La tabella di parsing LALR(1) si ottiene da quella $LR(1)$ **fondendo insieme gli stati con lo stesso nucleo**. Ciò comporta:

- **Meno righe**, poiché gli stati dell'automa $LR(0)$ vengono combinati.
- **Meno reduce** rispetto alla tabella SLR(1).

Example 8.6.6

Sia la grammatica degli esempi precedenti, ecco qui il nostro automa

Tabella LR(1)

	c	d	\$	S	C	
0	s3	s4		g1	g2	(0) $S^* \rightarrow S$
1			acc			(1) $S \rightarrow CC$
2	s6	s7			g5	G (2) $C \rightarrow cC$
3	s3	s4			g8	(3) $C \rightarrow d$
4	z3	z3				
5			z1			Guardando
6	s6	s7			g9	l'automa LR(1) a
7			z3			nf. 33, si vede
8	z2	z2				che 3-6
9			z2			4-7
						δ^{-g} nonno essere fusi !!

T : MN (A D F L)

Passando da lr(1) a LALR(1)

La fusione di due stati $LR(1)$ con lo stesso core può causare conflitti

Theorem 8.6.1 Babbo natale Theorem

Quando fondo due stati $LR(1)$ per generare un parser $LALR(1)$ sono possibili solo conflitti **reduce/reduce**

Dimostrazione: Supponiamo che in s , stato ottenuto dalla fusione di due stati $LR(1)$, s_1 ed s_2 . Allora in s esiste un item $[S \rightarrow \alpha.a]$ e un item $[B \rightarrow \beta.a\gamma, b]$ (H1)

Suppongo che $[A \rightarrow \alpha., a] \in s_1$ (H2)

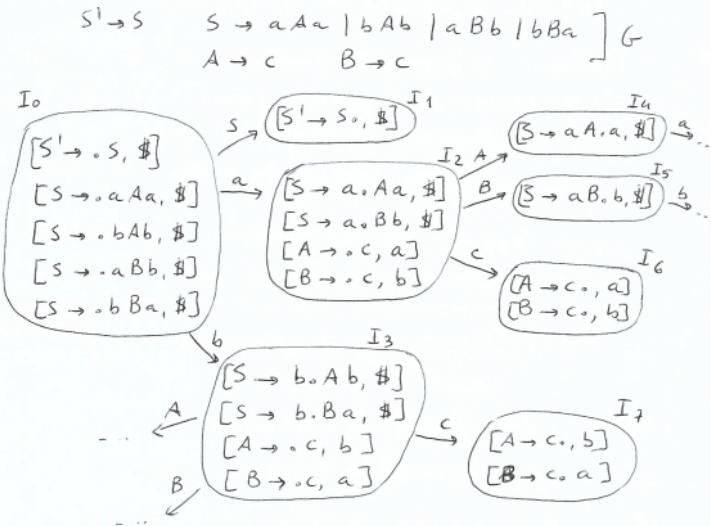
per (H1) $[A \rightarrow \alpha., a] \in s_1$ e $[B \rightarrow \beta.a\gamma, c] \in s_1$ (H3) dato che se ho fuso insieme s_1 ed s_2 allora il core $[B \rightarrow \beta.a\gamma]$ deve per forza esserci in entrambi \heartsuit

Per (H3) si ha che anche in $LR(1)$ si verificherebbe un conflitto shift-reduce, contro l'ipotesi che la tabella $lr(1)$ non presenti conflitti, pertanto se $LR(1)$ è senza conflitti, $LALR(1)$ potrebbe solo presentare conflitti reduce-reduce \heartsuit

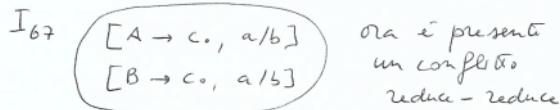
Note:

Se si generano conflitti, allora G non è $LALR(1)$, pur essendo $LR(1)$

Example 8.6.7



I₆ e I₇ hanno lo stesso core - ma se li fonda



$$M'[67, a] = \{ A \xrightarrow{\text{reduce}} c, B \xrightarrow{\text{reduce}} c \}$$

$$M'[67, b] = \{ A \xrightarrow{\text{reduce}} c, B \xrightarrow{\text{reduce}} c \}$$

N.B. Nello stato I₆ non c'è conflitto, e neanche nello stato I₇! $\Rightarrow G$ è davvero LR(1), ma G non è LALR(1)

8.7 Grammatiche LR(k)

Ci si può concentrare adesso alle grammatiche LR con k simboli di look-ahead, nessuno le usa perché molto confusionarie e macchinose, ma si possono andare a definire

Prima della definizione di grammatica occorre definire gli **item** $LR(k)$

Definition 8.7.1: Item $LR(k)$

Si definisce un **item** $LR(k)$, un item formato da una coppia formata da un item $LR(0)$ e una stringa β la cui lunghezza è inferiore a k , e si denota nel seguente modo:

$$[item\ LR(0), \beta] \text{ con } |\beta| \leq k$$

Tali item hanno queste caratteristiche:

- Lo stato iniziale è sempre $[S' \rightarrow .S, \$]$
- Sia s uno stato dell'automa canonico $LR(k)$, si che $([A \rightarrow \alpha.X\gamma, \beta] \in s \wedge \exists \text{ la produzione } X \rightarrow \gamma \wedge w \in First_k(\gamma\beta)) \implies [X \rightarrow .\gamma, w] \in s$

Si può subito intuire che un automa canonico $LR(k)$ genera una quantità molto grande di stati molto difficili da gestire a mano

Una volta costruito l'automa canonico $LR(k)$ si passa alla tabella di parsing $LR(k)$ con colonne su T^k (cioè su stringhe di terminali lunghe k)

Definition 8.7.2: Grammatica $LR(k)$

Si definisce G una grammatica $LR(k)$ una grammatica costruita a partire da una tabella di parsing $LR(k)$, ottenuta a partire dall'automa canonico $LR(k)$, con le seguenti caratteristiche:

- Contiene al più un'azione per ogni entrata della tabella
- non esistono sue w_1 e w_2 sulle colonne tali che w_1 è prefisso di w_2 e per almeno una righe le 2 corrispondenti entrate contengono un'azione

Note:

Ovviamente la tabella viene riempita nel modo ovvio per le azioni di shift/accept, mentre le reduce sono messe solo in corrispondenza del lookahead, ad esempio $[A \rightarrow \alpha., w]$

8.7.1 Grammatiche $SLR(k)$

Si parte dall'automa canonico $LR(0)$ e si riempie la tabella di parsing $SLR(k)$ - che ha colonne su T^k - secondo la seguente legge:

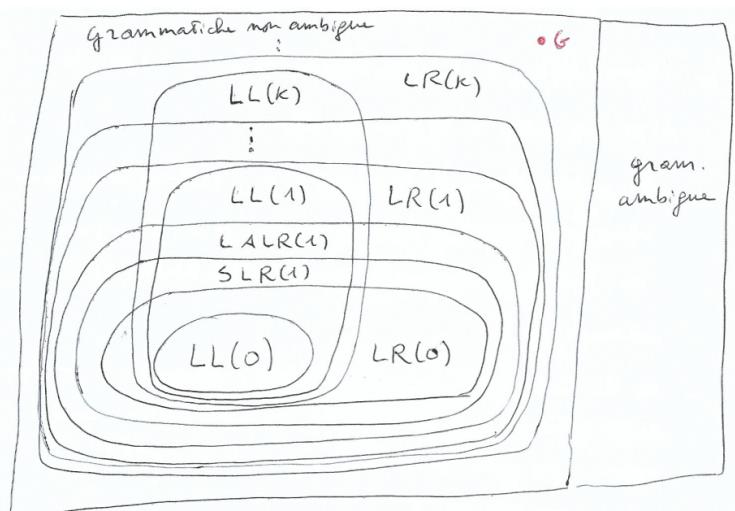
$$([A \rightarrow \alpha.] \in s \wedge A \neq S') \implies \forall w \in Follow_k(A). (M[s, w] = \text{reduce } A \rightarrow \alpha)$$

8.7.2 Grammatiche $LALR(k)$

Si parte dall'automa canonico $LR(k)$ e si fondono insieme due stati con lo stesso nucleo, se non vi sono conflitti G è $LALR(k)$

8.7.3 relazione tra le varie grammatiche

Si può notare il seguente schema riassuntivo:



Si può concludere che:

$$LR : \begin{cases} SLR(k) \subset LALR(k) \subset LR(k) & \text{per ogni } k \geq 1 \\ SLR(1) \subset SLR(2) \subset \dots \subset SLR(k) \\ LALR(1) \subset LALR(2) \subset \dots \subset LALR(k) \\ LR(0) \subset LR(1) \subset \dots \subset LR(k) \end{cases}$$

LL vs LR:

$$LLvsLR : \begin{cases} LL(k) \subset LR(k) & \text{per ogni } k \geq 0 \\ LL(k) \not\subset LR(k-1) & \text{per ogni } k \geq 1 \end{cases}$$

Proposition 8.7.1

- Se G è $LL(k)$, allora G è non ambigua e $L(G)$ è deterministico
- Se G è $LR(k)$, allora G è non ambigua e $L(G)$ è deterministico

Note:

Esistono linguaggi generati da grammatiche non ambigue, ma nondeterministici

Example 8.7.1

Sia G la seguente grammatica:

$$S \rightarrow aSa \mid bSb \mid \epsilon$$

Allora

$$L(G) = \{ww^R \mid w \in \{a, b\}^*\}$$

8.8 Linguaggi generati dalle varie grammatiche

Definition 8.8.1: Linguaggio di classe X

Un linguaggio L si definisce di classe X se $\exists G$ di classe X tale che $L = L(G)$
Dove X sta per $LR(0)/SLR(1)/LL(1)/\dots$

Se classifichiamo i linguaggi anziché le grammatica, il diagramma si semplifica molto

Example 8.8.1

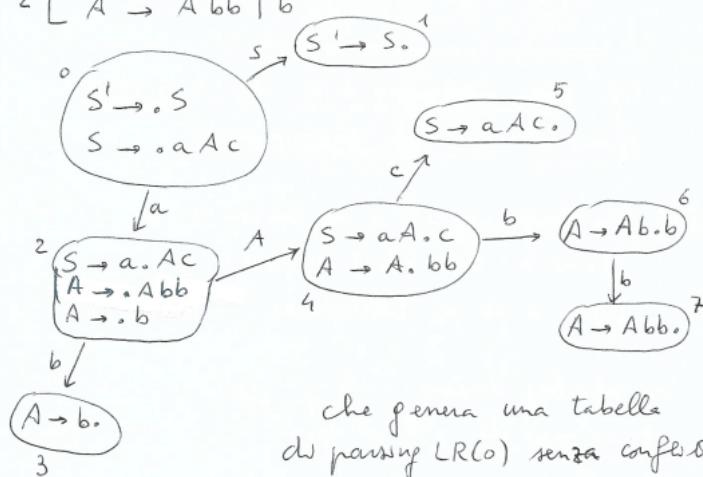
$S \rightarrow aAc$
 $A \rightarrow bAb \mid b$] G_1 si può dimostrare non
essere $LR(k)$ per nessun k !

$L(G_1) = \{a^nb^{2n+1}c \mid n \geq 0\}$

Infatti, non si può mai
sapere se in "a b^n " si
debbra ridurre con $A \rightarrow b$,
fino a quando non si
incontra "c"; cioè dovrei
sapere quando sono nel
"metto".

Ma il lang. generato da G_1
è generabile da una gram. $LR(0)$!

G_2 [$S \rightarrow aAc$
 $A \rightarrow A.bbb \mid b$



che genera una tabella
di parsing $LR(0)$ senza conflitti.

$\Rightarrow L(G_1) = L(G_2)$ è un lang. $LR(0)$!

Si considerino queste note

Note:

Un linguaggio è libero deterministico se è accettato, per stato finale

Note:

Ogni linguaggio regolare è generato da una grammatica di classe $LL(1)$

Note:

esistono linguaggi regolari che non sono $LR(0)$
Ad es. $L = \{a, ab\}$

Si considerino i seguenti teoremi:

Theorem 8.8.1

Un linguaggio è libero deterministico sse è generato da una grammatica $LR(1)$ per qualche $k \geq 0$

Theorem 8.8.2

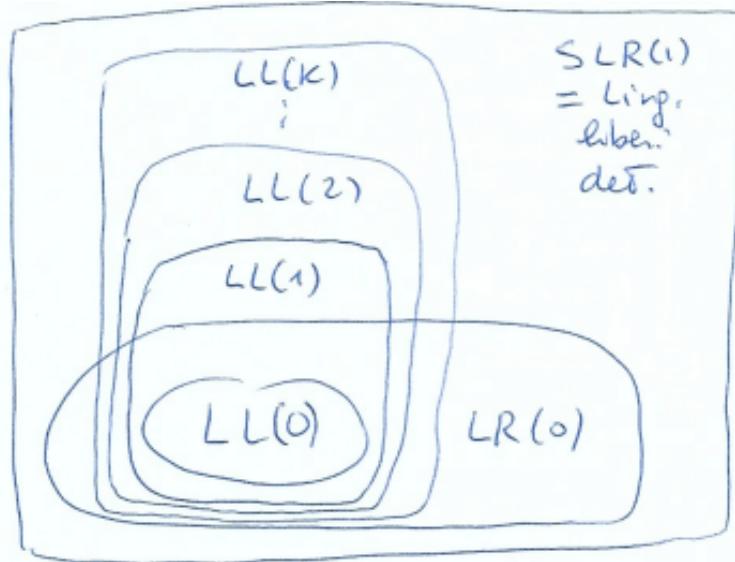
Un linguaggio è libero deterministico sse è generato da una grammatica $SLR(1)$

Theorem 8.8.3

I linguaggi generati da grammatica $LL(k)$ sono esattamente contenuti nei linguaggi generati da grammatiche $SLR(1), \forall k \geq 0$

classificazione dei linguaggi

Si noti il seguente schema:



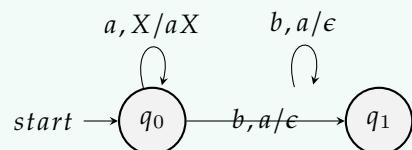
Note:

Se G è $LR(k)$, esiste $G' \in SLR(1)$ equivalente, ma G' può essere molto più complessa di G

Example 8.8.2

Sia $L = \{a^i b^j \mid i \geq j \geq 0\}$ libero deterministico

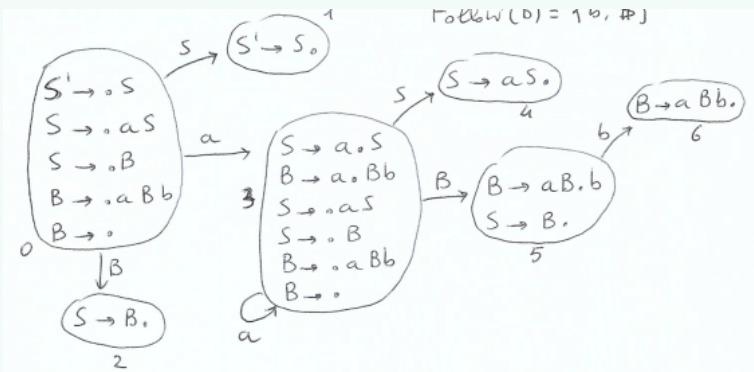
Si può notare che



è il DPDA che riconosce L per stato finale

L però non è $LL(k)$ per nessun k , tuttavia L è $SLR(1)$, infatti esiste G $SLR(1)$ tale che $L(G) = L$:

$$\begin{aligned} S &\rightarrow aS \mid B \\ B &\rightarrow aBb \mid \epsilon \end{aligned}$$



La cui tabella di parsing è:

	a	b	$\$$	S	B
0	S_3	R_4	R_4	G_1	G_2
1			acc		
2			R_2		
3	S_3	R_4		G_4	G_5
4			R_4		
5	S_3		R_2		
6	R_3	R_3			

Tabella di parsing SLR(1) senza conflitti
 $\Rightarrow G$ è SLR(1)

Note:

- $L_1 = \{a^n b^n \mid n \geq 1\} \in \text{LL}(1) \in \text{LR}(0)$
 $L_2 = \{a^n c^m \mid n, m \geq 1\} \in \text{LL}(1) \in \text{LR}(0)$

ma $L_1 \cup L_2$ non è LL(k) per nessun k ,
mentre $L_1 \cup L_2 \in \text{LR}(0)$
 \Rightarrow l'unione di linguaggi LL(1) può non essere LL(1)

- $L_1 = \{a\} \in \text{LL}(1) \in \text{LR}(0)$
 $L_2 = \{ab\} \in \text{LL}(1) \in \text{LR}(0)$

ma $L_1 \cup L_2$ non è LR(0) perché non gode della prefix-property,
mentre $L_1 \cup L_2 \in \text{LL}(1)$
 \Rightarrow l'unione di linguaggi LR(0) può non essere LR(0)

- $L_1 = a^* = \{a^n \mid n \geq 0\} \in \text{LL}(1)$
 $L_2 = \{a^n b^n \mid n \geq 0\} \in \text{LL}(1)$

ma $L_1 \cdot L_2 = \{a^i b^j \mid i \geq j \geq 0\}$ non è LL(k) per nessun k
 \Rightarrow la concatenazione di due linguaggi LL(1) può non essere LL(1)

Chapter 9

Nomi e Ambiente

Nell'evoluzione dei linguaggi di programmazione, i *nomi* hanno avuto un ruolo fondamentale nella sempre maggiore astrazione rispetto al linguaggio macchina.

Definition 9.0.1: Nome

I nomi sono solo una sequenza (significativa o meno) di caratteri che sono usati per rappresentare un oggetto, che puo' essere uno spazio di memoria se vogliamo etichettare dei dati, o un insieme di comandi nel caso di una funzione.

9.1 Nomi e Oggetti denotabili

Spesso, i nomi sono *identificatori*, ovvero token alfanumerici, ma possono essere usati anche simboli (+,-,...). E' importante ricordare che il nome e l'oggetto denotato non sono la stessa cosa, infatti un oggetto puo' avere diversi nomi (*aliasing*) e lo stesso nome puo' essere attribuito a diversi oggetti in momenti diversi (*attivazione* e *deattivazione*).

Definition 9.1.1: Oggetti denotabili

Sono gli oggetti a cui e' possibile attribuire un nome.

Note:

Non centra con la programmazione ad oggetti

Possono essere:

- Predefiniti: tipi e operazioni primitivi, ...
- Definibili dall'utente: variabili, procedure, ...

Quindi il legame fra nome e oggetto (chiamato **binding**) puo' avvenire in momenti diversi:

- Statico: prima dell'esecuzione del programma
- Dinamico: durante l'esecuzione del programma

9.2 Ambienti e Blocchi

Non tutti i legami fra nomi e oggetti vengono creati all'inizio del programma restando immutati fino alla fine. Per capire come i binding si comportano, occorre introdurre il concetto di *ambiente*:

Definition 9.2.1: Ambiente

Insieme di associazioni nome/oggetto denotabile che esistono a runtime in un punto specifico del programma ad un momento specifico durante l'esecuzione.

Soltamente nell'ambiente non vengono considerati i legami predefiniti dal linguaggio, ma solo quelli creati dal programmatore utilizzando le *dichiarazioni*, costrutti che permettono di aggiungere un nuovo binding nell'ambiente corrente.

Notare che e' possibile che nomi diversi possano denotare lo stesso oggetto. Questo fenomeno e' detto *aliasing* e succede spesso quando si lavora con puntatori.

9.2.1 Blocchi

Tutti i linguaggi di programmazione importanti al giorno d'oggi utilizzano i *blocchi*, strutture introdotte da ALGOL 60 che servono per strutturare e organizzare l'ambiente:

Definition 9.2.2: Blocco

Pezzo contiguo del programma delimitato da un inizio e una fine che puo' contenere dichiarazioni **locali** a quella regione.

Puo' essere:

- In-line (o anonimo): puo' apparire in generale in qualunque punto nel programma e non corrisponde a una procedura.
- Associato a una procedura

Permettono di strutturare e riutilizzare il codice, oltre a ottimizzare l'occupazione di memoria e rendere possibile la ricorsione.

9.2.2 Tipi di Ambiente

Un'altro meccanismo importante che forniscono i blocchi e' il loro *annidamento*, ovvero l'inclusione di un blocco all'interno di un altro (non la sovrapposizione parziale). In questo caso, se i nomi locali del blocco esterno sono presenti nell'ambiente del blocco interno, si dice che i nomi sono *visibili*. Le regole che determinano se un nome e' visibile o meno a un blocco si chiamano *regole di visibilita'* e sono in generale:

- Un nome locale di un blocco e' visibile a esso e a tutti i blocchi annidati.
- Se in un blocco annidato viene creata una nuova dichiarazione con lo stesso nome, questa ridefinizione *nasconde* quella precedente.

Definition 9.2.3: Ambiente associato a un blocco

L'ambiente di un blocco e' diviso in:

- **locale**: associazioni create all'ingresso nel blocco:
 - variabili locali
 - parametri formali (nel caso di un blocco associato a una procedura)
- **non locale**: associazioni ereditate da altri blocchi (senza considerare il blocco globale), che quindi non sono state dichiarate nel blocco corrente
- **globale**: associazioni definite nel blocco globale (visibile a tutti gli altri blocchi)

9.2.3 Operazioni sull'ambiente

- Creazione: dichiarazione locale, in cui introduco nell'ambiente locale una nuova associazione
- Riferimento: uso di un nome di un oggetto denotato
- Disattivazione/Riattivazione: quando viene ridefinito un certo nome, all'interno del blocco viene disattivato. Quando esco dal blocco riattivo la definizione originale
- Distruzione: le associazioni locali del blocco dal quale si esce vengono distrutte

Note:

Creazione e distruzione di un *oggetto denotato* non coincide necessariamente con la creazione o distruzione sull'associazione tra il nome e l'oggetto stesso, per essere più precisi nemmeno la vita dell'oggetto e del legame è la stessa. Verrà quindi mostrato nel dettaglio

9.2.4 Vita di un oggetto

Definition 9.2.4: Vita

Si definisce **tempo di vita** o **lifetime** di un oggetto o legame il tempo che intercorre tra la sua creazione e la sua distruzione

Per comprendere meglio questo concetto, i seguenti notino gli *eventi fondamentali*

| Creazione di un oggetto

| Creazione di un legame per loggetto

| Riferimento alloggetto, tramite il legame

| Disattivazione di un legame

| Riattivazione di un legame

| Distruzione di un legame

| Distruzione di un oggetto

Dal punto 1 e 7 è *la vita dell'oggetto*, mentre dall'evento 2 al 6 è *la vita dell'associazione*

Note:

È pertanto vero, quindi, che la vita di un oggetto non coincide con la vita dei legami per quell'oggetto

Esistono 2 modi per categorizzare il tempo di vita di un legame/associazione:

- Vita delloggetto più **lunga** di quella del legame

Si consideri questo codice

```
1 program ExampleCode;
2
3     procedure P(var X: integer); begin ... end;
4     ...
5     var A:integer;
6     ...
7
8     P(A); {chiamata a P con A}
```

Nel codice dato, inizialmente il nome A viene associato a un oggetto (un valore intero). Quando si chiama la procedura P(A), l'argomento A viene passato per riferimento, il che significa che all'interno della procedura non viene creato un nuovo oggetto, ma semplicemente un nuovo nome per lo stesso oggetto: X.

Durante l'esecuzione della procedura, X e A sono quindi due nomi che fanno riferimento allo stesso valore in memoria. Qualsiasi modifica apportata a X all'interno della procedura si riflette direttamente su A.

Una volta terminata l'esecuzione della procedura, il legame tra X e l'oggetto viene distrutto, mentre A continua a riferirsi allo stesso valore, eventualmente modificato dalla procedura. Questo è un classico esempio in cui la durata del legame tra un nome (X) e un oggetto è più breve della vita dell'oggetto stesso

- Vita delloggetto più **breve** di quella del legame

Si consideri questo codice, piuttosto nasty in C:

```

1      #include <stdio.h>
2      #include <stdlib.h>
3
4      int main() {
5          int *X, *Y;
6          X = (int *) malloc(sizeof(int));
7          Y = X;
8          free(X);
9          X = NULL;
10         return 0;
11     }
12 }
```

Nel codice illustrato vengono creati due puntatori. L'oggetto puntato da `X`, attraverso il comando `malloc`, punta a un'area di memoria allocata dinamicamente. Di conseguenza, assegnando `Y = X`, anche `Y` farà riferimento allo stesso oggetto puntato da `X`.

Col comando `free(X)`, l'oggetto alla fine della catena viene deallocated, ovvero la memoria precedentemente allocata viene liberata. Successivamente, l'istruzione `X = NULL` imposta `X` a `NULL`, indicando che non punta più a un'area valida di memoria.

Tuttavia, il puntatore `Y` continua a riferirsi all'oggetto che è stato deallocated. Questo crea un *dangling pointer* (puntatore pendente), poiché il legame tra `Y` e l'oggetto non esiste più in modo sicuro. Accedere a `Y` dopo la deallocazione può portare a comportamenti indefiniti e DA EVITARE CAZZO

9.3 Regole di scope

Innanzi tutto fornirò la definizione di scope

Definition 9.3.1: Scope

Lo **scope** (o **ambito**) è un concetto semantico che determina in quali porzioni di un programma una variabile o un nome è visibile e utilizzabile. Le regole di scope stabiliscono come i riferimenti ai nomi vengono risolti all'interno di un determinato contesto di esecuzione, garantendo che l'uso delle variabili sia coerente e prevedibile.

Come detto in precedenza una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno ché intervenga in tali blocchi una nuova dichiarazione dello stesso nome che nasconderà quello precedente (shadowing)

Occorre tuttavia determinare come interpretare le regole di visibilità di una variabile in presenza di porzioni di blocchi eseguiti in posizioni diverse dalle loro definizioni e in presenza di ambienti non locali... nasty vero?

Vi sono due filosofie principali

- **Statico:** Basato sul testo del programma
- **Dinamico:** Basato sul flow di esecuzione

Prima di andare avanti, si noti la seguente annotazione

Note:

Entrambe gli apprezzano solo in presenza congiunta di ambiente non locale e non globale e procedura

Vabbùò, è normale non capirci un catso solo a parole, si consideri il seguente testo:

```

1 A:{int x = 0;
2     void pippo(int n){
3         x = n+1;
4     }
5     pippo(3);
6     write(x);
7     {
8         int x = 0;
```

```

9     pippo(3);
10    write(x);
11 }
12 write(x);
13 }
```

Che cosa caspita scriveremo a riga 10? Ebbene dipenderà dal tipo di regola di scope, o statica o dinamica
Di seguito sono riportati nel dettaglio

9.3.1 Scope statico

Definition 9.3.2: Scope statico

La regola dello **scope statico** (o **regola dello scope annidato più vicino**) si basa sui seguenti principi:

1. **Ambiente locale di un blocco:** Le dichiarazioni all'interno di un blocco definiscono il suo ambiente locale. Questo include solo le dichiarazioni presenti direttamente nel blocco stesso e non quelle eventualmente presenti nei blocchi annidati al suo interno.
2. **Ricerca delle associazioni di un nome:** Se un nome viene utilizzato all'interno di un blocco, si segue questa gerarchia per determinare quale dichiarazione è valida:
 - Se esiste una dichiarazione del nome nel **blocco locale**, questa è quella valida.
 - Se il nome non è dichiarato nel blocco locale, si cerca nel **blocco immediatamente contenitore**.
 - Se il nome non è ancora trovato, si continua a risalire nei blocchi contenitori fino al più esterno.
 - Se il nome non è dichiarato nemmeno nel blocco più esterno, si cerca nell'**ambiente predefinito del linguaggio**.
 - Se il nome non è presente neanche nell'ambiente predefinito, si genera un errore.
3. **Blocchi con nome:** Un blocco può essere **assegnato a un nome**, e in questo caso tale nome diventa parte dell'ambiente locale del **blocco immediatamente contenitore**. Questo vale anche per i blocchi associati a procedure o funzioni.

Molto più semplicemente si può dire che

Note:

Un nome non locale è risolto nel blocco che *testualmente* lo racchiude

Pertanto nel codice d'esempio nel primo `write(x)` verrà stampato 4, nel secondo 0 e nel terzo 4, in quanto la `x` che la funzione `pippo(3)` modifica è quella dichiarata all'interno del blocco che lo racchiude, in questo caso A. Nel blocco B non verrà modificata la `x` racchiusa nello stesso quindi si stamperà, quindi 0.

Si ha quindi una forte indipendenza dalla posizione della posizione da parte dei nomi. Ad esempio se si dichiara una funzione all'interno di un blocco, il corpo della procedura si riferirà sempre alle regole di scope medesime del blocco in cui è stata dichiarata, pertanto dovunque la funzione verrà chiamata lo scope a cui riferisce sarà sempre lo stesso

Tra i vantaggi dello scope statico troviamo una maggiore comprensione per il programmatore, ogni nome può essere collegato alla sua dichiarazione semplicemente analizzando la struttura del codice, senza dover simulare lesecuzione e la facilità di analisi del programma da parte del compilatore che può determinare tutte le occorrenze di un nome e fare controlli di correttezza sui tipi di dati ed eseguire ottimizzazioni del codice prima dellesecuzione

9.3.2 Scope dinamico

Definition 9.3.3: Regole di scope dinamico

Secondo le regole di scope dinamico, l'associazione valida per un nome x ad un punto P del programma è la più recente (in senso temporale) associazione creata per x ancora attiva appena il controllo di flusso arriva a P

In pratica occorre andare indietro *nell'esecuzione* per cercare l'occorrenza d'interesse (è l'ultima che è stata introdotta) blocco attivato per ultimo (che deve essere ancora attivo), come riassunto in questa nota quindi:

Note:

Un nome non locale è risolto nel blocco attivato più di recente e non ancora disattivato

Quando un nome non è dichiarato localmente in un blocco, viene cercato nel blocco attivato più recentemente che lo contiene. Questo significa che la risoluzione dei nomi segue una logica *LIFO* (*Last In First Out*), cioè una gestione a stack basata sull'ordine di chiamata delle funzioni. Questo approccio è più semplice da gestire a runtime perché si basa solo sulla pila di attivazione, senza necessità di altre strutture dati

9.4 determinare l'ambiente

L'ambiente è quindi determinato da:

- Regole di scope (statico o dinamico)
- Regole di visibilità
- Regole di binding (intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X)
- Regole per il passaggio di parametri

Chapter 10

Gestione Memoria

Prima di discutere su i diversi modi in cui la memoria (RAM e HD solo se necessario) puo' essere gestita dal compilatore di un linguaggio, e' importante identificare cosa esattamente deve essere contenuto all'interno di essa. Sicuramente, ogni *dato* che deve essere salvato durante l'esecuzione del programma dovrà avere un posto nella memoria, come ad esempio le variabili, ma ci sono anche informazioni per il *controllo dell'esecuzione* che necessitano di essere memorizzati.

La vita di un oggetto corrisponde con tre meccanismi di allocazione di memoria:

- **Allocazione statica:** l'oggetto viene allocato una volta sola, prima dell'inizio dell'esecuzione del programma, e deallocato alla fine dell'esecuzione,
pertanto è una memoria allocata a tempo di compilazione
- **Allocazione automatica:** l'oggetto viene allocato all'entrata di un blocco (tipicamente una funzione) e deallocato all'uscita del blocco.
- **Allocazione dinamica:** l'oggetto viene allocato e deallocato esplicitamente dal programmatore tramite chiamate a funzioni di allocazione e deallocazione (ad esempio, `malloc` e `free` in C). *pertanto è una memoria allocata a tempo di esecuzione*

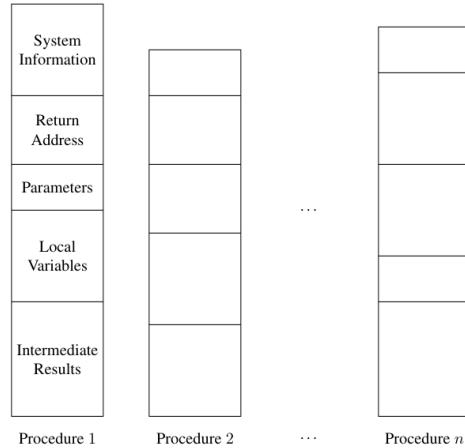
Questo tipo di allocazione di serve di due aree di memoria:

- pila (stack): gli oggetti sono allocati con una politica LIFO, utilizzato per le variabili locali e i parametri formali delle funzioni
- heap: gli oggetti sono allocati e deallocati in qualsiasi ordine, utilizzato per gli oggetti dinamici (puntatori)

10.1 Allocazione Statica

Non sarebbe possibile utilizzare solo una memoria con allocazione statica? Sicuramente per le variabili *globali*, *costanti* che non dipendono da altri valori non noti inizialmente e anche per le *tabelle* che utilizza il compiler, l'allocazione statica funziona benissimo.

Per quanto riguarda i sottoprogrammi, si puo' pensare di allocare per ognuna di esse tutto lo spazio necessario per i parametri e le variabili locali (e altre informazioni necessarie), dato che possiamo determinare tutto cio' prima di eseguire il programma:



Questo metodo funziona solo se siamo sicuri che, se una procedura e' attiva, allora non puo' chiamare la stessa procedura ricorsivamente. Questo e' perche' esiste solo un'istanza per ogni procedura allocata in memoria e non e' possibile sapere quante volte una funzione verra' chiamata ricorsivamente. Quindi, se vogliamo implementare la ricorsione, serve l'allocazione *dinamica* (stack di chiamate):

10.2 Allocazione Dinamica

10.2.1 Allocazione Dinamica con Pila

Ogni istanza di sottoprogramma viene memorizzata con un *frame* (o *record di attivazione*) che contiene tutte le informazioni necessarie. Quando un'istanza viene attivata, il relativo frame viene messo in cima a una **pila**, la struttura dati naturale in quanto se una procedura A chiama una procedura B, allora siamo sicuri che B deve terminare prima che A possa continuare l'esecuzione e terminare anch'esso.

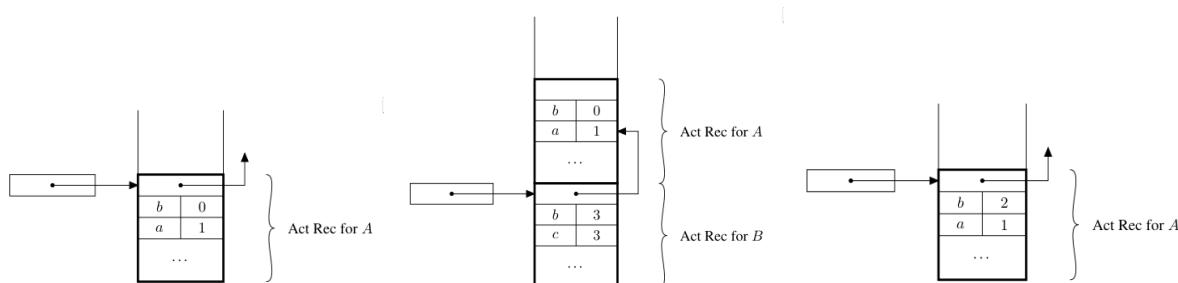
Note:

L'allocazione dinamica e' utile anche quando non c'e' ricorsione come meccanismo per risparmiare memoria.

Vediamo un esempio:

```

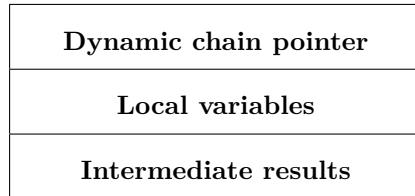
1 A:{  
2     int a = 1;  
3     int b = 0;  
4  
5 B:{  
6     int c = 3;  
7     int b = 3;  
8 }  
9     b = a + 1;  
10 }
```



Vediamo piu' in dettaglio cosa viene memorizzato nei record di attivazione:

Record di attivazione

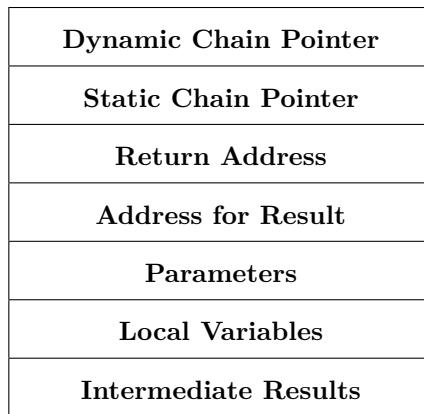
Per un semplice blocco anonimo, il corrispondente frame ha tale forma:



Note:

Nella realta' la maggior parte dei linguaggi usa l'allocazione statica per blocchi anonimi per maggiore efficienza di calcolo (sacrificando pero' l'efficienza di memoria).

Mentre per le procedure e' un po' piu' complesso:

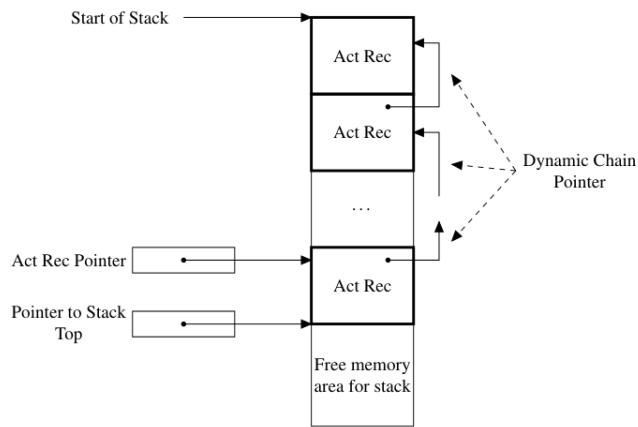


Vediamo in dettaglio cosa sono tutti sti dati:

- **Intermediate Results:** serve per memorizzare risultati intermedi di equazioni complicate e per risultati di chiamate ricorsive.
- **Local Variables e Parameters:** per le variabili locali e parametri.
- **Dynamic Chain Pointer:** puntatore all'ultimo RdA creato sulla stack, l'insieme di tutti i puntatori dinamici e' chiamata *catena dinamica*.
- **Static Chain Pointer:** informazione necessaria per implementare lo scope statico, vedremo piu' avanti.
- **Return Address:** indirizzo di memoria della prima istruzione da eseguire quando termina la procedura corrente.
- **Address for Result:** indirizzo dove puo' essere salvato il valore di ritorno della funzione (sara' un indirizzo interno al frame della funzione chiamante).

Gestione della Pila

Vediamo la struttura di un sistema a pila (discendente):



Come puo' notare dall'immagine, ci sono due puntatori esterni che puntano a posti specifici sull'ultimo record di attivazione:

- **Pointer al RdA:** e' un puntatore ad un luogo predeterminato all'interno del frame usato come base da cui si puo' calcolare l'offset per accedere alle variabili locali. Questo offset e' determinabile staticamente dal compilatore (ad eccezione del caso di varialli di dimensione variabile).
- **Stack Top Pointer:** come si puo' indovinare, e' il puntatore al primo indirizzo libero di memoria dopo l'ultimo RdA. Puo' essere omesso se e' possibile calcolare lo stesso indirizzo partendo dal pointer al RdA.

Il funzionamento corretto della pila di RdA e' dato dalla collaborazione fra il chiamante e il blocco chiamato, che eseguono dei blocchi di codice inseriti dal compilatore (o interprete) prima e dopo chiamate a procedure e blocchi anonimi:

- **Sequenza di Chiamata:** eseguito dal chiamante subito prima della chiamata
- **Prologo:** eseguito immediatamente all'inizio del blocco chiamato
- **Epilogo:** eseguito alla fine del blocco
- **Sequenza di Ritorno:** eseguito dal chiamante immediatamente dopo la chiamata

Al momento della chiamata, la Sequenza di Chiamata e il Prologo devono:

- **Modificare PC**
- **Allocare spazio sulla pila**
- **Modifica pointer RdA**
- **Passaggio parametri**
- **Memorizzare registri**

Quando il blocco o processo chiamato termina, l'Epilogo e la Sequenza di Ritorno devono:

- **Modificare PC**
- **Ritornare Valori**
- **Recuperare Registri**
- **Deallocare spazio sulla pila**

Note:

Sono stati omessi meccanismi per l'implementazione delle regole di scope. Vedremo queste piu' avanti.

10.2.2 Allocazione Dinamica con Heap

Nel caso in cui vogliamo dare la possibilita' a chi usa il linguaggio di allocare esplicitamente memoria a run-time o di usare oggetti di dimensioni variabili sorge il seguente problema: la vita degli oggetti non e' per forza LIFO, ovvero un oggetto creato prima di un altro puo' essere rimosso dalla memoria prima di un'altro oggetto creato dopo, come nel seguente blocco di codice:

```
1 int *p, *q;
2 p = malloc(sizeof(int));
3 q = malloc(sizeof(int));
4 *p = 0;
5 *q = 1;
6 free(p);
7 free(q);
```

Dobbiamo usare quindi la *heap*, ovvero una regione di memoria i cui blocchi possono essere allocati/deallocati in momenti arbitrari.

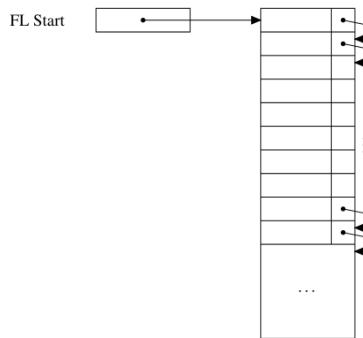
Note:

La heap che abbiamo appena definito non centra niente con la struttura dati usata per la "heap sort".

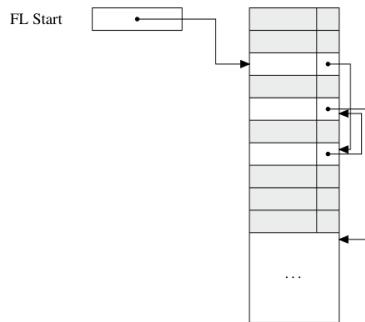
Esistono due categorie principali di metodi di gestione della heap a seconda della lunghezza dei blocchi che memorizza, che possono essere di *dimensione fissa* o *variabile*.

Dimensione fissa

La heap viene suddivisa in blocchi di dimensione fissa abbastanza limitata che sono inizialmente collegati tutti assieme nella *lista libera*:



A run-time, quando viene richiesto un blocco di memoria, il primo elemento della lista libera viene rimosso e restituito al processo che ha richiesto la memoria, mentre la testa della lista libera si sposta al prossimo elemento della lista. Vediamo un esempio di heap a dimensione fissa dopo alcune operazioni di allocazione/deallocazione (i blocchi grigi sono in uso):

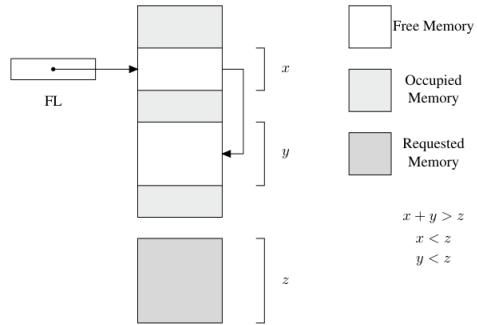


Dimensione variabile

Nel caso volessimo poter allocare array la cui dimensione e' determinata solo a runtime, la soluzione a dimensione fissa non sarebbe adeguata, dato che l'array puo' essere di dimensioni maggiori rispetto ai blocchi fissi e non si possono allocare piu' blocchi perchc la memoria deve essere per forza contigua.

In questi casi e' necessario poter richiedere un blocco di dimensione arbitraria. In un sistema di questo tipo, bisogna prestare attenzione ad usare *operazioni efficienti* e a limitare lo *spreco di memoria* che puo' avvenire in due situazioni:

- **Frammentazione interna:** viene richiesto un blocco di dimensione n ma ne viene restituito uno di dimensione $k > n$, quindi $k - n$ parole si perdono.
- **Frammentazione esterna:** lo spazio totale nella memoria sarebbe abbastanza per soddisfare una richiesta di dimensione n , ma i blocchi liberi sono separati o non contigui.



Esistono due meccanismi diversi per gestire blocchi di dimensione variabile:

- **Unica lista:**

Inizialmente, la lista libera e' composta da un solo blocco che occupa tutta la heap. Quando viene fatta la richiesta per un blocco di dimensione n , le prime n parole dell'heap vengono allocate e l'inizio della lista si sposta di n . Si va avanti cosi' finche' la memoria rimasta dall'inizio della lista libera alla fine della heap non e' abbastanza per soddisfare la richiesta. In questo caso dobbiamo riutilizzare memoria deallocated, che puo' essere fatto in due modi:

- **Uso diretto della lista libera:** si scorre lungo la lista finche' si trova un blocco di dimensioni $k > n$. Se la differenza fra la grandezza del blocco e della memoria effettivamente usata e' maggiore di una tolleranza, allora il blocco viene diviso ed il blocco di dimensione $k - n$ viene reinserito nella lista. Possono essere usate due politiche di ricerca diverse per scegliere quale blocco prendere: *first fit* e *best fit*. Quando un blocco viene deallocated, si guarda se blocchi adiacenti sono anch'essi liberi e in caso affermativo si fondono per formare un blocco piu' grande (*compattazione parziale*).
- **Compattazione della memoria libera:** tutti i blocchi attivi vengono spostati alla fine della heap, molto efficiente ma funziona solo quando possiamo spostare la memoria.

- **Liste libere Multiple:**

Per ridurre il costo operativo di cercare un blocco di dimensione arbitraria, e' possibile utilizzare diverse liste per diverse dimensioni di blocchi. Quando viene richiesto un blocco di dimensione n , si scorrono le liste finche' una che contiene blocchi di dimensioni adeguate non e' vuota. Anche in questo caso e' possibile riudurre la dimensione dei blocchi per ridurre la frammentazione interna, esistono due metodi:

- **Buddy system:** la dimensione dei blocchi aumenta per potenze di 2. Si calcola l'intero minore k tale che $2^k \geq n$ e si controlla se la relativa lista ha blocchi liberi. Altrimenti, si va a cercare nella lista $k + 1$ e si divide il blocco in due blocchi da 2^k (la dimensione che volevamo), uno viene allocato e l'altro viene spostato nella lista corretta. Quando viene deallocated, la meta' cerca il suo compagno nella lista libera e se lo trova si uniscono e tornano nella lista originale.
- **Fibonacci:** funzionamento equivalente ma le dimensioni dei blocchi seguono la sequenza di Fibonacci, che sale piu' lentamente quindi porta a meno frammentazione ma piu' tempo.

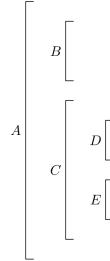
TODO: disegni esplicativi

10.3 Implementazione delle Regole di Scope

Ora che abbiamo capito come viene gestita la memoria di un programma dal compilatore, soprattutto per quanto riguarda i RdA, vediamo come possiamo implementare le regole di scope quando un blocco deve accedere a variabili non-locali.

10.3.1 Scope Statico

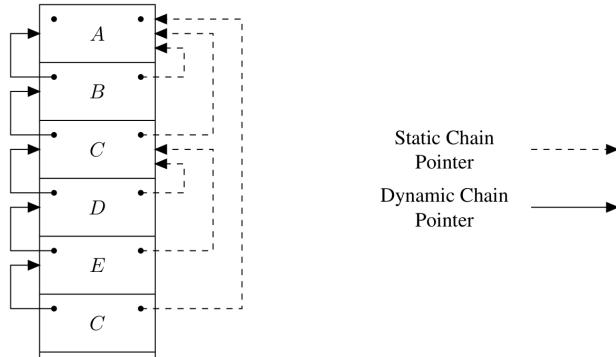
Se vengono implementate le regole di scope statico, allora l'ordine da seguire quando stiamo cercando il valore di una variabile non-locale non e' necessariamente quello dettato dalla catena dinamica, ovvero l'ordine delle chiamate. Infatti, l'RdA corretto e' determinato dalla struttura statica (testuale) del programma, seguendo quindi l'ordine di annidamento dei blocchi. Vediamo un esempio:



Data la struttura soprastante, immaginiamo di chiamare in ordine A, B, C, D, E, C e che rimangano tutte attive:

- Aggiungiamo inizialmente l'RdA di A allo stack di sistema, aggiornando lo SP e l'RdA pointer. Dato che e' il primo sulla pila, non ha nessun link.
- Man mano che aggiungo i RdA di B, C, \dots aggiorno sempre SP e RdA pointer, ma e' anche necessario determinare il link dinamico e statico.

Quindi con tutte le chiamate aperte la situazione e' questa:



Come al solito, il puntatore di catena dinamica punta all'RdA *temporalmente precedente* (quello appena sotto nella pila), mentre il puntatore di catena **statica** indica l'RdA del blocco che *contiene strutturalmente* quello in cui siamo.

Note:

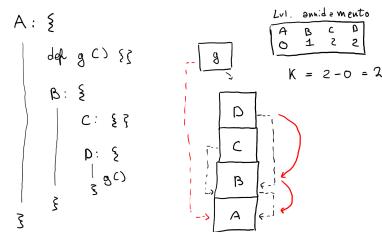
Se un sottoprogramma e' annidato a livello k, allora la catena statica sara' lunga k.

Supponiamo di essere in E e di voler accedere alla variabile x in modo statico dichiarata in A :

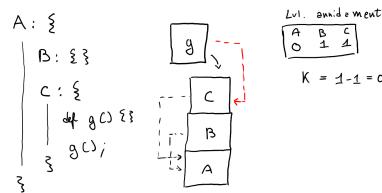
- Seguendo la catena statica, controllo se x e' dichiarata in C .
- Non c'e', quindi continuo a seguire i link statici e arrivo in A , dove trovo il valore cercato.

Il supporto a run-time della catena statica e' compito della sequenza di chiamata, prologo e l'epilogo che abbiamo visto prima per le chiamate. L'approccio piu' comune e' quello dove il chiamante calcola il puntatore a catena statica che passa poi al chiamato, e' abbastanza semplice e puo' essere diviso in due casi:

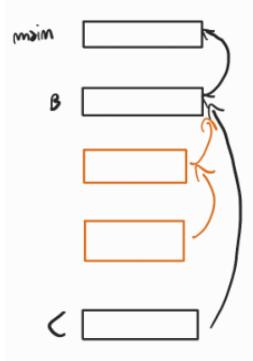
- **Il chiamato e' esterno al chiamante:** secondo le regole di visibilita', questa situazione e' possibile solo se il chiamante e' annidato internamente rispetto al blocco del chiamato. Cio' significa che tale blocco e' ancora attivo sulla pila, come tutti i blocchi annidati fino a quello del chiamante. Dato che sappiamo il livello di annidamento di ciascun blocco (puo' essere calcolato staticamente), basta trovare la differenza fra il livello del chiamante meno il chiamato e fare questo numero di salti sulla catena statica del chiamante (che e' in cima alla stack). In questo modo ci troviamo nel RdA del blocco contenente la definizione del chiamato, e ci basta solo passare l'indirizzo di tale RdA al chiamato che lo imposta come link statico.



- **Il chiamato e' interno al chiamante:** tale situazione puo' succedere solo se il chiamato e' definito nello stesso blocco dove viene chiamato. Siamo quindi in un caso speciale del punto precedente dove la distanza di annidamento e' 0, quindi basta passare come link statico al chiamato il puntatore RdA del record contenente la chiamata (che sara' quello in cima).



Il numero di "salti" da fare per raggiungere il RdA corretto e' calcolabile staticamente dal compiler usando la *tabella dei simboli*, che memorizza anche il livello di annidamento al quale e' stata dichiarata la variabile cercata. Pero' non e' possibile stabilire staticamente la locazione esatta di memoria dove si trovera' tale variabile non-locale, dato che, pur sapendo il numero di "salti", non sappiamo quanti e quali RdA si trovano fra ogni salto a run-time. Per questo motivo ci serve la catena statica.



Il Display

E' un po' uno schifo dover seguire sta catena statica, dato che se siamo a un livello k di annidamento, dobbiamo per forza eseguire k accessi a memoria per arrivare finalmente al frame giusto. Anche se nella pratica generalmente non e' un problema, possiamo ottimizzarlo quindi lo facciamo.

TODO: Fai Display, CRT e A-List

Chapter 11

Strutturare il controllo - espressioni, comandi, iterazione, ricorsione

11.1 Espressioni

Come al solito prima fornisco la definizione

Definition 11.1.1: Espressione

si definisce **espressione** un'unità sintattica la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita

11.1.1 sintassi delle espressioni

In generale, un'espressione è composta da una singola entità (costante, variabile, ecc...) o da un operatore (+, cons, ecc...) applicato ad una serie di argomenti anch'essi espressioni. Si tenga presente che la sintassi di un'espressione può essere descritta da una grammatica libera e può essere rappresentata da un albero di derivazione che ne esprime anche la semantica. Con Gabrielli (e non Morbidelli) verranno utilizzate le notazioni lineari per scrivere le espressioni quindi non dovrò stare qui a disegnare alberi o automi su latex grazie a dio. Queste notazioni differiscono tra loro per il modo in cui rappresentano l'applicazione (quindi la semantica) di un operatore ai suoi operandi. Possiamo distinguere tre tipi principali di notazione:

- **infissa**: il simbolo dell'operatore è posto in mezzo a due espressioni, es. $a+b$.

Sintassi ambigua, e richiede l'utilizzo di parentesi e regole di precedenza per la disambiguazione. Quasi tutti i linguaggi di programmazione insitono sulla notazione infissa, ma spesso questa è solo uno *zucchero sintattico* per rendere il codice più digeribile a chi lo legge. In C++ l'espressione $a+b$ è un'abbreviazione di $a.\text{operator}+(b)$

- **prefissa (polacca¹)**: il simbolo dell'operatore è posto prima a due espressioni, es. $+ab$

Intuitivamente è la sintassi delle funzioni ($f(a,b)$ o $+(a,b)$) e non richiede parentesi o regole di precedenza in quanto l'arietà di ogni operatore è conosciuta. Inoltre non c'è ambiguità su quale operatore applicare ad ogni operando perché è sempre quello che precede immediatamente gli operandi. es. $* + a b + c d$

- **postfissa (polacca inversa)**: l'operatore è posto dopo le espressioni, es. $ab+$

Un vantaggio delle due notazioni polacche rispetto a quella infix è che possono essere utilizzate in modo uniforme per rappresentare operatori con qualsiasi numero di operandi. Nella notazione infix, invece, rappresentare operatori con più di due operandi significa dover introdurre operatori ausiliari. Un secondo vantaggio, come abbiamo già detto, è che possiamo omettere completamente le parentesi. Un ultimo vantaggio della notazione polacca è che rende particolarmente semplice la valutazione di un'espressione, che adesso vediamo

¹In onore di W. Łukasiewicz, il brodo che l'ha utilizzato estensivamente

11.1.2 Semantica delle espressioni

Adesso verrà esposta la semantica delle tre notazioni lineari sintattiche prima presentate

notazione infissa

Quando utilizziamo la notazione infissa paghiamo per la semplicità di lettura con una maggiore complicazione nel meccanismo di valutazione. Ecco qui presentati i vari problemucci:

- **Precedenza** fra gli operatori:

Se le parentesi non sono utilizzate sistematicamente (o altri tipi di *zucchero sintattico*) è necessario specificare la precedenza di ogni operatore. I linguaggi di programmazione, quindi, impiegano delle *regole di precedenza* per definire una gerarchia tra l'ordine di valutazione e i vari operatori

Esempio:

```
1      a+ b * c ** d ** e / f \|??
2      if A < B and C < D then \|??
```

Che fare prima?

- **Associatività**:

Un altro problema che nella valutazione di espressioni che concerne gli operatori, se infatti noi scriviamo 15-5-3 potremmo intendere sia $(15-5)-3$ o anche $15-(5-3)$. In questo caso la normale convenzione matematica impone che la prima espressione sia quella corretta e che l'operatore - associ da **destra a sinistra**. Non è sempre ovvio, in APL $15-5-3$ è interpretato come $15-(5-3)!!!! CAZZO$

Si può quindi concludere che valutazione di un'espressione infissa non è semplice, andiamo dai polacchi vah, che è mejo

notazione postfissa

La valutazione è molto più semplice di quella infissa:

- non servono regole di precedenza
- non servono regole di associatività
- non servono le parentesi

Infatti questa notazione prevede una semplice strategia di valutazione che percorre l'espressione da sinistra a destra usando una pila per memorizzare i vari componenti. Si può quindi desceivere l'algoritmo di valutazione in questo modo:

1. Leggi il prossimo simbolo nell'espressione e pushalo nella pila
2. Se il simbolo appena letto è un operatore applicalo agli operandi immediatamente prima nello stack, memorizza il risultato in R , e fai il pop degli operatori e operandi dalla pila e pusha il valore in R
3. Se la sequenza da leggere non è vuota torna a (1)
4. Se il simbolo letto un operando torna a (1).

Valutazione prefissa

Un po' più complessa di quella postfissa, qui mostrato l'algoritmo: L'algoritmo di valutazione è descritto dai seguenti passaggi, dove utilizziamo uno stack ordinario (con le operazioni push e pop) e un contatore C per memorizzare il numero di operandi richiesti dall'ultimo operatore letto:

1. Leggi un simbolo dall'espressione e inseriscilo nello stack;
2. Se il simbolo appena letto è un operatore, inizializza il contatore C con il numero di argomenti dell'operatore e vai al passaggio 1;

3. Se il simbolo appena letto è un operando, decrementa C ;
4. Se $C \neq 0$, vai a 1;
5. Se $C = 0$, esegui le seguenti operazioni:
 - (a) Applica l'ultimo operatore memorizzato nello stack agli operandi appena inseriti nello stack, memorizzando i risultati in un registro R , elimina operatore e operandi dallo stack e memorizza il valore di R nello stack;
 - (b) Se non ci sono simboli di operatore nello stack, vai a 6;
 - (c) Inizializza il contatore C a $n - m$, dove n è il numero di argomenti dell'operatore in cima allo stack e m è il numero di operandi presenti nello stack sopra questo operatore;
 - (d) Vai a 4;
6. Se la sequenza rimanente da leggere non è vuota, vai a 1;

Chapter 12

Astrazione sul controllo: sottoprogrammi ed eccezioni

A cosa servono gli array? Il linguaggio assembly non ce li ha, ma riesce comunque a svolgere tutto quello che possono fare. Sono quindi un'*astrazione* che rende la vita dei programmatore piu' facile. Questo e' l'obiettivo dei linguaggi di programmazione di alto livello, che astrae sul controllo e sui dati.

L'astrazione, quindi, consiste nell'identificare proprietà importanti di cosa si vuole descrivere, concentrarsi su quelle e ignorare le altre. Che cosa ignorare e cosa no dipende dallo scopo del progetto

I linguaggi di programmazione (oltre al fatto che essi stessi sono astrazioni più sono ad alto livello) forniscono strumenti per implementare astrazioni e modelli astratti, questi sono chiamati, appunto, **astrazioni sul controllo e sui dati**. Adesso diamo una definizione

Definition 12.0.1: astrazione sul controllo

Si definisce **astrazione del controllo** una serie di istruzioni per svolgere un compito a prescindere dal contesto in cui questo opera, specificandone modalità e fine

Queste astrazioni sono ad esempio funzioni o blocchi. Tra le proprietà più importanti di questi costrutti è che ogni componente fornisce servizi al suo ambiente, nascondendo i dettagli interni necessari a produrlo

Un meccanismo fondamentale con cui si può definire come ogni sottoprogramma (funzione) comunica con il programma principale `main` è attraverso i parametri o un'ambiente globale (da preferire i primi perché quest'ultimo rende nulla l'astrazione)

12.1 parametri

Introduciamo due definizioni terminologiche:

Definition 12.1.1: Parametro formale

Un parametro formale è una variabile utilizzata nella definizione di un sottoprogramma, che viene sostituita dal valore o riferimento del parametro attuale quando il sottoprogramma viene chiamato.

Pertanto si trova nella dichiarazione/definizione: `int f (int n) return n+1;`

Definition 12.1.2: Parametro attuale

Un parametro attuale è il valore o riferimento passato a un sottoprogramma quando viene chiamato. Questo valore o riferimento sostituisce il parametro formale nella definizione del sottoprogramma.

Ad esempio, nell'espressione `f(5)`, il valore 5 è il parametro attuale che sostituisce il parametro formale `n` nella definizione del sottoprogramma `f`.

Definition 12.1.3: Pragamtica

La pragamtica rappresenta il flusso di informazioni tra chiamante e chiamato

Rappresentiamo il chiamate con `main` e il chiamato `proc`, questi sono i possibili di informazioni tra i comunicanti:

- `main→proc`, es. `x=f(y+1)`
- `proc→main`, es. `procedure Uno (var y:integer); begin y:=1 end;`
- `proc↔main`, es. `procedure Succ (var y:integer); begin y:=y+1 end;`

Example 12.1.1 (Una funzione comunica col chiamante)

Valore restituito

```
int f() return 1;
```

12.1.1 Modalità di passaggio dei parametri

Vi sono due modi principali per passare i parametri ai sottoprogrammi:

- **Passaggio per valore:** In questo caso, il valore del parametro attuale viene copiato nel parametro formale del sottoprogramma. Le modifiche apportate al parametro formale all'interno del sottoprogramma non influenzano il parametro attuale.

La pragamtica è `main → proc`

Example 12.1.2 (Esempio di passaggio per valore)

```
void increment(int n) { n = n + 1; }
int main() { int x = 5; increment(x); }
```

In questo esempio, il valore di `x` non cambia dopo la chiamata a `increment`.

- **Passaggio per riferimento:** In questo caso, il parametro formale del sottoprogramma diventa un riferimento al parametro attuale. Le modifiche apportate al parametro formale all'interno del sottoprogramma influenzano direttamente il parametro attuale

La pragamtica è `main ↔ proc`

Example 12.1.3 (Esempio di passaggio per riferimento)

```
void increment(int &n) { n = n + 1; }
int main() { int x = 5; increment(x); }
```

In questo esempio, il valore di `x` cambia a 6 dopo la chiamata a `increment`.

Passaggio per valore

Si prenda come esempio il seguente codice

```
1 void foo (int x) { x = x+1; }
2 {...}
3 y = 1;
4 foo(y+1);
```

In questo caso il parametro formale `x`, mentre quello attuale è `y`. Alla chiamata di `foo`, viene valutato `y+1` e assegnato al formalex. Ovviamente non vi sarà nessun legame tra `x` e `y` e alla fine della computazione `x` verrà deallocata e distrutta.

Tuttavia si ha che nel record di attivazione di `foo`, appena viene eseguita la funzione, viene creata una copia di `y` per `x`. La cosa è influente dato che `x` è un intero, ma potrebbe essere estremamente costoso per parametri attuali di grandi dimensioni, si pensi ad un array di 1000 elementi

Passaggio per riferimento

Si prenda come esempio il seguente codice

```
1 void foo (reference int x){ x = x+1; }
2 ...
3 y = 1;
4 foo(y);
```

In questo caso, nella funzione viene passato un riferimento, ovvero un puntatore a un intero. Pertanto, qualsiasi modifica apportata al parametro formale `x` all'interno della funzione `foo` influenzerà direttamente il parametro attuale `y`. Tra la variabile `x` e `y` si verifica il cosiddetto *aliasing* alla stessa cella

Questo metodo è efficiente in termini di memoria, poiché non viene creata una copia del parametro attuale. Tuttavia, bisogna fare attenzione alle modifiche non intenzionali ai parametri attuali, poiché queste possono portare a effetti collaterali indesiderati

Passaggio per costante

Il passaggio per costante è una variante del passaggio per valore, tuttavia viene garantito che nella procedura non è permessa la modifica del parametro formale

```
1 void foo (final int x){
2     // qui x non puo' essere modificato
3 }
```

Se l'oggetto passato è di grandi dimensioni, il compiler puo' evitare di fare la copia usando il passaggio a riferimento, mantenendo sempre la semantica del passaggio per valore.

Passaggio per Risultato

```
1 void foo (result int x) {x = 8;}
2 ...
3 y = 1;
4 foo(y);
```

Il passaggio per risultato è la tecnica *complementare* al passaggio per valore. In questo caso, il parametro formale viene utilizzato per restituire un valore al termine dell'esecuzione del sottoprogramma

Al momento della chiamata e della computazione (all'interno del corpo di `foo`) non vi sarà alcun legame tra `x` e `y`, ma al ritorno di `foo` verrà fatta una copia di del formale sull'attuale `y=x`. La pragamtica sarà dunque: `proc → main`

Passaggio per valore risultato

È un mix tra il passaggio per risultato e per valore, pertanto verrà fatta una copia dall'attuale a formale all'inizio e una copia dal formale all'attuale alla fine e, dato che non vi è alcun riferimento, non vi è alcun legame tra il formale e l'attuale durante la computazione dei dati nel corpo della funzione

Pragmaticamente: `main↔proc`

Si noti che il passaggio valore-risultato ≠ riferimento, ad esempio in questo codice:

```
1 void foo (int x, int y, int z) {
2     x = 2;
3     y = 2;
4     x = 4;
5     if (x == y) z = 1;
6 }
7 int a = 3;
8 int b = 0;
9 foo(a,a,b);
```

Per valore risultato il valore delle variabili istruzione per istruzione è

Value-result					
z	0	0	0	0	
y	3	2	2	2	
x	3	2	4	4	
b	0	0	0	0	0
a	3	3	3	3	2 o 4

Mentre per riferimento

Riferimento					
z b	0	0	0	1	1
y x a	3	2	4	4	4

Morale

Il passaggio per valore ha come pro:

- Semantica semplice: il corpo non ha necessità di conoscere come la procedura verrà chiamata
- trasparenza referenziale: ovvero la proprietà di un'espressione che garantisce che verrà restituito sempre lo stesso risultato ogni qualvolto il gli verrà fornito lo stesso imput indipendentemente dal contesto
- implementazione abbastanza semplice

e come contro un costo potenzialmente alto per la copia del parametro attuale.

Invece il passaggio per riferimento ha come pro:

- implementazione semplice
- efficienza nel passaggio da parametro attuale a formale

E come contro una semantica complessa a causa dell'aliasing

Passaggio per nome

Il passaggio per nome è una modalità di passaggio parametri introdotta da Algol negli anni 60 che vede la chiamata come una *macro espansione* ovvero un meccanismo dove la semantica di una chiamata di funzione è definita in modo **prescrittivo** e consiste nell'esecuzione del corpo come se fosse testualmente presente nel chiamante, anche la gestione dei parametri segue lo stesso principio, ovvero il corpo della procedura viene eseguito dopo che i parametri attuali sono stati sostituiti testualmente al posto dei parametri formali, quindi quest'ultimi vengono valutati dopo questo passaggio.

Più in particolare il passaggio per nome segue la cosiddetta *regola della copia*: una chiamata alla procedura P è la stessa cosa che eseguire il corpo di P dopo aver sostituito i parametri attuali al posto dei parametri formali. Si noti il seguente codice:

```

1 int y;
2 void foo (name int x) {x= x + 1; }
3 ...
4 y = 1;
5 foo(y);

```

In questo caso dopo la chiamata `foo(y)` viene inserito il corpo della funzione nel `main` cambiando il parametro formale con quello attuale, quindi `y=y+1`. Tuttavia questa regola per come è definita nasconde una criticità, infatti se nel corpo della funzione è presente un nome `y` potrebbe provocare un conflitto, si noti questo codice:

```

1 int y;
2 void fie (name int x){
3     int y;
4     x = x + 1; y = 0;
5 }
6 ...
7 y = 1;
8 fie(y);

```

Dopo la chiamata verrà eseguita la macro espansione sostituendo il parametro formale con quello attuale:

```

1 int y;
2 y = y+1;
3 y =0;

```

Tuttavia all'interno di `fie` vi erano due variabili diverse che abbiamo reso uguali perché il nome di `y` all'interno di `fie` è uguale al parametro attuale, provocando così un conflitto se si esegue il programma con scope statico.

Per ovviare a questo problema viene passato, ai vari parametri, una coppia `<exp, amb>` detta *closure*, dove:

- **exp**: è il parametro attuale (testo, non valutato)
- **amb**: è l'ambiente di valutazione (in scoping statico)

Si prenda come esempio il seguente codice:

```

1 int y;
2 void fie (int x ){
3     int y;
4     x = x + 1; y = 0;
5 }
6 ...
7 y = 1;
8 fie(y);

```

Quando viene eseguita la macro-expansione si avrà che `x→<y, int y;>` dove l'ambiente non è altro che la dichiarazione della variabile in riga 1, gg. Si ha, quindi, che *ogni volta che il formale è usato, exp è valutata in amb*

La pragmatica in questo è `main↔proc`, inoltre si tratta di una pratica costosa infatti

- **vi è il passaggio di una struttura complessa** (soprattutto l'ambiente)
- **è prevista una rivalutazione ripetuta del parametro**, infatti differenza del passaggio per valore, in cui il parametro viene valutato una sola volta prima di entrare nella funzione, nel passaggio per nome il parametro viene rivalutato ogni volta che viene usato nel corpo della funzione

Per implementare il passaggio per nome, la coppia closure è formata, dal lato pratico, da:

- un puntatore al testo di `exp`
- un puntatore di catena statica (sullo stack) al record di attivazione del blocco di chiamata

12.2 Funzioni di ordine superiore

Alcuni linguaggi di programmazione consentono di passare funzioni come argomenti di altre funzioni o restituire funzioni come risultato di una funzione:

Definition 12.2.1: Funzione di ordine superiore

Funzione che accetta una funzione come parametro o che restituisce una funzione come risultato.

è quindi lecito chiedersi come viene gestito lo scope in questi casi

12.2.1 Funzione come argomento

Vediamo un esempio:

```
1 A :{  
2     int x = 1;  
3     int f (int y){  
4         return x+y; // Quale "x"?  
5     }  
6     void g(int h(int b)){  
7         int x = 2;  
8         return h(3) + x;  
9     }  
10    ...  
11 B :{  
12     int x = 4;  
13     int z = g(f)  
14 }  
15 }
```

Notare che f utilizza una variabile non-locale x, che e' stata dichiarata piu' volte: nel blocco A, B e nella funzione g. Dobbiamo quindi capire in quale ambiente risolverla, proviamo ad applicare le regole di scope che abbiamo visto fin'ora:

- se lo scope è statico: si usa x definita nel punto in cui f è stata dichiarata (x=1)
- Se lo scope è dinamico: si potrebbe usare sia la x della chiamata x=4, sia la x di g (x=2)

L'incertezza su quale ambiente non-locale scegliere e' dovuta dal fatto che e' possibile immaginare due implementazioni diverse della chiamata di una funzione passata come parametro (f) tramite il parametro formale (h):

- La chiamata della funzione passata avviene nel blocco della funzione di ordine superiore (la chiamata h(3) chiama la funzione f all'interno di g)
- La chiamata della funzione passata avviene nel blocco dove viene creato il legame fra parametro formale e attuale (la chiamata h(3) chiama la funzione f all'interno del blocco B)

Quindi notiamo che le regole di scope non sono abbastanza specifiche per determinare l'ambiente da utilizzare, dato che non e' chiaro da quale blocco si vuole che avvenga la chiamata di f (in questo caso lo scope statico ha solo un'opzione, ma vedremo che non e' sempre così). Ci servono quindi delle ulteriori regole specifiche alle funzioni parametro, dette regole di *binding*:

Definition 12.2.2: Binding

Data una funzione di ordine superiore g che ammette una funzione parametro f, in base al blocco dal quale vogliamo che f venga chiamato quando chiamiamo il parametro formale associato h, abbiamo due casi:

- *Deep binding*, se la chiamata avviene nel blocco attivo nel momento in cui e' stato instaurato il legame fra parametro formale e attuale.
- *Shallow binding*, se la chiamata avviene nel blocco attivo nel momento in cui viene chiamato il parametro formale.

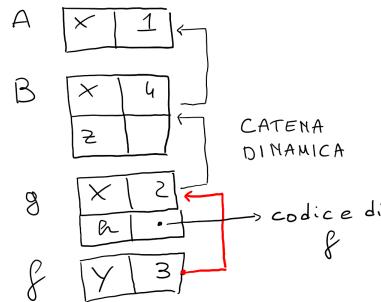
Quindi, applicando cio' all'esempio di prima:

- Usando deep binding, si simula la chiamata di f dal blocco B, quindi con scope statico $x = 1$ e con scope dinamico $x = 4$.
- Usando shallow binding, f viene chiamata dalla funzione g, quindi se si usa scope statico abbiamo comunque $x = 1$, ma con scope dinamico $x = 2$.

Vediamo in specifico l'implementazione delle due regole di binding applicate in linguaggi con scope dinamico e statico:

Binding con scope dinamico

Consideriamo prima il caso di un linguaggio con scope dinamico e shallow binding. Per definizione di shallow binding, vogliamo chiamare la funzione parametro f all'interno del corpo della funzione g. Per le regole dello scope dinamico, dobbiamo risolvere le variabili non-locali nell'ambiente immediatamente precedente alla chiamata, che quindi corrisponde all'ultimo record di attivazione sulla pila (quello di g). In questo caso quindi basta la normale implementazione di scope dinamico.

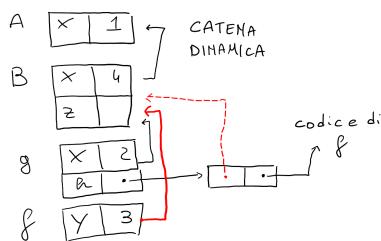


Note:

Ovviamente puo' anche essere implementato lo scope dinamico con A-list o CRT.

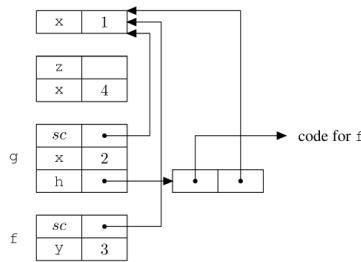
Nel caso di deep binding con scope dinamico, ci serve l'ambiente del blocco attivo al momento del legame fra la funzione e il parametro formale. Ma in questo caso, l'RdA di questo blocco (che nell'esempio e' B) non sara' direttamente sotto a quello della funzione f quando viene chiamata, dato che ci sara' sicuramente almeno il frame della funzione di ordine superiore. Dobbiamo quindi passare un puntatore a questo ambiente insieme alla funzione. Tale struttura viene chiamata *closure*, ovvero l'accoppiata $\langle f, \text{amb} \rangle$, dove:

- f è il puntatore alla funzione che vogliamo passare
- amb è il puntatore all'ambiente non-locale in cui è da valutare il corpo della funzione f



Binding con scope statico

Anche nel caso generale dello scope statico e' necessario usare una closure per poter passare alla funzione di ordine superiore anche il puntatore di catena statica, che puo' essere calcolata al momento del legame fra funzione e parametro formale, come abbiamo visto studiando l'implementazione dello scope statico (10.3.1).

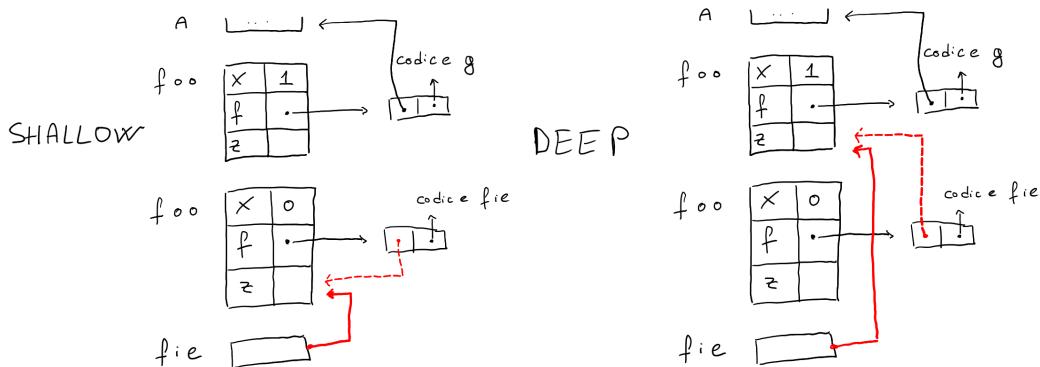


Puo' sembrare che le regole di scope statico siano rigide e che non servano le regole di binding per distinguere vari casi. In effetti per la maggior parte delle situazioni e' cosi, ma ci possono essere discussioni per quanto riguarda un caso specifico con funzioni ricorsive. Vediamo un esempio:

```

1  A : {
2      void foo(int f(), int x){
3          int fie(){
4              return x;
5          }
6          int z;
7          if (x==0) z = f();
8          else foo(fie, 0);
9      }
10     int g(){
11         return 1;
12     }
13     foo(g,1);
14 }
```

Il problema in questo caso e' che la funzione fie e' definita in due istanze diverse della funzione foo, alle quali sono associate due ambienti diversi: nel primo $x = 1$, nel secondo $x = 0$. Dato che possono essere considerate valide entrambi per le regole di scope, bisogna utilizzare le regole di binding. Usando shallow binding, consideriamo la definizione piu' vicina temporalmente; usando deep binding, scegliamo l'ambiente attivo al momento della prima definizione di fie.



Chapter 13

Esercitazioni

13.1 Scope

13.1.1 Esercizio 1

Testo

```
1      {
2          int x = 2;
3          int func ( int y){
4              x = x+y;
5              write(x);
6          }
7          {
8              int x = 5;
9              func(x);
10             write(x);
11         }
12         write(x);
13     }
```

- Si descriva il comportamento del programma assumendo uno scope statico.
- Si descriva il comportamento del programma assumendo uno scope dinamico.

Soluzione

- **Scope statico:** il programma stampa 7, 5, 7
- **Scope dinamico:** il programma stampa 10, 10, 2

13.1.2 Esercizio 2

Testo

```
1      {
2          int x = 0;
3          int next () {
4              x = x+1;
5              write(x);
6          }
7          int exec(){
8              int x = 3;
9              next();
10             write(x);
```

```

11     }
12     exec();
13     write(x);
14 }
```

- Si descriva il comportamento del programma assumendo uno scope statico
- Si descriva il comportamento del programma assumendo uno scope dinamico

Soluzione

- **Scope statico:** il programma stampa 1, 3, 1
- **Scope dinamico:** il programma stampa 4 infatti `next` prenderà l'istanza di `x` dell'ultimo ambiente non disattivato, ovvero `exec`, modifco poi il valore di `x` nell'ambiente `exec`, 4 dato che l'ambiente di `exec` è stato modificato, 0 dato che `exec` è stato disattivato si ha che `x = 0`

13.1.3 Esercizio 3

Testo

```

1 {
2     int x = 0;
3     void pippo(value int y, rif int z){
4         z= x+y+z;
5     }
6     {
7         int x = 1;
8         int y = 100;
9         int z = 30;
10        pippo(x++,x); // ricordarsi che il x++ prima passa il valore di x nudo e
11            crudo poi come side-effect modifica x staticamente in questo caso,
12            pertanto il secondo parametro sara' 2: pippo(1, 2) -> 3 = 0+ 1 +2
13        pippo(x++, x); // pippo(3, 4) -> 7 = 0 + 3 + 4
14        write(x); // 7
15    }
16    write(x); // 0 perche' il blocco e' finito, terminato e riprende la variabile
17        x dichiarata all'inizio
18 }
```

Si descriva il comportamento del programma assumendo uno scope statico

Soluzione

Il programma stampa 7, 0

13.1.4 Esercizio 4

Testo

```

1 {
2     void pippo(value int y, value int z){
3         x=y+z;
4     }
5     int x = 100;
6     pippo(x++, x); // pippo(100, 101) -> 201, quindi x = 201
7     pippo(x++, x); // pippo(201, 202) -> 403, quindi x = 403
8     write(x); // 403
9 }
```

Si descriva il comportamento del programma assumendo uno scope dinamico

Soluzione

Il programma stampa 403

13.1.5 Esercizio 5

Testo

```
1  {
2      int f(value int k){ //1. 2, 1. 1
3          int g (value int n){
4              return n+y //
5          };
6          int x=10;
7          int y=10;
8          if k==1 return g(x) + g(y); // 40
9          else {
10             int x = 30;
11             int y= 30;
12             return f(k-1);
13         }
14     }
15     int x =50;
16     int y=50;
17     x= f(2); // x = 40
18     write(x); // 40
19 }
```

si risolva utilizzando uno scope dinamico

Soluzione

Il programma stampa 40

13.1.6 Esercizio 6

Testo

```
1  {
2      int x =10;
3      int v =5;
4      void B(){
5          write(x);
6      }
7      void A(z){
8          int x = z * v;
9          B();
10     }
11     A(??) // si ha che qualsiasi valore di al posto di ?? si scrivera' 10
12 }
```

Fornire una chiamata alla funzione A di modo che il programma, usando scope statico stami il valore 10

```
1  {
2      int x =10;
3      int v =5;
4      void B(){
5          write(x);
6      }
7      void A(z){
8          int x = z * v;
9          B();
10 }
```

```

10      }
11      A(??) // si scrive 2 cazzo
12  }

```

fornire una chiamata per stampare il valore 10, ma usando uno scope dinamico

Soluzione

Per la prima parte si stamperà sempre 10, per la seconda parte si passa il parametro 2

13.1.7 Esercizio 7

Testo

```

1 {
2     int x =1;
3     int y=2;
4     void A(){
5         int x = 2;
6         int k = 3;
7     }
8     void B(){
9         int x = 5; // 2
10    A();
11    // ** 1**
12    x=2;
13    C();
14 }
15 void C(){
16     int z =5;
17     // ** 2 **
18 }
B();
}

```

Si descriva qual è il contenuto delle variabili attive al punto ****1**** e ****2****, utilizzando scope statico e dinamico

Soluzione

- Scope statico:

- Punto 1: x = 5, y = 2
- Punto 2: x = 1, y = 2, z = 5

- Scope dinamico:

- Punto 1: x = 5: nel momento in cui si attiva A() lo shadowing impone che x in memoria è 2, ma appena A termina si "disattiva" e si riprende x=5 definito nel punto B, y = 2
- Punto 2: x = 2, y = 2, z = 5

13.2 sottoprogrammi

13.2.1 Es. 1

Testo

Considerare un linguaggio che ammette il passaggio per nome. Dire cosa stampa il codice:

```

1     int k = 2;
2     int A[5];
3     A = {1, 2, 4, 7, 3};
4     void swap(int name x, int name y) {
5         int temp = x; // temp = 2
6         x = y; // x = A[k] -> x = A[2] -> x = 4
7         y = temp; // A[4] = 2
8         write(A); // A = {1, 2, 4, 7, 4};
9     }
10    swap(k, A[k]);

```

svolgimento

è probabile che scriverà $A = 1, 2, 4, 7, 4$;

13.2.2 Es. 2

Testo

L'esecuzione del seguente frammento di codice su una certa implementazione risulta nella stampa dei valori 4 e 10.

```

1     int W[10];
2     int x = 4;
3     for (int i=0; i<10; i++) W[i]=i; // quindi, [0,1,2,3,4,5,6,7,8,9]
4
5     void foo(int x; int y){
6         x = x+1;
7         y=10;
8     }
9
10    foo (x, W[x])
11    write (W[4]) // 4
12    write (W[5]) // 10

```

Si fornisca una possibile spiegazione.

svolgimento

Una possibile soluzione è che entrambi i parametri siano passati per nome, infatti se andiamo a valutare i parametri dopo la macro espansione tutto torna, provare per credere

13.2.3 Es. 3

Testo

La definizione di un certo linguaggio di programmazione specifica che la valutazione procede da sinistra a destra, ma nel valutare un'espressione complessa eventuali sottoespressioni che vi compaiono più di una volta devono essere valutate una sola volta, usando il valore così calcolato anche per le altre occorrenze della stessa sottoespressione. Un assegnamento è una particolare forma di espressione complessa e il passaggio dei parametri avviene per nome. Si consideri il seguente frammento di codice:

```

1     int x = 1;
2     int A[5];
3     for (int i=0; i<5; i++) A[i] = i; // A = {0,1,2,3,4}
4
5     void f(int name x, int name y){
6         x= y + x + x;
7     }
8
9     f(A[x++],x) // dopo la chiamata avremo:

```

```

11     // A[2] = A[2] + 1 +1 (dato che avremo modificato x nello scope del main)
12     //Quindi A[2] = 1 + 2 +2

```

Qual'è lo stato del vettore A dopo la chiamata della funzione f?

Soluzione

A = 0,5,2,3,4

13.2.4 Es. 4

Testo

È dato il seguente frammento di codice in uno pseudolinguaggio con goto, scope dinamico e blocchi annidati etichettati (indicati con A ...):

```

1 A: {
2     int x = 5;
3     int y = 4;
4     goto B;
5     B: {
6         int x = 4;
7         int z = 3;
8         goto C;
9     }
10    C: {int x = 3;
11        D: {int x = 2;}
12        goto E;
13    }
14    E: {
15        int x = 1; // (***)
16    }
17 }
18 }

```

Lo scope dinamico è gestito mediante CRT. Si illustri graficamente la situazione della CRT nel momento in cui l'esecuzione raggiunge il punto segnato con il commento (***)

13.2.5 Es. 5

Testo

Si consideri il seguente schema di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio per riferimento

```

1 {
2     int x = 0;
3     int A(reference int y) {
4         int x =2;
5         y=y+1;
6         return B(y)+x;
7     }
8     int B(reference int y){
9         int C(reference int y){
10            int x = 3;
11            return A(y)+x+y;
12        }
13        if (y==1) return C(x)+y;
14        else return x+y;
15    }
16    write (A(x));
17 }

```

Supponiamo che lo scope statico sia implementato mediante display. Si dia graficamente la situazione del display e della pila dei record di attivazione al momento in cui il controllo entra per la seconda volta nella funzione A. Per ogni record di attivazione si dia solo il valore del campo destinato a salvare il valore precedente del display

zio pera