Linguaggi di programmazione
Appunti

GioLaPalma

Contents

Chapter 1

Introduzione ai linguaggi normali: grammatiche

1.1 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
- **Pregmatica**: in quale modo frasi corrette e sensate sono usate
- **Implementazione**: come eseguire una frase corretta rispettandone la semantica

Partiamo con il descrivere questi elementi

1.1.1 Sintassi

Definition 1.1.1: 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 cotruire 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"

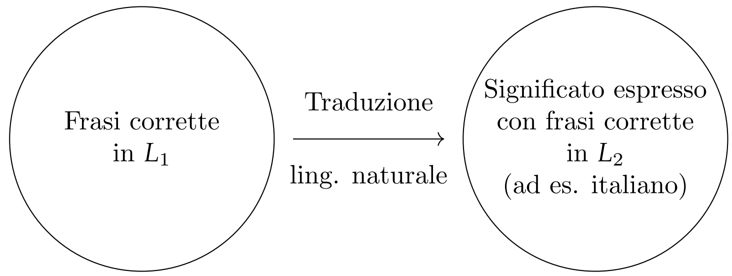
1.1.2 Semantica

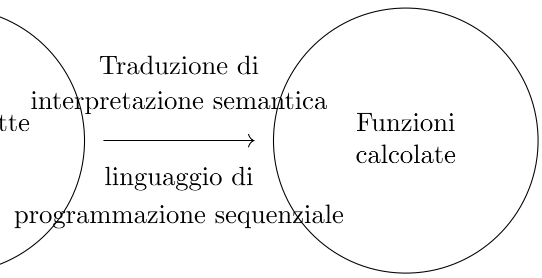
Definition 1.1.2: Semantica

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





1.1.3 Pragmatica

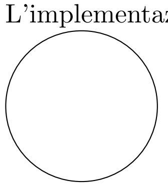
Definition 1.1.3: Pregmatica

La **pragramtica** 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

1.1.4 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!

1.2 Lessico e frasi di un linguaggio

Innanzi tutto diamo tre definizioni

Definition 1.2.1: alfabeto

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

Definire l'alfabeto ci porta alla definizione di lessico:

Definition 1.2.2: 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 1.2.3: 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 1.2.4: linguaggio formale

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

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

e

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

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

Si osservi che A^* è un insieme infinito contabile dato un ordinamento ¡ sui simboli di A, possiamo elencare tutt 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. $a, b, c, \ldots$
3. poi elenco le parole in A^2 secondo ¡

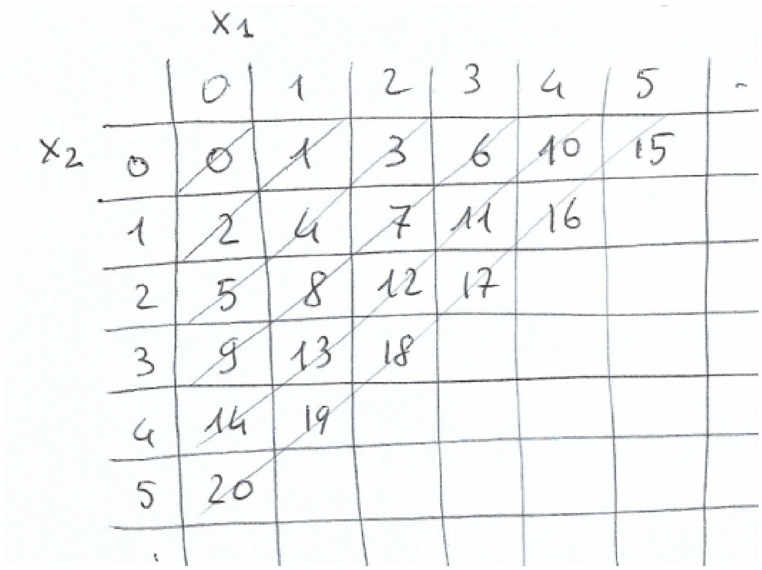
   Es. $aa, ab, ac, \ldots, ba, bb, bc, \ldots$
4. così via

Anche se l'alfabeto A fosse infinito (quindi $A = \{a_0, a_1, \ldots\}$) 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 cosìdetta **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"



è pertanto vero, quindi, che

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

è biunivoca

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

$$f^k(x_1, x_2, \ldots, x_k) = \text{ if } (k = 2) \text{ then } f^2(x_1, x_2) \quad \text{else } f^2(x_1, f^{k-1}(x_2, \ldots, x_k))$$

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

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

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

1.3 Notazioni e definizioni ausiliarie

Vengono riportate qui alcune definizioni/notazioni

1.3.1 Lunghezza

Definition 1.3.1: lunghezza

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

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

Es: $|abc| = 2$

1.3.2 Concatenazione

Definition 1.3.2: Concatenazione

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

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

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

Questi sono le leggi della concatenazione

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

1.3.3 Sottostringa

Definition 1.3.3: Sottostringa

La stringa v si dive **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

1.3.4 Suffisso

Definition 1.3.4: suffisso

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

1.3.5 Prefisso

Definition 1.3.5: prefisso

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

1.3.6 Potenza n-esima

Definition 1.3.6: potenza n-esima

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

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

1.3.7 Linguaggio

Definition 1.3.7: linguaggio

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

Vengono riportati qui alcuni esempi

Example 1.3.1

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

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

1.4 Operazione sui linguaggi

Qui sono elencati le varie operazioni

1.4.1 Complemento

Definition 1.4.1: complemento

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

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

1.4.2 Unione e intersezione

Definition 1.4.2: unione e intersezione

Ovvi:

$$L_1 \cup L_2 \{w | w \in L_1 \vee w \in L_2\}$$
$$L_1 \cap L_2 \{w | w \in L_1 \wedge w \in L_2\}$$

1.4.3 Concatenazione

Definition 1.4.3: concatenazione

È definita **concatenazione** tale operazione:

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

Ecco alcuni esempi:

Example 1.4.1

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

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

1.4.4 Potenza di un linguaggio

Definition 1.4.4: Potenza di un linguaggio

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

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

1.4.5 Stella di kleene

Definition 1.4.5: stella di kleene

Si dice **stella di kleene**:

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

Oppure

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

Quest'ultima detta **chiusura positiva**

1.5 Definire finitamente un linguaggio

1.5.1 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"

- $A = \{a, b\} \quad L = \{\epsilon, a, b, aa, bb, aba, bab, dots\}$. 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"

1.5.2 Esempio 2

espressioni aritmetiche formate a partire dalle variabili a e b con gli operatori $\times, +$ e le parantesi $(,)$
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:

$$E \rightarrow a \mid b \mid E \times E \mid E + E \mid (E)$$

1.6 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 (in rapporto tra facilità di analisi ed espressività)

Definition 1.6.1: grammatiche libere

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:

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

Alcuni esempi:

Example 1.6.1

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

Con

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

1.7 Derivazioni

Definition 1.7.1: 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:

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

Definition 1.7.2: derivazione

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

$$v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w$$

Cioè:

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

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

1.8 Linguaggio Generato

Definition 1.8.1: Linguaggio Generato

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

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

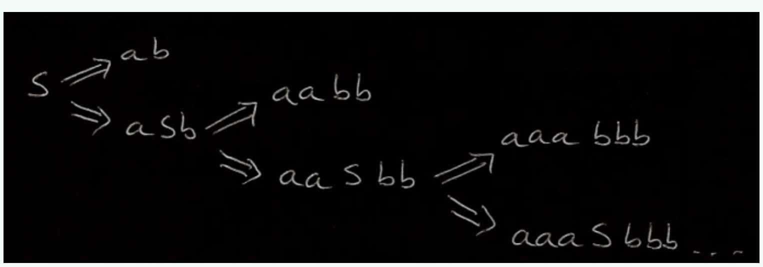
1.8.1 Algoritmo di Naif

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 produzione (regole) per trovare una derivazione che generi w

In certi casi questa verifica è semplice

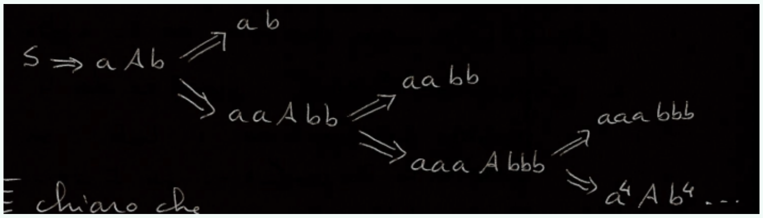
Example 1.8.1

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

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

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



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

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

1.9 Alberi di derivazione

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

Definition 1.9.1: 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 $m_1, \ldots, m_k$ con ,etichetta $x_1, \ldots, x_k$ (in $NT \cup T$), allora

$$A \longrightarrow x_1, \ldots, x_k \text{ è una produzione in R}$$

- se il nodo n ha etichetta ϵ, allora n è una foglia, è figlio unico e, dato A suo padre, $A \longrightarrow \epsilon$ 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 1.9.1

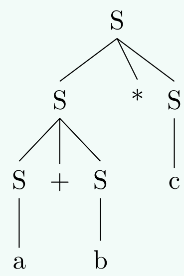
Consideriamo la grammatica

$$s \longrightarrow a|b|c|S+S|S\times S$$

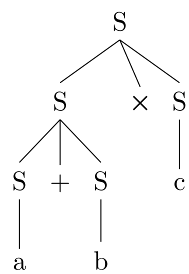
Inoltre si consideri la derivazione

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

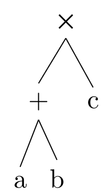
Allora il suo albero di derivazione è:



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



e



Theorem 1.9.1

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

1.9.1 Ambiguità

Per spiegare cos'è l'ambiguità partiamo con un esempio

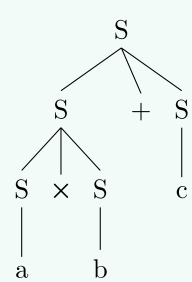
Example 1.9.2

Si consideri la seguente grammatica

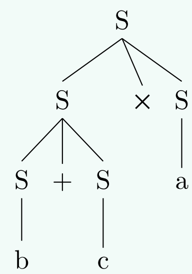
$$S \longrightarrow a|b|c|S+S|S \times S$$

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:



Per questa grammatica, la stringa $a{\times}b{+}c$ ha più di un albero di derivazione in questi casi si dice che la grammatica è quindi **ambigua** e inutilizzabile per dare semantica a $a \times b + c$. Bisogna utilizzare grammatiche non ambigue, o manipolare grammatiche ambigue per disambiguarle

Grammatica ambigua

Definition 1.9.2: Grammatica ambigua

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

Definition 1.9.3: Linguaggio ambiguo

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

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))

1.9.2 Rimuovere l'ambiguità

Questa CAZZATA viene spiegata con solo degli esempi

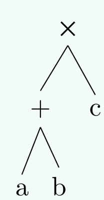
Example 1.9.3

Sia S la grammatica

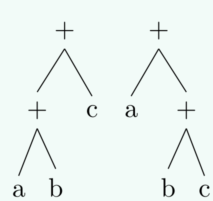
$$S \longrightarrow a \mid b \mid c \mid S+S \mid S \times S \text{ è ambigua!}$$

Problemi:

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



- associatività del + e del ×



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

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

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

Chapter 2

Struttura di un compilatore, semantica statica, semantica dinamica

2.1 Vincoli contestuali

Definition 2.1.1: vincoli sintattici contestuali

I vincoli sintattici contestuali sono termini o parole riservate non esprimibili per mezzo di grammatiche libere (perché non possono descrivere vincoli 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 esecuzioni
Il compilatore delega questi controlli di semantica statica alla cosiddetta **analisi semantica**

2.2 Semantica statica

Definition 2.2.1: Semantica statica

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

Example 2.2.1

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

2.3 semantica dinamica

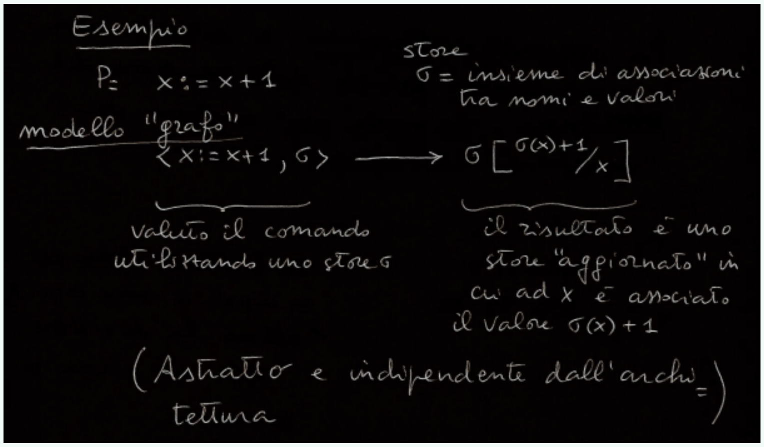
Per **semantica dinamica** si intende una rappresentazione formale dell'esecuzione del programma, la quale può mostrare errori durante l'esecuzione

Example 2.3.1

```
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 2.3.2



2.3.1 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

2.3.2 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

2.4 Pragmatica nella descrizione di un linguaggio

Definition 2.4.1: 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

Esempietti:

Example 2.4.1

- 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 paramatri ad una funzione
- scelta tra iterazione determinata (`for`) e indeterminata (`while`)

2.5 implementazione

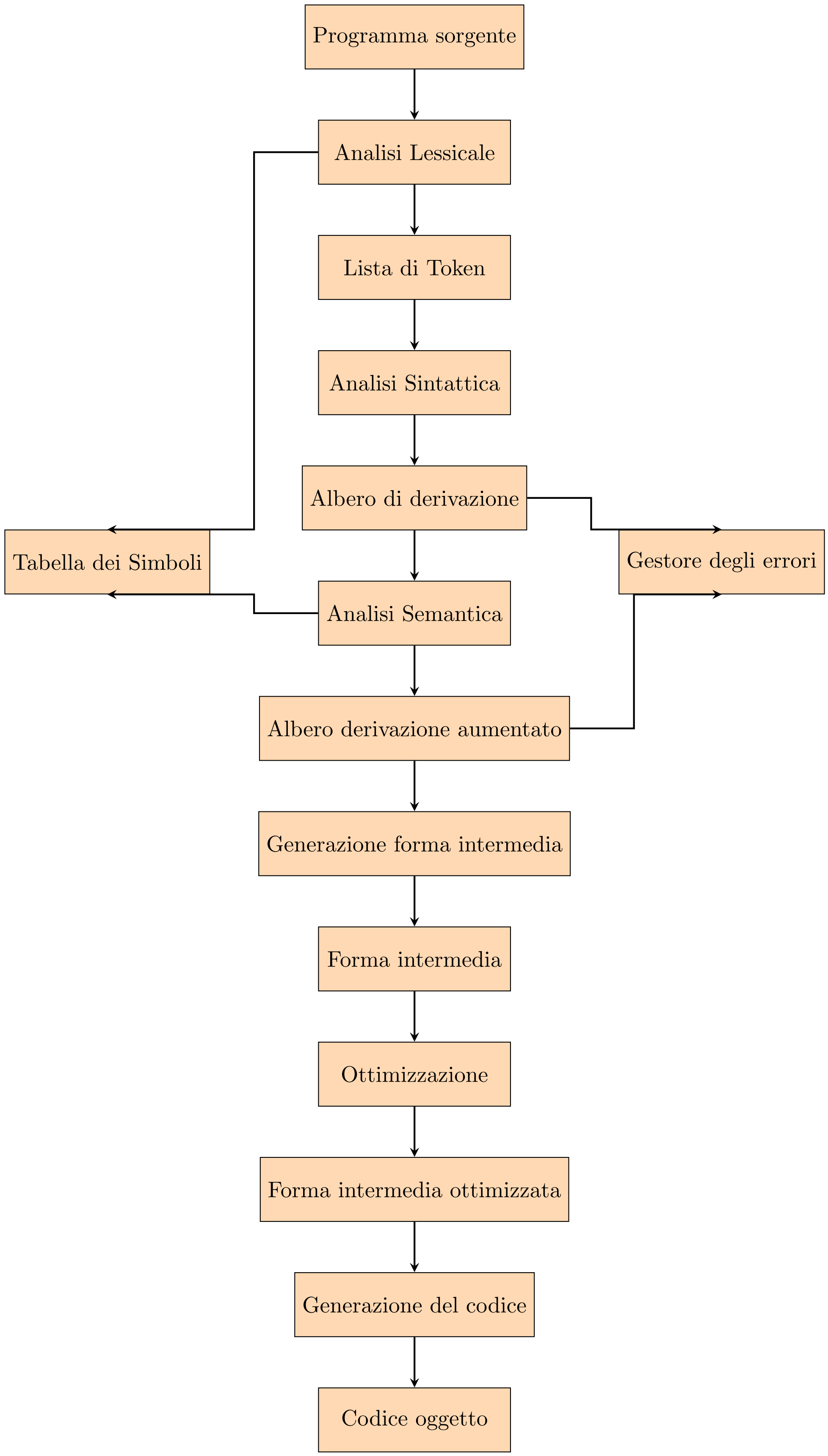
Definition 2.5.1: implementazione

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

2.5.1 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

2.5.2 Struttura di un compilatore



2.6 fasi principali della compilazione

2.6.1 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 tabello 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

2.6.2 analisi sintattica (parser)

A partire dalla lista di tokens, generata dallo scanne, 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

2.6.3 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

2.6.4 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

2.6.5 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**

2.6.6 Generazione del codice

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

2.6.7 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

2.7 semantica operazionale strutturata

2.7.1 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 atratta 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: $\{\mathtt{tt},\mathtt{ff}\}$. Le metavariabili sono $t, t_1, t' \in \mathbb{T}$
- **numeri naturali**: $\{0, 1, 2, \dots\} \quad n, m, p \in \mathbb{N}$
- **variabili**: $a, b, c, \dots, z \quad v \in Var$

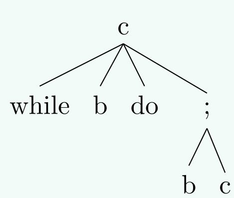
Per descrivere espressioni più complesse, vengono definiti alcuni **insiemi derivati** utilizzando la notazione BNF (Backus-Naur Form)

- **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 partite dagli alberi di sintassi astratta (ambigua) e ignorare la parte di anali del parser

Example 2.7.1
Riportiamo qui un esempio di sintassi astratta.
Che tipo di albero sintattico vogliamo intendere con la seguente espressione?

$$\text{while } b \text{ do } c_1; c_2$$



2.8 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 2.8.1: sistema di transizione

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

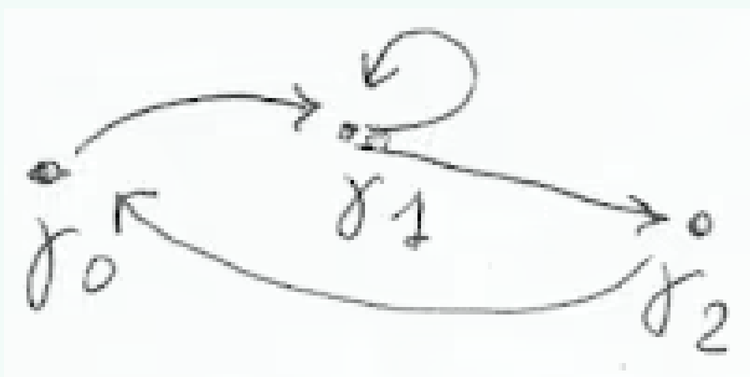
- Γ è l'insieme di stati (o configurazione)
- $T \subseteq \Gamma$ è l'insieme degli stati terminali (ovvero tutti quegli stati in cui il calcolo è stato terminato con successo)
- $\rightarrow \subseteq \Gamma \times \Gamma$ è la relazione di transazione che prende in input uno stato $\in \Gamma$ e restituisce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

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

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

Example 2.8.1



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 2.8.2

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

   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 $var = \{x_1, x_2, \dots, x_k\}$

2.8.1 Semantica delle espressioni artimetiche

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**:

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

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**:

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

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**:

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

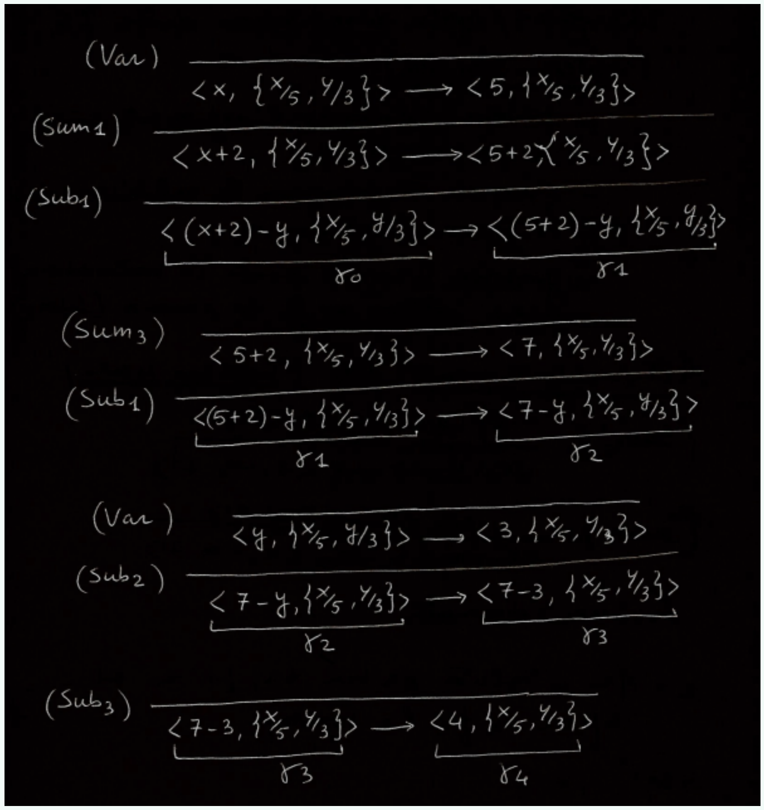
7. **Sottrazione 3**:

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

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 2.8.3

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



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

Theorem 2.8.1

Vogliamo dimostrare che $\to_e$ è deterministico, ovvero:

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

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

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

1. $e = m \in \mathbb{N}$: se $\langle e, \sigma\rangle \not\to_e e$ allora la conclusione è vera perché la premessa è falsa

2. $e = v \in Var$: Per la regola (Var), l'unica transizione derivabile per $\langle v, \sigma\rangle$ è $\langle v, \sigma\rangle \to_e \langle \sigma(v), \sigma\rangle$ 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 $\langle e_0 + e_1, \sigma\rangle \to \gamma'$ e $\langle e_0 + e_1, \sigma\rangle \to \gamma''$
   Ci sono 3 sottocasi da esaminare in accordo nel modo in cui derivo $\langle e_0 + e_1, \sigma\rangle \to \gamma'$:

   (a) $\langle e_0, \sigma\rangle \to \langle e_0', \sigma'\rangle$ 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 $\langle e_0 + e_1, \sigma\rangle \to \gamma''$ è necessario che $\langle e_0, \sigma\rangle \to \langle e_0'', \sigma''\rangle$ 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 PRECEDETE

Q.e.d. 😆

Questo teorema ci porta ad un dio boia di corollario:

Corollary 2.8.1

poiché $\rightarrow_e$ è determiniscа, 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

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

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 2.8.4

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

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

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

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

Inoltre sia introdotta la definizione di equivalenza:

Definition 2.8.2: 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 2.8.5

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

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 $Eval_{id}(e,\sigma) = \begin{cases} m & \text{se } \langle e,\sigma\rangle \rightarrow^*_{id} \langle m,\sigma\rangle \\ \text{indefinita} & \text{altrimenti} \end{cases}$
Come vedremo, è possibile definire anche altre siscipline di valutazione come Esterna Sinistra, Esterne Destra, Esterna parallela.

2.8.2 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**

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

- **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**

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

- **Or3**

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

- **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**

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

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 2.8.2

$\rightarrow_b$ è deterministica, ovvero

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

Che porta al seguente corollario:

Corollary 2.8.2

si può, quindi, definire:

$$eval_b(b, \sigma) = \begin{cases} t & \text{se } \langle b, \sigma \rangle \rightarrow^* \langle t, \sigma \rangle \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

E si ha anche la seguente definizione:

Definition 2.8.3: Equivalenza booleani

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

Example 2.8.6

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

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'**:

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

- **Or3'**:

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

Example 2.8.7

$\gamma = \langle \rangle$

Chapter 3

linguaggi liberi deterministici

3.1 PDA e linguaggi deterministici

3.1.1 PDA deterministici

Definition 3.1.1: PDA deterministico

Un PDA $N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ si dice **deterministico** sse:

1. $\forall q \in Q,\ \forall z \in \Gamma,\ \big(\forall a \in \Sigma,\ (\delta(q, \epsilon, z) \neq \varnothing \implies \delta(q, a, z) = \varnothing)\big)$
2. $\forall q \in Q,\ \forall z \in \Gamma,\ \forall a \in (\Sigma \cup \{\epsilon\}),\ \big(|\delta(q, a, z)| \leqslant 1\big)$

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

3.1.2 linguaggi liberi deterministici

Definition 3.1.2: Linguaggio libero deterministico

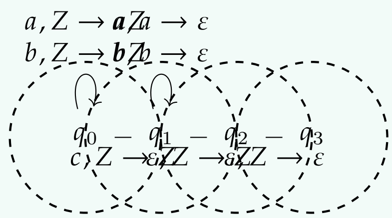
Un linguaggio è **libero deterministico** se è accettato per stato finale da un DPDA

Theorem 3.1.1
la classe dei linguaggi liberi deterministici è includa propriamente nella classe dei linguaggi liberi :)

Example 3.1.1

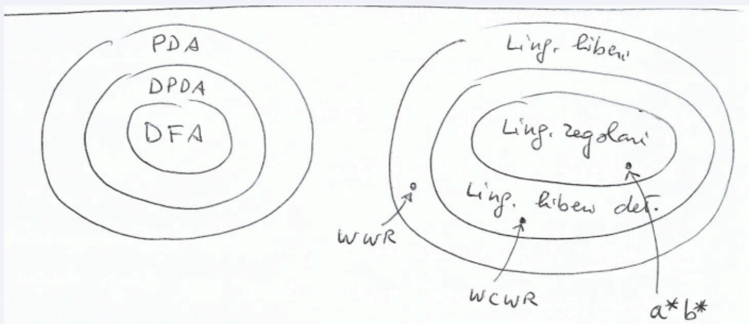
- Sia $L_1 = \{ww^R \mid w \in \{a,b\}^*\}$ è libero, am si può dimostrare che non esiste un DPDA che lo riconosca, infatti con un DPDA non esiste un modo deterministico per riconoscere quando finisce w e inizia w^R
- $L_2 = \{wcw^R \mid w \in \{a,b\}^*\}$ è libero deterministico grazie al segnaposto c è possibile riconoscere

quando inizia w^R, si ha infatti:



Theorem 3.1.2

Se L è regolare, allora $\exists$ DPDA. N tale che $L = L[N]$ per stato finale



Si giunge così alla seguente osservazione:

Note:

Un linguaggio libero deterministico L è riconosciuto da un $DPDA$ per pila vuota sse L gode della "perfix propriety", ovvero

$$\nexists x, y \in L : x \text{ è prefisso di } y$$

Pertanto si ha che:

- Se L è non gode della prefix propriety non può essere riconosciuto da un $PDPA$ per pila vuota
- Se L è libero deterministico gode della prefix propriety, 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 propriety, infatti $L\$$ può essere riconosciuto da un $PDPA$ per pila vuota

dimostrazione: Se L è regoalre, allora $\exists$ DFA M tale che $L = L[M]$. A partire da M, posso costruire un DPDA N si compore come M senza mai manipolare lo stack, allora si che $L = L[N]$ per stato finale 😆

3.2 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 $A \to \epsilon$) inadatte al bottom up parsing
- Eliminare le produzione unitarie (del tipo $A \to B$ 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 inziale S

  es. gli stati d'errore
- Eleminare la ricorsione sinistra (del tipo $A \to A\alpha$), perché inadatte al top - down parsing
- fattorizzare le grammatiche, per ottenere grammatiche con meno non determinismo nel top-down parsing

3.2.1 Eliminare le produzioni ϵ

Per fare ciò si usi un algoritmo che ha:

- in input: una G libera con produzione ϵ
- in output: una G' libera senza produzione ϵ tale che $L(G') = L(G) \backslash \{\epsilon\}$

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 $G'' = G' \cup \{S' \to \epsilon | S\}$ t.c.

$$G'' = (NT \cup \{S'\}, T, S', R' \cup \{S' \to \epsilon | S\})$$

simboli annullabili

Per l'algoritmo occorre innanzi tutto definire i **simboli annullabili**

Definition 3.2.1: 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:

- $N_0(G) = \{A \in NT | A \to \epsilon\}$, questo è il caso in cui un non terminale A viene riscritto direttamente in ϵ tramite una produzione
- $N_{i+1}(G) = N_i(G) \cup \{B \in NT | B \to c_1, \ldots, c_k \in \mathbb{R}$ e $c_1, \ldots, c_k \in N_i(G)\}$ questo è il caso in cui un non terminale B possa essere ricondotto a 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 3.2.1

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 $A \to \alpha \in R$ con $\epsilon \notin \alpha$, in cui occorrono simboli annullabili $Z_1, \ldots, Z_k$, mettiamo in R' tutte le produzioni del tipo $A \to \alpha'$ dove α' si ottiene da α cancellando tutti i possibili sottoinsiemi di $Z_1, \ldots, Z_k$ (incluso $\varnothing$), 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 $Z_1, \ldots, Z_n$:

- in G' non mettiamo produzioni $A \to \epsilon \in R$,
- in G' non introduciamo mai produzioni del tipo $A \to \epsilon$.

Theorem 3.2.2

Data una grammatica libera G, la grammatica G' determinata dall'algoritmo sopra non ha ϵ-produzioni, e $L(G') = L(G) \setminus \{\epsilon\}$

Example 3.2.1

Sia G una grammatica tale che:

$$G = \begin{cases} S \rightarrow AB \\ A \rightarrow aAA|\epsilon \\ B \rightarrow bBB|\epsilon \end{cases} \qquad N_0(G) = \{A, B\} \text{ e } N_1(G) = \{A, B, S\} = N(G)$$

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 Anella 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:
  $$G' = \begin{cases} S \rightarrow AB|A|B \\ A \rightarrow aAA|aA|a \\ B \rightarrow bBB|bB|b \end{cases}$$

3.2.2 Eliminazione delle produzioni unitarie

Definition 3.2.2: 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 3.2.3: Coppia unitaria

Una coppia (A,B) si dice **unitaria** qunado $A \implies {}^*B$ (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
- $U_{i+1}(G) = U_i(G) \cup \{(A,C)|(A,B) \in U_i(G)\}$ e $B \longrightarrow C \in R$, 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 $B \longrightarrow C \in R$

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 $A \longrightarrow \alpha$, dove $B \longrightarrow \alpha \in R$ 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 3.2.3

Sia $G = (NT,T,R,S)$ libera e sia $U(G)$ l'insieme selle sue coppie unitarie. Sia $G' = (NT,T,R',S)$ la grammatica ottenuro dall'algoritmo G' non ha produzione unitarie e $L(G) = L(G')$

Esempietto:

Example 3.2.2

Prediamo con esempio la grammatica non ambigua E delle espressioni aritmetiche:

$$E = \begin{cases} E \longrightarrow E + T|T \\ T \longrightarrow T * A|A \\ A \longrightarrow a|b|(E) \end{cases}$$

Si noti subito che ha 2 produzioni unitarie: $E \longrightarrow T$ e $T \longrightarrow A$. 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 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:

$$G' = \begin{cases} E \longrightarrow E + T \\ T \longrightarrow T \times A \\ A \longrightarrow a|b|(E) \end{cases}$$

In aggiunta:

$$\begin{cases} E \longrightarrow E + T & \text{perché } (E,T) \in U_1(G) \\ T \longrightarrow a|b|(E) & \text{perché } (T,A) \in U_1(G) \\ E \longrightarrow a|b|(E) & \text{perché } (E,A) \in U_2(G) \end{cases}$$

Pertanto G' sarà:

$$G' = \begin{cases} E \longrightarrow E + T|T \times A|a|b|(E) \\ T \longrightarrow T \times A|a|b|(E) \\ A \longrightarrow a|b|(E) \end{cases}$$

Sia ha che non contiene produzioni unitarie ed è equivalente a G

3.2.3 Rimuovere i simboli inutili

Definition 3.2.4: Simboli generatori, raggiungibili e utili

Un simbolo $X \in T \cup NT$ è

- Un generatore $\iff \exists w \in T^*$ con $x \implies {}^*w$

  Quindi un generatore è o un terminale (un simbolo può riscriversi 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 3.2.4

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 a 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 w) 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 inverto l'ordine, allora può capitare che non elimino tutti i simboli inutili

Esempietto di eliminazione di tutti quei simboli non utili (inutili)

Example 3.2.3

Si parta da questa grammatica:

$$G = \begin{cases} S \to AB|a \\ B \to b \end{cases}$$

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 eleminare tutte le produzioni che includono A:

$$G' = \begin{cases} S \to a \\ B \to b \end{cases}$$

Adesso posso eliminare tutti i non raggiungibili da S, che in questo caso l'unico è solo B. Si ha che:

$$G'' = S \to a$$

Si ha che G''è equivalente a G, ma non contiene simboli utili

Esempio secondo:

Example 3.2.4

$$G = \begin{cases} S \to aC \\ A \to a \\ B \to bB \\ C \to b \mid AC \\ D \to a \mid aS \end{cases}$$

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:

$$G_1 = \begin{cases} S \to aC \\ A \to a \\ C \to b \mid AC \\ D \to a \mid aS \end{cases}$$

A questo punto, notiamo che solo D non è raggiungibile, quindi possiamo eliminarlo. Si ottiene:

$$G_2 = \begin{cases} S \to aC \\ A \to a \\ C \to b \mid AC \end{cases}$$

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:

$$S \rightarrow aS|ab$$

3.2.4 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 nè 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 3.2.5

$$G = \begin{cases} S \rightarrow aAa \mid aa \\ A \rightarrow C \\ C \rightarrow S \mid \varepsilon \end{cases}$$

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 $(aa)^+$:

$$S \to aSa|aa$$

o anche

$$S \to aaS|aa$$

3.2.5 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 $A \to BC$ o $A \to a$, 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 $A \to a\alpha$, 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 3.2.5: Forma normale di Chomsky

Una grammatica si dice in **forma normale di Chomsky** se sono nella forma:

$$A \to BC$$
$$A \to a$$

Dove ϵ è trattato a parte $S \to \epsilon|BC$ 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 3.2.6: forma normale di Greibach

Una grammatica si dice in **forma normale di Greibach** se sono nella forma:

$$A \to aBC$$
$$A \to aB$$
$$A \to a$$

Dove ϵ è trattato a parte $S \to \epsilon|BC$ 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

3.2.6 Eliminare la ricorsione a sinistra

L'eliminazione della ricorsione a sinistra è un problema tipico dei parser top-down

Definition 3.2.7: produzione ricorsiva a sinistra

Si definisce **una produzione ricorsiva a sinistra** una produzione del tipo

$$A \to A\alpha \in R$$

Definition 3.2.8: 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:

$$A \to A_{\alpha_1}|\dots|A_{\alpha_n}|\beta_1|\dots|\beta_n$$

Dove le stringhe β_i non cominciano per A. Queste produzioni possono essere rimpiazzate da

$$\begin{aligned} &A \to \beta_1 A'|\dots|\beta_m A' \\ &A' \to \alpha_1 A'|\dots|\alpha_n A'|\epsilon \end{aligned}$$

Se nella grammatica originale avviamo la derivazione

$$A \implies A\alpha_{i_1} \implies A\alpha_{i_2}\alpha_{i_1} \implies \cdots \implies A\alpha_{i_k}\dots\alpha_{i_2}\alpha_{i_1} \implies \beta_i\alpha_{i_k}\dots\alpha_{i_2}\alpha_{i_1}$$

Con la nuova grammatica si ha:

$$A \implies \beta_i A' \implies \beta_i\alpha_{i_k}A' \implies \cdots \implies \beta_i\alpha_{i_k}\dots\alpha_{i_2}A' \implies \beta_i\alpha_{i_k}\dots\alpha_{i_1}A' \implies \beta_i\alpha_{i_k}\dots\alpha_{i_1}$$

Esempi concreti:

Example 3.2.6

$$\begin{aligned} &A \to Aa|b \\ &\Rightarrow \\ &A \to bA' \\ &A' \to aA'|\epsilon \end{aligned}$$

Poi

$$\begin{aligned} &A \to Ab|Ac|d \\ &\Rightarrow \\ &A \to dA' \\ &A' \to bA'|cA'|\epsilon \end{aligned}$$

Note:

Se $G = [A \to Aa]$, non si può applicare l'algoritmo perché mancano le produzione di base da cui partire $(A \to \beta_1|\dots|\beta_m)$. Infatti, $L(G) = \emptyset$ e la grammatica corrispente non ha produzioni

3.2.7 Ricorsione sx non-immediata

Consideriamo

$$G = \begin{cases} S \to Ba|b \\ B \to Bc|Sc|d \end{cases}$$

In G c'è ricorsione sx immediata ($B \to Bc$) ma anche non immediata ($S \implies Ba \implies Sca$)

algoritmo per il calcolo della ricorsione non immediata

Algorithm 1: 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, ..., A_n} in un ordine fissato;
2 for i = 1to n do
3     for j = 1to i − 1 do
4         Sostituisci ogni produzione della forma A_i → A_j α con le produzioni A_i → β_1 α| ... |β_k α,dove
            A_j → β_1| ... |β_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 $A_i \to A_k\alpha$ 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, $A_1 \to A_k\alpha$ 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 $A_i \to A_k\alpha$ è tale che $i < k$.

  Ora l'istruzione 2) rimuove l'eventuale ricorsione sx immediata da A_i, sicché ogni produzione $A_i \to A_k\alpha$ è tale che $i < k$.
- Quindi al termine dell'algoritmo, avremo che ogni produzione $A_i \to A_k\alpha$ è tale che $i < k$, garantendo l'impossibilità di creare ricorsione sx non immediata.

Example 3.2.7

Come esempio si consideri la grammatica di prima, ovvero

$$G = \begin{cases} S \to Ba|b \\ B \to Bc|Scd \end{cases}$$

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 $B \to Sc$ viene rimpiazzata con:

  $$B \to Bac|bc$$

  Ora le produzioni complessive per B sono :

  $$B \to Bc|Bac|bc|d$$

Dalla quale dobbiamo eliminare la ricorsione immediata, il risultato è:

$$B \longrightarrow bcB'|sB'$$
$$B' \longrightarrow cB'|acB'|\epsilon$$

Pertanto la gigagrammatica risultante è:

$$S \longrightarrow Ba|B$$
$$B \longrightarrow bcB'|sB'$$
$$B' \longrightarrow cB'|acB'|\epsilon$$

3.2.8 Fattorizzazione a sinistra

Si prendi in esempio la seguente grammatica:

$$A \longrightarrow aBbC|aBd$$

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 raccogliore la parte comune (aB) alle 2 produzioni e introduco un nuovo nonterminale per rappresentare il resto delle produzione, quindi:

$$A \longrightarrow aBA'$$
$$A' \longrightarrow bC|d$$

algortimo per il calcolo della fattorizzazione

Algorithm 2: Fattorizzazione LU

Input: Grammatica G non fattorizzata
Output: Grammatica G' fattorizzata

```
1  Let N be a new variable;
2  N ← NT;
3  while è pssibile modificare a N o all'insieme delle produzione do
4      foreach A ∈ N do
5          Sia α il prefissio più lungo comune alle parti destre di alcune produzione di A;
6          if α ≠ ϵ then
7              Sia A un nuovo non terminale ;
8              N ← N ∪ {A};
9              rimpiazza tutte le produzione per A del tipo
                   A → αβ_1|...|αβ_k|γ_1|...|γ_h
               con le produzioni:
                   A → αA'|γ_1|...|γ_h
                   A' → β_1|...|β_k
```

Example 3.2.8

Riporto una grammatica da fattorizzare:

$$E \longrightarrow T|T+E|T-E$$
$$T \longrightarrow A|A*T$$
$$A \longrightarrow a|b|(E)$$

Dove sia E che T si possono fattorizzare, perciò diventa:

$$
\begin{aligned}
&E \longrightarrow TE' \\
&E' \longrightarrow \epsilon | + E | - E \\
&T \longrightarrow AT' | \\
&T' \longrightarrow \epsilon | * T \\
&A \longrightarrow a | b | (E)
\end{aligned}
$$

Chapter 4

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

4.1 Parser a discesa ricorsiva

Data una grammatica libera $G = (NT, T, S, R), \forall A \in NT$ con produzioni:

$$A \to X_1^1 \dots X_{n_1}^1 | \dots | X_k^1 \dots X_{n_k}^k$$

Definisce la funzione

Algorithm 3: 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** $X_i^h =$ *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

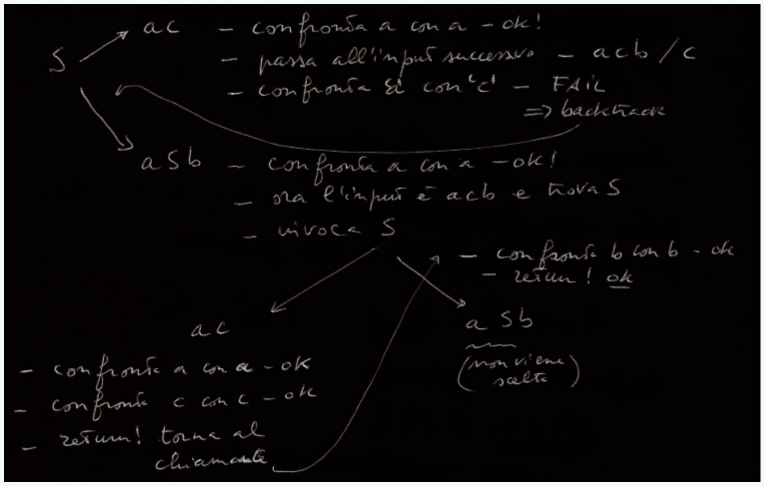
Example 4.1.1
Sia G la grammatica:

$$S \to ac|aSb$$

Col linguaggio:

$$L = \{a^{n+1}cb^n | n \geqslant 0\}$$

e sia $aacb$ un input. Si ha



input	**Stack delle chiamate**
$\underline{\text{a}}$ a c b	S
	$\underline{\text{a}}$ c
$\underline{\text{a}}$ c b	c *fail*
$\underline{\text{a}}$ a c b	S
	$\underline{\text{a}}$ S b
$\underline{\text{a}}$ c b	S b
	$\underline{\text{a}}$ c b
$\underline{\text{c}}$ b	$\underline{\text{c}}$ b
$\underline{\text{b}}$	b
	$\underline{\text{ok}}$

Tuttavia il parser a discesa ricorsiva è parecchio inefficiente a causa della sua natura nondeterminista, vi è infatti la necessita nel peggiore dei casi di esplorare tutte le alternative

Theorem 4.1.1

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

4.2 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

4.2.1 First

Definition 4.2.1: First

Data una grammatica libera G e $\alpha \in (T \cup NT)^*$, di definisce **First**(α) come l'insieme dei terminali che possono stare in prima posizione in una stringa che si deriva da α

- per $a \in T, a \in First(\alpha) \iff \alpha \implies {}^*a\beta$ per $\beta \in (T \cup NT)^*$
- inoltre $(\alpha \implies {}^*\epsilon) \implies \epsilon \in First(\alpha)$

Note:

Sia la grammatica

$$A \to \alpha_1 | \alpha_2$$

Se $First(\alpha_1) \cap First(\alpha_2) = \emptyset$ la scelta della produzione è deterministica

Qui vi è riportato un esempietto:

Example 4.2.1

$$A \to aB|bC$$

Si ha che:

$$First(aB) = \{a\}$$
$$First(bC) = \{b\}$$

Pertanto abbiamo del determinismo con un solo carattere in lettura

algoritmo per calcolare il first

Algorithm 4: First()

Input: Una grammatica credo
Output: bho

for $x \in T$ **do**
  $First(x) \leftarrow \{x\}$; // un terminale è il primo elemento di se stesso
for $X \in NT$ **do**
  $First(X) \leftarrow \emptyset$; // per ogni x non terminale si inizializza il suo first a "0"
while *almeno un* $First(X)$ *può essere modificato in una iterazione* **do**
  foreach $x \to Y_1, \ldots, Y_k$ **do**
    foreach $i = 1$**to** k **do**
      // se ciascuno di questi simboli $y_1, \ldots, Y_{i-1}$ può derivare la stringa vuota ϵ
      if $Y_1, \ldots, Y_{i-1} \in N(G)$ **then**
        $First(X) \leftarrow First(X) \cup (First(Y_i) \backslash \{\epsilon\})$; // allora è possibile aggiungere gli elementi di $FIRST(Y_i)$ a $FIRST(X)$ per la produzione $y_1, \ldots, y_k$
      // 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
foreach $X \in N(G)$ **do**
  $First(X) = First(X) \cup \{\epsilon\}$;

In generale per una stringa α si ha che:

- Se $\alpha = \varepsilon$, allora $FIRST(\alpha) = \{\varepsilon\}$.
- Se $\alpha = X\beta$ e $X \notin N(G)$, allora $FIRST(X\beta) = FIRST(X)$.
- Se $\alpha = X\beta$ e $X \in N(G)$, allora $FIRST(X\beta) = (FIRST(X) \setminus \{\varepsilon\}) \cup FIRST(\beta)$

In pratica se

$$A \longrightarrow \alpha_1 | \dots | \alpha_k$$

si ha che

$$First(A) = First(\alpha_1) \cup \dots \cup First(\alpha_k)$$

Example 4.2.2

Si ossrvi la seguente grammatica:

$$S \longrightarrow Ab|c$$
$$A \longrightarrow aA|\epsilon$$

$$\begin{aligned} FIRST(S) &= FIRST(Ab) \cup FIRST(c) \\ &= (FIRST(A) \setminus \{\epsilon\}) \cup FIRST(b) \cup \{\epsilon\} \\ &= \{a\} \cup \{b\} \cup \{c\} = \{a, b, c\} \end{aligned}$$

$$\begin{aligned} FIRST(A) &= FIRST(aA) \cup FIRST(\epsilon) \\ &= \{a\} \cup \{\epsilon\} = \{a, \epsilon\} \end{aligned}$$

4.2.2 Follow

Definition 4.2.2: Follow

Data una grammatica libera G e $A \in NT$, definiamo che $Follow(A)$ è l'insieme dei terminali che possono comparire immediatamente a destra di A in una forma sentenziale.

- Per ogni $a \in T$, $a \in Follow(A)$ se $S \Longrightarrow^* \alpha A a \beta$ per qualche α e $\beta \in (T \cup NT)^*$.
- $\$ \in Follow(A)$ se $S \Longrightarrow^* \alpha A$ (Poiché $S \Longrightarrow^* S$, allora $\$ \in Follow(S)$!)

Riporto qui un esempio

Example 4.2.3

$$S \longrightarrow Ab \mid c$$
$$A \longrightarrow aA \mid \varepsilon$$

$$Follow(S) = \{\$\} \quad Follow(A) = \{b\}$$

- $S \Longrightarrow^* S$
- $S \Longrightarrow^* Ab$

algoritmo per calcolare il Follow

Algorithm 5: Follow()

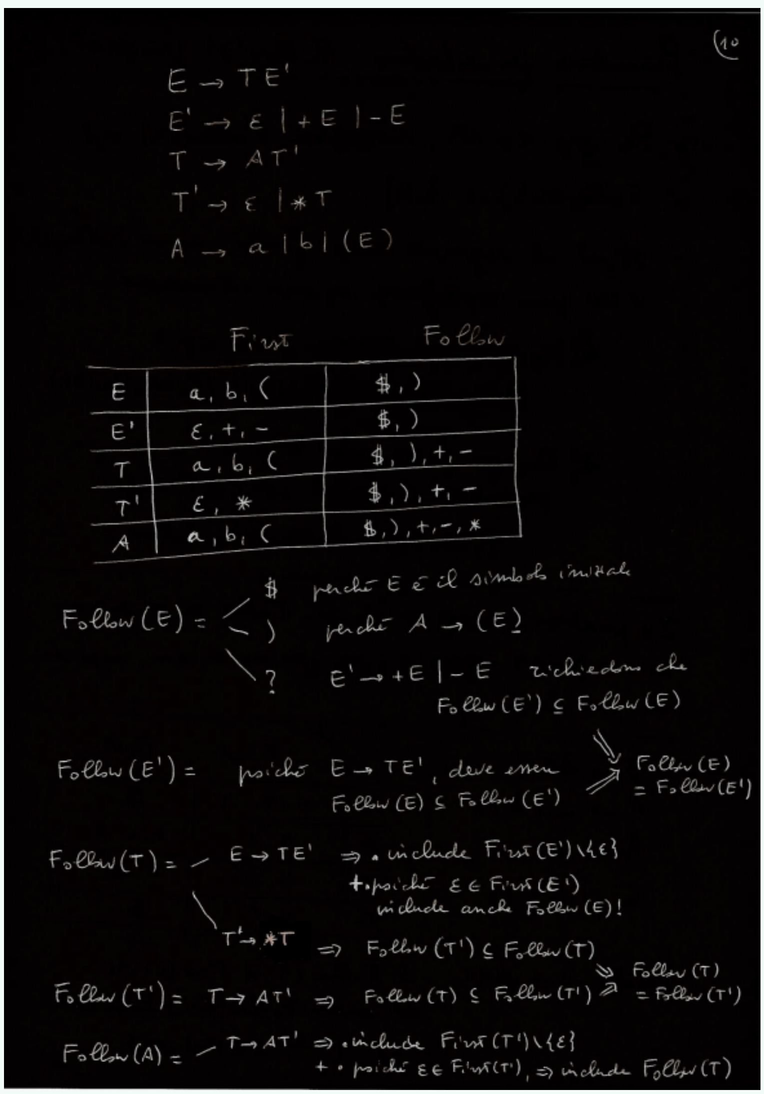
Input: Una grammatica credo
Output: bho

```
1 foreach X ∈ NT do
2 |  First(X) ← ∅;                    // per ogni X non terminale si inizializza il suo first a "0"
3 Follow(S) ← {$};
4 while almeno un Follow(X) può essere modificato in una iterazione do
5 |  foreach X → αYβ do
6 |  |  Follow(Y) ← Follow(Y) ∪ (First(β)\{ε});
7 |  foreach X → αY do
8 |  |  foreach X → αβ, ε ∈ First(β) do
9 |  |  |  Follow(Y) ← Folloe(Y) ∪ 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 4.2.4



Adesso che abbiamo introdotto i le procedure First e Follow occorre fare un passo in più per definire i parser

4.2.3 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 $A \to \alpha$:

1. per ogni $a \in T$ e $a \in \text{First}(\alpha)$, inserisci $A \to \alpha$ nella casella $M[A, a]$
2. se $\varepsilon \in \text{First}(\alpha)$, inserisci $A \to \alpha$ 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 4.2.3: grammatica $LL(1)$

Una grammatica si definisce $LL(1)$ sse ogni casella della tabella di parsing $LL(1)$ contirne 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 4.2.1
G è $LL(1)$ sse per ogni coppia di produzioni distinte con la stessa testa

$$A \to \alpha | \beta$$

si ha che

1. $First(\alpha) \cap First(\beta) = \emptyset$
2. (a) $(\epsilon \in First(\alpha)) \implies (First(\beta) \cap Follow(A) = \emptyset)$
   (b) $(\epsilon \in First(\beta)) \implies (First(\alpha) \cap Follow(A) = \emptyset)$

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 prodizone in ogni cassella. Ma vale anche viceversa! 😊

Linguaggio $LL(1)$

Definition 4.2.4: Linguaggio $LL(1)$

Un linguaggio si definisce $LL(1) \iff \exists G'$ grammatica $= LL(1)$ che lo genera

Example 4.2.5

Sia G la segunete grammatica:

$$S \longrightarrow A|B$$
$$A \longrightarrow ab|cd$$
$$B \longrightarrow ad|cb$$

Si può notare che G non è $LL(1)$ dato che $S \longrightarrow A|B$ e

$$First(A) = \{a, c\}$$
$$First(B) = \{a, c\}$$
$$First(A) \cap First(B) = \{a, c\}$$

Dal teorema sopra fornito si può dimostrare che non è $LL(1)$
Tuttavia si può manipolarla per farla diventare $LL(1)$, quindi espando S:

$$S \longrightarrow ab|cd|ad|cb$$

$$S \longrightarrow aT|cT'$$
$$T \longrightarrow b|d$$
$$T' \longrightarrow b|d$$

Poi osservo che T e T' sono identici, sia quindi G' la nuova grammatica:

$$S \longrightarrow aT|cT$$
$$T \longrightarrow b|d$$

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 4.2.2

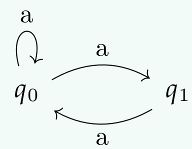
Ogni linguaggio regolare è generabile da una grammatica G di classe $LL(1)$

Dimostrazione: Sia L un linguaggio regolare, allora $\exists$ DFA $M = (Q, \Sigma, \delta, q_0, F) : L = [M]$.
A partira 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 $[q] \longrightarrow a[q'] \in R$
    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 $[q] \longrightarrow a[q']$ 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 $[q] \longrightarrow \epsilon \in R$

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 $[q] \longrightarrow a[q']$ che "inizia" per a e dato che se q è finale, allora $[q] \longrightarrow \epsilon$ è applicabile solo per i Follow$([q]) = \{\$\} \implies$ nessun conflitto, dato che nessuna produzione genera \$, si ha che G è $LL(1)$ 😃

Example 4.2.6



Le produzioni corrispondenti sono: (TODO: perche' si rompe se metto due righe??)

$$[q_0] \to a[q_1] \mid a[q_0][q_1] \to a[q_0] \mid \epsilon$$

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 4.2.7

Sia G la seguente grammatica:

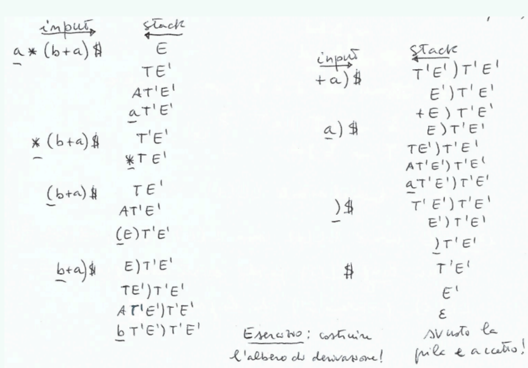
$$\begin{aligned}
&E \to TE' \\
&E' \to \varepsilon \mid +TE' \mid -TE' \\
&T \to AT' \\
&T' \to \varepsilon \mid *T \\
&A \to a \mid b \mid (E)
\end{aligned}$$

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	$E \to TE'$	$E \to TE'$	$E \to TE'$			
E'				$E' \to +TE'$	$E' \to -TE'$	$E' \to \varepsilon$
T	$T \to AT'$	$T \to AT'$	$T \to AT'$			
T'				$T' \to \varepsilon$	$T' \to *T$	$T' \to \varepsilon$
A	$A \to a$	$A \to b$	$A \to (E)$			



Algorithm 6: Parser $LL(1)$

```
Input: Stringa w
Output: Niente
Pila ← S$;                                   // cima della pila a sinistra
X ← S ;                                      // top della pila
// lettura in input
input ← w$;
i_c = primo carattere dell'input;
// viene eseguito il ciclo While finché la pila non è vuota o l'input non è stato consumato
while X ≠ $ do
    if X è un terminale then
        // si controlla se X fa "match" con i_c
        if X = i_c then
            Pop X dalla pila;                // rimuovo X dalla pila
            avanza i_c sull'input;
        else
            Errore();                        // no match
    else
        // se nella tabella di Parsing M esiste una regola X → Y_1,...,Y_n
        if M[X, i_c] = X → Y_1,...,Y_n then
            Pop X dalla pila;
            Push Y_1,...,Y_n sulla pila ;    // mette sulla pila i simboli Y_1,...,Y_n con Y_1 in cima
            In output la produzione X → Y_1,...,Y_n;
        else
            Errore ();                       // se non esiste una regola in M[X, i_c] si genera un errore, viene definito caso "bianco"
        X ← top della pila;                  // Aggiorna X con il nuovo simbolo in cima alla pila
if i_c ≠ $ then
    Errore();                                // pila svuotata ma vi è ancora dell'input da leggere
```

Algoritmo Per il calcolo di un parser $LL(1)$

Un altro esempio:

Example 4.2.8

Sia G la seguente grammatica:

$$S \to aAB \mid bS$$
$$A \to a$$
$$B \to b$$

Che genera il seguente linguaggio

$$L(G) = L\left(b^* aab\right)$$

Si ha che questa grammatica G è LL(1), perché:

$$First(aAB) \cap First(bS) = \varnothing, \quad \{a\} \cap \{b\} = \varnothing$$

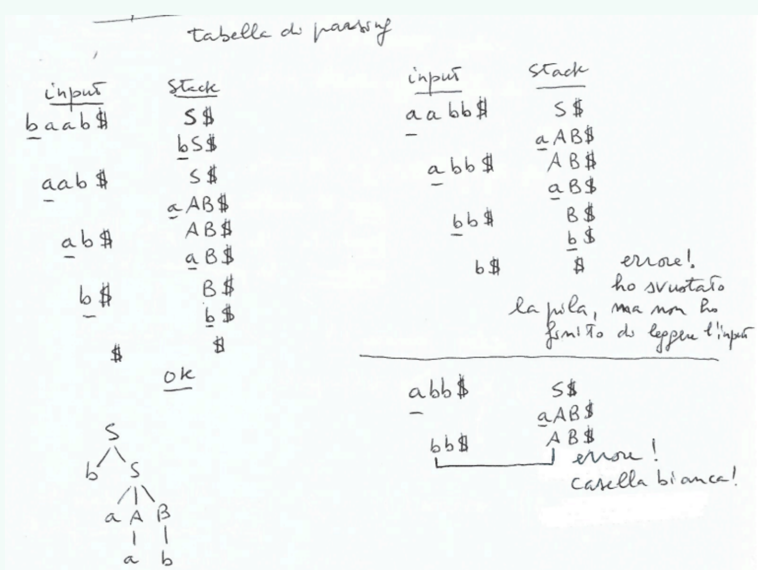
Si può prosegure con la tabella First e follow:

$Simbolo$	$First$	$Follow$
S	a, b	$\$$
A	a	b
B	b	$\$$

Da cui si può costruire la seguente tabella di parsing:

	a	b	$\$$
S	$S \to aAB$	$S \to bS$	
A	$A \to a$		
B		$B \to b$	

Viene qui descritto il funzionamento del parser con pila con diversi input:



4.2.4 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 4.2.5: 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 $w \in First_k(\alpha) \iff \alpha \implies {}^* w\beta$ con $|w| = k, w \in T^*, \beta \in (T \cup NT)^*$ oppure $\alpha \Rightarrow^* w$ con $|w| \leqslant k$ e $w \in T^*$

Definition 4.2.6: Follow $LL(K)$

L'insieme $\text{Follow}_k(A)$ definisce quali stringhe possono apparire immediatamente dopo un simbolo non terminale A in una derivazione a partire dal simbolo iniziale S. In particolare: $w \in \text{Follow}_k(A)$ se $S \Rightarrow^* \alpha A w \beta$ con $|w| = k$, $w \in T^*$, e $\alpha, \beta \in (T \cup NT)^*$, oppure, $S \Rightarrow^* \alpha A w$ con $|w| \leqslant k$ e $w \in T^*$.

Tabella di Parsing $LL(K)$

- **Righe**: non terminali.
- **Colonne**: $\{w \in T^* \mid |w| \leqslant k\}$ (solo quelle necessarie).

Per ogni produzione $A \to \alpha$, la tabella $M[A, w]$ contiene:

- $A \to \alpha$, 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 $A \to \alpha$ o a $\text{Follow}_k(A)$ per $A \to \alpha$. Questo va verificato per tutte le produzioni:

$$w \in \text{First}_k(\alpha).$$

Example 4.2.9

Sia G la seguente grammatica

$$S \to aSb \mid ab \mid c$$

Si ha che $L(G) = \{a^n b^n \mid n \geqslant 1\} \cup \{a^n c b^n \mid n \geqslant 0\}$, inoltre:

- $First_2(aSb) = \{aa, ac\}$
- $First_2(ab) = \{ab\}$
- $First_2(c) = \{c\}$

Si può dimostrare che G è $LL(2)$ dato che:

- $First_2(aSb) \cap First_2(ab) = \varnothing$
- $First_2(aSb) \cap First_2(ab) = \varnothing$
- $First_2(aSb) \cap First_2(ab) = \varnothing$

Da cui si può ricavare la tabella di parsing:

$Simbolo$	aa	ab	ac	c
S	$S \to aSb$	$S \to ab$	$S \to aSb$	$S \to c$

Theorem 4.2.3

- 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 4.2.7: Linguaggio $LL(K)$

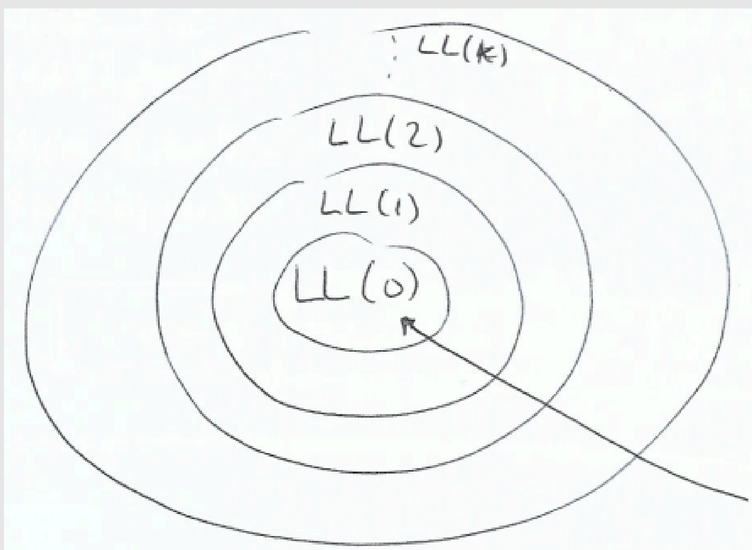
un linguaggio L è di classe $LL(K)$ se G di classe $LL(K)$ tale che $L = L(G)$

Theorem 4.2.4

$\forall k \geqslant 0$, la classe dei linguaggi $LL(K)$ contiene strettamente la classe dei linguaggi $LL(K)$

Note:

$\forall k \geqslant 0$, la classe dei linguaggi $LL(K+1)$ contiene strettamente la classe dei linguaggi $LL(K)$



Si ha che: $(\forall A \in NT, \quad \exists! \alpha \in (T \cup NT)^*, \quad A \to \alpha \implies G \text{ è } LL(0)) \implies L(G) = \{w\}$ ovvero una sola parola al massimo

Nella pratica tuttavia si usano solo $LL(1)$, spesso la si può manipolare trasformandola in $LL(1)$

Example 4.2.10

$$S \to Asb \mid ab \mid c$$

G è un $LL(2)$ ma non $LL(1)$ Si fattorizza

$$S \to aT|c$$
$$T \to Sb|b$$

Ottenuta fattorizzando G è $LL(1)$ infatti:

- $First(aT) \cap First(c) = \varnothing$
- $First(Sb) \cap First(b) = \varnothing$

Si ha che $L = \{a^n b^n \mid n \geqslant 1\} \cup \{a^n c b^n \mid n \geqslant 0\}$ è un linguaggio di classe $LL(1)$ perché G' è $LL(1)$

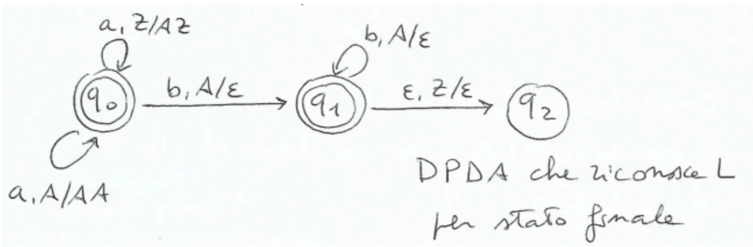
Casi speciali

Sia

$$L = \{a^i b^j | i \geqslant j\} \text{ è libero deterministico}$$

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:

$$S \longrightarrow aS|B$$
$$B \longrightarrow aBb|\epsilon$$



Sia G una grammatica libero per L

$$S \longrightarrow aS \mid BB \longrightarrow aBb \mid \epsilon$$

e poniamo $L = L(G)$
Per scegliere tra $S \longrightarrow aS$ e $S \longrightarrow B$ 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