# Linguaggi Formali e Traduttori

Daniel Biasiotto

June 14, 2022

CONTENTS

- Teacher: Sproston Jeremy
- PDF Version
- Prova di laboratorio (progetto, interrogazione singola anche in caso di progetto di gruppo con gruppi da 3)
  - Sostenibile dopo aver superato Teoria

  1/3 del voto
  - LFTCompiler

# 1 TESTI

## 1.1 Compilatori

Principi tecniche e strumenti

## 1.2 Automi

Automi, Linguaggi e Calcolabilita'

# 2 FASI COMPILATORE

- NB
  - $$\text{DFA} = \text{NFA} = \epsilon\text{-NFA} = \text{RE}$$
  - $$\text{DFA} \subset \text{DPDA} \subset \text{CFG non ambigue} \subset \text{CFG} = \text{PDA}$$

## 2.1 Analisi Lessicale

sequenze di caratteri | `token` o `lessemi`
Si passa da

1. Programma come sequenza di caratteri
2. Programma come sequenza di token

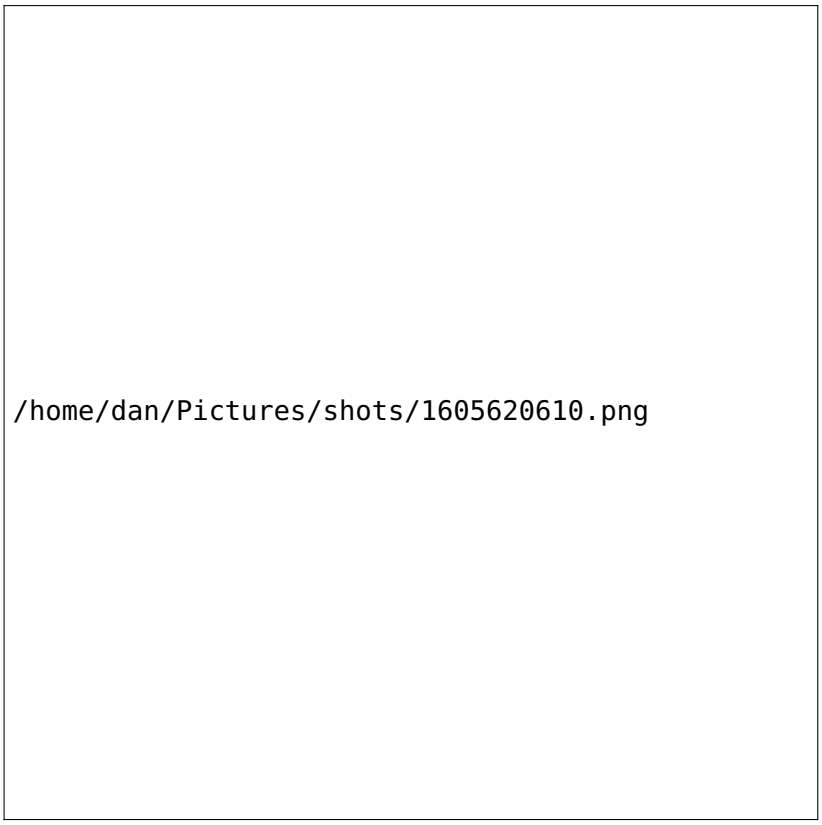
### *2.1.1 Token*

Costante numerica intera sequenza non vuota di cifre decimali, preceduta da + o - Costante numerica con virgola due sequenza (almeno 1 non vuota) di cifre decimali separate da . Identificatore sequenza on vuota di lettere numeri e _ e non inizia con un numero

### *2.1.2 Lexer*

`Analizzatore lessicale` —-> Codice <—- La visione del programma passa da "carattere per carattere" a "token per token"

- spazi e commenti vengono scartati dal lexer
  - * *
  - / /
    - finiscono con a capo o EOF
- Token




  - Identificatori
    * non comincia con un numero
    * non é composto solamente dal simbolo _
    * Ovvero corrisponde all'espressione regolare
      · ([a-zA-Z] | (_)*[a-zA-Z0-9])) ([a-zA-Z0-9] | _)*

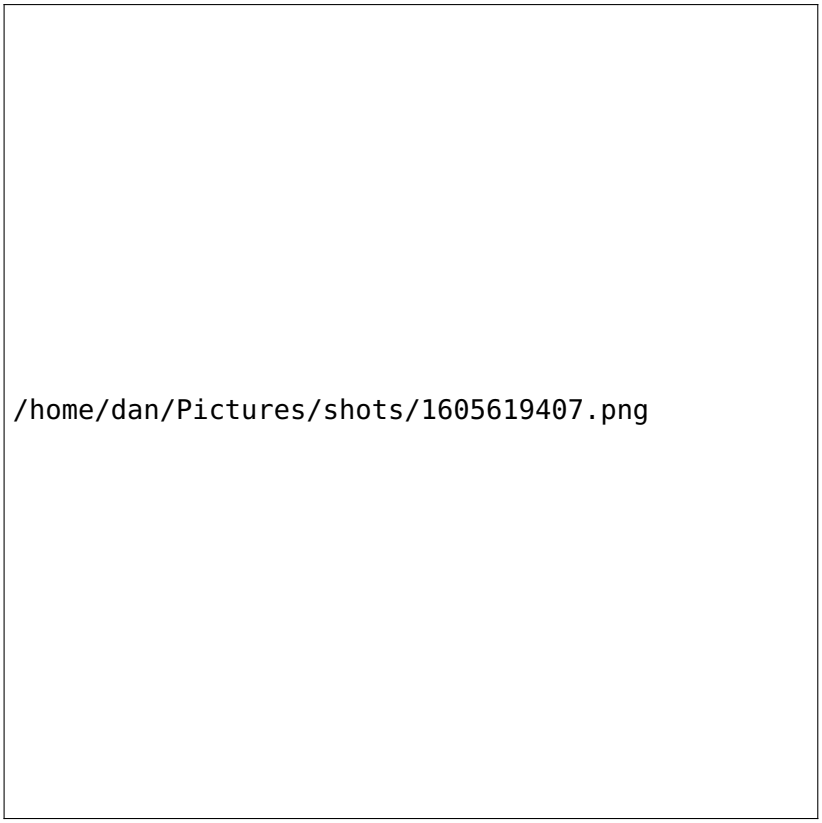
## 2.2 Analisi Sintattica

Vedi: Parser Top-Down e Codice Parser a Discesa Ricorsiva

- Prende input dall'analizzatore lessicale
- Crea un Albero Sintatico
  - che sará utilizzato poi dal Analizzatore Semantico
    * vedi Analisi Semantica Non affrontata nel corso
- In caso l'input non corrisponda ad un albero lessicale
  - deve restituire un errore
- Espressioni Booleane
  - RELOP $\in \{==, <>, <=, >=, <, >\}$
- Separatore

  - punto e virgola
    * non un terminatore di istruzione

- Produzioni




  - In caso di rami annullabili attenzione ai FOLLOW

- Sintassi scheme-like
  - espressioni aritmetiche
    * notazione prefissa
    * puó comprendere ID
    * operatori
      · * + : varianti n-arie: n>=1
      · - / : binarie
    * compaiono nelle espressioni booleane
      · impatto sull'insieme GUIDA
  - assegnamento
    * notazione prefissa
  - espressioni relazionali
    * notazione prefissa

## 2.3 Analisi Semantica

- Si occupa della valutazione delle espressioni

### *2.3.1 SDD*

`Syntax Directed Definition` Definizioni dirette dalla sintassi strumento che permette la `traduzione`

- consistono in
  - grammatica libera
    - specifica la sintassi
      - gli operatori qui sono sintattici
  - attributi
    - risultati della traduzione
      - sono riferiti dall'analizzatore lessicale
    - (nome, valore)
    - rappresentano una qualunque informazione associata ad un nodo
  - regole semantiche
    - come calcolare gli attributi
    - specificano regole di calcolo e assegnamento tra attributi per ogni produzione
      - gli operatori qui sono semantici/matematici
    - sono valutate in ordine arbitrario
      - richiedono la costruzione di un albero sintattico annotato

Con cui si definisce un `albero sintattico annotato`

- i cui nodi possono essere annotati con o o piú attributi

**ATTRIBUTI**

- Sintetizzati Il suo valore dipende da quello di attributi dei figli ed eventualmente da altri attributi di se stesso
- Ereditati Il suo valore dipende da quello dal padre e dai fratelli del nodo

**GRAFO DELLE DIPENDENZE** Alcuni attributi dipendono da altri, questo impone un'ordine tra questi

- se il grado contiene dei cicli non é possibile trovare un'ordine di valutazione degli attributi

**S-ATTRIBUITE** Contiene solo attributi sintetizzati

- ogni S-attribuita é a sua volta L-attribuita

**L-ATTRIBUITE** Per ogni produzione $A \to X_1X_2...X_n$ e ogni attributo ereditato $X_i.e$ la regola semantica che definisce il valore di $X_i.e$ dipende solo da

- attributi ereditati da $A$
- attributi sintetizzati ed ereditati dai simboli $X_1, X_2, ..., X_{i-1}$ alla sinistra di $X$

### *2.3.2 SDT*

`Syntax-Directed Translation scheme` Schema di traduzione, variante SDD che rende esplicito l'ordine di valutazione degli attributi

- grammatica in cui le produzioni sono arricchite da frammenti di codice
  - azioni semantiche
    - eseguite nel momento che i simboli alla loro sinistra sono stati riconosciuti
    - simili alle regole semantiche degli SDD
      - specificano il calcolo degli attributi ma anche codice arbitrario
      - l'ordine di esecuzione é esplicito a differenza delle regole semantiche
      - essendo eseguite da sinistra verso destra non richiedono la costruzione dell'albero sintattico annotato

**DA SDD L-ATTRIBUTE A SDT** data $A \to X_1X_2...X_n$

1. subito prima di $X_i$
   - azione semantica che calcola il valore degli attributi ereditati
     - che possono solo dipendere da attributi ereditati di $A$ e attributi dei nodi fratelli alla sua sinistra
2. in fondo alla produzione
   a) azione semantica che calcola il valore degli attributi sintetizzati di $A$

### 2.3.3 *Traduzione on the fly*

Attributi sintetizzati `principali`

- il loro valore include sempre la concatenazione dei valori dello stesso attributo per tutte le variabili nel corpo di ogni produzione oltre che eventuali variabili ausiliarie
- la concatenzazione rispetta l'ordine delle variabili nel corpo delle produzioni Es, trasformazione da forma infissa a postfissa

$E \rightarrow E_1 + T\{E.post = E_1.post||T.post||"..."\}$ Questo viene tradotto on the fly in { print("...") }

# 3 AUTOMI

## 3.1 Esempio

automa: riconosce stringhe stati finiti: memoria finita input: stringa output: "si" se riconosciuta "no" altrimenti

L'automa ha visione `locale e limitata` , legge un simbolo alla volta

L'automa altera il suo stato in base al simbolo letto

Se alla fine della stringa l'automa si trova in uno `stato finale` la stringa é accettata, altrimenti rifiutata

## 3.2 Automi a stati finiti deterministici `DFA`

Deterministico: lo stato in cui si sposta é univocamente determinato dallo stato corrente e dal input

Quintupla composta da:

1. Q - insieme finito di stati
2. $\Sigma$ - alfabeto riconosciuto
3. $\delta$ - funzione di transizione
4. $q_0$ - e' lo stato iniziale
5. F - insieme di stati finali

### 3.2.1 *Funzione di transizione estesa*

funzione definita su stringhe invece che singoli simboli definito per induzione

### *3.2.2 Linguaggio riconosciuto*

Stringhe definite sull'alfabeto che per mezzo della F di transizione estesa portano ad uno `stato finale` dell'automa

## 3.3 Automi a stati finiti non deterministici NFA

Non deterministico: l'automa puo' scegliere di spostarsi in 0 o piu' stati possibili

- Il codominio della funzione di transizione e' l'insieme delle parti degli stati Q

Quintupla composta da:

1. Q - insieme finito di stati
2. $\Sigma$ - alfabeto riconosciuto
3. $\delta$ - funzione di transizione il cui codominio e' un'insieme delle parti di Q
4. $q_0$ - e' lo stato iniziale
5. F - insieme di stati finali

   Insiemi singoletto indicano transizioni deterministiche (da funzione di transizione estesa) Automi che possono eseguire transizioni spontanee senza leggere alcun simbolo nella stringa da riconoscere

   - passa di stato anche senza consumare alcun simbolo

### *3.3.1 epsilon-chiusura*

calcolare l'insieme di stati raggiungibili solo con transizioni-epsilon `ECLOSE`

- la chiusura e' transitiva
- la chiusura di q include q ECLOSE(S) = Unione di ECLOSE($q_i$)

  Gli NFA sono un caso particolare di epsilon-NFA in cui non ci sono transizioni epsilon

  - il potere riconoscitivo degli epsilon-NFA e' almeno pari a quello dei DFA/NFA

**TEOREMA** Dato un eNFA E esiste un DFA D tale che L(D) = L(E)

## 3.4 Passaggio da DFA a NFA e viceversa

Da NFA a DFA sono possibili `fino` a $2^n$ stati

Da un DFA con piu' stati finali e' possibile ricavare un e-NFA equivalente con un unico stato finale

## 3.5 Espressioni regolari RE

Sono un approccio generativo alle classi di Linguaggi E' sempre possibile creare un e-NFA a partire da una RE

Denotano un Linguaggio con $L(E)$ Definito per induzione

$L(0) = \emptyset$ $L(\epsilon) = \{\epsilon\}$ / *la stringa vuota* $L(a) = a$ $L(E + F) = L(E) \cup L(F)$ $L(EF) = L(E)L(F)$ $L(E*) = L(E)^*$ / chiusura di Kleene

### *3.5.1 precedenza*

1. *
2. concatenazione
3. +

### *3.5.2 Proprietá*

#### UNIONE

- Commutativa
- Associativa
- Idempotenza
- Identitá

#### CONCATENAZIONE

- Associativa
- Identitá
- Assorbimento
- distributivitá

#### CHIUSURA DI KLEENE

- Idempotenza

## 3.6 Indistinguibilitá tra stati

`Equivalenza` (relazione riflessiva, simmetrica e transitiva) Due stati hanno lo stesso potere discriminante se presa una qualunque stringa del linguaggio si arriva ad uno stato finale in entrambi i casi o no in entrambi i casi, la indichiamo con ~

- Puó esserci una stringa che `distingue` i due stati
- Uno stato finale é distinto da altri stati non finali dalla stringa vuota

### *3.6.1 Minimizzazione di Automi*

si raggiunge un automa minimo: $(Q/\tilde{},\Sigma,\delta,[q_0],F/\tilde{})$ in cui $\delta([p],a) = [\delta(p,a)]$ Non esiste un automa corrispondente con meno stati dell'automa minimo

### *3.6.2 Equivalenza di Automi*

Puó essere usato l'algoritmo riempi tabella per decidere se due automi sono equivalenti Si crea l'unione dei due DFA: $A = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_1 \cup F_2)$ $\delta(q,a) = \delta_1 \cup \delta_2$ Se $q_1$ e $q_2$ risultano indistinguibili in $A$ allora $A_1$ e $A_2$ sono equivalenti

## 3.7 Automi a Pila `PDA`

`Approccio Riconoscitivo` Utilizza operazioni push e pop su una pila di dimensione illimitata

- Simbolo sentinella $Z_0$ che indica la fine della stringa, é il simbolo della pila con cui quest'ultima viene inizializzata
- Ad ogni lettura di un simbolo l'automa fa push(x) o push(b) dipendentemente dal Linguaggio
- La $\epsilon$ transizione finale puó eseguire solo se peek restituisce $Z_0$

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- $\Sigma$ = alfabeto di input
- $\Gamma$ = alfabeto della pila
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to p(Q \times \Gamma^*)$ = funzione di transizione

### 3.7.1 Descrizioni istantanee

Fissato un automa a pila P D.I. $= (q, w, \alpha)$

- stato in cui si trova l'automa
- ció che rimane da riconoscere nella stringa di input
- contenuto della pila dalla cima al fondo (sx a dx)

**MOSSE** relazioni da D.I. a D.I. $I \vdash_P J$ chiusura riflessiva e transitiva $I \vdash_P^* J$

### 3.7.2 Linguaggio Accettato

Per stato finale: $L(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \alpha), q \in F\}$ Per pila vuota: $N(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon)\}$

- Per stato finale il contenuto della pila nella D.I. finale é irrilevante
- Per pila vuoto lo stato nela D.I. finale puó non essere finale

In ogni caso la stringa di input deve essere consumata completamente

### 3.7.3 Automi a Pila Deterministici

`DPDA` Strettamente meno espressivi dei `PDA`

- riconoscono comunque ogni Linguaggio Regolare
- riconoscono i linguaggi liberi non ineremente ambigui

Dimostrabile:

1. Per ogni CFG G esiste un PDA P tale che $N(P) = L(G)$
2. Per ogni PDA P esiste una CFG G tale che $L(G) = N(P)$

I DPDA a paritá di stato simbolo letto e simbolo sulla pila possono fare al massimo una mossa.

- $\delta(q, a, X) \cup \delta(q, \epsilon, X)$ deve contenere al massimo un elemento

Mentre il linguaggio $ww^R$ non é riconoscibile in quanto fa uso chiave del non determinismo mentre $wcw^R$ é riconoscibile grazie al simbolo sentinella c

- Dim - Ogni linguagio regolare é riconosciuto da un DPDA
  - $A = (Q, \Sigma, \delta_A, q_0, F)$
  - $P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$

dove

- $\delta_P(q, a, Z_0) = \{(\delta_A(q, a, Z_0))\}$ per ogni $q \in Q, a \in \Sigma$
- $\delta_P(q, \epsilon, Z_0) = \emptyset$

Dimostrabile

1. Per ogni DPDA P esiste una grammatica libera non ambigua G tale che $L(G) = N(P)$
2. Il viceversa non vale

*La famiglia dei linguaggi riconoscibili da DPDA é inclusa in - ma non concide con - quella dei linguaggi generabili da grammatiche libere non ambigue*

### 3.8 Parser Top-Down

Vedi:File dedicato

## 4 GRAMMATICHE LIBERE

`Teorema`
Per ogni linguaggio regolare L esiste una grammatica G tale che $L(G) = L$

- dove $L(G)$ é il linguaggio generato da G
- le grammatiche possono generare tutti i linguaggi regolari
- possono anche generare linguaggi non regolari
  - stringhe palindrome
  - parentesi bilanciate

*I linguaggi liberi includono propriamente i linguaggi regolari*

### 4.1 LL(1)

### 4.2 Non LL(1)

#### *4.2.1 Fattorizzazione*

$A \to \alpha\beta_1|\alpha\beta_2$ quindi $\text{GUIDA}(A \to \alpha\beta_1) \cap \text{GUIDA}(A \to \alpha\beta_2) = / = \emptyset$
Soluzione Fattorizzare il previsso comune in una variabile a parte $A'$

### 4.2.2 Ricorsione immediata a sinistra

$A \to A\alpha|\beta$

Soluzione Nuova variabile $A'$ per spostare la ricorsione da sinistra a destra $A \to \beta A'$ $A' \to \epsilon|\alpha A'$

In generale l'eliminazione della ricorsione a sinistra non garantisce che la grammatica risultante sia LL(1)

### 4.2.3 Ricorsione indiretta a sinistra

$S \to Aa|b$ $A \to Ac|Sd|\epsilon$

Soluzione

1. si impone un ordine arbitrario alle variabili
2. considerando ogni variabile nell'ordine imposto si elimina la ricorsione immediata per quella variabile e si riscrivono le occorrenze di quella variabile che compaiono nei corpi delle produzione delle variabili seguenti

# 5 LINGUAGGI

## 5.1 Linguaggio regolare

Esiste almeno un Automa A che lo riconosce

### 5.1.1 Linguaggi Regolari

`def` Un Linguaggio riconoscibile da un `DFA`

**I LINGUAGGI REGOLARI SONO CHIUSI RISPETTO ALL'OPERAZIONE DI UNIONE** 'Collego' i due automi deterministici attraverso uno stato q0 che con epsilon-transizioni passa da uno o dall'altro

**I LINGUAGGI REGOLARI SONO CHIUSI RISPETTO ALL'OPERAZIONE DI CONCATENAZIONE** 'Collego' lo stato finale (che non sara' piu' finale) del e-NFA corrispondente al primo automa con quello iniziale di quello e-NFA del successivo, con una epsilon-transizione

**CHIUSURA dim** p- $L \cup L'$

- Dati $E_1$ e $E_2$
  - Si dimostra che $E_1 + E_2$ genera $L'$
  - Essendo quella ancora un'espressione regolare anche il linguaggio generato sará regolare

- $LL'$
- Simile all'unione
- $notL$
- $notL = \Sigma^* - L$
- si crea un automa $B = (Q, \Sigma, \delta, q_0, Q - F)$
  - abbiamo complementato l'insieme degli stati finali
- $iL \cap L'$
- Si utilizzano le leggi di De Morgan
  - ci si riconduce al caso dell'unione e della complementazione
- O si construisce un automa B che riconosce una simulazione dei due automi iniziali $A_1$ e $A_2$
- $L - L'$
- $L_1 - L_2 = L_1 \cap notL_2$
- $L^R$
  - L rovesciato
- Si ricava un $E^R$ per induzione
  $\emptyset^R = \emptyset$ $\epsilon^R = \epsilon$ $a^R = a$ $(E_1 + E_2)^R = {E_1}^R + {E_2}^R$ $(E_1E_2)^R = {E_2}^R{E_1}^R$ $(E^*)^R = (E^R)^*$ Facile poi dimostrare che $L(E^R) = L(E)^R$ Tutti questi sono ancora regolari

## 5.2 Linguaggi non Regolari

### *5.2.1 Pumping Lemma*

Per ogni linguaggio regolare L esiste $n$ appartenente a N tale che per ogni $w$ appartenente a L con $|w| >= n$ esistono $x, y, z$ tc $w = xyz$ :

1. $y \neq \epsilon$
2. $|xy| \leqslant n$
3. $xy^kz$ appartiene L per ogni $k \geqslant 0$ Abbiamo una stringa media $y$ non vuota che puó essere replicata un numero arbitrario di volte sempre ottenendo un Liguaggio Regolare.
   - Esempio
     - $L = \{a^kb^k \mid k >= 0\}$ non é regolare

**DIM**

- L regolare
- $A = (Q, \Sigma, \delta, q_0, F)$ tc $L = L(A)$
- $n = |Q|$
- $|w| >= n$ tc $w = a_1 a_2 ... a_m$ con $m >= n$
- Dopo $m$ passaggi lo stato $q_m$ deve essere `finale` per definizione
- Il numero di stati attraversati sará $m + 1$
- $m >= n$ implica $m + 1 > n$ quindi gli stati attraversati non possono essere tutti distinti
- $q_i = q_j$ ( $i < j$ ) é il primo `stato che si ripete` nel cammino dell'automa

Allora concludiamo identificando $x, y, z$

- $x = a_1 a_2 ... a_i$
- $y = a_{i+1} a_{i+2} ... a_j$
- $z = a_{j+1} a_{j+2} ... a_m$
- $y! = \epsilon$ in quanto $i < j$
- $|xy| <= n$ in quanto $q_i = q_j$ é il primo stato che si ripete e sono al massimo $n + 1$
- $xy^k z$ appartiene a L per ogni $k >= 0$

## 5.3 Linguaggi Liberi dal Contesto

Le grammatiche libere sono un approccio generativo alle stringhe $L = a^n b^n \mid n \in$ non e' regolare:

- e' il inguaggio delle parentesi bilanciate

$G = (V, T, P, S)$ e' una `grammatica libera`

- V variabili o simboli non terminali
- T terminali
- P produzioni $A \to \alpha$
  - testa
  - corpo

        * La riscrittura della $A$ in $\alpha$ (sequenza arbitraria di simboli terminali o non) é libera dal contesto

- $S$ simbolo iniziale

`Derivazioni`:

- derivazione in un solo passo
- derivazione in zero o piu' passi

Il potere riconoscitivo delle grammatiche libere e' almeno tanto quanto quello dei linguaggi regolari

`Derivazioni canoniche`

- leftmost
  - $\Rightarrow_{lm}$
- rightmost
  - $\Rightarrow_{rm}$

  Se esistono due derivazioni canoniche distinte (entrambe `lm` o `rm`) per la stessa stringa allora G e' `ambigua`

### 5.3.1 Alberi Sintattici

Derivazioni differenti possono generare lo stesso programma

- anche imponendo regole all'ordine delle riscritture

Gli alberi sintattici (alternativa alle generazioni) astraggono dall'ordine delle riscritture e permettono di ragionare sulla `struttura` delle stringhe

- grammatiche ambigue
  - piú alberi con lo stesso prodotto
  - non é avere derivazioni distinte che mi porta ad alberi diversi e quindi ambiguitá

Data una grammatica $G = (V, T, P, S)$ gli alberi sintattici di G:

- ogni nodo etichettato con una var in $V$
- ogni foglia etichettata da $V$ o $T$ o $\epsilon$
- $\epsilon$ significa unico figlio del genitore
- se un nodo $A$ i suoi figli sono etichettati (sx a dx)
  - $X_1, X_2, ..., X_n$
  - $A \to X_1, X_2, ..., X_n$ e' una produzione in $P$

Il `prodotto` é la stringa ottenuta concatenando(sx verso dx) le etichette di tutte le foglie

**TEOREMA** $A \rightarrow_G^* \alpha$ se e solo se esiste un albero sintattico di G con radice $A$ e prodotto $\alpha$

#### RISOLUZIONE DELLE AMBIGUITÁ (GRAMMATICHE IN FORMA INFISSA)

- `Precedenza` degli operatori
- `Associativitá` degli operatori
  - per operatori associativi questo non é un problema
  - lo é per altri operatori

`Soluzione ad hoc` Utilizziamo associativitá a sinistra, sbilanciamo le espressioni e le stratifichiamo

- Espressione = somma di termini
- Termine = prodotto di fattori
- Fattore = costante o espressione tra parentesi

Nuova grammatica: $(\{E, T, F\}, \{0, 1, ..., 9, +, *, (, )\}, P, E)$ Produzioni:

- $E \rightarrow T \mid E + T$
- $T \rightarrow F \mid T \times F$
- $F \rightarrow 0 \mid 1 \mid ... \mid 9 \mid (E)$

#### LINGUAGGI INERENTEMENTE AMBIGUI

$$L = \{a^n b^n c^m d^m \mid n \geqslant 1, m \geqslant 1\} \cup \{a^n b^m c^m d^n \mid n \geqslant 1, m \geqslant 1\}$$

Qualunque Grammatica che genera L ha sempre almeno due derivazioni canoniche distinte che generano una stringa della forma

$$a^n b^n c^n d^n$$

### 5.3.2 *Pumping Lemma*

### 5.3.3 *Chiusura*

**UNIONE & CONCATENAZIONE** SI dati $L_1 = L(G_1)$ e $L_2 = L(G_2)$ dove $V_1 \cap V_2 = \emptyset$ costruiamo la grammatica $(V_1 \cup V_2, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S)$ che genera $L_1 \cup L_2$ e la grammatica $(V_1 \cup V_2, T_1 \cap T_2, P_1 \cap P_2 \cap \{S \rightarrow S_1 S_2\}, S)$ che genera $L_1 L_2$

**INTERSEZIONE** NO tra 2 Linguaggi Liberi $L_1 = \{a^n b^n c^m \mid \geqslant 0\}$ $L_1 = \{a^m b^n c^n \mid \geqslant 0\}$ Sono liberi ma $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geqslant 0\}$ Non é libero, dimostrabile con il pumping lemma SI tra linguaggio Libero e linguaggio Regolare NB: L'intersezione non é piú un linguaggio regolare es. $L = \{a^n b^n \mid n \geqslant 0\}$ e $R = L(a^* b^*)$ $L \cap R = L$ il quale non é regolare

**COMPLEMENTO & DIFFERENZA** NO Se fossero chiusi per complemento allora $L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$ Contrario a ció dimostrato Il complemento é esprimibile per differenze e quindi nemmeno la differenza é chiusa

**INVERSIONE** SI $G^R = (V, T, P^R S)$ dove $P^R = \{A \to \alpha^R \mid A \to \alpha \in P\}$ Si dimostra che $\overline{L}(G^R) = L(G)^R$

# 6 JVM

Vedi: IJVM, Bytecode Instruction Listing Progetto: Translator.java

- Interprete `bytecode`
- macchina virtuale basata su `pila`
- basso e alto livello (gestione della pila / oggetti)
- `Garbage Collector`

Pipeline del corso: .lft → .j → .class → output

## 6.1 Pila

Composta da Frames

- uno per ogni metodo in esecuzione
  - `NB` I metodi non statici hanno come primo argomento il riferimento all'oggetto ricevente
- argomenti e variabili riferite con il loro indirizzo nella pila
- `Instruction Set` *Gestione della Pila*
  - istore
  - iload
  - swap

  *Aritmetica*
  - ineg
  - iadd
  - isub
  - imul

  *Gestione Array*
  - newarray

- arraylength
- iaload
- iastore

*Controllo del Flusso*

- goto
- $\text{if}_{\text{icmpeq}}$
- $\text{if}_{\text{icmpne}}$
- $\text{if}_{\text{icmple}}$
- $\text{if}_{\text{icmpge}}$
- $\text{if}_{\text{icmplt}}$
- $\text{if}_{\text{cmpgt}}$
- invokestatic
- return
- ireturn

## 6.2 Espressioni

### *6.2.1 Aritmetiche*

### *6.2.2 Logiche*

Implementazione di `Valutazione Corto-Circuitata`

## 6.3 Problemi

la compilazione di un metodo comporta il calcolo della `dimensione del suo frame`

- variabili locali
- pila degli operandi

inoltre deve assicurarsi che se il `tipo di ritorno` é diverso da void ci sia un valore restituito Questo senza eseguire il codice, utilizzando l'$_{\text{analisi}}$ statica del codice_ Nello sviluppo ci occupiamo di

- metodi statici
- con tipo di ritorno int o void

### 6.3.1 Verifica del Return

Analisi di ogni cammino per verificare che alla fine di ogni metodo ci sia una istruzione return

- l'analisi é statica in quanto non tiene conto dell'effettivo flusso di esecuzione del metodo
  - non garantisce che il return sia eseguito
    - in caso di ciclo infinito
    - in caso di eccezione

Vengono fatte delle `approssimazioni`:

- non sono valutate `espressioni booleane` anche se banali: il problema é `indecidibile`
- non viene controllato se il tipo di ritorno é giusto o meno
  - necessita un'altra analisi dei tipi

Questo é implementato con un attributo

- `S.ret`
  - true se l'espressione di S termina é perché esegue una return
  - in caso di liste di Comandi
    - l'attributo é determinato dall'OR tra i Comandi che compongono la lista:
      - questa informazione puó essere utile per individuare la presenza di codice morto
      - warning o errore

### 6.3.2 Allocazione delle variabili locali

Il piú piccolo numero di slot necessari all'interno di un frame per la memorizzazione di argomenti e variabili locali

- determinare il numero massimio di variabili che sono *contemporaneamente* attive
  - tener conto della localitá delle variabili

Questo é implementato con un attributo

- `S.locals`
  - max{ S1.locals, S2.locals }
    - nel caso di *if else* o *liste di comandi*

### 6.3.3 Calcolo dimensione massima della pila

Numero massimo di slot occupati sulla pila degli operandi durante l'esecuzione di un metodo

- tenendo conto del codice prodotto
  - approssimare per eccesso la dimensione massima della pila

Implementato con l'attributo *stack* per `E`, `B`, `S`

- `E.stack`
  - >= 1
- `E_list.stack`
  - >= 0

*NB* L'`associativitá a sinistra` mantiene la `pila piccola` perché le sottoespressioni vengono valutate man mano che si incontrano da sinistra verso destra