LFC (Linguaggi Formali e Compilatori) - Note del Corso

Edoardo Lenzi

January 17, 2018

Contents

Chapter 1

Introduzione

Un compilatore é un programma che legge un **linguaggio source** e lo traduce in un `equilvalente` **linguaggio di programmazione target**. Solitamente il compilatore compila in `assembly` e poi un **assembler** produce codice macchina. Se il target language é un programma eseguibile puó processare input e produrre output.

Un **interprete** é un altro tipo di language processor, invece di tradurre il linguaggio lo esegue direttamente quindi piglia sia il source program che gli input e processa l'output

Infine il **preprocessore** risolve le macro nel sorgente codificandole in linguaggio nativo (espandendole) prima di compilare.

> Solitamente il compilato va piú veloce mentre l'interprete ti da diagnosi piu accurate dato che esegue il codice. Nel caso di Java compilo il sorgente in linguaggio intermedio `bytecode` che poi interpreto sulla JVM.

Il **linker** "linka" assieme moduli e librerie dove ho riferimenti ad altri file (risolve gli indirizzi). Il **loader** invece fa il merge in memoria per l'esecuzione.

1.1 Front-End of the Compiler

La **parte analitica** del processo di compilazione spacca la sorgente in parti costituenti e impone su di esse una struttura grammaticale (stile dtd); sfrutta questa struttura per creare una rappresentazione intermedia. Se non passa la validazione grammaticale mi tira errori. Il sorgente viene storicizzato in una struttura dati chiamata **symbol table**.

1.2 Back-End of the Compiler

La **parte di sintesi** invece traduce il sorgente guardando la rappresentazione intermedia e la symbol table; le parti di analisi e sintesi sono chiamate anche **front-end of the copiler** mentre le restanti **back-end**.

1.3 Lexical analysis

Fa uno scan e raggruppa le parole in **lexems**, per ogni lexem genera un **token** della forma

`(token name, attribute value)`

Il `token name` é un simbolo astratto usato nella syntax analysis mentre il `value` é un puntatore alla symbol table entry.

ie)

`position = initial + rate * 60` diventa `(id, 1) (=) (id, 2) (+) (id, 3) (*) (60)` gli operatori matematici sono simboli astratti che non hanno attribute value (?non sono referenziati nella symbol table?).

1.4 Session syntax analyzer

É un parsing, con i token crea una **rappresentazione ad albero (syntax tree)** nel quale il nodo é un operatore e i figli gli operandi.

gli operatori devono avere prioritá per costruire l'albero, la struttura grammaticale serve anche a definire le prioritá degli operatori.

1.5 Semantic analyzer

Piglia il `syntax tree` e guarda se é semanticamente consistente con la definizione del linguaggio. (ie `type checking`). Il linguaggio puo ammettere cast impliciti chiamati **coercizioni** o tirare cogne.

"*intofloat*"é una coercizione dell'intero 60 in float dato che gli altri operandi sono float.

1.6 Intermediate code generation

Nel processo di compilazione posso avere varie rappresentazioni intermedie come alberi etc.. Dopo semantic analysis solitmente creo un codice basso livello, machine-like, *"easy to produce and esay to translate int target machine code"*. Nella figura ho un tree address code ricavato dal syntax tree.

In un tree address code a destra ho al massimo un operatore (assembly like), e le operazioni sono in ordine.

Devo avere variabiline intermedie

1.7 Code generation

Segue la fase opzionale di **code optimization**, prende la rappresentazione intermedia e la mappa in un target language. Le istruzioni intermedie vengono tradote in istruzioni macchina (presumibilmente). Devo capire come mappare variabili su registri

Nella symbol table devo storicizzare tutti gli attributi di un variable name.

Solitamente posso agglomerare le fasi di analisi in front end pass e le altre in back end pass.

Chapter 2

2

Syntax descrive la forma appropriata di un programma Semantic descrive il significato...

Per specificare la sintassi uso BNF (Backus Naur Form) context-free grammar

Analisi consiste nel (guardando la **sintassi**) spaccare il sorgente in parti costituenti (lexems) e generare tokens che li rappresentano (ho un linguaggio intermedio). La **Sintesi** invece parte dal linguaggio intermedio e sintetizza il target program.

Per specificare la **sintassi** uso la notazione della **context-free grammar** o BNF (Backus Naur Form).

2.1 Syntax Analysis

Vedo una sequenza di caratteri come entitá chiamate tokens

Creo un **abstract syntax tree** con entitá sulle foglie e operatori sugli altri nodi (intermedi).

assign **three address instruction** per via del fatto che ho tre variabili (istruzione assembly).

a +

b c

```
if(expression) statement else statement

stmt -> if(expr) stmt else stmt
// -> la traduco in "can have the form"
```

La regola sopra puó essere chiamata **produzione**.

In una **produzione** elementi lessicali come if e parentesi (keywords) cono chiamati **terminali** mentre le variabili sono **non terminali** (ulteriormente espandibili con produzioni).

Una **context-free grammar** ha:

- un **set di terminali** (tokens), set di simboli elementari del linguaggoi definiti dalla grammatica
- un **set di non terminali** o syntactic variables
- un **set di produzioni** ($Head \to Boby$)
  head é il costrutto, body la forma scritta del costrutto
- un non terminale chiamato **start symbol**

In un compilatore il lexical analyzer legge i caratteri del sorgente, li raggruppa in lexems e produce tokens (della forma **(TokenName, AttributeValue)**).

Specifico, nella pratica, una grammatica come lista di produzioni (con quelle contenenti lo start symbol per prime). Simboli come $<$, $>$, $=$ e le keyword del linguaggio sono terminali.

Per convenzione scrivo in *italic i non terminali* ed in **boldface per i terminali**. Uso l'operatore OR | (pipeline) per separare gli elementi nel body. Definisco ε come empty string.

2.1.1 ie)

ho $5 + 9 - 3 + 5 - 6 - 7 + 1$

```
list -> list + digit | list - digit | digit
digit -> 0|1|2|...|9
```

I terminali sono $\{+ - 0123...9\}$, i non terminali $\{list,\ digit\}$, start symbol é *list*

2.2 Derivations

Una grammatica deriva stringhe partendo dallo start symbol e ricorsivamente applicando le produzioni sui non terminali. Il linguaggio definito da una grammatica é il {stringhe ottenute}.

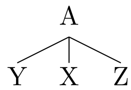
2.2.1 ie)

```
//argomenti di una funzione
call -> id(optparams)
optparams -> params | Epsilon
params -> params, param | param
```

2.3 Parsing

Il **parsing** é il problema secondo cui, data una stringa di terminali, devo capire come é stata costruita partendo da uno start symbol (tirare eccezione altrimenti).

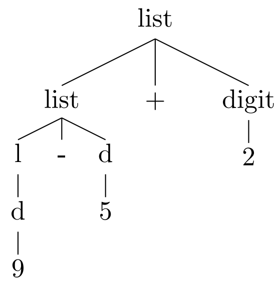
Uso **parse trees** $A \to XYZ \implies$



Regole di costruzione:

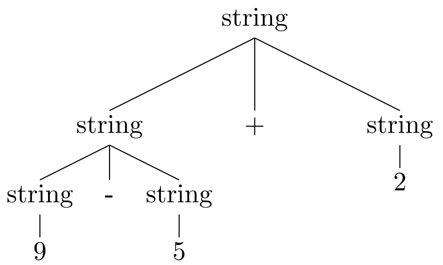
- la root é lo start symbol
- le foglie sono terminali o ε
- i nodi interni sono non-terminali
- se A non-terminali ha figli $x_1, ..., x_n \implies A \to x_1...x_n$

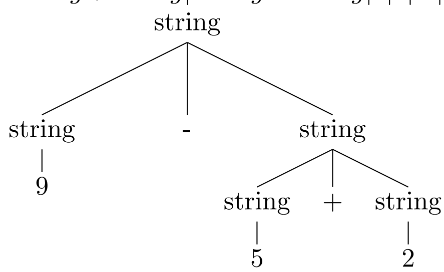
9-5+2



Una stringa puó avere piú parse tree ma ció implica che la **grammatica é ambigua**; la presenza di piú alberi implica l'esistenza di piú significati diversi.

Mergiando le nozioni di list e digit ottengo $string \to string + string | string - string | 1|2|...|9$ 9-5+1 ha





2.3.1 Associativitá a sinistra

L'operatore + assicia a sinistra perché se ho un pezzo di espressione +5+ il + a sinistra viene applicato al 5 mentre ad esempio per l'elevamento a potenza o l'assengnazione di una variabile (=) l'associativitá é a destra (right associative).

```
a = b = c -> a = (b = c)
2^3^4 -> (2^3)^4
1 + 2 + 3 -> (1 + 2) + 3
```

Stringhe right associative (l'abero crese a destra) sono generate dalla grammatica:

```
right -> letter = right | letter
letter -> [ab...z]
```

2.4 Precedenze degli operatori

L'associativitá vale per operandi uguali ma non risolve $a + b * c$. In questo caso ho due livelli di precedenza uno per $+-$ ed uno per $*/$. Creo *expr e term* per i due livelli e *factor* per i base units.

```
factor -> digit | (expr)
term -> term * factor
       | term / factor
       | factor
expr -> expr + term
       | expr - term
       | term
```

La grammatica sará quindi

```
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digit | (expr)
// non posso avere un operatore vicino ad un fattore
```

> Posso generalizzare il concetto per n livelli di precedenza (per n livelli mi servono n+1 non terminali)

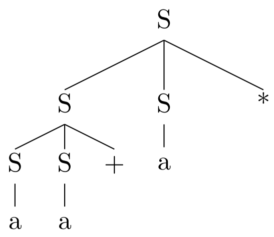
expr = list {terms separati da $*/$}

2.5 Java statements

```
stmt -> id = expr
       | if (expr) stmt
       | if (expr) stmt else stmt
       | while (expr) stmt
       | do stmt while (expr)
       | {stmts}
stmts -> stmts stmt
       | Epsilon
```

2.5.1 ie) notazione postfissa

```
//prefissa con   SS -> +SS|*SS|a
SS -> SS+|SS*|a
genera aa+a*
S->SS*-> Sa*->SS+a*->aa+a*
```



2.5.2 ie)

```
S -> 0S1 | 01
000000111111
0^n 1^n

S -> S(S)S | E
S -> S(S)S -> S(S)S(S)S -> S(S(S(S(S)S)S)S)S(S)S(S)S(S)S
E(E(E(E(E)E)E)E)E(E)E(E)E(E)E
avro' sempre una E(E all'inizio ed una E)E alla fine

S -> aSbS | bSaS | E
avro' sempre un numero uguale di a e b, lunghezza totale pari
```

2.6 Syntax Directed Translation

Fatte attaccando regole o frammenti di programma a produzioni.

Attributi sono proprietá di espressioni del linguaggio (length).

Syntax Directed Translation Schemes

2.6.1 Postfix notation

(ab+), 9-5+2 diventa 95-2+, 9-(5+2) diventa 952+-. La notazione postfissa non necessita di parentesizzazioni, non puó avere ambiguitá. Per leggere l'espressione faccio uno scan da destra fino al primo operatore e lo applico (vado avanti ricorsivamente).

Syntax Directed Definition associa

- ad ogni simbolo un set di **attributes**

- ad ogni produzione un set di **semantic rules** per computare i valori degli attributi associati ai simboli che compaiono nella produzione

Per una stringa x faccio un **parse tree** poi applico le regole semantiche per valutare gli attibutes ad ogni nodo dell'albero. x.a é l'attributo a di x $\rightarrow$ **annoted parse tree**

Un attributo é detto **sintetizzato** se il suo valore in un nodo é determinato dai valori dei suoi figli e dal nodo stesso. Un attributo é detto **inherited** se il suo valore in un nodo é determinato dai valori del padre, dei fratelli e di se stesso.

Uso l'operatore || per concatenare le stringhe. Le regole semantiche sono applicate agli attributi.

```
expr -> term   =>  expr.t = term.t
expr -> expr1 + term  => expr.t = expr1.t || term.t || '+'
// l'attributo nella head = attributi nel body concatenati con strighe extra ('+')
```

2.7 Tree traversals

Chapter 3

Link

Vecchio sito
Note Drive

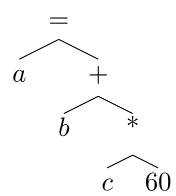
Chapter 4

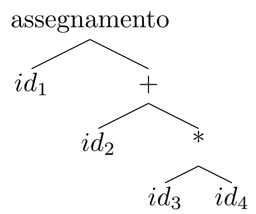
Introduzione

I linguaggi di **analisi lessicale** lavorano con simboli e caratteri; devo costruire una **tavola dei simboli** (specifica per un dato programma e compilatore). L'analisi restituisce dei **tokens** (puntatori) a record nella tavola dei simboli. La maggior parte delle implementazioni usano un numero come **identificatore**.

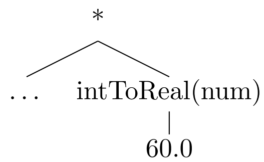
L'analisi sintattica invece studia la **grammatica del linguaggio**. Viene costruito un **abstract syntax tree**:

$$a = b + c \cdot 60$$

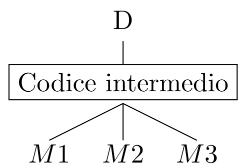




L'analisi semantica si occupa di vedere se c'é una corretta semantica (variabili dichiarate precedentemente). Se * necessita di un float allora 60 dev'essere convertito a float.



Generazione di **codice intermedio**:
$$\begin{array}{ll|l} temp_1 = & \text{intToReal}(60) & \text{VISITA DELL'ALBERO} \\ temp_2 = & id_3 * temp1 & \\ temp_3 = & id_2 + temp2 & \\ id_1 = & temp_3 & \end{array}$$



4.1 Grammatiche

Una **grammatica** é una tupla G(V, T, S, P) con:

V	vocabolario
T	set simboli terminali
S	start symbol
P	set delle produzioni
V\T	simboli
ε	parola vuota, non puó essere un terminale!

Le produzioni sono della forma $\alpha \rightarrow \beta$ con α **stringa non vuota di simboli V con almeno un non terminale**, β stringa su V o ε.

> Per convenzione i caratteri in maiuscolo denotano simboli non terminali mentre in minuscolo terminali. Quindi i simboli in T sono tutti lettere minuscole.

Considero X, Y variabili, generico simbolo in V e α β δ stringhe su V^*

$S \to aSb$
$S \to \varepsilon$ $\quad T = \{a, b\}$ (terminali), $(V \backslash T) = \{S, A\}$ (non terminali)
$S \to A$

4.1.1 Derivazioni

Date $\mu = \mu_1\alpha\mu_2,\ \alpha \to \beta$, produzioni della grammatica G, $\gamma = \mu_1\beta\mu_2$ é una derivazione in uno o piú passi partendo da μ.

Scrivendo $\mu \to^+ \gamma$ intendo che posso derivare γ da μ in uno o piú passi di derivazione.

4.1.2 Linguaggio generato da una grammatica

$$L(G) = \{w \ / \ w \in T^*, S \to_G^+ w\}$$

Dato il linguaggio L possono esistere piú grammatiche diverse tra loro che generano L. Pertanto dal linguaggio non posso risalire con certezza alla grammatica.

> In generale dato un linguaggio generale L ed una grammatica G $\nexists$ un algoritmo per dimostrare che L = L(G).

Chapter 5

Linguaggi liberi

5.1 Grammatica libera dal contesto (context free)

Una grammatica generata é libera dal contesto (context free) se ogni sua produzione ha la forma:

$$A \to \beta$$

Con A non terminale. ($\alpha \to \beta$, α deve essere un solo simbolo non terminale, altrimenti é condizionata). Con β qualsiasi (terminali, non terminali o ε).

Grammatiche libere si prestano in modo naturale a descrivere derivazioni in viste ad albero.

5.1.1 Esempi

$S \to aAb$
$aA \to aaAb$ Non é context free, genera $L(G) = \{a^n b^n \ / \ n > 0\}$.
$A \to \varepsilon$

$S \to aSb|\varepsilon$
$S \to aSb \to aaSbb \to aabb \implies \{a^n b^n \ / \ n \geq 0\}$ Context free, genera lo stesso L di quella sopra.
Serve per parentesizzare correttamente codice (genera $a^n b^n$).

$S \to AB$
$A \to Aa|a$ Tutto ció che deriva da A é indipendente da ció che deriva da B. $a^n b^m, \ n, m \geq 0$
$B \to Bb|b$

Se ho produzioni impossibili che non finiscono in terminali la grammatica genera un linguaggio vuoto ($(\{S, B, a\}, \ \{A\}, \ S, \ \{S \to aB\})$).

$(\{S\}, \ \{\}, \ S, \ \{S \to \varepsilon\})$ invece genera un linguaggio $\{\varepsilon\} \neq \emptyset$

$S \to 0B|1A$
$A \to 0|0S|1AA$ L(G) = {w / count(0,w)=count(1,w) }
$B \to 1|1S|0BB$

Definire una grammatica per $L = (a^k b^n \ / \ k, n > 0)$

$S \to aS|aB$
$B \to bB|b$

$S \to AB$
$A \to aA|a$
$B \to bB|b$

$S \to ab|aS|Sb$

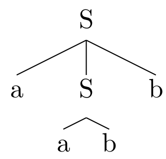
Definire G tale che $L(G) = \{a^k b^n c^{2n} \ / \ k, n > 0\}$
$S \to AB$
$A \to aA|a$
$B \to bBcc|bcc$

Definire G tale che $L(G) = \{a^k b^n d^{2k} \ / \ k, n > 0\}$
$S \to aSdd|B$
$B \to bB|b$

$S \to aSdd|aBdd$
$B \to bB|b$

5.2 Abstract syntax Tree

$$S \to aSb|ab$$



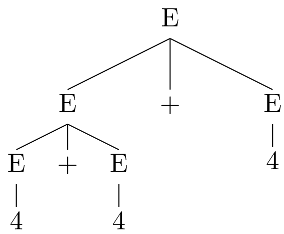
aabb, derivazione canonica $\mu \to \gamma$

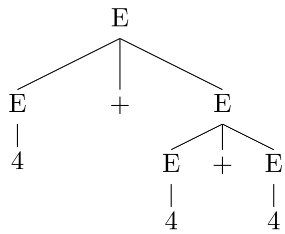
5.3 Grammatiche ambigue

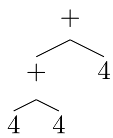
Nel caso di grammatiche libere si definiscono le **derivazioni canoniche destre e sinistre**, nel caso **rightmost** si richiede che ad ogni passo di derivazione $\mu \to \gamma$ venga rimpiazzato il terminale piú a destra di μ; nel caso **leftmost** quello piú a sinistra.

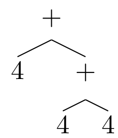
> G é **ambigua** se esiste una parola del linguaggio generato da G, generabile con due derivazioni canoniche distinte entrambe destre o entrambe sinistre.

$E \to E + E|E * E|4$ (il + associa a sinistra)









[Analogo con il *] Essendo derivazioni entrambe leftmost **G é ambigua**.

> Occhio a non confondere la struttura dell'albero di derivazione con la sua sequenza di derivazioni.
> La derivazione leftmost sostituisce prima i non terminali a sinistra e poi procede con i successivi.
> Quello a sx prima spacca la E in E+E che poi viene spaccato in E+E dove poi vanno sostituiti alle E i non terminali 4.
> Quello a dx invece spacca E in E+E, sostituisce alla prima E il 4 e poi passa alla sostituzione della seconda E con E+E.
> In entrambi i casi la sostituzione dei non terminali avviene sempre prendendo il primo non terminale della stringa, cioé quello piú a sinistra.
>
> Finché considero sostituzioni con un solo carattere a destra della produzione leftmost e rightmost sono del tutto equivalenti; la differenza arriva quando considero produzioni con sostituzioni su piú di un carattere perché mangi caratteri a possibili derivazioni future.

$S \to if\ b\ then\ S \mid if\ b\ then\ S\ else\ S \mid altro$
$if\ b\ then\ if\ b\ then\ altro\ else\ altro$
$if\ b\ then\ S\ else\ S$

5.4 Linguaggi liberi

> Un linguaggio é libero se esiste una grammatica libera che lo genera.

Lemma 1: La classe dei linguaggi liberi é **chiusa all'unione** (se L_1 e L_2 sono liberi allora $L1 \cup L2$ é ancora libero)

$$\begin{aligned} L_1 \ libero &\implies \exists G_1 = (V_1, T_1, S_1, P_1) \ / \ L_1 = L(G_1) \\ L_2 \ libero &\implies \exists G_2 = (V_2, T_2, S_2, P_2) \ / \ L_2 = L(G_2) \\ (\{V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, & S, P_1 \cup P_2 \cup (S \to S_1 | S_2)\}) \end{aligned} \tag{5.1}$$

Devo rinominare i non terminali di G1 e G2 in modo da non avere omonimie (clash). Notare la produzione che agginge un nuovo start symbol per agganciarsi ai vecchi.

Lemma 2: La classe dei linguaggi liberi é **chiusa rispetto alla concatenazione** (se L1 e L2 sono liberi allora $\{w_1 w_2 \ / \ w_1 \in L_1, \ w_2 \in L_2\}$ é esso stesso un linguaggio libero).

$$G = (V_1 \cup V_2' \cup \{S\}, \ T_1 \cup T_2, \ S, \ P_1 \cup P_2' \cup \{S \to S_1 S_2'\})$$

$P_1 \cup P_2 \cup \{S \to S_1 S_2\}$ anche in questo caso bisogna stare attenti ad eliminare possibili clash dei simboli non terminali di G_2. Metto un apice per indicare una rinominazione dei non terminali ($V_2', \ S_2', \ P_2'$). In pratica concateno gli start symbols.

5.4.1 Esempio

Il linguaggio $\{a^n b^n \ / \ n > 0\}$ é libero perché $\exists$ una grammatica libera che lo genera (G_1):

$$\begin{array}{cc} G_1 & S \to aSb/ab \ G_1 \text{ libera} \\ G_2 & s \to aAb, \ A \to aaAb, \ A \to \varepsilon \ G_2 \text{ non libera } \square \end{array}$$

5.5 Pumping Lemma per Linguaggi Liberi

> Serve per dimostrare che un linguaggio non é libero. Ossia non esiste alcuna grammatica libera che lo genera.

Pumping lemma:
Sia L un linguaggio libero allora $\exists p \in \mathbb{N}^+ \ / \ \forall z \in L \ / \ |z| > p, \exists uvwxy$:

i) $z = uvwxy \ \wedge$

ii) $|vwx| \geq p \ \wedge$

iii) $|vx| > 0 \ \wedge$

$\forall i \in \mathbb{N} \ / \ uv^i wx^i y \in L$

5.5.1 Commento definizione

$\exists \ p \in \mathbb{N}^+$	esiste una costante $p > 0$
$\forall \ z \in L : \ |z| > p$	ogni parola con piú elementi di p
$\exists \ uvwxy \ / \ z = uvwxy$	esistono 5 sottostringhe che compongono z
$|vwx| \leq p$	la lunghezza delle 3 stringhe centrali é minore di p
$|vx| > 0$	la seconda e la quarta non sono mai entrambe nulle
$\forall \ i \in \mathbb{N} uv^i wx^i y \in L$	se ripeto i volte (i puó essere 0) la 2 e la 4 sono ancora in L

5.5.2 Dimostrazione Pumping Lemma

Osservazione 1: Data una grammatica G posso sempre creare una altra grammatica G' modificata dalla prima che genera lo stesso linguaggio.

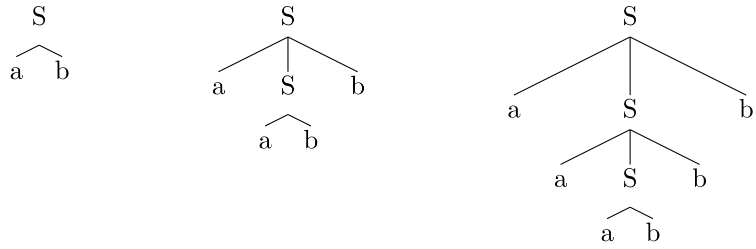
Osservazione 2: Una grammatica si puó portare in forma normale (di Chomsky) togliendo eventuali ridondanze o produzioni che terminano per forza con ε (a meno che non debba fare parte del linguaggio, ma a quel punto si scrive come $S \to \varepsilon$).

CNF

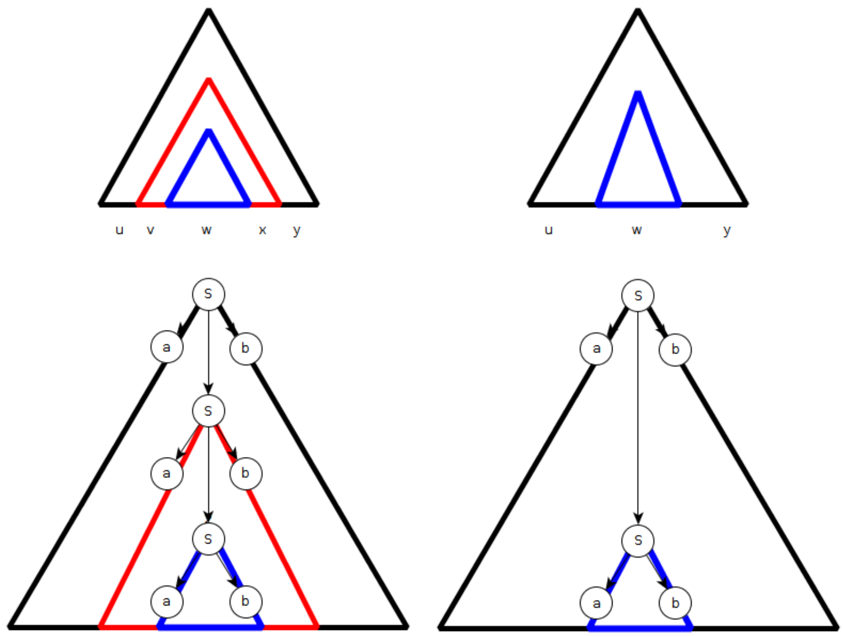
Chomsky Normal Form esige che una grammatica non abbia produzioni ridondanti:
ie) G_1 $S \to aSb|ab|B,\ B \to aBb|ab \leftarrow$ doppioni

Dimostrazione: L é un linguaggio libero $\implies \exists\, G$ in Chomsky Normal Form tale che L = L(G).

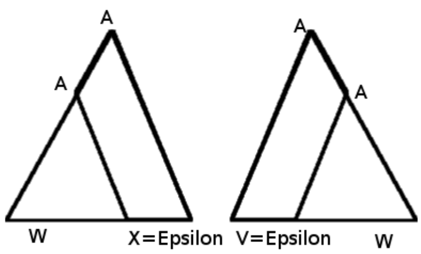
Definisco p come la lunghezza della parola piú lunga che puó essere derivata usando un albero di derivazione i cui cammini dalla radice sono lunghi al piú come il numero di simboli non terminali della grammatica ($|V \backslash T|$).
$S \to aSb|ab$, ha due non terminali, p = 2



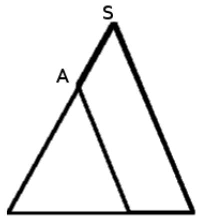
Guardo il cammino S $\to S_1 \to S_2 \to ... \to S_K$ di lunghezza K Se $z \in L \ \wedge\ |z| > p \implies$ nell'albero di derivazione di z esiste almeno un cammino la cui lunghezza é maggiore di $|V \backslash T| \implies$ allora esiste almeno un non-terminale che occorre almeno due volte lungo quel cammino (S, nell'esempio sotto).



ie) p = 2, se prendo una qualunque parola piú lunga di 3 generata da G: aaaabbbb, la posso dividere in 5 con due pumpable: u=aa, v=a, w=abb, x=b, y=b
Se prendo $z \in L \wedge |z| > p \implies$ ho dovuto usare un albero di derivazione $/\ \exists$ al meno un cammino piú lungo di $|V \backslash T|$ per definizione di z. $\implies$ ho un non terminale ripetuto al meno due volte $\implies \exists$ al meno un non terminale che occorre al meno 2 volte lungo quel cammino
con **l'un-pumping** la parola sta sempre nel linguaggio (**taglio un pezzo di albero**) Dato che w e x non possono essere entrambi nulli al massimo avró $A \to aA,\ o\ A \to Aa$



ie) linguaggio libero $\{a, b\},\ S \to ab$



5.6 Pumping Lemma per assurdo

(tesi) = L libero $\implies \exists p \in \mathbb{N}^+ \ / \ \forall z \in L \ / \ |z| > p, \exists uvwxy :$

i) $z = uvwxy \wedge$

ii) $|vwx| \geq p \wedge$

iii) $|vx| > 0 \wedge$

$\forall i \in \mathbb{N} \ / \ uv^iwx^iy \in L$

$\neg(tesi) = \forall p \in \mathbb{N}^+ \ / \ \exists z \in L \ / \ |z| > p \ \forall \ uvwxy \ / \ \ [(z = uvwxy \wedge |vwx| \leq p \wedge |vx| > 0)$
$\implies \exists i \in N \ / \ uv^iwx^iy \notin L]$

Se si verifica la tesi negata il linguaggio non é libero.

Suppongo $L_1 = w_1w_2 \ / \ w_1 = w_2, \ w_1, w_2 \in \{a, b\}^*$ **libero**;
Sia p un naturale qualunque sempre positivo;
Sia $z = a^pb^pa^pb^p \ (|z| = 4p)$;
Allora $z \in L_1, \ |z| > p$
Siano $uvwxy \ / \ z = uvwxy, \ |vwx| \leq p \wedge |vx| > 0$ distinguiamo vari casi:

1) vwx é composto da ‘a’ che occorrono a sinistra (w_1)

2) vwx é a cavallo e contiene sia ‘a’ che ‘b’ in w_1

3) vwx contiene solo ‘b’ in w_1

4) ‘b’ in w_1 e ‘a’$\in w_2$

5,6,7) ...speculare su w_2

- Nei casi 1, 3, 5, 7 considero le parole $z^1 = uv^0wx^0y$ (i=0);

- nel caso 1 sono certo di togliere alcune occorrenze di a quindi avró $z^1 = a^kb^pa^pb^p, \ k < p \implies z^1 \notin L$.

- Nel caso 3 $z^1 = a^pb^ka^pb^p, \ k < p \implies z^1 \notin L$.

- Nel caso 5 $z^1 = a^pb^pa^kb^p, \ k < p \implies z^1 \notin L$.

- Nel caso 7 $z^1 = a^pb^pa^pb^k, \ k < p \implies z^1 \notin L$.

- Nei casi 2, 4, 6 invece avró ancora $z^1 = uv^0wx^0y$ (i=0);

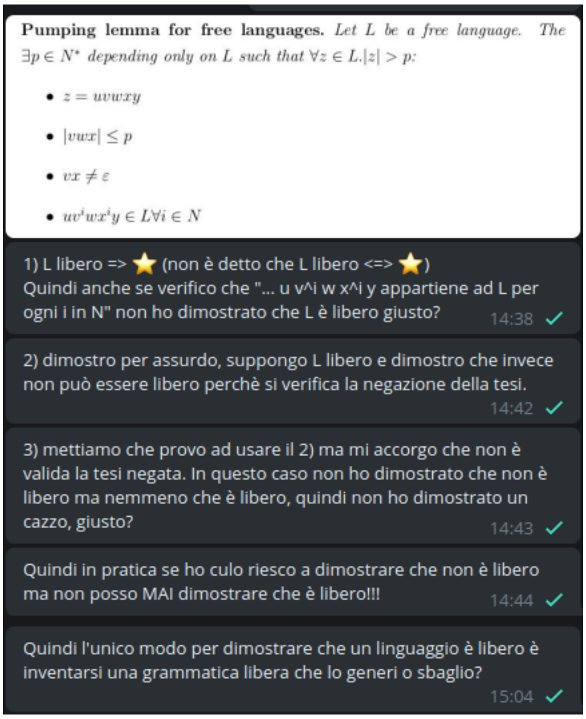
- Nel caso 2 $z^1 = a^kb^pa^pb^p,\ o\ a^pb^ka^pb^p,\ o\ a^jb^ka^pb^p,\ j,k < p \implies z^1 \notin L$.

- ...

In pratica facendo l'unpumping in tutti i casi $z_1 \notin L$ quindi L non é libero.

5.6.1 How not to use Pumping Lemma

Se avessi preso $\{ww\ /\ w \in \{a,b\}^*\}$ sia $z = (ab)^p(ab)^p$, prendo p=4, $v \in \varepsilon, x = a, i = 0$ cosí non dimostro niente perché se voglio dimostrare con il pumping lemma la negazione della tesi devo dimostrare un asserto che vale $\forall p \in \mathbb{N}^+$. Pertanto non posso prendere un arbitrario p=4.



Pumping lemma for free languages. *Let L be a free language. The* $\exists p \in N^*$ *depending only on L such that* $\forall z \in L.|z| > p$:

- $z = uvwxy$
- $|vwx| \leq p$
- $vx \neq \varepsilon$
- $uv^iwx^iy \in L \forall i \in N$

1) L libero => ⭐ (non è detto che L libero <=> ⭐)
Quindi anche se verifico che "... u v^i w x^i y appartiene ad L per ogni i in N" non ho dimostrato che L è libero giusto? 14:38

2) dimostro per assurdo, suppongo L libero e dimostro che invece non può essere libero perchè si verifica la negazione della tesi. 14:42

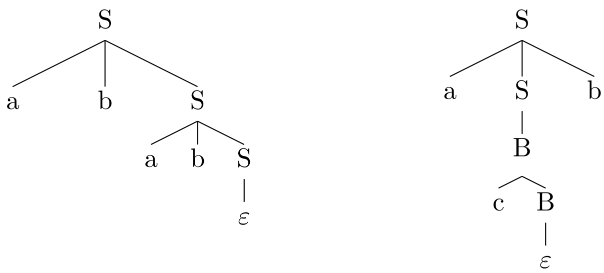
3) mettiamo che provo ad usare il 2) ma mi accorgo che non è valida la tesi negata. In questo caso non ho dimostrato che non è libero ma nemmeno che è libero, quindi non ho dimostrato un cazzo, giusto? 14:43

Quindi in pratica se ho culo riesco a dimostrare che non è libero ma non posso MAI dimostrare che è libero!!! 14:44

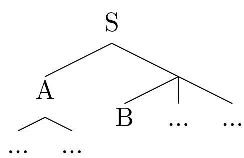
Quindi l'unico modo per dimostrare che un linguaggio è libero è inventarsi una grammatica libera che lo generi o sbaglio? 15:04

5.6.2 Esempi

ie) $\{a^nb^nc^j/n, j \geq 0\} = L_{17}$
$S \rightarrow aSb|B$
$B \rightarrow cB|\varepsilon$
$acb \in L_{17}$



$S \rightarrow abA|B$
$B \rightarrow cB|\varepsilon$
s



$S \to AB$
$A \to aAb|ab|\varepsilon$
$B \to Ab|c|\varepsilon$

5.6.3 Esempi Linguaggi Liberi

Essendo un linguaggio libero chiuso rispetto alla concatenazione, dati:

$L_1 = \{a^n b^n c^j \ / \ n, j \geq 0\}$ Libero perché concatenazione di $\{a^n b^n \ / \ n \geq 0\}$ e $\{c^j \ / \ j \geq 0\}$, entrambi liberi
$L_2 = \{a^j b^n c^n \ / \ n, j \geq 0\}$ Libero, inverso di L_1
$L_3 = \{a^n b^n c^n \ / \ n \geq 0\}$ Non é libero:
Suppongo L_3 libero, sia $p \in \mathbb{N}^+$, $z = a^p b^p c^p$ Allora $z \in L_3,\ |z| = 3p > p$
Spacco z in $A = a...a,\ B = b...b, C = c...c$
Siano $z = uvwxy \ \wedge \ |vwx| \leq p \ \wedge \ |vx| > 0$:

- vwx é composto da sole a in A
- vwx é composto da a in A e b in B
- vwx é composto da sole b in B
- vwx é composto da b in B e c in C
- vwx é composto da sole c in C

Considero la parola $z' = uv^0wx^0y$

1. $z' = a^k b^p c^p,\ k < p,\ z' \notin L_3$

3. $z' = a^p b^k c^p,\ k < p,\ z' \notin L_3$

5. $z' = a^p b^p c^k,\ k < p,\ z' \notin L_3$

2. $z' = a^k b^j c^p,\ k < p \ \vee \ j < p,\ z' \notin L_3$

4. $z' = a^p b^k c^j,\ k < p \ \vee \ j < p,\ z' \notin L_3$

Quindi visto che la parola non appartiene mai ad L_3 il linguaggio non é libero. □

5.6.4 Conferma $a^n b^n c^n$ non é libero

In effetti se provo ad applicare il pumping lemma mi accorgo che non ce la faccio:
considero aaaabbbbcccc, u = a^3, v = ab, w = b^3, x = c, y = c^3 é la cosa piú ragionevole ma quando faccio uv^2wx^2y mi viene fuori aaa abab bbb cc ccc. Se invece considero v = a e x = c alla fine $|a| = |c| \neq |b|$.

Quindi é vero che concatenando $a^n b^n$ libero con c^n libero ho un linguaggio libero ma mi viene $a^n b^n c^{n^1} \neq a^n b^n c^n$!

5.6.5 La classe dei linguaggi liberi non é chiusa all'intersezione

Nota che $L1$ ed $L2$ risultano liberi anche facendo pumping lemma per assurdo perché nel caso in cui vwx cada nel terminale ripetuto j volte con l'unpumping la stringa appartiene comunque al linguaggio (quindi ho almeno un caso in cui appartiene al linguaggio e non posso applicare il pumping lemma per assurdo).

> Quindi la classe di linguaggi liberi **non é chiusa rispetto all'intersezione**

$L_4 = \{a^n b^m c^{n+m} \ / \ n, m > 0\}$ Libero
$S \rightarrow aSc|aBc$
$B \rightarrow bBc|bc$

$L_5 = \{a^n b^m c^n d^m | n, m > 0\}$ Non libero
$L_6 = \{wcw^R \ / \ w \in \{a, b\}^+\}$ Libero
$S \rightarrow aSa|bSb|aca|bcb$

Chapter 6

Automi a stati finiti

Un NFA accetta/riconosce un certo linguaggio.

Sia N un NFA, allora il linguaggio riconosciuto/accettato da N é il set delle parole per le quali esiste almeno un cammino dallo stato iniziale di N ad uno stato finale di N.

notare che $\forall\ a \in A,\ a\varepsilon = \varepsilon a = a$.

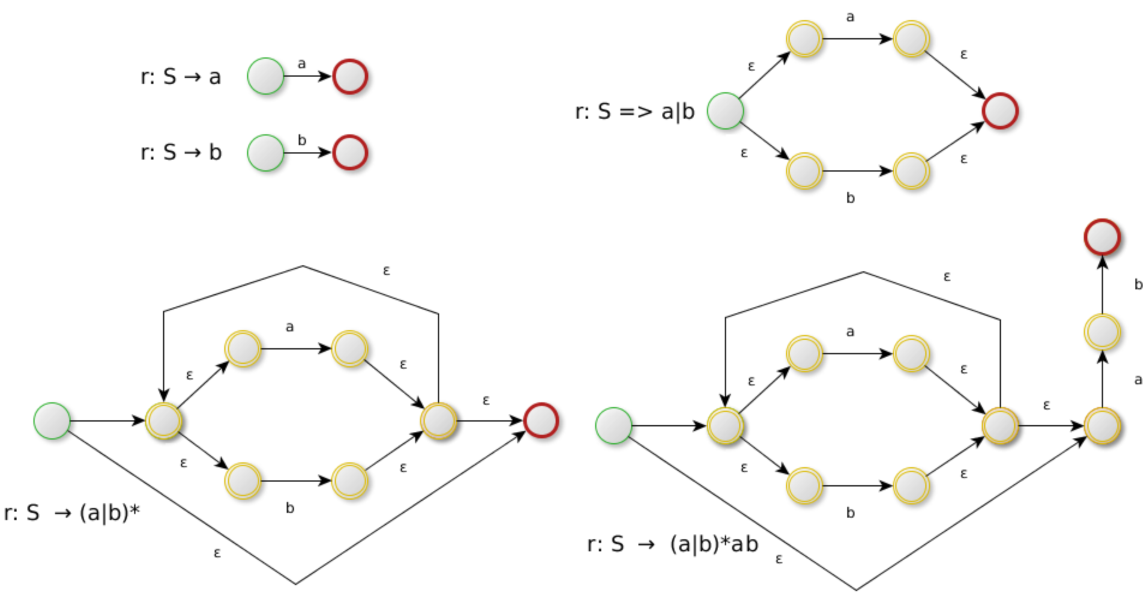
6.1 Thompson construction

input	regular expression r
output	NFA N $\ /\ L(N) = L(r)$

Gli NFA usati nei passi della costruzione hanno:

- un solo stato finale
- non hanno archi entranti sul nodo iniziale
- non hanno archi uscenti dal nodo finale

Lemma: Lo NFA ottenuto dalle costruzini di Thompson ha al massimo 2k stati e 4k archi, con k lunghezza della re. r. **Osservazione**: Ogni passo della costruzione introduce al massimo 2 nodi e 4 archi.



Algoritmo a complessitá $O(|r|)$

6.2 Simulare un NFA

Il backtracking consiste nel seguire un percorso e se non va bene tornare in dietro e provarne un altro finché alla fine li provo tutti mal che vada.

$N = (S, A, move_n, S_0, F)$, S insieme stati, A i non terminali (label degli archi), S_0 stato iniziale, F set stati finali, $move_n$ funzione di transizione $(S \otimes A \rightarrow S)$
$t \in S, T \subset S$

6.2.1 $\varepsilon - closure$

> $\varepsilon - closure(\{t\})$ il set degli stati S raggiungibili tramite zero o piú $\varepsilon - transizioni$ da t (in pratica il nodo stesso e tutti i nodi raggiungibili con una $\varepsilon - transition$ [**applicato ricorsivamente**]).

Nota che $\forall t \in S,\ t \in \varepsilon - closure(t)$ $\quad \varepsilon - closure(T) = \cup_{t \in T} \varepsilon - closure(t)$

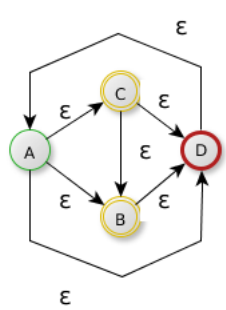
Questo algoritmo é piú performante del backtracking.

6.2.2 Algoritmo per la computazione $\varepsilon - closure$

Strutture dati:

- pila
- bool[] alreadyOn, dimensione $|S|$
- array[][] $move_n$

```
for(int i = 0; i < |S|; i++){
    alreadyOn[i] = false;
}
closure(t, stack){
    push t onto stack;
    alreadyOn[t] = true; //posso sempre arrivare a me stesso con una epsilon-transition
    foreach(i in move_n(t, epsilon)){
        if(!alreadyOn[i]){
            closure(i, stack);
        }
    }
}
```




```
alredyOn[F F F F];
closure(A, pila vuota)
    [A] [T F F F]
        //B non e' ancora nella pila
        closure(B, [A])
            [A, B] [T T F F]
```

```
        closure(D, [A, B])
            [A, B, D] [T T F T]
    closure(C, [A, B, D])
        [A, B, C, D] [T T T T]
```

6.2.3 Algoritmo per la simulazione di un NFA

input	NFA N, w$
output	yes se $w \in L(N)$, no altrimenti

```
N = (S, A, move_n, S_0, F)
states = epsilon-closure({S_0})
symbol = nextchar()
while(symbol != $){
    states = epsilon-closure(Unione_{t in states} di move_n(t, symbol));
    //l'insieme di tutti gli stati raggiungibili con la sottostringa letta fin ora
    symbol = nextchar();
}
if(states intersecato F != emptyset){
    return yes;
}
return no;
```

Algoritmo a complessitá $O(|w|(n+m))$

6.2.4 Sink

non serve semplicemente per avere una funzione di transizione totale inserisco un nodo pozzo sink dove confluiscono tutte le transizioni non presenti nel DFA. Dato che non é possibile uscire dal pozzo le parole che finiscono in sink non arriveranno mai ad uno stato finale, quindi non sono riconosciute da L.

6.3 DFA

Automa a stati finiti, deterministico; una sottoclasse degli NFA che rispettano:

$$\text{DFA} \triangleq (S, A, move_d, s_0, F)$$
$$move_d \triangleq (S \otimes A) \to S$$

- non hanno $\varepsilon - transizioni$
- $\forall a \in A, s \in S,\ move_n(s, a)$ é un unico stato se ho una **funzione di transizione totale** (al piú uno stato se ho una **funzione di transizione parziale**)

> **Sink** è il nodo pozzo dove **confluiscono tutte le transizioni non segnate** (per ogni stato se mi manca una transizione per un determinato terminale ne faccio una su sink); viene aggiunto per rendere la funzione di transizione una funzione di transizione totale

6.3.1 Linguaggio riconosciuto dal DFA

Dato il DFA D, L(D) é il linguaggio riconosciuto da D.
$L(D) = \{w = a_1, ..., a_k$ / $\exists$ cammino in D dallo stato iniziale al finale$\}$. $\varepsilon \in L(D) \iff s_0 \in F$.

6.3.2 Simulazione di un DFA con $move_d$ totale

input	w$, DFA $D = (S, A, move_d, F)$
output	yes se $w \in L(D)$, no altrimenti

```
state = s_0;
while(symbol != $ && state != bottom){
   //move_d(s, a) = bottom <=> move_d non e' definita su (s,a)
   //se sono in bottom sono in sink
   state = move_d(state, symbol);
   symbol = nextchar();
}
if(state in F)
   return yes;
return no;
```

Simulazione NFA costa $O(|w|(n+m))$ Simulazione DFA costa $O(|w|)$

6.4 Subset Construction

input $NFA(S^n, A, move_n, S_0^n, F^n)$
output $DFA(S^d, A, move_d, S_0^d, F^d)$

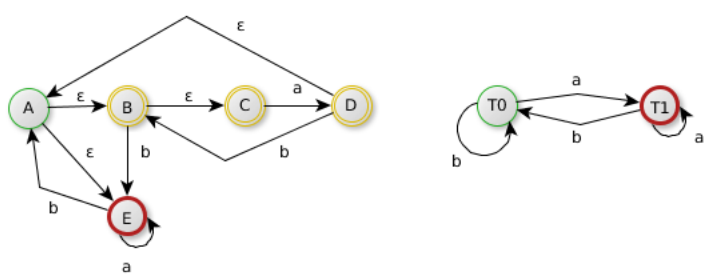
```
S_0^d = epsilon-closure({S_0^n});
 //raggruppo stati della epsilon closure in un unico stato S_0^d del DFA
states = {S_0^d};
tag S_0^d come non marcato;

while(exist T in states non marcato){
   marco T;
   foreach(a in A){ //guardo ogni arco
      T_1 = epsilon-closure(U_{t in T} di move_n(t,a));
         //tutti gli stati raggiungibili con una a-transition da uno stato in T
         //poi la loro epsilon closure
      if(T_1 != emptySet){
         move_d(T, a) = T_1;
         if(T_1 !in states){
            aggiungi T_1 a states come non marcato;
         }
      }
   }
}

foreach(T in states){
   if( (T intersecato F^n) != 0){
      metti T in F^d;
   }
}
```

Lo stato iniziale del DFA sará la $\varepsilon - closure$ dallo stato iniziale del NFA (quindi un set di stati). Considero lo stato iniziale del NFA e lo marco in grassetto poi espando T_0 con la $\varepsilon - closure$ dello stato iniziale.

Dallo stato T_0 guardo per ogni arco gli stati in cui arrivo e li marco in grassetto (T_1, T_2, ...); poi espando quelli in grassetto guardando le rispettive $\varepsilon - closure$.

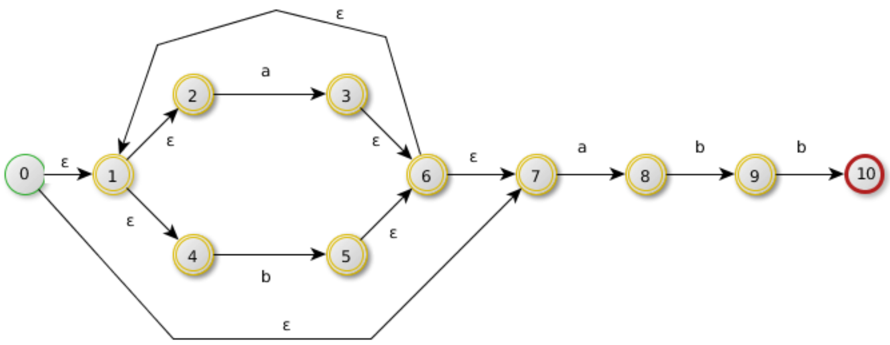
Alla fine guardo i set degli stati se due set coincidono mergio gli stati.

6.4.1 Esercizio

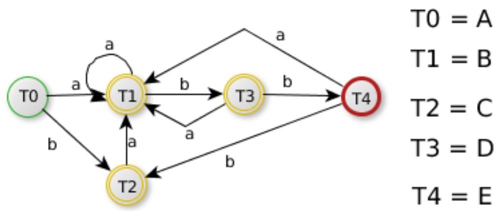


States	**a**	**b**
T0 = { **A** B C E }	T1	T2 = T0
T1 = { A B C **D** **E** }	T1	T0
T2 = { **A** B C **E** } = T0 (quindi T0 va in T0 tramite b)	come T0	come T0

6.4.2 Esercizio



States	**a**	**b**
S_0^d = { **0** 1 2 4 7 }	T1	T2
T1 = { 1 2 **3** 4 6 7 **8** }	T1	T3
T2 = { 1 2 4 **5** 6 7 }	T1	T2
T3 = { 1 2 4 **5** 6 7 9 }	T1	T4
T4 = { 1 2 4 **5** 6 7 **10** }	T1	T2



Quindi sono in uno stato T e guardo un terminale, prima guardo per ogni stato in T se ci sono transizioni per quel terminale (in caso scrivo lo stato in cui arrivo in grassetto). A questo punto espando con ε-transition ogni stato grassettato. Scrivo uno stato in grassetto anche se giá contentuto in T.

6.5 Partition Refinement

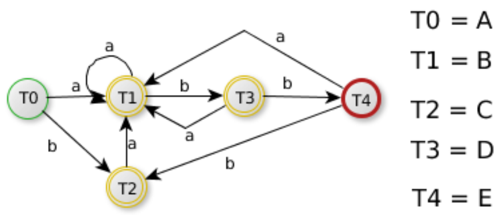
Guado gli archi, se tutta la partizione punta ad un nodo dell'altra transizione con lo stesso non terminale allora va bene; altrimenti spacco la partizione.

6.5.1 Algoritmo di Partition Refinement

Input	DFA D = $\{S, A, move_d, s_0, F\}$
Output	partizione di S in blocchi equidistanti

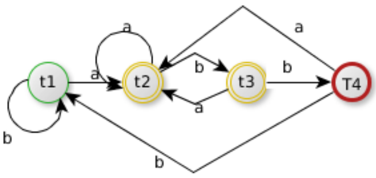
```
B_1 = F;
B_2 = S \ F;
P = {B_1, B_2};
while(exists B_i, B_j in P, exists a in A, B_i e'' partizionabile rispetto a (B_j, a)){
   sostituire B_i in P con split(B_i, (B_j, a));
}
```

6.5.2 Esempio



{ A B C D } { E }	Considero le partizioni dei finali e non finali
{ A B C D } { E }	Con a-transizione non esco dal primo set
{ A B C } { D } { E }	Con b-transizione vado da D in E (e A B C non vanno in E con b-transizioni)
{ A B C } { D } { E }	Con a-transizione non esco
{ A C } { B } { D } { E }	Con b-transizione vado da B in D e gli altri no quindi splitto
{ A C } { B } { D } { E }	vanno bene

Rinomino { A C } { B } { D } { E } in $t_1,\ t_2,\ t_3,\ t_4$



6.6 Algoritmo di minimizzazione di DFA

Input	DFA $\mathbf{D} = \{S, A, move_d, s_0, F\}$ con $move_d$ totale
Output	minimo DFA (**min(D)**) che riconosce lo stesso linguaggio del primo

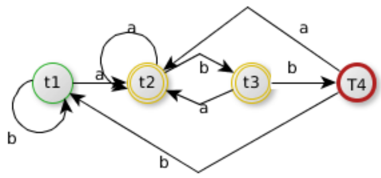
```
P = PartitionRefinement(DFA D);
// P = (B_1, ..., B_k);
foreach(B_i in P){
    var t_i = B_i;          //do un nome alla partizione, un alias
    if(s_o in B_i){
        t_i e'' iniziale per min(D); //setto lo stato iniziale di min(D)
    }
}

foreach(B_i in P/ B_i subset F){
    t_i = B_i;
    t_i e'' lo stato finale di min(D); //setto lo stato finale di min(D), t_i = B_i
}

foreach( (B_i, a, B_j) / esiste s_i in B_i, s_j in B_j / che move_d(s_i, a) = s_j){
    //per ogni tupla (stato, arco, stato) faccio la rispettiva transizione in min(D)
    setto una transizione temporanea in min(D) da t_i a t_j secondo il simbolo a;
}

foreach(dead state t_i){ //uno stato che non potra' mai arrivare in un finale
    rimuovere t_i e tutte le transizioni da/verso t_i;
}
tutti i temporanei residui (sia stati che transizioni) sono gli stati e le transizioni
     di min(D);
```

Complessitá $O(nlgn)$.

Un **dead state** é uno stato che non puó essere raggiunto, nel nostro caso anche sink é un dead state.

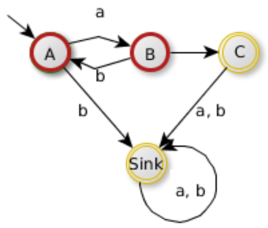
6.6.1 Esempio

Arrivato qua: $t_1 = \{$ A C $\}$, $t_2 = \{$ B $\}$, $t_3 = \{$ D $\}$, $t_4 = \{$ E $\}$, applico la minimizzazione del DFA.

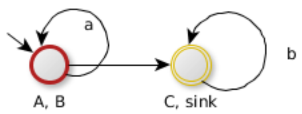


6.6.2 Esempio

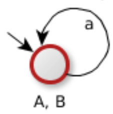
Aggiungo il nodo sink (se richiesta fn transizione completa)



Ho le partizioni $\{A,\ B\}$, $\{C,\ sink\}$, applico partition refinement ma sono giá partizionati correttamente.

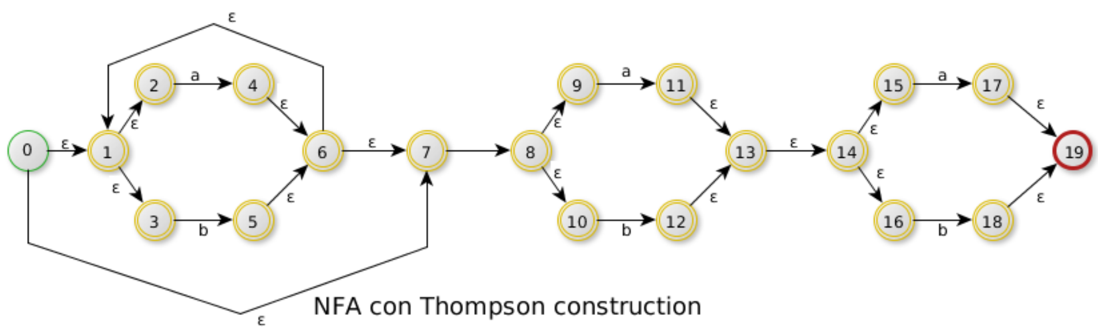


Visto {C, sink} un dead state per il grafo, posso eliminarlo

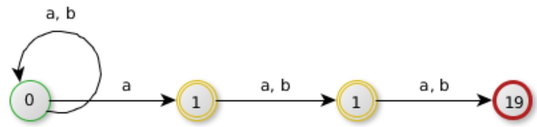


6.6.3 Esempio

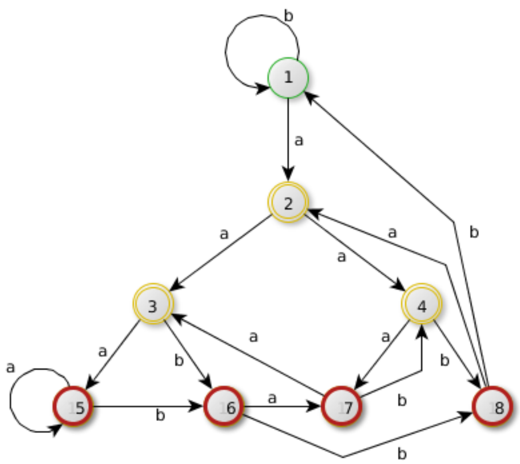
Sia $r = (a|b) * a(a|b)(a|b)$, per determinare il minimo DFA di riconoscimento di L(r) posso usare Thompson e spararmi in faccia o andare a naso.



NFA con Thompson construction



NFA a naso

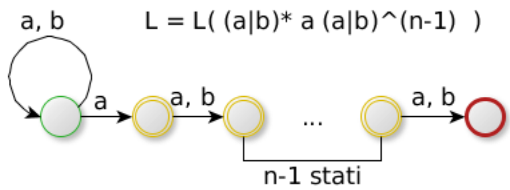


DFA Totale

6.6.4 Lemma

Lemma: $\forall n \in \mathbb{N}^{+}$, $\exists$ un NFA con (n+1) stati il cui minimo DFA equivalente ha almeno 2^n stati.

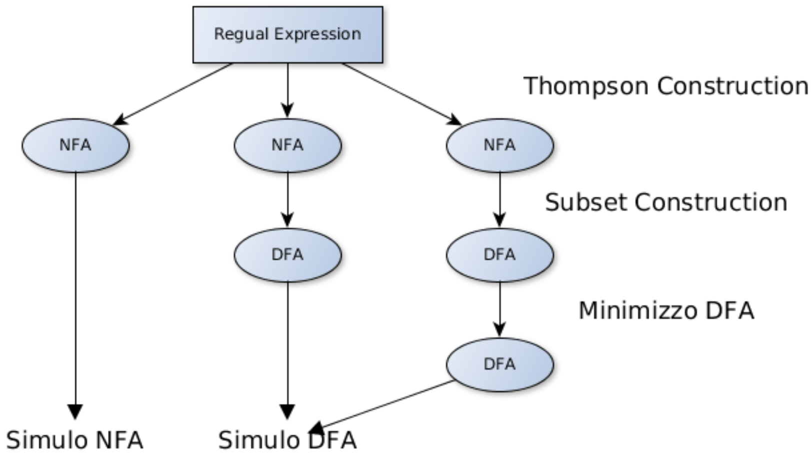
Dim:



Per assurdo suppongo esista un DFA minimo con meno di 2^n stati.
Osservo che esistono 2^n possibili parole di lunghezza n con simboli $\{a, b\}$ (osservazione indipendente dal grafo, tutte le possibili combinazioni).
$\Longrightarrow$ $\exists\ w_1 \neq w_2$ / $|w_1| = |w_2| = n$ e il loro riconoscimento conduce allo stesso stato del DFA.
$\Longrightarrow$ esiste almeno una posizione per cui w_1 e w_2 differiscono (considero quella piú a destra).

$w_1 = w_1'ax$, $w_2 = w_2'bx$ iniziano diversi ma finiscono con x entrambe. Considero $w_1'' = w_1'ab^{n-1}$ $w_2'' = w_2'bb^{n-1}$ raggiungo uno stato finale t; la seconda parola peró non appartiene al linguaggio, nonostante possa comunque raggiungere lo stato t. Pertanto contraddiciamo che t sia finale. Quindi sono ad un assurdo, il DFA minimo deve avere per forza almeno 2^n stati.

6.7 Ricapitolando



Algoritmo	Complessitá nello spazio	Complessitá nel tempo
Thompson Construction	$O(\|r\|)$	$O(\|r\|)$ || $O(n_n + m_n)$, [nel caso di ε-closure]
Simulazione NFA	-	$O(\|w\|(n_n + m_n))$
Subset Construction	-	$O(n_d\|A\|(n_n + m_n))$, [spesso $\|A\| = O(\|r\|)$]
Minimazzazione DFA	-	$O(n_d lg(n_d))$
Simulazione DFA	-	$O(\|w\|)$

Chapter 7

Linguaggi Regolari o Lineari

7.1 Da DFA a Grammatica Regolare

Una grammatica é regolare se le produzioni sono della forma: $A \to \beta$, con β terminale non-terminale, viceversa o terminale e basta.

$A \to aB$	$B \to b$	grammatica lineare destra
$B \to Ab$	$A \to a$	grammatica lineare sinistra

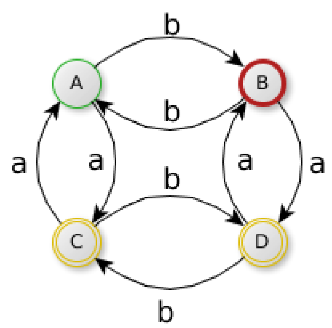
In pratica é la diretta trascrizione di un DFA in regex!

Dato un DFA D voglio trovare una grammatica regolare G tale che L(G) = L(D). Se ho una transizione $A \to B$ con una a-transizione diventerá $A \to aB$. Segno il nome del nodo che sto considerando prima della freccia e, dopo la freccia, il non terminale ed il nodo destinazione. Se ho un nodo foglia C avró $C \to \varepsilon$.

Se invece ho una grammatica regolare e voglio trovare un DFA D / $L(G) = L(D)$, faccio il procedimento inverso a prima; se ottengo un NFA basta fare Subset Construction.

7.1.1 Esempio

$L = \{w\ /\ w \in \{a,b\}^* \&\& |a|\ pari,\ \ |b|\ dispari\}$, L é regolare?



Sí é regolare.

7.1.2 Considerazioni

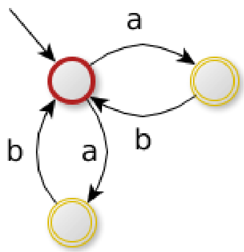
Regular expression, NFA e DFA hanno la stessa potenza espressiva, sono solo notazioni diverse.

Dal DFA posso sempre costruirmi una **grammatica regolare** equivalente.

Non devo fare l'errore di assumere che qualsiasi grammatica sia esprimibile attraverso un NFA.

7.1.3 Esempio

$L = \{w \ / \ w \in \{a, b\}^* \&\& |a| = |b|\}$, L é regolare?



Non potrá mai essere regolare, per il pumping lemma per i linguaggi regolari.

7.2 Pumping Lemma per Linguaggi Regolari

Sia L un linguaggio regolare $\implies \exists \ p \in \mathbb{N}^+ \ / \ \forall \ z \in L \ / \ |z| > p, \ \exists \ u, v, w \ / :$

i) $z = uvw \ \wedge$

i) $|uw| \leq p \ \wedge$

i) $|v| > 0, \ \forall i \in \mathbb{N}, \ uv^i w \in L$

7.2.1 Dimostrazione

L é regolare quindi puó essere riconosciuto da un automa a stati finiti.
Sia D il min DFA $\ / \ L(D) = L, \ p = |S|$, allora i cammini piú lunghi che non passano piú di una volta nel medesimo stato hanno al piú lunghezza (p-1).
Allora se $z \in L$ con $|z| > p$, z é riconosciuta tramite un cammino che attraversa almeno due volte uno stato.

7.2.2 Negazione testi Pumping Lemma per linguaggi regolari

$\forall p \in \mathbb{N}^+ \ / \ \exists \ z \in L \ / \ |z| > p. \ \forall \ uvw \ z = uvw \wedge |uw| \leq p \wedge |v| > 0) \implies \exists \ i \in \mathbb{N} \ / \ uv^i w \notin L)$

Lemma: $L = \{a^n b^n \ / \ n \geq 0\}$ non é regolare

Dim: Assumo per assurdo che L sia regolare, dato p un qualunque numero positivo e $z = a^p b^p$ allora $\forall \ uvw \ / \ z = uvw \wedge |uw| \leq p \wedge |v| > 0$ (la stringa v contiene solo (e almeno una) ‘a’).

allora uv^2w ha la forma $a^{p+k}b^p, \ k > 0$ allora $uv^2w \notin L$ il che contraddice il Pumping Lemma per linguaggi regolari.

[v puó assumere a^i, $a^i b^j$, b^i, in ogni caso per qualunque potenza di v non appartiene ad L (con $(a^i b^j)^2$ ho)]

7.2.3 Esercizio

$L_1 = \{w \ / \ w \in \{a, b\}^*$e contiene almeno una occorrenza di “aa ”$\}$

L_1: $A \to aA|bA|aB$
$B \to aC$
$C \to aC|bC|\varepsilon$

$L_2 = \{ww \ / \quad \in \{a, b\}^*\}$
non é libero per il pumping lemma (giá dimostrato), quindi non é regolare.

$\neg$ L Libero $\implies$ $\neg$ L Regolare
$\neg$ L Libero $\not\Longleftrightarrow$ $\neg$ L Regolare

$L_3 = \{ww^r \ / \ w \in \{a, b\}^*\}$
Non é regolare ma libero. $z = a^p b^p b^p a^p \in L_3$ visto che $uv < p$, uv é composta solo da a $uv^i w = a^p b^{2p} a^p \notin L_3$ quindi non puó essere regolare.
[w^r é w rovesciato]

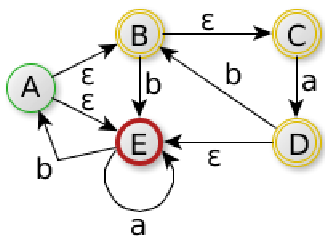
7.2.4 Esercizi di esame

Sia N_1 lo NFA con stato iniziale A e finale E con la seguente funzione di transizione:

	ε	a	b
A	$\{B, E\}$	$\emptyset$	$\emptyset$
B	$\{C\}$	$\emptyset$	$\{E\}$
C	$\emptyset$	$\{D\}$	$\emptyset$
D	$\{E\}$	$\emptyset$	$\{B\}$
E	$\emptyset$	$\{E\}$	$\{A\}$

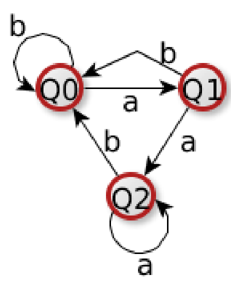
1) $aa \in L(N_1)$?

2) D é il DFA ottenuto da N_1, per subset construction, Q stato iniziale di D, $Q_{ab\_}$ lo stato di D che si raggiunge da Q tramite il cammino ab. Dire a quale sottoinsieme degli stati di N_1 corrisponde $Q_{ab\_}$.



1) Sí facendo $A \to B \to C \to D \to E \to E$ 2) Facendo la subset construction:

	a	b
$Q0 = \{A, B, C, D\}$	Q1	Q0
$Q1 = \{D, E\}$	Q2	Q0
$Q2 = \{E\}$	Q2	Q0



Chapter 8

Analisi Sintattica

$S \to cAd$
$A \to ab|a$

8.1 Parsing Top-down

Parto dal starting symbol ed espando le derivazioni dando prioritá alle derivazioni piú a sinistra. Cerco quindi di ricostruire una derivazione leftmost della stringa w data in input.

$w\$, \$ \notin V$
$w = cabd$

Per ricostruire la parola w parto dalla prima derivazione $S \to cAd$ derivo la A piú a sinistra (leftmost) e posso scegliere fra a ed ab; scelgo a e mi accorgo che ho sbagliato, torno in dietro e scelgo ab.

8.2 Parsing Top-down predittivo (o non ricorsivo)

Cambio la grammatica sopra in:
$S \to cAd$
$A \to aB$
$B \to b|\varepsilon$

$S \to cAd \to caBd \to$ vedo che mi serve una b, escludo a priori ε

8.3 Grammatica LL(1)

Le grammatiche LL(1) sono un subset delle grammatiche libere.

prima **L**	leggiamo la input string da sinistra (left)
seconda **L**	ricostruiamo una leftmost derivazione
(1)	decidiamo quale operazione effettuare guardando un solo simbolo in input

8.4 First

Data una generica $\alpha \in V^*$ per G=(V, T, S, P), first(α) é l'insieme dei simboli terminali b tali che $\alpha \implies bv$. Inoltre se $\alpha \implies \varepsilon$ allora $\varepsilon \in first(\alpha)$

8.4.1 Esercizio

$S \to A|B$
$A \to a|C$
$C \to \varepsilon$
Allora first(A) = $\{a, \varepsilon\}$ (ε perché posso fare $A \implies C \implies \varepsilon$).

8.4.2 Esercizio

$S \to A|B$
$A \to a|C$

$C \to bB$
Allora first(A) = $\{a, b\}$ (b perché posso fare $A \implies C \implies bB$, ma B non esiste).

8.4.3 Esercizio

$S \to A|B$
$A \to a|C$
$C \to bB$
$B \to c$
Allora first(A) = $\{a, b\}$ ($A \implies C \implies bB \implies bc$, ma tengo solo il primo simbolo (b))

8.4.4 Esercizio

$A \to A|C$
$C \to bB|\varepsilon$
$B \to c$
Allora first(A) = $\{a, b, \varepsilon\}$

8.4.5 Algoritmo calcolo dei first

G=(V,T,S,P) Sia $X \in V$. L'insieme first(X) viene calcolato come segue:

1) inizializzo first(X) vuoto $\forall\ X \in V$

2) se $X \in T$ allora first(X) = $\{X\}$

3) se $X \to \varepsilon \in P$ allora aggiungere ε ai first(X)

4) se $X \to Y_1...Y_n \in P$, con $n \geq 1$ allora uso la seguente procedura:

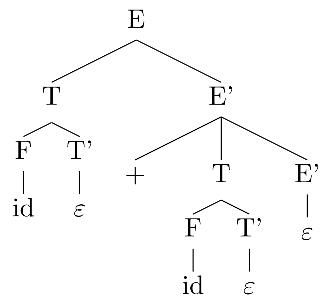
```
j = 1;
while(j <= n){
    aggiungere ai first(X) ogni b tale che b in first(Yj)
    if(epsilon in first(Yj)){
        j++;
    } else {
        break;
    }
}

if(j == n+1){
    aggiungere epsilon ai first(X);
}
```

8.4.6 Esercizio

$E \to TE'$
$E' \to +TE'|\varepsilon$
$T \to FT'$
$T' \to *FT'|\varepsilon$
$F \to (E)|id$

First:
$E = \{id, (\}$ ovviamente ha gli stessi first di T per $E \to TE'$
$E' = \{+, \varepsilon\}$
$T = \{id, (\}$ ha gli stessi first di F per $T \to FT'$
$T' = \{*, \varepsilon\}$
$F = \{id, (\}$

Per generare id + id:



Mancano le parentesi fra i terminali nella tabella...

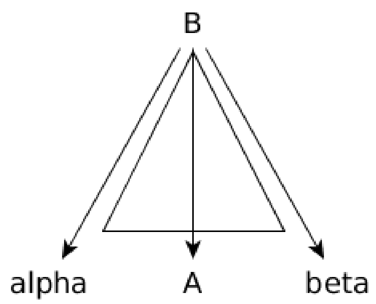
	id	+	*	$
E	$E \rightarrow TE'$			
E		$E' \rightarrow TE'$		$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			

8.5 Follow

$\forall\ A \in V \backslash T$, follow(A):

```
follow(A) = emptySet per ogni A in (V \ T);
follow(S).push($);

repeat{
    foreach(B -> alpha A beta in P){
        if(beta == epsilon){
            follow(A).push(follow(B));
        } else {
            follow(A).push(first(beta) \ epsilon);
            if(epsilon in first(beta)){
                follow(A).push(follow(B));
            }
        }
    }
} until (saturazione);
```

Nei follow non potró mai avere ε



Quindi in pratica i follow(A) li trovo guardando le produzioni con A dopo la freccia. α e β sono espressioni con lunghezza qualsiasi mentre A é un non treminale.

Se tipo ho abCdEFGhi come lo spacco in $\alpha B \beta$? Vai in ordine, se vuoi calcolarti i follow di C allora C sarà la tua B.

Calcolo follow(B), guardo produzioni $A \rightarrow \alpha B \beta$
Metti $ per tutti i non terminali
Per le produzioni $A \rightarrow aB$ tutti i follow di A vanno in B
Per $A \rightarrow aBb$ tutti i first(b) meno epsilon vanno in follow(B)
Per $A \rightarrow aBb$ con epsilon appartenente ai first(b) allora aggiungi anche i follow(A) ai follow(B)

8.5.1 Esempio

$S \to aABb$
$A \to Ac|d$
$B \to CD$
$C \to e|\varepsilon$
$D \to f|\varepsilon$

	First	Follow
$S =$	$\{a\}$	$\{\$\}$
$A =$	$\{d\}$	$\{e, f, b$ (da $S \to aABb$), c (da $A \to Ac$)$\}$
$B =$	$\{e, f, \varepsilon\}$	$\{b$ (da $S \to aABb$)$\}$
$C =$	$\{a, \varepsilon\}$	$\{f$ (da $B \to CD$)$\}$
$D =$	$\{f, \varepsilon\}$	$\{\}$

Poi i follow(B) vanno in D perché ho $B \to CD$ e anche in C perché D puó essere ϵ.

8.5.2 Esempio

$S \to aA|bBc$
$A \to Bd|Cc$
$B \to e|\varepsilon$
$C \to f|\varepsilon$

	First	Follow
$S =$	$\{a, b\}$	$\{\$\}$
$A =$	$\{e, d, f, c\}$	$\{\$, (?)\}$
$B =$	$\{e, \varepsilon\}$	$\{c, d\}$
$C =$	$\{f, \varepsilon\}$	$\{c\}$

Occhio che nei first di A non ci va ε perché ci può essere dalla sostituzione con B ma subito dopo hai d quindi in questo caso il first é d

8.5.3 Esempio

$E \to TE'$
$E' \to +TE'|\varepsilon$
$T \to FT'$
$T' \to *FT'|\varepsilon$
$F \to (E)|id$

	First	Follow	
$E =$	$\{id, (\}$	$\{\$,)\}$	
$E' =$	$\{+, \varepsilon\}$	$\{\}$	ed eredita i follow di E
$T =$	$\{id, (\}$	$\{+\}$	ed eredita i follow di E,E'
$T' =$	$\{*, \varepsilon\}$	$\{\}$	ed eredita i follow di T
$F =$	$\{id, (\}$	$\{*\}$	ed eredita i follow di T, T'
		Quindi diventa:	
$E =$	$\{id, (\}$	$\{\$,)\}$	
$E' =$	$\{+, \varepsilon\}$	$\{\$,)\}$	
$T =$	$\{id, (\}$	$\{+, \$,)\}$	
$T' =$	$\{+, \varepsilon\}$	$\{+, \$,)\}$	
$F =$	$\{id, (\}$	$\{*, +, \$,)\}$	

Parsing di "$id + id * id\$$" $E \to TE' \to FT'E' \to idT'E' \to ...$

8.6 Tabella di parsing

Nella cella [A,b] metto le produzioni $A \to \beta$ / $b \in first(\beta)$. Se epsilon appartiene ai first devi anche controllare che il terminale sia contenuto nei follow del non terminale

> Quindi in [A, b] metto $(A \to \beta) \in P$ / $b \in first(\beta)$ e se $\varepsilon \in first(\beta) \implies b \in follow(A)$.

8.6.1 Algoritmo di costruzione della tabella di parsing predittivo top-down

input	G=(V,T,S,P)
output	Tabella T di parsing predittivo top-down se G é LL(1)

```
foreach((A -> alpha) in P){
    forall b in first(alpha), poniamo A -> alpha in T[A, b];
    if(epsilon in first(alpha)){
        forall x in follow(A) poniamo A -> alpha in T[A, x];
    }
}

poniamo error() in tutte le entry di T che sono rimaste vuote;

if(la tabella non ha entry multiply-defined)
    G e'' LL(1);
```

8.6.2 Esempio

$E \to E + T|T$
$T \to T * F|T$
$F \to (E)|id$

	First	Follow	
$E =$	$\{(, id\}$	$\{\$, +,)\}$	$\{\$, +,)\}$
$T =$	$\{(, id\}$	$\{*\}$ ed eredita i follow di E	$\{\$, +, *,)\}$
$F =$	$\{(, id\}$	$\{\}$ ed eredita i follow di T	$\{\$, +, *,)\}$

	id
E	$E \to E + T$ $E \to T$

Guardo se é LL(1)

Pur non sviluppando tutta la tabella si vede che ci sono entry multiple **quindi non é LL(1)**.

8.7 Algoritmi di Parsing

input buffer	w$
stack	bottom [$] top
parsing table	con tante righe quante non terminali, tante colonne quante terminali ($ incluso) in ogni cella metto un'eventuale trasformazione o “error ”

8.7.1 Algoritmo di parsing non-ricorsivo

input	stringa w, tabella parsing non ricorsivo T, per G
output	derivazione leftmost di w se $w \in L(G)$, error() altrimenti

```
//non terminali e gli stati del grafo sono la stessa cosa
//init
buffer = {w$}; //meglio $w^r
stack.push($S); //stack di terminali e non terminali

let b = buffer.pop() //il primo simbolo di w
let x = stack.top()

while(x != $){
    if(x == b){ //ho il carattere giusto e lo brucio
        stack.pop(x);
        b = buffer.nextChar();
    } else if(x e'' terminale){
        //sono arrivato ad un terminale diverso da quello della stringa
        error();
    } else if(T[x,b] contiene x -> Y1...Yn){ //x e' un non terminale (nello stack)
        //se la tabella di parsing contiene una entry
        cout << x -> Y1...Yn;
        stack.pop(x);
        stack.push(Yn...Y1); //li pusha al contrario
                        //se ho una epsilon non la pusho
    }
```

```
        x = stack.top()
    }
```

8.7.2 Esempio

$E \rightarrow TE'$
$E' \rightarrow +TE'|\varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'|\varepsilon$
$F \rightarrow id$

First and Follow

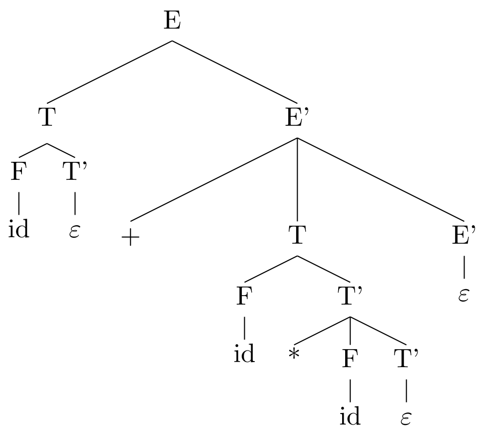
Non terminale	**First**	**Follow**
E	id	\$
E'	+, ε	\$
T	id	+, \$
T'	*, ε	+, \$
F	id	+, *, \$

Tabella di parsing

	id	+	*	\$
E	$E \rightarrow TE'$			
E'		$E' \rightarrow +TE'$		$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			

Passi dell'algoritmo

pila	input	output
$\$\ \underline{E}$	$\underline{id}+id*id\ \$$	$E \rightarrow TE'$
$\$\ E\ \underline{T'}$		$T \rightarrow FT'$
$\$\ E\ T'\ \underline{F}$		$F \rightarrow id$
$\$\ E\ T'\ \underline{id}$		
$\$\ E'\ \underline{T'}$	$\underline{+}id*id\ \$$	$T' \rightarrow \varepsilon$
$\$\ \underline{E'}$		$E' \rightarrow TE'$
$\$\ E'\ T\ \underline{+}$	$\underline{id}*id\ \$$	
$\$\ E'\ \underline{T}$		
	...Avanti cosı	



8.7.3 Esercizio

$S \rightarrow aA|bB$
$A \rightarrow c$
$B \rightarrow d$

$w = ac\$$

Parsing: $S \implies aA \implies ac$

First and Follow

Non terminali	First	Follow
S	a, b	$
A	c	$
B	d	$

Tabella di parsing con le produzioni di G

	a	b	c	d	$
S	$S \to aA$	$S \to bB$			
A			$A \to c$		
B				$B \to d$	

8.8 Grammatica Ricorsiva Sinistra

Una grammatica G esibisce **left recursion** se $\exists\ A \in (V \backslash T)\ /\ A \to^* A\alpha,\ \alpha \in V^*$

La left recursion é immediata se G ha almeno una produzione del tipo $A \to A\alpha$ (ovvero se succede nel passato). Nell'esempio di prima c'era left recursion immediata nei primi due casi ($E \to E\alpha \wedge T \to T\alpha$).

Proposizione: ogni grammatica che esibisce left recursion **non é LL(1)**.

Proposizione: ogni grammatica che nella tabella di parsing ha piú di una entry in una cella **non é LL(1). Altrimenti lo é**.

Se una grammatica é LL(1) $\implies$ non da conflitti nella tabella di parsing $\implies$ é una grammatica libera.

$A \to B$
$B \to Aa$
Esempio di left recursion in piú passi.

8.9 Eliminazione Left Recursion immediata

8.9.1 Esempio

$A \to A\alpha|\beta$, con $\beta \neq A \wedge \alpha \neq \varepsilon$ diventa:
$A \to \beta A'$
$A' \to \alpha A'|\varepsilon$

Piú in generale $A \to A\alpha_1|...|A\alpha_n|\beta_1|...|\beta_n, \quad \beta_1, ..., \beta_n \neq A_y, \quad \alpha_1, ..., \alpha_n \neq \varepsilon$ diventa
$A \to \beta_1 A'|...|\beta_n A'$
$A' \to \alpha_1 A'|...|\alpha_n A'|\varepsilon$
Ho introdotto A' nuovo non terminale in G.

8.9.2 Esempio

Eliminare Left Recursion immediata da:	Diventa:
$E \to E + T\|T$ $T \to T * F\|F$ $F \to (E)\|id$	$E \to TE'$ $E' \to +TE'\|\varepsilon$ $T \to FT'$ $T' \to *FT'\|\varepsilon$ $F \to (E)\|id$

	First	Follow
$E =$	$\{id, (\}$	$\{\$,)\}$
$E' =$	$\{+, \varepsilon\}$	$\{\$,)\}$
$T =$	$\{id, (\}$	$\{+, \$,)\}$
$T' =$	$\{*, \varepsilon\}$	$\{+, \$,)\}$
$F =$	$\{id, (\}$	$\{*, +, \$,)\}$

Tabella di parsing

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		$
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		$
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	)	$

I campi vuoti sono error, non ci sono multiple entries quindi é LL(1).

8.9.3 Esempio

Eliminare Left Recursion immediata da:
$E \to E + E|E * E|(E)|id$

Diventa:
$E \to (E)E'|idE'$
$E' \to +EE'| * EE'|\varepsilon$

(Parte della tabella tanto non serve tutta)

	First	Follow	
$E =$	$\{id, (\}$	$\{\$,), +, *\}$ ed i follow di E'	$\{\$,), +, *\}$
$E' =$	$\{+, *, \varepsilon\}$	$\{\}$ ed i follow di E	$\{\$,), +, *\}$

Visto che ho almeno una entry multipla la grammatica non é LL(1).

L'eliminazione della left recursion ci ha dato un grammatica che non é comunque LL(1). Nel nostro caso é anche ambigua.

Lemma: L'eliminazione della left recursion NON elimina **l'ambiguitá**.

8.10 Left Factoring

$S \to aSb|ab$

	a	b	$
S	$S \to aSb$ $S \to ab$		

La grammatica non é LL(1).

Possiamo peró fattorizzare le produzioni considerando una parte che é a sinistra ed é comune a piú produzioni, per ottenere una **grammatica LL(1)** che genera lo stesso linguaggio.
$S \to aA'$
$A' \to Sb|b$

DEF

Una grammatica G puó essere fattorizzata a sinistra quando esistono almeno due produzioni $A \to \alpha\beta_1$ e $A \to \alpha\beta_2$ per qualche $A \in V\backslash T,\ \alpha, \beta_1, \beta_2 \in V^* \wedge \alpha$ non comincia per A.

DEF

G puó essere fattorizzata a sinistra se: $A \to \alpha\beta_1,\ A \to \alpha\beta_2 \in P$ con
$\alpha, \beta_1, \beta_2 \in V^*, \alpha$ non ha A come primo simbolo, $A \in V\backslash T$

Lemma

Se G puó essere fattorizzata a sinistra allora **G non é LL(1)**.

8.11 Algoritmo di fattorizzazione a sinistra

```
foreach(A in V\T){
    trovare il prefisso piu lungo comune a due o piu produzioni per A, chiamato alpha
    if(alpha != epsilon){
        sostituire A -> alpha beta_1|...|alpha beta_n|Y_1|...|Y_k
```

```
            con A -> alpha A' | Y_1 | Y_k
            con A' -> beta_1 | ... | beta_n e A' nuovo simbolo
        }
    }
```

manca roba

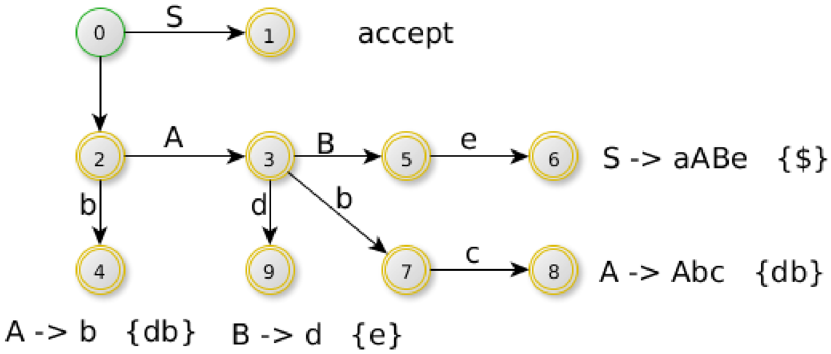
8.12 Bottom Up

Ricostruire, se $w \in L(G)$, una rightmost derivation al contrario

8.12.1 Esempio

$S \to aABe$
$A \to Abc|b$
$B \to d$

w = abbcde visto che é rightmost devo espandere B dato che é il non terminale piú a destra. $S \to aABe \to aAde \to aAbcde \to abbcde$



La sottolineatura significa che se arrivo in questo stato e sto leggendo come prossimo input una d o una b posso fare la riduzione della b usando A. Lo stesso vale per le altre, ovviamente con i loro simboli. La roba fra parentesi graffe si chiama look-ahead set.

Nel grafo faccio quindi i seguenti passi (i numeri sono i nodi):

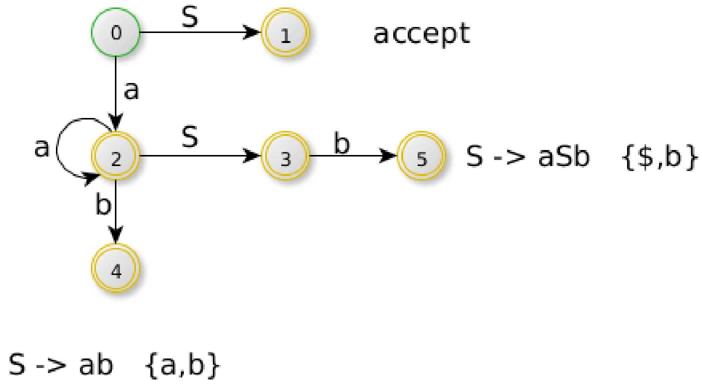
0	$abbcde\$$
$0 \to 2$ consumando 'a '	$a\|\|bbcde\$$
$2 \to 4$ consumando 'b '	$ab\|\|bcde\$$
4 riduco $A \to b$	$aA\|\|bcde$

A questo punto torno al nodo 2 ovvero il precedente. Vado quindi in 3, perché ho la A al posto della b che avevo prima.

torno a 2, vado in 3

$3 \to 7$ consumando 'b '	$aAb\|\|cde\$$
$7 \to 8$ consumando 'c '	$aAbc\|\|de\$$
8 riduco $A \to Abc$	$aA\|\|de$
torno a 7, torno in 3, vado in 9	
$3 \to 9$ consumando 'd '	$aAd\|\|e\$$
riduco $B \to d$	$aAB\|\|e\$$
torno a 3, vado in 5, vado in 6	
$5 \to 6$ consumando 'e '	$aABe\|\|\$$
6 riduco $S \to aABe$	$S\|\|\$$
torno a 0, vado in 1, ho finito	

Noi vogliamo avere grammatiche di tipo LALR(1). Grammatiche: $SLR(1) \subset LALR(1) \subset LR(1)$



$S \to aSb|ab$

$w = aaabbb\$$

0	$aaabbb\$$
$0 \to 2$	$a\|\|aabbb\$$
$2 \to 2$	$aa\|\|abbb\$$
$2 \to 2$	$aaa\|\|bbb\$$
$2 \to 4$	$aaab\|\|bb\$$
4 riduco $S \to ab$	$aaS\|\|bb\$$
torno a 2, vado in 3, vado in 5	
$3 \to 5$	$aaSb\|\|b\$$
5 riduco $S \to aSb$	$aS\|\|b\$$
torno a 3, vado in 5	
$3 \to 5$	$aSb\|\|\$$
5 riduco $S \to aSb$	$S\|\|\$$
torno a 0, vado in 1, ho finito	

Questa é una tabella:

	terminali $\cup\$$	$V \backslash T$
stati	shift-k: leggi un simbolo di input e vai allo stato reduce $A \to b$	goto-k: descrive le funzioni di tr identificate dai non terminali qu

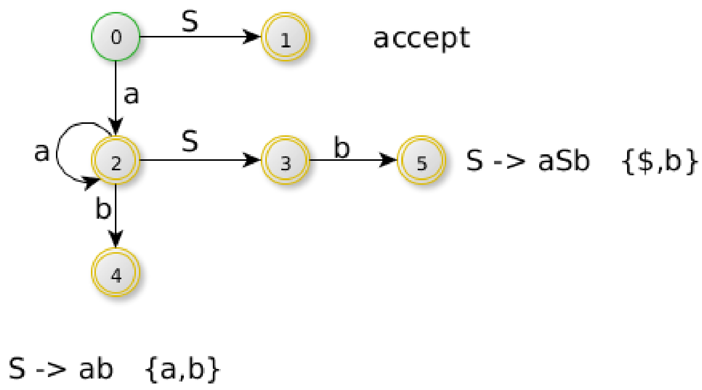
8.13 Algoritmo di shift/reduce

(comune a SLR(1), LR(1), LALR(1))

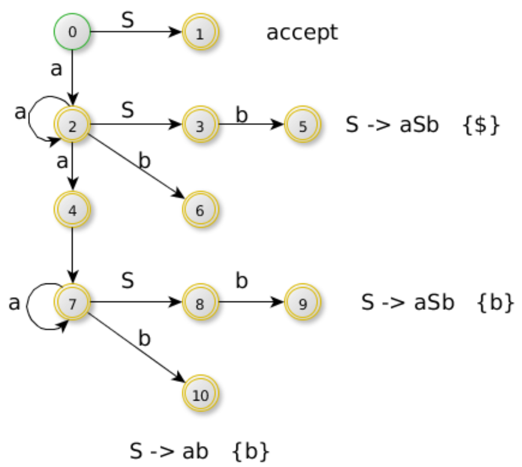
input	w, tabella di parsing bottom-up di tipo $\diamond$, con $\diamond$ scelto fra $\{SLR(1), LALR(1), LR(1)\}$ G.
output	derivazione rightmost di w se $w \in L(G)$, altrimenti error()

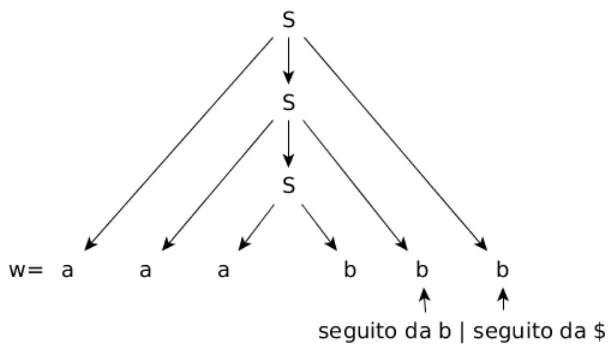
```
stack.push(s_0);
buffer = w$;
while(true){
    let s = stack.top();
    if(M[s,b] == shift-k){
        stack.push(b);
        stack.push(k);
        let b = buffer.readNext();
    } else if(M[s,b] == "reduce A -> beta"){
        stack.pop() 2|beta| simboli;
        let j tale che M[m, A] = gj;
        push(A);
        push(j);
        output "A -> beta";
    } else if(M[s,b] = accetta){
        break;
    } else {
        error();
    }
}
```

sketo $S \to aSb|ab$



Questo é uguale ma scritto diversamente per separare $\{\$, b\}$ in $\{4\}$ e $\{b\}$. Nel caso di $w = aaabbb\$$. Una caso rappresenta il rao piú in alto, mentre l'altro il secondo ramo (piú interno).





- Automa caratteristico
- Lookahead Function

Coppie diverse di questi due insiemi ci danno tipi di grammatiche diverse.

Gli automi che stiamo utilizzando devono essere in grado di ricordare abbastanza da essere in grado di tornare indietro fino al punto in cui abbiamo sostituito una certa sequenza di terminali/non terminali con un'altra.

$G = (V, T, S, P)$, aggiungo una produzione $S' \to S$, $G' = (V \cup \{S'\}, T, S', P \cup \{S' \to S\})$ $A \to \alpha.\beta$

All'inizio ho .S, ovvero non ho ancora letto nulla e devo leggere S.

All'inizio (il nodo iniziale), non ho ancora visto nulla. Visto che S puó iniziare con aSb o ab non sappiamo davanti a quale sviluppo ci troviamo. Il primo stato é quindi

$S' \to .S$
$S \to .aSb$
$S \to .ab$

Questo puó essere visto come un nodo. Da questo stato mi muovo verso un altro stato (con una a-transizione, perché vedo che iniziano quasi tutte con a). In questo stato avró: $S \to a.Sb$
$S \to a.b$

Adesso mi aspetto di vedere l'espansione di una S. Devo quindi aggiungere a questo nodo anche quelle produzioni, e diventa quindi: $S \to a.Sb$
$S \to a.b$
$S \to .aSb$
$S \to .ab$

Notare che le ultime due sono le stesse delle ultime due del nodo prima. Quella é la chiusura, mentre le due prima sono i generatori dello stato (kernel dello stato, **kernel items**).

Gli stati che sono terminali (ovvero che nol disegno prima avevano le transizioni scritte vicino), sono del tipo $S \to ab.$, ovvero che hanno incontrato di tutto e di cui si puó eseguire la riduzione. Questi si chiamano **reducing items**.

Dallo stato con 4 items che avevo prima, si puó fare una b-transizione che va in uno di quelli stati terminali, ovvero: $S \to ab.$
Questo perché la seconda produzione si aspetta b, che poi completa quello che viene generato da quella produzione. Sempre da quello stato con 4 produzioni partirá anche una a-transizione ed una S-transizione. Per vedere che transizioni devo avere, devo vedere la prima lettera dopo il punto per ogni item di quel nodo.

8.14 Items

$G = (V, T, S, P)$
$G' = (V \cup \{S'\}, T, S', P \cup \{S' \to S\})$, con $S' \notin V$.

Un LR(0)-item di G' é una produzione di G con un punto in qualche posizione del body, ovvero $A \to \alpha.\beta$. Alla produzione della forma $A \to \varepsilon$ corrisponde un solo LR(0)-item, ovvero $A \to .$

	iniziale	se $A = S' \wedge \alpha = \varepsilon \wedge \beta = S$, cioé se l'item é $S' \to .S$
	accepting	se $A = S' \wedge \alpha = S \wedge \beta = \varepsilon$, cioé se l'item é $S' \to S.$
L'item $A \to \alpha.\beta$ é detto:	kernel	se é un iniziale o tale che $\alpha! = \varepsilon$
	closure	se $\alpha = \varepsilon$ e non é iniziale
	reducing	se non é accepting e $\beta = \varepsilon$, cioé se il punto é in fondo $\wedge !accepting$