LFC (Linguaggi Formali e Compilatori) - Note del Corso

Edoardo Lenzi

January 31, 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}.

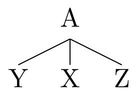
2.2.1 ie)

```
//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).

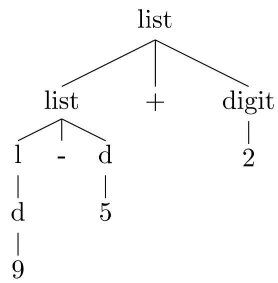
Uso **parse trees** $A \to XYZ \implies$



Regole di costruzione:

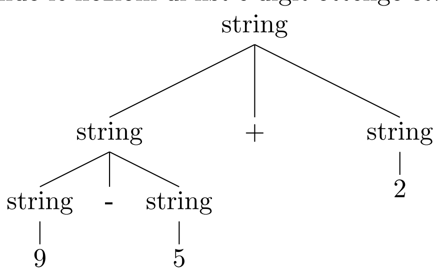
- la root é lo start symbol
- le foglie sono terminali o ε
- i nodi interni sono non-terminali
- se A non-terminali ha figli $x_1, ..., x_n \implies A \to x_1...x_n$

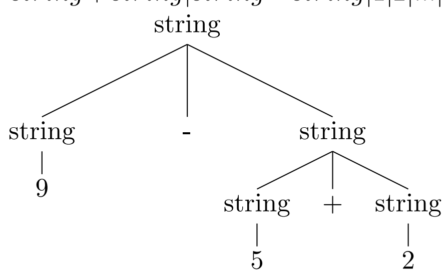
9-5+2



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)

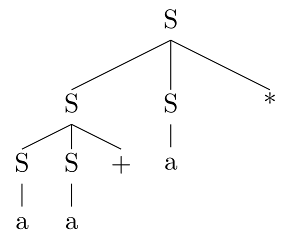
expr = list {terms separati da $*/$}

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

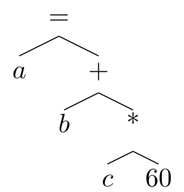
Chapter 4

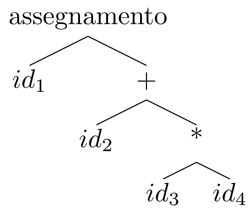
Introduzione

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**.

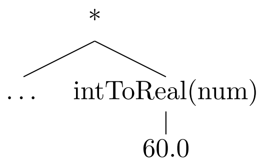
L'analisi sintattica invece studia la **grammatica del linguaggio**. Viene costruito un **abstract syntax tree**:

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

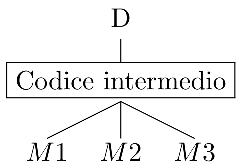




L'analisi semantica si occupa di vedere se c'é una corretta semantica (variabili dichiarate precedentemente). Se * necessita di un float allora 60 dev'essere convertito a float.



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 $\alpha \to \beta$ 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 $L = (a^k b^n \ / \ k, n > 0)$

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

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

$S \to ab|aS|Sb$

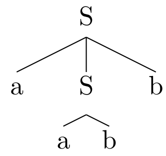
Definire G tale che $L(G) = \{a^k b^n c^{2n} \ / \ k, n > 0\}$
$S \to AB$
$A \to aA|a$
$B \to bBcc|bcc$

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

$$S \to aSb|ab$$



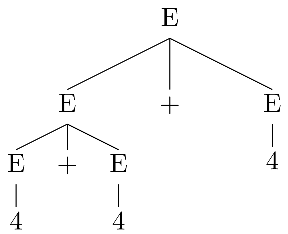
aabb, derivazione canonica $\mu \to \gamma$

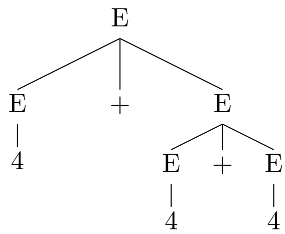
5.3 Grammatiche ambigue

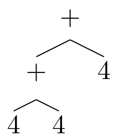
Nel caso di grammatiche libere si definiscono le **derivazioni canoniche destre e sinistre**, nel caso **rightmost** si richiede che ad ogni passo di derivazione $\mu \to \gamma$ venga rimpiazzato il terminale piú a destra di μ; nel caso **leftmost** quello piú a sinistra.

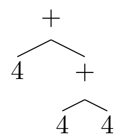
G é **ambigua** se esiste una parola del linguaggio generato da G, generabile con due derivazioni canoniche distinte entrambe destre o entrambe sinistre.

$E \to E + E|E * E|4$ (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.

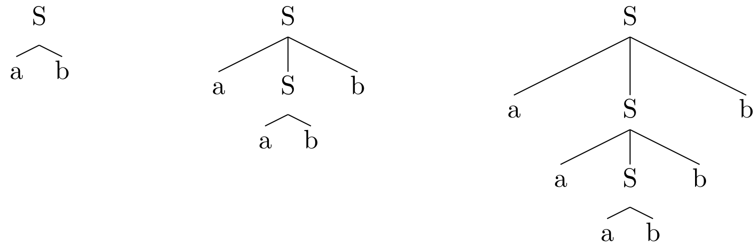
Osservazione 2: Una grammatica si puó portare in forma normale (di Chomsky) togliendo eventuali ridondanze o produzioni che terminano per forza con ε (a meno che non debba fare parte del linguaggio, ma a quel punto si scrive come $S \to \varepsilon$).

CNF

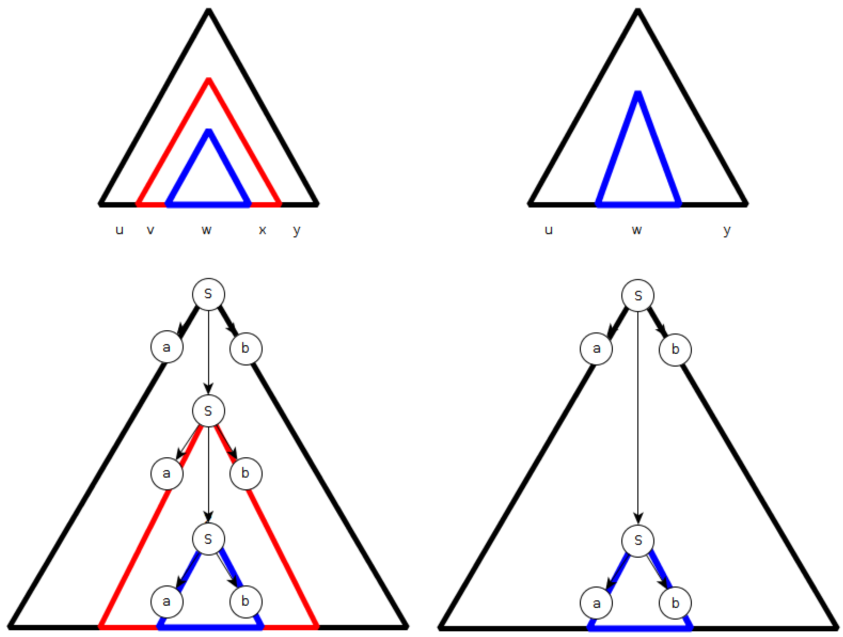
Chomsky Normal Form esige che una grammatica non abbia produzioni ridondanti:
ie) G_1 $S \rightarrow aSb|ab|B,\ B \rightarrow aBb|ab \leftarrow$ doppioni

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 ($|V \backslash T|$).
$S \rightarrow aSb|ab$, ha due non terminali, p = 2



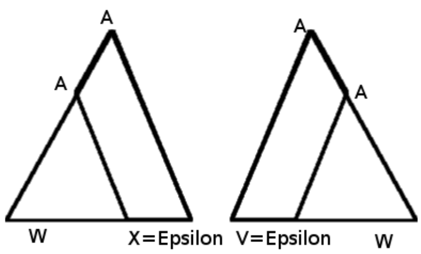
Guardo il cammino S $\rightarrow S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_K$ di lunghezza K Se $z \in L\ \wedge\ |z| > p \implies$ nell'albero di derivazione di z esiste almeno un cammino la cui lunghezza é maggiore di $|V \backslash T| \implies$ allora esiste almeno un non-terminale che occorre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



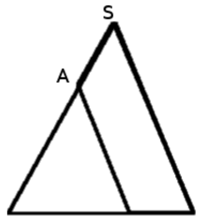
ie) p = 2, se prendo una qualunque parola piú lunga di 3 generata da G: aaaabbbb, la posso dividere in 5 con due pumpabile: u=aa, v=a, w=abb, x=b, y=b

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 $A \rightarrow aA,\ o\ A \rightarrow Aa$



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



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) $|vx| > 0 \wedge$

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

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

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

Suppongo $L_1 = w_1 w_2 \ / \ w_1 = w_2, \ w_1, w_2 \in \{a, b\}^*$ **libero**;
Sia p un naturale qualunque sempre positivo;
Sia $z = a^p b^p a^p b^p$ $(|z| = 4p)$;
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 $z^1 = uv^0 wx^0 y$ (i=0);

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

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

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

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

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

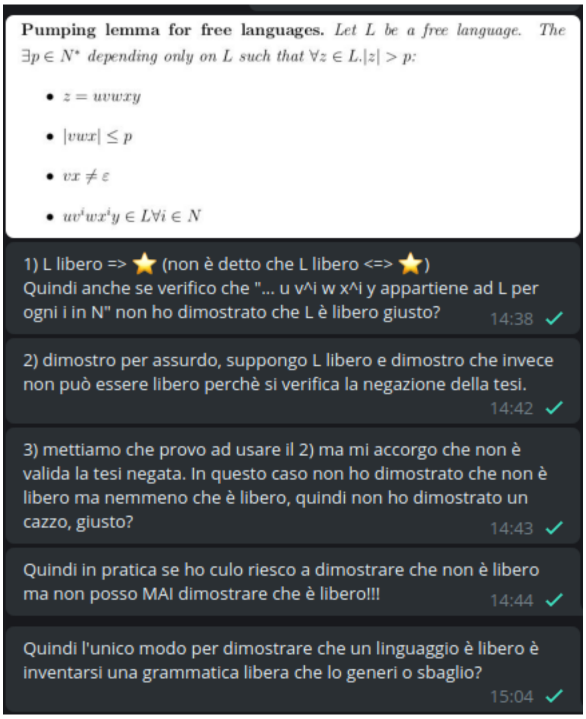
- Nel caso 2 $z^1 = a^kb^pa^pb^p,\ o\ a^pb^ka^pb^p,\ o\ a^jb^ka^pb^p,\ j,k < p \implies z^1 \notin L$.

- ...

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

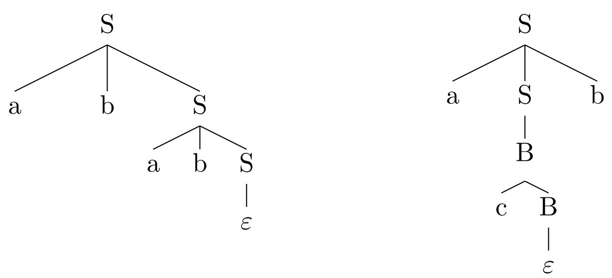
5.6.1 How not to use Pumping Lemma

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.

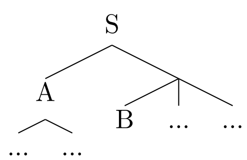


5.6.2 Esempi

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



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



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

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
$S \to aSc|aBc$
$B \to bBc|bc$

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

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

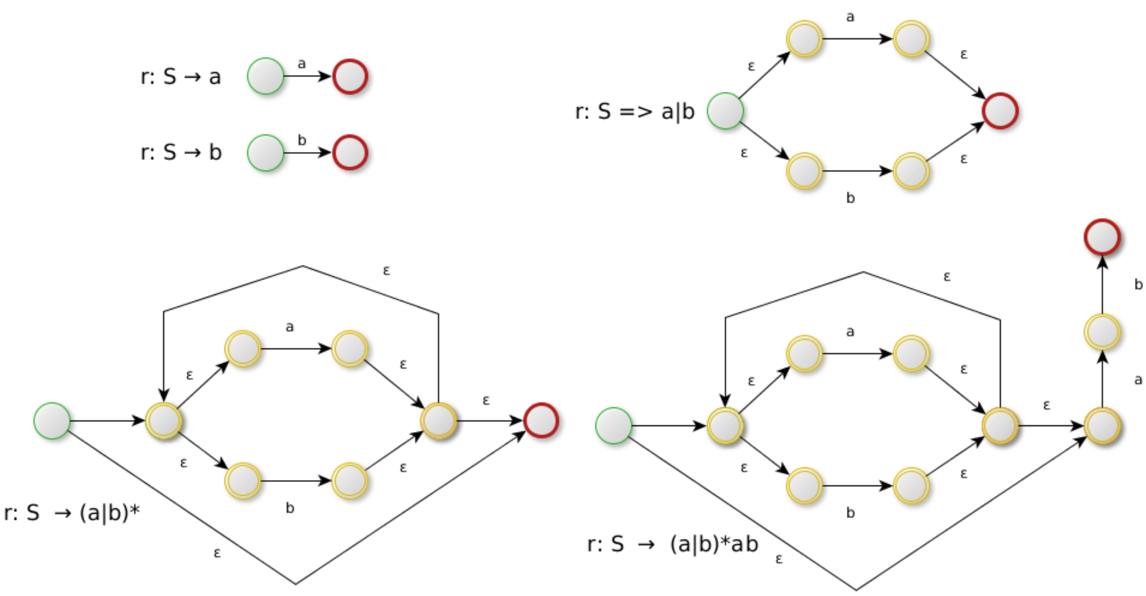
6.1 Thompson construction

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 ($S \otimes A \to S$)
$t \in S, T \subset S$

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

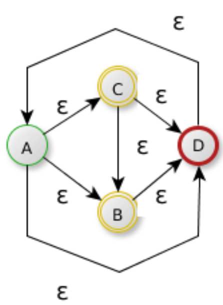
Questo algoritmo é piú performante del backtracking.

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:

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

- 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	$DFA(S^d, A, move_d, S_0^d, F^d)$

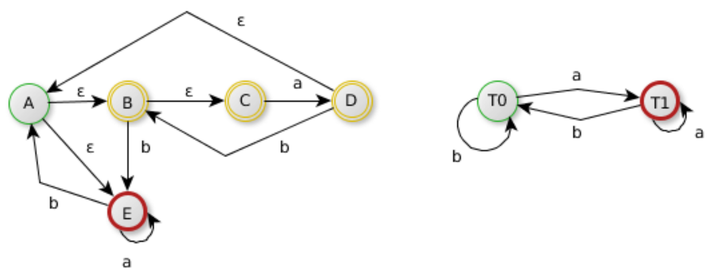
```
S_0^d = epsilon-closure({S_0^n});
 //raggruppo stati della epsilon closure in un unico stato S_0^d del DFA
states = {S_0^d};
tag S_0^d come non marcato;

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 $\varepsilon - closure$.

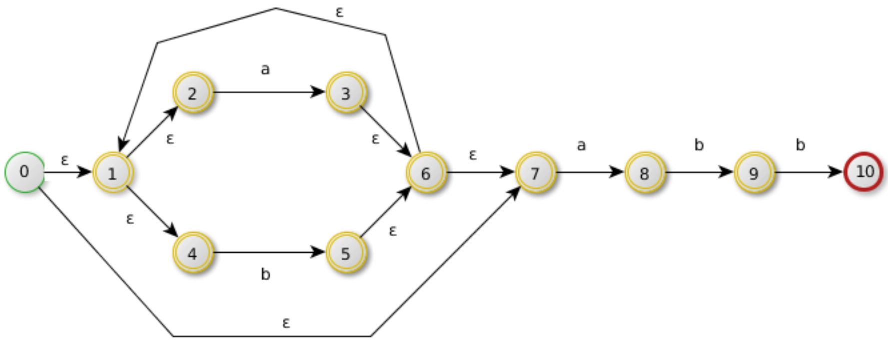
Alla fine guardo i set degli stati se due set coincidono mergio gli stati.

6.4.1 Esercizio

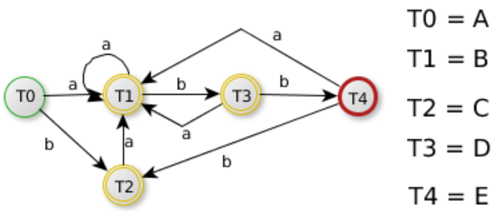


States	**a**	**b**
T0 = { **A** B C E }	T1	T2 = T0
T1 = { A B C **D** **E** }	T1	T0
T2 = { **A** B C **E** } = T0 (quindi T0 va in T0 tramite b)	come T0	come T0

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

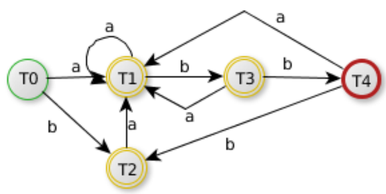
> Guado gli archi, se tutta la partizione punta ad un nodo dell'altra transizione con lo stesso non terminale allora va bene; altrimenti spacco la partizione.

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));
}
```

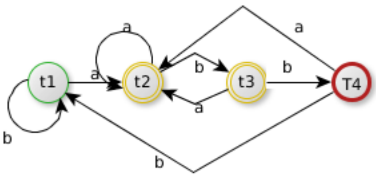
6.5.2 Esempio



T0 = A
T1 = B
T2 = C
T3 = D
T4 = E

{ 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 $(\mathbf{min(D)})$ che riconosce lo stesso linguaggio del primo

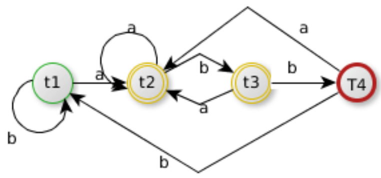
```
P = PartitionRefinement(DFA D);
// P = (B_1, ..., B_k);
foreach(B_i in P){
    var t_i = B_i;          //do un nome alla partizione, un alias
    if(s_o in B_i){
        t_i e'' iniziale per min(D); //setto lo stato iniziale di min(D)
    }
}

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.

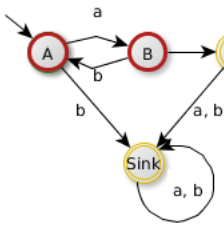
6.6.1 Esempio

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

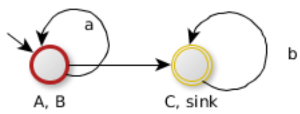


6.6.2 Esempio

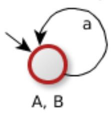
Aggiungo il nodo sink (se richiesta fn transizione completa)



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

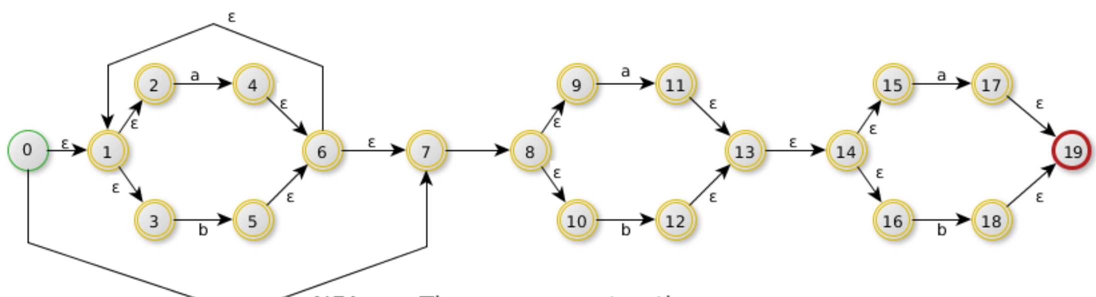


Visto {C, sink} un dead state per il grafo, posso eliminarlo

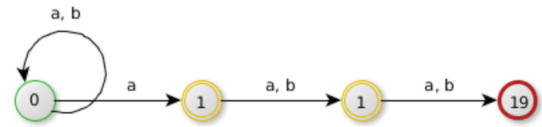


6.6.3 Esempio

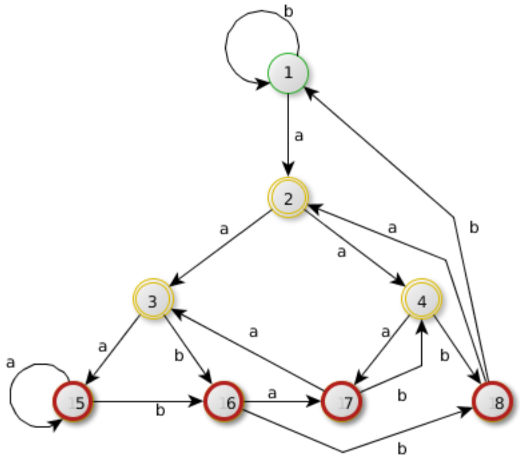
Sia $r = (a|b) * a(a|b)(a|b)$, per determinare il minimo DFA di riconoscimento di L(r) posso usare Thompson e spararmi in faccia o andare a naso.



NFA con Thompson construction



NFA a naso

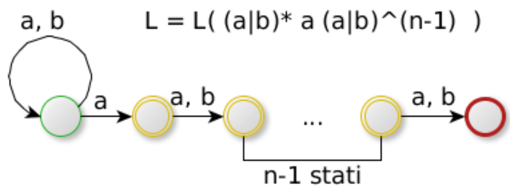


DFA Totale

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.

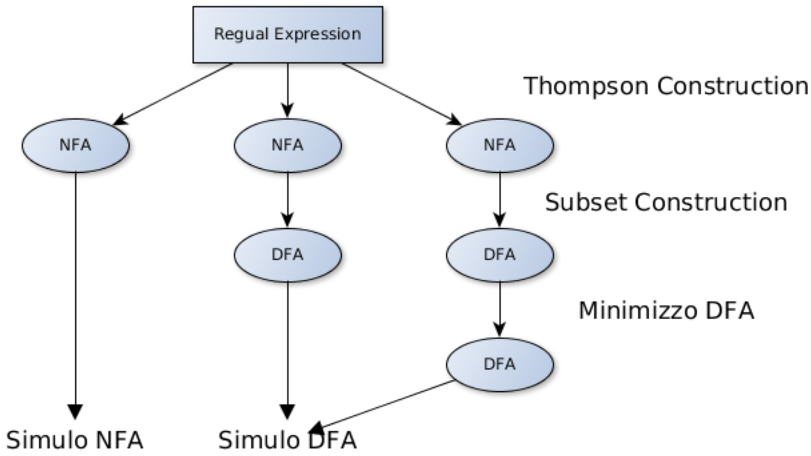
Dim:



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

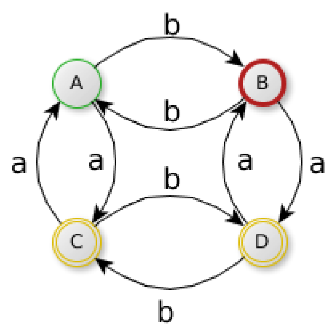
In pratica é la diretta trascrizione di un DFA in regex!

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

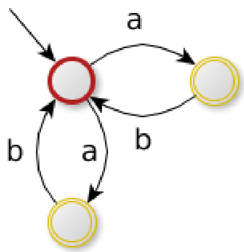
Regular expression, NFA e DFA hanno la stessa potenza espressiva, sono solo notazioni diverse.

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]

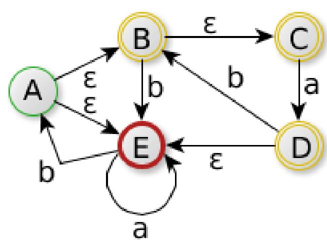
7.2.4 Esercizi di esame

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

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

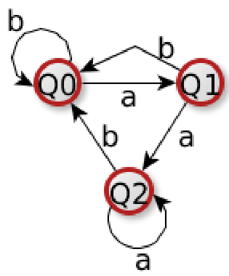
1) $aa \in L(N_1)$?

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:

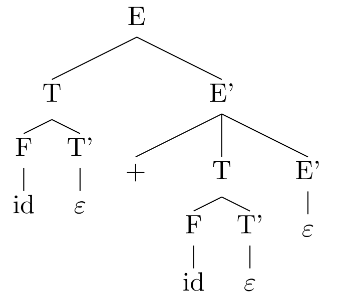
```
j = 1;
while(j <= n){
    aggiungere ai first(X) ogni b tale che b in first(Yj)
    if(epsilon in first(Yj)){
        j++;
    } else {
        break;
    }
}

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...

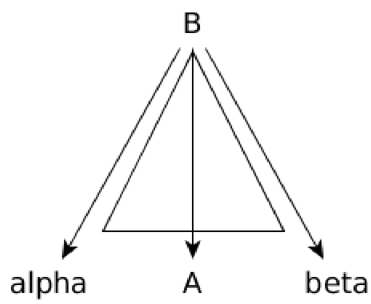
	id	+	*	$
E	$E \rightarrow TE'$			
E		$E' \rightarrow TE'$		$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			

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	
$S =$	$\{a, b\}$	$\{\$\}$	
$A =$	$\{e, d, f, c\}$	$\{\$, (?)\}$	Occhio che nei first di A non ci va ε perché ci può essere dalla
$B =$	$\{e, \varepsilon\}$	$\{c, d\}$	
$C =$	$\{f, \varepsilon\}$	$\{c\}$	

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

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

First and Follow

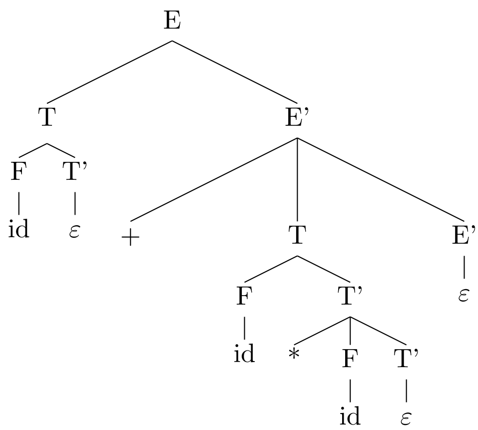
Non terminale	**First**	**Follow**
E	id	\$
E'	+, ε	\$
T	id	+, \$
T'	*, ε	+, \$
F	id	+, *, \$

Tabella di parsing

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

Passi dell'algoritmo

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



8.7.3 Esercizio

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

$w = ac\$$

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

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

8.9.3 Esempio

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

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

(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

$S \rightarrow aSb|ab$

	a	b	$
S	$S \rightarrow aSb$ $S \rightarrow ab$		

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.
$S \rightarrow aA'$
$A' \rightarrow Sb|b$

DEF

Una grammatica G puó essere fattorizzata a sinistra quando esistono almeno due produzioni $A \rightarrow \alpha\beta_1$ e $A \rightarrow \alpha\beta_2$ 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: $A \rightarrow \alpha\beta_1,\ A \rightarrow \alpha\beta_2 \in P$ 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
```

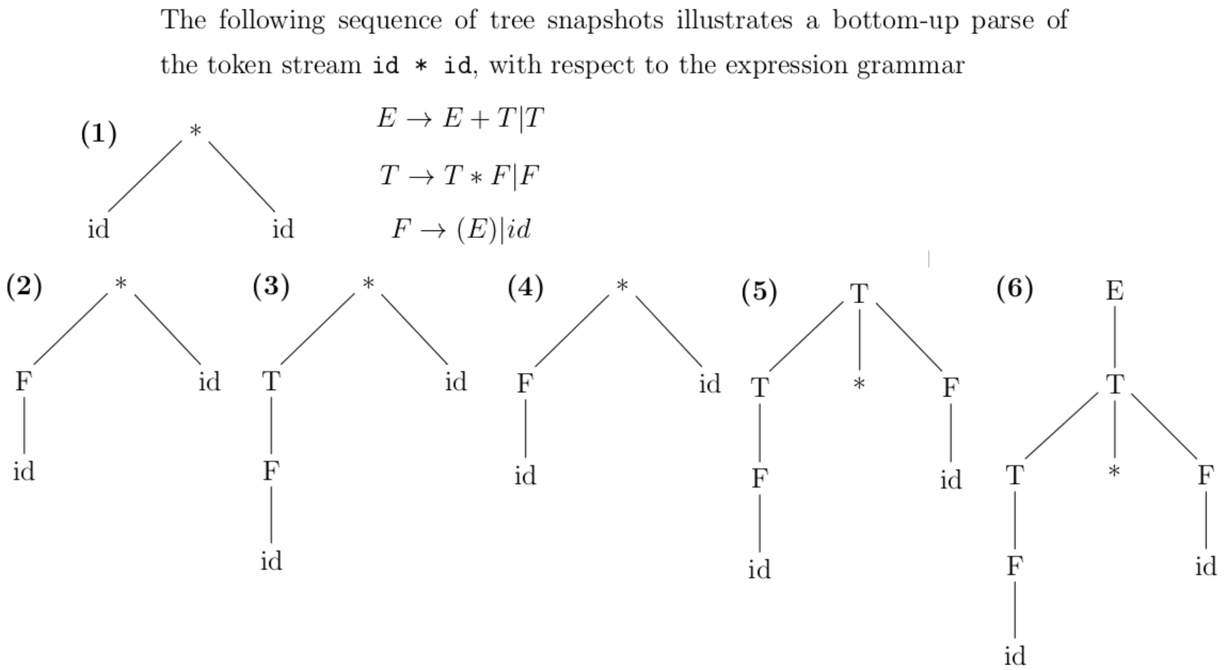
```
        con A -> alpha A' | Y_1 | Y_k
        con A' -> beta_1 | ... | beta_n e A' nuovo simbolo
    }
}
```

---

manca roba

Chapter 9

Parsing Bottom Up (Zampedri)



Riduzione:	$id * id,\ F * id,\ T * id,\ T * F,\ T,\ E$
Derivazione:	$E \to T \to T * F \to T * id \to F * id \to id * id$

9.1 Reductions

La **riduzione** é l'operazione inversa della **derivazione**. Posso pensare al parsing bottom up come al processo di riduzione di una stringa w allo start symbol della grammatica. In ogni step di riduzione una sottostringa che matcha con il body di una produzione viene ridotta con la testa della produzione.

Le scelte critiche sono quando ridurre e cosa ridurre.

9.2 Handle Pruning

Chiamo handle una sottostringa che matcha con il body di una produzione, e la sua riduzione é un passo inverso, al contrario, di una rightmost derivation.

Right Sentential Form	Handle	Reducing Production
id_1 * id_2	id_1	F → id
F * id_2	F	T → F
T * id_2	id_2	F → id
T * F	T * F	E → T * F

9.3 Shift reduce parsing

É una forma di bottom up parsing in cui mi appoggio ad uno stack contenente i simboli della grammatica e un buffer per la input string da parsare.

```
//init
stack = $;
buffer = w$;
//faccio uno scan da sx a dx e shifto simboli dal buffer allo stack
//quando nello stack posso fare una riduzione sostituisco
//vado avanti fince' non ho un error o finisce il buffer
```

Sembra che qualsiasi stringa possa essere parsata, basta continuare a shiftare???

`id₁ * id₂`

$E \rightarrow E + T|T$

$T \rightarrow T * F|F$

$F \rightarrow (E)|id$

Stack	Input	Action
\$	$id_1 * id_2$ \$	shift
\$ id_1	$* id_2$ \$	reduce by F → id
\$ F	$* id_2$ \$	reduce by T → F
\$ T	$* id_2$ \$	shift
\$ T *	id_2 \$	shift
\$ T * id_2	\$	reduce by F → id
\$ T * F	\$	reduce by T → T * F
\$ T	\$	reduce by E → T
\$ E	\$	accept

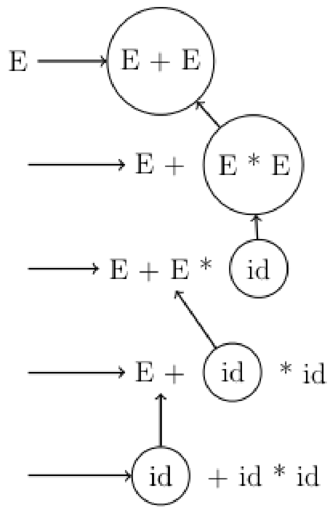
Ho **shift** quando passo un simbolo dal buffer allo stack, **reduce** quando applico una riduzione sul top dello stack (mai inside), **accept** se sono in fondo al buffer e ho quindi parsato correttamente la stringa, **error** nol va!

L'algoritmo shift reduce non funziona per tutte le grammatiche, puó essere che in certi casi mi trovi in situazioni dove non so decidere se fare shift o reduce o non so quale reduce scegliere (reduce conflicts).

9.3.1 Esempio

$E \rightarrow E + E|E * E|(E)|id$

`id + id * id`



stack	input	
\$	id + id * id \$	(S)
\$ id	+ id * id \$	(R: $E \rightarrow id$)
\$ E	+ id * id \$	(S)
\$ E +	id * id \$	(S)
\$ E + id	* id \$	(R: $E \rightarrow id$)
\$ E + E	* id \$	(R: $E \rightarrow E + E$)
\$ E	* id \$	(S)
\$ E *	id \$	(S)
\$ E * id	\$	(R: $E \rightarrow id$)
\$ E * E	\$	(R: $E \rightarrow E * E$)
\$ E	\$	

9.4 LR parsing

Il metodo piú usato é LR(k) parsing con

L	left to right input scan
R	rightmost derivation costruita inversamente
(k)	numero di input symbol lookahead messi per prendere le decisioni

Se ometto k assumo che sia k = 1 (LR = LR(1)). Una grammatica é LR(1) se capace di parsare in shift reduce le stringhe (riconosce gli handles sul top dello stack).

9.4.1 LR(0) Automaton

Per prendere decisioni critiche mi serve tener traccia dello stato in cui mi trovo nel parsing. Per rappresentare gli stati uso gli items; un item é una produzione con un punto: $A \to XYZ$ codifica 4 items: $A \to .XYZ$, $A \to X.YZ$, $A \to XY.Z$, $A \to XYZ.$. La produzione $A \to \varepsilon$, mi produce solo un item $A \to$ . .

Intuitivamente $A \to X.YZ$ mi dice che ho giá visto X ma non YZ.

canonical LR(0) collection é un set di item LR(0) che mi permettono di costruire DFA in grado di prendere decisioni precise per il parsing; in particolare ogni stato nel automa rappresenta un set di item del canonical LR(0) collection.

Per costruire l'automa definisco una grammatica aumentata e le funzioni CLOSURE e GOTO. Data G con start symbol S, G' é una grammatica aumentata di G ed ha la produzione $S' \to S$ (start symbol S'). Questo serve essenzialmente per poter annunciare la fine del parsing.

Closure of Item Sets

Dato I set di item per G, CLOSURE(I) é il set degli item costruito da I /

i) inizialmente closure(I) = I

ii) se $(A \to \alpha.B\beta) \in closure(I) \wedge (B \to \gamma) \in P$ aggiungi $B \to .\gamma$ a closure(I) (se non giá presente).

Vado avanti ricorsivamente finché non ho aggiunto tutto quello che potevo.

Example.

Consider the augmented expression grammar:

$$E^{'} \to E$$

$$E \to E + T|T$$

$$T \to T * F|F$$

$$E \to (E)|id$$

To see how closure is compute, $E^{'} \to E$ is put in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, we add the E-productions with dots at the left ends: $E \to .E + T$ and $E \to .T$. Now, there is a T immediately to the right of a dot in the latter item, so we add $T \to .T * F$ and $T \to .F$. Next, the F to the right of a dot forces us to add $F \to .(E)$ and $F \to .id$, but no other items need to be added.

Sembra che $E' \to E$ sia lo stesso di $E' \to .E$; in pratica per ogni item guardo se ho un non terminale subito a destra del punto e casomai inserisco tutti gli item con tale simbolo in testa.

```
J = I;
repeat{
    foreach( [A -> alpha . B beta] in J)
        foreach( [B -> gamma] in G)
            if( [B -> .gamma] not in J)
                J.add( [B -> .gamma] );
} until (saturation) //nothing more can be added!
```

GOTO funtion

GOTO(I, X), I set of items, $X \in V$, $\triangleq closure(\{[A \to \alpha X.\beta] \ / \ [A \to \alpha.X\beta] \in I\})$.

In pratica lo uso per definire le transizioni nell'automa LR(0); gli stati sono set di item e le transizioni sono definite dai GOTO(I, X) (dallo stato I per il simbolo X).

9.4.2 Esempio

$$I = \{[E' \to E.],\ [E \to E. + T]\} \quad \Bigg| \quad G = \begin{array}{l} E' \to E \\ E \to E + T | T \\ T \to T * F | F \\ E \to (E) | id \end{array}$$

$$GOTO(I, +) = \left| \begin{array}{ll} [E \to E + .T], & \text{Mi arriva +, scarto } [E' \to E.]\text{, tengo l'altro'} \\ [T \to .T * F], & \text{Includo i primi item per le produzioni con T in testa} \\ [T \to .F], & \\ [F \to .(E)], & \text{Visto che ho } [T \to .F] \text{ includo gli item con F} \\ [F \to .id], & \end{array} \right|$$

9.5 Algoritmo generazione canonical collection C

Per trovare C, canonical collection of sets of LR(0) items for an augmented grammar G'.

```
C = [CLOSURE{S' -> S}]
repeat{
    foreach(I in C){ //set of item I
        foreach(X in V'){
            if(GOTO(I,X) != empty && not in C yet){
                C.add(GOTO(I,X));
            }
        }
    }
} until (saturation)
```

9.6 Simple LR (SLR)

SLR parsing consiste nella costruzione della grammatica dell'automa LR(0). Gli stati dell'automa sono set di item derivanti da C (canonical LR(0) collection) e le transizioni derivano dalla GOTO functions.

Lo start state dell'automa LR(0) é $CLOSURE([S' \to .S])$ (S' start symbol di G' grammatica aumentata).

Per prendere decisioni, immaginando di arrivare con la stringa γ dallo stato 0 allo stato j; a questo punto posso shiftare il prossimo carattere a se j ha transizioni per a oppure ridurre. Gli item in j mi dicono cosa fare.

9.6.1 Esempio

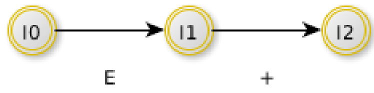
$$G = \left| \begin{array}{l} E \to E + T | T \\ T \to T * F | T \\ F \to (E) | id \end{array} \right.$$

Aumento la grammatica con $E' \to E$.

$$\text{Computo } CLOSURE(\{E' \to E\}) : \boxed{I_0 = \begin{array}{l} E' \to .E, \\ E \to .E + T, \\ E \to .T, \\ T \to .T * F, \\ T \to .F, \\ F \to .(E), \\ F \to .id \end{array}}$$

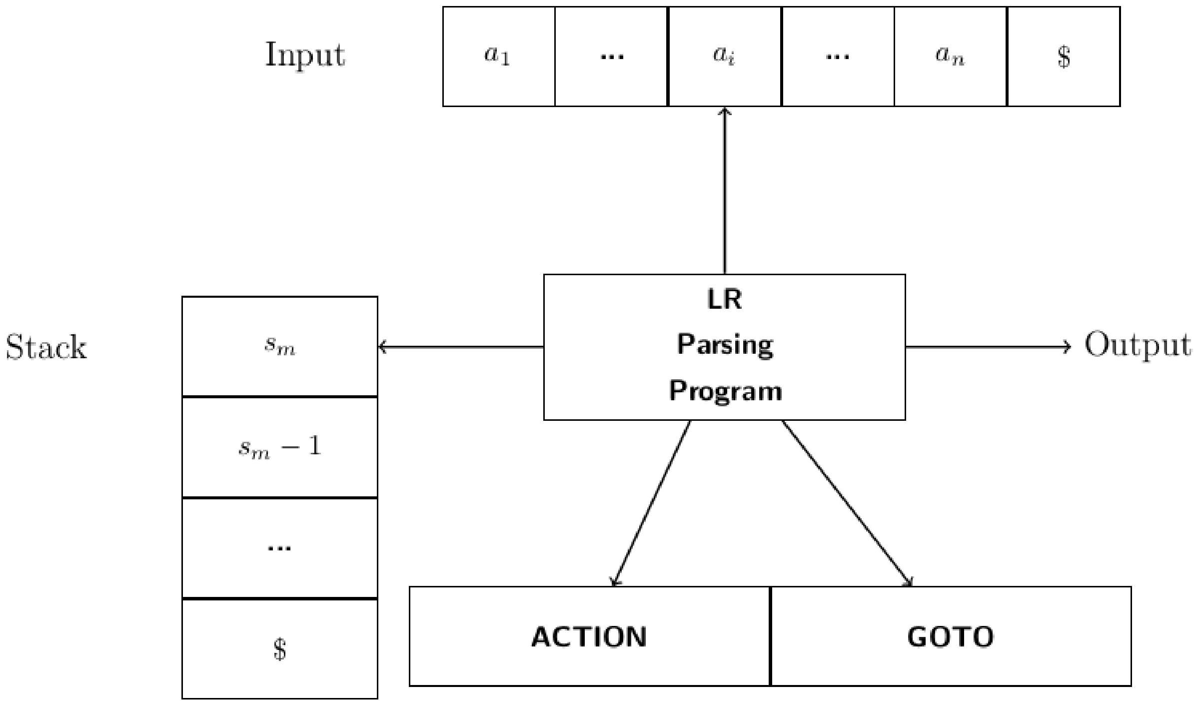
$$\text{Computo i sets} \begin{array}{ll} I_1 = & GOTO(I_0, E) = \{E' \to E.,\quad E \to E. + T\} \\ I_2 = & GOTO(I_1, +) = \{E \to E + .T,\quad T \to .T * F,\quad T \to .F,\quad F \to .(E),\quad F \to .id\} \end{array}$$

Quindi l'automa inizia con una forma di questo genere:



9.7 The LR-Parsing Algorithm

An LR parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO).



The driver program is the same for all LR parsers; only the parsing table changes from one parser to another.

La differenza fra ACTION e GOTO é che le action sono applicate per input di terminali, GOTO per input di non terminali!

9.7.1 Structure of LR Parsing Table

La parsing table é strutturata in due parti: a parsing-action function **ACTION** and a goto funtion **GOTO**. $ACTION[i, a], i \in States, a \in T \cup \{\$\}$ puó assumere varie forme:

- **Shift** j, $j \in States$ (shifta un carattere a ma usa j per rappresentarlo);
- **Reduce** $A \to \beta$ (riduce β ad A)
- **Accept**
- **Error**

La prossima mossa del parser sará leggere il simbolo corrente a_i e lo stato in cima allo stack s_m. Guardo $ACTION[s_m,\ a_i]$ nella parsing action table.

9.7.2 LR Parsing Algorithm

Input	string w, LR-parsing table con ACTION e GOTO per la grammatica G
Output	gli step di riduzione bottom-up per w o un errore

```
stack.push(0); //start state 0
buffer.push(w$);
* ip = leftChar(w); //pointer sul buffer
repeat{
    n = stack.top();
    a = simbolo puntato da ip;
    if(T[n,a] == s_m){ //shift m
        push(a)
        push(m)
        move ip right
    } else if(T[n, a] = r_k con (k): A -> beta){ //reduce k
        pop 2^{|beta|} simboli;
        n' = stack.top();
        push(A);
        push m / T[A, n'] = g_m;
        cout << "A -> beta";
    } else if (T[n, a] == accept){
        accept();
    } else {
        error();
    }
} until (accept() || error());
```

9.7.3 Costruire SLR-Parsing Table

Il metodo SLR inizia con items LR(0) ed un automa LR(0). Poi aumento la grammatica G, mi trovo C, GOTO e ACTION.

G' non é SLR se costruendo la parsing table ho dei **conflitti**.

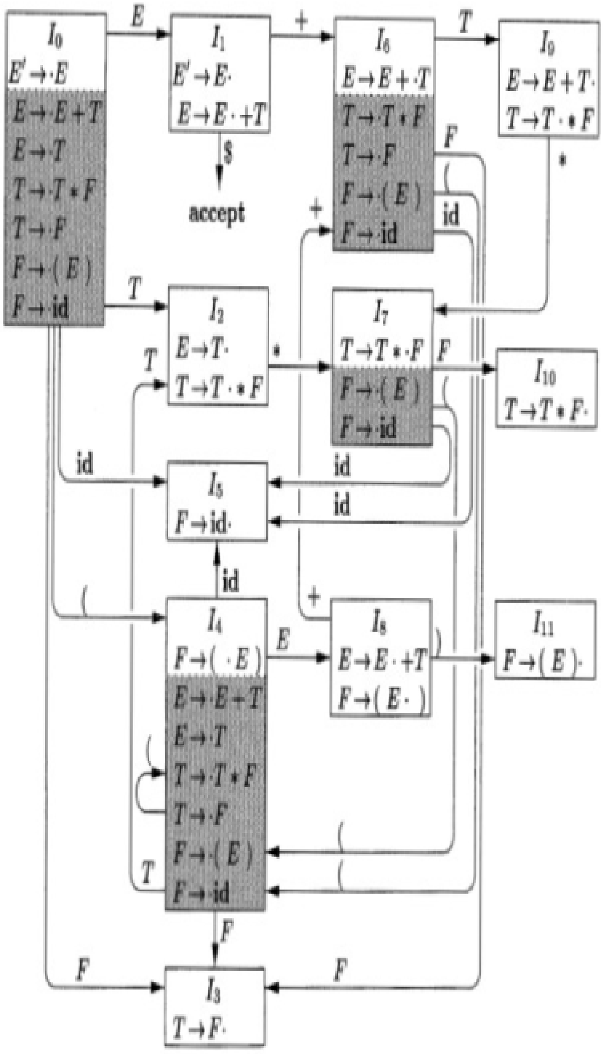
Input	G' grammatica arricchita
Output	Parsing table o un errore

```
Define [I_0, ..., I_n] set items LR(0) per G'
Define lo stato i nella tabella (ie row i) dopo l'elemento I_i come segue:

* if (A -> alpha . a beta in I_i && GOTO(I_i, a) == I_j){
    T[i, a] = s_j; // shift corrisponde ad un cambio di stato
}
* if (A -> alpha . in I_i){
    foreach(x in FOLLOW(A)){
        T[i,X] = r"A -> alpha"; // reduce
    }
}
* if (S' -> S. in I_i){ //ACCEPT
    T[i, $] = accept;
}

if(le azioni precedenti generano conflitti){
    throw Exception("G is not SLR"); //G non e' SLR
} else {
    if(GOTO(I_i, A) == I_j){
        T[i, a] = g_j; // goto
    }
    Put error() nelle entry vuote //ERROR
    I_0 = CLOSURE(S' -> .S)
}
```

9.7.4 Esempio

1)$E \to R + T$
2)$E \to T$
3)$T \to T * F$
4)$T \to F$
5)$F \to (E)$
6)$F \to id$

Arricchisco la grammatica con $E' \to E$, trovo **l'automa LR(0)**:



Osservazioni
Per ogni stato guardo solo i simboli dopo il punto negli item
quindi ie l'item $F \to .(E)$ mi genera ACTION[0,(] = shift 4
Da notare che da I_1 con \$ vado in ACCEPT perché ho l'item $E' \to E$.
La closure di $T \to T. * F = \emptyset$, **non espando i terminali**
Considero I_2, dato l'item $E \to T.$ e dato che $FOLLOW(E) = \{\$, +,)\}$
$\implies ACTION[2, \$] = ACTION[2, +] = ACTION[2,)] = REDUCE E \to T$

In pratica l'automa mi crea la parsing table

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

ie. $w = id * id + id$

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		id * id + id $	shift
(2)	0 5	id	* id + id $	reduce by $F \to$ id
(3)	0 3	F	* id + id $	reduce by $T \to F$
(4)	0 2	T	* id + id $	shift
(5)	0 2 7	T *	id + id $	shift
(6)	0 2 7 5	T * id	+ id $	reduce by $F \to$ id
(7)	0 2 7 10	T * F	+ id $	reduce by $T \to T * F$
(8)	0 2	T	+ id $	reduce by $E \to T$
(9)	0 1	R	+ id $	shift
(10)	0 1 6	E +	id $	shift
(11)	0 1 6 5	E + id	$	reduce by $F \to$ id
(12)	0 1 6 3	E + F	$	reduce by $T \to$ F
(13)	0 1 6 9	E + T	$	reduce by $E \to E + T$
(14)	0 1	E	$	accept

Sono in 0, mi arriva id faccio shift 5, pusho 5 sullo stack e id nei simboli; mi arriva *, faccio reduce $F \to id$, sostituisco F a id nei simboli, faccio una pop dallo stack e mi trovo nello stato 0 con simbolo F che é una goto 3 (pusho 3). Occhio che in (7) sono nello stato 10 e mi arriva +, faccio la reduce $T \to T * F$ e devo fare $|T * F| = 3$ pop dello stack.

9.7.5 Shift/Reduce Conflicts

SLR é una grammatica non ambigua, notare che ci sono grammatiche non ambigue che non sono SLR(1). Considero G:

$S \to L = R | R$

$L \to *R|id$
$R \to L$

La Canonical collection C é:
$I_0 = \{S' \to .S,\ S \to .L = R,\ S \to .R,\ L \to . * R,\ L \to .id,\ R \to .L\}$
$I_1 = \{S' \to S.\}$ (da I_0 mi arriva S)
$I_2 = \{S \to L. = R,\ R \to L.\}$ (da I_0 mi arriva L)
$I_3 = \{S \to R.\}$ (da I_0 mi arriva R)
$I_4 = \{L \to *.R,\ R \to .L,\ L \to .*R,\ L \to .id\}$ (da I_0 mi arriva *, poi espando con le closure a cascata)
$I_5 = \{L \to id.\}$ (da I_0 mi arriva id)
$I_6 = \{S \to L = .R,\ R \to .L,\ L \to . * R, L \to .id\}$ (da I_2 mi arriva =)
$I_7 = \{L \to *R.\}$ (da I_4 mi arriva R)
$I_8 = \{R \to L.\}$ (da I_4 mi arriva L)
$I_9 = \{S \to L = R.\}$ (da I_6 mi arriva R)

Se guardo I_2 con il simbolo = il primo item mi suggerisce di fare shift 6, il secondo una reduce $R \to L$ quindi ho un conflitto. In ogni caso la grammatica non é ambigua, é solo che SLR parser non ha abbastanza memoria per decidere l'azione corretta.

9.8 Viable Prefixes

A **Viable Prefix** é un prefisso di una right-sentential form che non continua oltre l'estremità destra dell'handle più a destra di quella sentential form (forma sentenziosa).

Per definizione é possibile aggiungere simboli terminali alla fine di un viable prefix per ottenere una right-sentential form.

Dico che un item $A \to \beta_1 \otimes \beta_2$ é valido per un viable prefix $\alpha\beta_1$ se ho una derivazione $S' \to^*_{rm} \alpha A\omega \to \alpha\beta_1\beta_2\omega$.

Questo fatto mi dice molto in caso di shift/reduce conflicts infatti:

- se $\beta_1 \neq \varepsilon$ mi suggerisce che non abbiamo ancora shiftato l'handle nello stack quindi faccio uno shift

- se $\beta_1 = \varepsilon$ allora $A \to \beta_1$ é l'handle e devo ridurre

Sfortunatamente non possiamo supporre che tutte i conflitti possono essere risolti se il metodo LR é applicato su una grammatica arbitraria.

9.8.1 Resolve Shift/Reduce Conflicts

$(0)E' \to E$
$(1)E \to E + E$
$(2)E' \to E * E$
$(3)E' \to id$
$\Longrightarrow\ FOLLOW(E) = \{+, *, \$\}$ calcolo C LR(0) su G':
$I_0 = \{E' \to .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}$
$I_1 = \{E' \to E.,\ E \to E. + E,\ E \to E. * E\}$ (E)
$I_2 = \{E \to id.\}$ (id)
$I_3 = \{E \to E + .E,\ E \to .E * E,\ E \to .E + E,\ E \to .id\}$ (da I_1 +)
$I_4 = \{E \to E * .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}$ (da I_1 *)
$I_5 = \{E \to E + E.,\ E \to E. * E,\ E \to E. + E\}$ (da I_3 E)
$I_6 = \{E \to E * E.,\ E \to E. + E,\ E \to E. * E\}$ (da I_4 E)

	id	+	*	$	E
0	s2				1
1		s3	s4	acc	
2		r3	r3	r3	
3	s2				5
4	s2				6
5		s3/r1	s4/r1	r1	
6		s3/r2	s4/r2	r2	

Noto che ci sono dei conflitti, in action [5,+] se voglio favorire l'associativitá a sinistra del + scelgo r1, se voglio far valere la precedenza del * scelgo s4 per la action[5,*].

9.9 More Powerful LR Parsers

Estendo il parser di prima aggiungendo un simbolo di lookahead; ci sono due metodi:

- **canonical-LR** usa grandi set di items LR(1) per il lookahead

- **lookahead-LR o LALR method** basato su set di items LR(0) ed ha pochi stati rispetto ai parsers basati su items LR(1)

9.10 Canonical LR(1) items

Cerco di portarmi dietro piú informazioni, splitto gli stati del SLR. Definisco un **LR(1) item** come un oggetto della forma $[A \to \alpha\beta,\ a]$ con a un terminale o un end-marker $ chiamato **lookahead** dell'item. Un item $[A \to \alpha,\ a]$ viene ridotto solo se il simbolo dopo é una a. Il lookahead set dev'essere un subset dei follow di A.

9.10.1 Closure Function for LR(1) Sets of Items

```
repeat{
    foreach([A -> alpha B beta, a] in I){
        foreach(B -> gamma in P){
            foreach(b in T / b in first(beta alpha))
                I.add[B -> .gamma, b]
        }
    }
} until (no more items are added to I)
return I;
```

9.10.2 Costructing LR(1) Goto Function

```
goto(I,X){
    J = emptySet;
    foreach([A -> alpha X beta, a] in I){
        J.add([A -> alpha X beta, a]);
    }
    return closure(J);
}
```

9.10.3 Items(G')

```
C = closure({[S' -> .S, $]});
repeat{
    foreach(set of items I in C){
        foreach(X in V){
            if(goto(I,X) != empty & not in C){
                C.add(goto(I,X));
            }
        }
    }
} until(no new sets of items are added to C);
```

9.10.4 Esempio

$S' \to S$
$S \to CC$
$C \to cC|d$

Inizio computando la chiusura di $\{[S' \to .S, \$]\}$ che mi deve matchare con $[A \to \alpha B\beta,\ a]$ $(A = S',\ B = S,\ a = \$)$. Devo quindi aggiungere $[B \to \gamma,\ b]$ con $b \in first(\beta\alpha)$. Devo per forza aggiungere $S \to CC$ e dato che $\beta = varepsilon$, b puó essere solo \$. Quindi aggiungo $[S \to .CC,\ \$]$.

Continuo la closure con $[C \to .\gamma,\ b]$ con $b \in first(C\$) = first(C) = \{c,\ d\} \implies$ aggiungo gli items $[C \to .cC,\ c]$, $[C \to .cC,\ d]$, $[C \to .d,\ c]$, $[C \to .d,\ d]$; nessuno di questi puó essere ulteriormente espanso quindi ho finito per questo set.

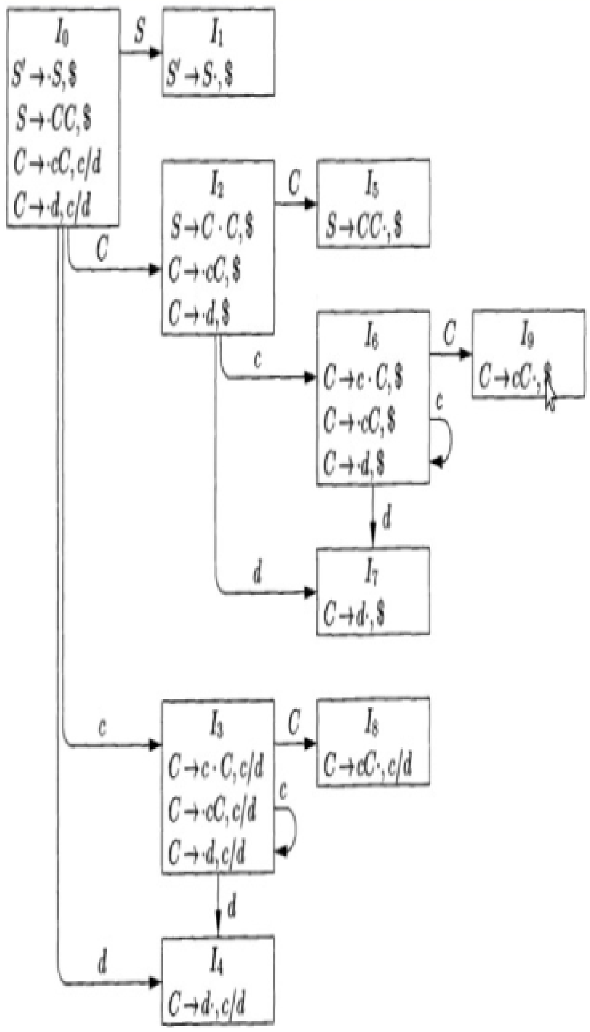
$I_0 = \{[S' \to .S,\ \$],\ [S \to .CC,\ \$],\ [C \to .cC,\ c|d],\ [C \to .d,\ c|d]\}$

Computo $goto(I_0, S)$, devo aggiungere $[S' \to S.,\ \$]$ che non puó esere ampliato.

$I_1 = \{[S' \to S.,\ \$]\}$

Computo $goto(I_0, C)$ aggiungo $[S \to C.C,\ \$]$ ed espando

$I_2 = \{[S \to C.C,\ \$],\ [C \to .cC,\ \$],\ [C \to .d,\ \$]\}$
$I_3 = \{[C \to c.C, c|d],\ [C \to .CC, c|d],\ [C \to .d, c|d]\}$
$I_4 = \{[C \to d.,\ c|d]\}$



9.10.5 Canonical LR(1) Parsing Table

- $C' = \{I_0,\ I_1,\ ...,\ I_n\}$ la collection dei set LR(1)

- lo stato i del parser é costruito da I_i. La parsing action dello stato i é determinata:

a) se $[A \to \alpha a\beta,\ b] \in I_i$ e $goto(I_i,\ a) = I_j \implies ACTION[i,a] = shift\ j,\ a \in T$

b) se $[A \to \alpha.,\ a] \in I_i, A \neq S' \implies action[i,a] = reduce\ A \to \alpha''$

c) se $[S' \to S.,\ \$] \in I_i \implies action[i,\$] = accept$

- Se $goto(I_i, A) = I_j \implies goto(i, A) = j$

- Tutte le entry non definite generano error()

- Lo stato iniziale é costruito a partire da $[S' \to S, \$]$

Se emergono conflitti nella costruzione della parsing table la grammatica **non é LR(1)**.

9.10.6 Esempio

$S' \to S$
$S \to CC$
$C \to cC|d$

	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

9.11 LALR (LookAhead LR) Parsing Table

É spesso usata perché le tabelle LALR sono piú piccole delle canonical LR tables.
$I_3 = \{[C \to c.C,\ c|d],\ [C \to .cC,\ c|d],\ [C \to .d,\ c|d]\}$
$I_4 = \{[C \to .d,\ c|d]\}$
$I_6 = \{[C \to c.C,\ \$],\ [C \to .cC,\ \$],\ [C \to .d,\ \$]\}$
$I_7 = \{[C \to .d,\ \$]\}$

Inoltre so che $goto(I_3, c) = I_3,\ goto(I_3, d) = I_4,\ goto(I_6, c) = I_6,\ goto(I_6, d) = I_7$; in questo caso I_4 e I_7 hannoun core comune $\{C \to .d\}$, I_3 ed I_6 hanno un core $\{C \to c.C,\ C \to .cC,\ C \to .d\}$.

In generale un **core** é un set di item LR(0) / la grammatica puó produrre piú di due set di item con lo stesso core.

Finché il core di goto(I, X) dipende solo dal core di I, le goto di set mergiati possono essere mergiate. Quindi non ho problemi a risolvere la goto di set mergiati. Quindi in pratica se ho una grammatica LR(1) senza conflitti e unisco gli stati con lo stesso core potró avere solo conflitti reduce/reduce ed é improbabile.

9.11.1 LALR Parsing Table

- genero la collection $C = \{I_0, ..., I_n\}$
- unisco gli stati in C con la loro unione generando $C' = \{J_0, ..., J_m\}$ (**LALR(1) items**)
- genero le action come per cLR su C' (**LALR(1) collection**)
- Se $I_n \subset J \implies goto(J, X) = gk,\ k = union(item\ LR(1)$ / hanno lo stesso core di $goto(I_n, X))$

Se emergono conflitti nella costruzione della parsing table la grammatica G **non é LALR(1)**; altrimenti G é LALR(1).

9.11.2 Esempio

$S' \to S$
$S \to CC$
$C \to cC|d$
Ottengo coppie con lo stesso core (I_3 e I_6, I_4 e I_7, I_8 e I_9). Ottengo questa parsing table:

	c	**d**	**$**	**S**	**C**
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Notare che goto(I_{36}, C) originariamente era (in LR(1)) $goto(I_3, C) = I_8$ e I_8 adesso é parte di I_{89}, allora metto $goto(I_{36}, C) = I_{89}$.

Naturalmente arrivavo alla stessa conclusione anche per I_6, parte di I_{36}, con $goto(I_6, C) = I_9$ che adesso appartiene a I_{89}.

Quando parso una stringa i parser LR e LALR fanno le stesse sequenze di riduzioni solo che in LALR hanno nomi diversi.

9.12 Efficient Construction of LALR Parsing Table

Posso evitare di costruire l'intera collection di LR(1) items; posso rappresentare LR(0) e LR(1) items con il loro **kernel** (il body della produzione dopo il punto). Posso costruire i kernel degli item LALR(1) partendo dai kernel degli item LR(0) per propagazione e spontanea generazione dei lookahead.

Chapter 10

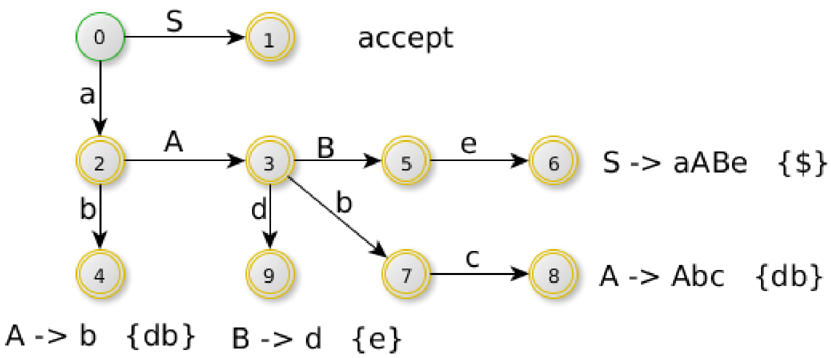
Bottom Up (Farina)

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

10.0.1 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):

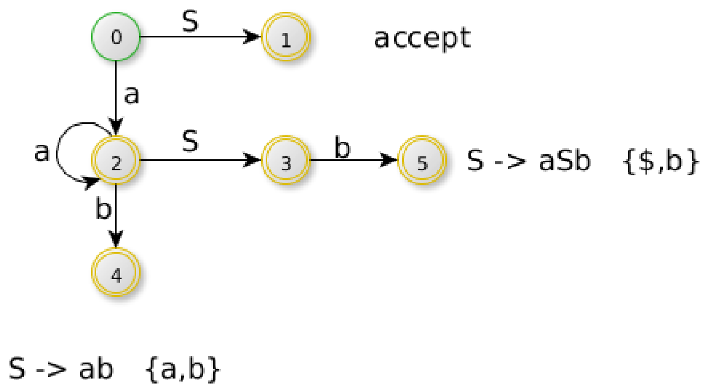
0	$abbcde\$$
$0 \to 2$ consumando 'a '	$a\|\|bbcde\$$
$2 \to 4$ consumando 'b '	$ab\|\|bcde\$$
4 riduco $A \to b$	$aA\|\|bcde$

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 $\cup$ \$	$V \setminus T$
stati	shift-k: leggi un simbolo di input e vai allo stato reduce $A \to b$	goto-k: descrive le funzioni di transizione identificate dai non terminali quando consumi roba

10.1 Algoritmo di shift/reduce

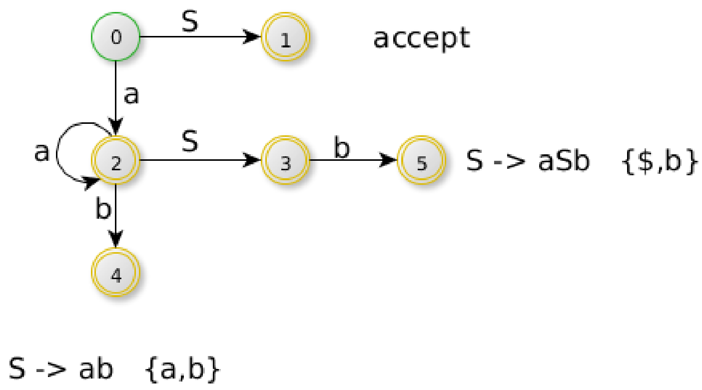
(comune a SLR(1), LR(1), LALR(1))

input w, tabella di parsing bottom-up di tipo $\diamond$, con $\diamond$ scelto fra $\{SLR(1), LALR(1), LR(1)\}$ G.

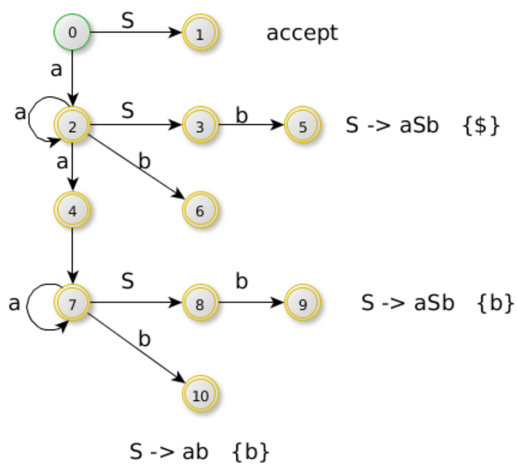
output derivazione rightmost di w se $w \in L(G)$, altrimenti error()

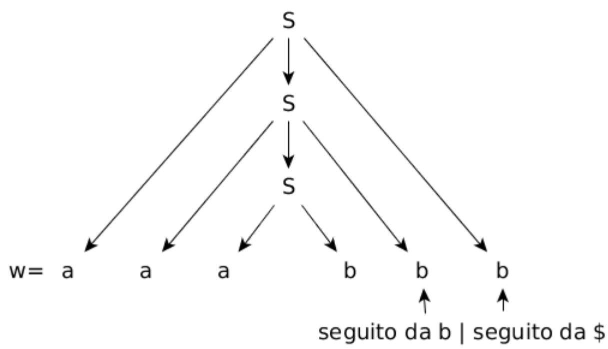
```
stack.push(s_0);
buffer = w$;
while(true){
    let s = stack.top();
    if(M[s,b] == shift-k){
        stack.push(b);
        stack.push(k);
        let b = buffer.readNext();
    } else if(M[s,b] == "reduce A -> beta"){
        stack.pop() 2|beta| simboli;
        let j tale che M[m, A] = gj;
        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 ramo 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 $S' \to S \implies G' = (V \cup \{S'\}, T, S', P \cup \{S' \to S\})$

10.1.1 Closure

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: $S \to a.Sb$

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

Quando computo uno stato ho i **kernel items** che poi vengono espansi con la **closure**. Notare che gli ultimi due items sono gli stessi degli ultimi due del nodo precedente. Quella é la chiusura, mentre i primi due sono i generatori dello stato (kernel dello stato o **kernel items**).

Gli stati che terminali dell'automa (le foglie) sono del tipo $S \to ab.$, quindi hanno incontrato tutto i simboli e possono essere ridotti (**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.

10.2 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 $\wedge ! accepting$

Chapter 11

Costruzione di un automa caratteristico LR(0) o LR(1)

(data un'appropriata istanziazione di P_0 (stato iniziale) e closure function)

Prima di tutto facciamo la costruzione dell'automa caratteristico, usando l'algoritmo sotto riportato.

Istanziando P_0, closure otteniamo:

- automi caratteristici LR(0)-automi $\leftarrow$ parsing SLR(1)

- automi caratteristici LR(1)-automi $\leftarrow$ parsing LR(1)

Lookahead function $LA : (FxP) \rightarrow p(T \cup \{\$\})$ con F = insieme degli stati finali dell'automa caratteristico considerato. Si considerano stati finali gli stati che contengono almeno un reducing item.

L'automa caratteristico sará necessario per la creazione della tabella di parsing.

```
Inizializzare la collezione Q di stati come {P_0};
flag P_0 come non-marcato;

while(ho uno stato non marcato P in Q){
   marca P;
   foreach(A -> a.Ybeta in P){
      tmp.add( A->aY.beta );
   }
   if(tmp == kernel(R)){ //per qualche R in Q
      tau(P, Y) = R; //goto function
   } else {
      newState = closure(tmp);
      tau(P, Y) = newState;
      Q.add(newState); //come non-marcato
   }
}
```

11.0.1 Esempi di closure, grammatica SLR

se ho la grammatica:
$S' \rightarrow S$
$S \rightarrow aSb|ab$
$closure_0(\{S' \rightarrow .S\})$ é composta da:
$S \rightarrow .aSb$
$S \rightarrow .ab$

Se invece ho una grammatica:
$E' \rightarrow E$
$E \rightarrow E + T|T$
$T \rightarrow T * F|F$
$F \rightarrow (E)|id$

Allora $closure_0(\{E' \to .E\})$ diventa
$E \to .E + T$ quelle con il punto prima della E le ho giá aggiunte, non faccio nulla
$E \to .T$ ora aggiungo quelle con il punto prima della T
$T \to .T * F$ quelle con il punto prima della T le ho giá aggiunte, non faccio nulla
$T \to .F$ ora aggiungo quelle con il punto prima della F
$F \to .(E)$ terminale dopo il punto, non devo aggiungere nulla
$F \to .id$ terminale dopo il punto, non devo aggiungere nulla
Si devono fare in ordine per non dimenticarsene in giro.

11.1 Algoritmo di $closure_0(P)$

```
tag ogni item in P come non-marcato

while(ho ancora un item I non marcato in P){
   marca I;
   if(I ha la forma A -> alpha .B beta){
      foreach ((B -> Y) in P){
         if(B -> Y not in P){
            add(B -> .Ya);
            segna P come non-marcato;
         }
      }
   }
}
return P;
```

11.2 LR(0) automaton

Si ricava utilizzando, nell'algoritmo di costruzione dell'automa caratteristico:

- $P_0 = closure_0(\{S' \to .S\})$ per la grammatica G arricchita con $S' \to S$

- $closure_0$ per closure

11.2.1 Esempio

$S' \to S$
$S \to aSb|ab$

$P_0 =$	$S' \to .S$	
	$S \to .aSb$	
	$S \to .ab$	
$P_1 =$	$S' \to S.$	ci si arriva con una S-transizione (ACCEPT)
$P_2 =$	$S \to a.Sb$	ci si arriva con una a-transizione
	$S \to a.b$	
	$S \to .aSb$	
	$S \to .ab$	
	P_0 e P_1 sono finiti, guardo P_2	
$P_3 =$	$S \to aS.b$	ci si arriva con una S-transizione da P_2
$P_4 =$	$S \to ab.$	ci si arriva con una b-transizione da P_2
$P_5 =$	$S \to .aSb$	ci si arriva con una a-transizione da P_2 (stesse righe di P_2)
	$S \to .ab$	la freccia quindi va da P_2 a P_2, perché $P_5 \subset P_2$
	ora guardo P_3	
$P_6 =$	$S \to aSb.$	ci si arriva con una b-transizione da P_3

11.3 Creazione della tabella di parsing

τ	funzione di transizione (GOTO) dell'automa caratteristico considerato.
LA	lookahead function considerata.

La tabella di parsing bottom-up per la coppia prescelta di automa e lookahead function é una matrice $Q \otimes (V \cup \{\$\})$ dove Q é l'insieme degli stati dell'automa prescelto.

$\forall$ entry (P,Y):	inserire **shift**	se $Y \in T \wedge \tau(P,Y) = R$
	inserire **reduce** $A \to \beta$	se $A \to \beta. \in P \wedge Y \in LA(P, A \to \beta)$
	inserire **accept()**	se $Y = \$ \wedge S' \to S. \in P$
	inserire **error()**	se $Y \in T \cup \{\$\} \wedge$ non é ancora stato inserito nulla
	applicando i passi precedenti	
	inserire **goto R**	se $Y \in V \backslash T \wedge \tau(P,Y) = R$

11.3.1 Esercizio

$SLR(1)$
$LR(0)$ automa
$LA(P, A \to \beta) = follow(A)\ \forall\ P \in Q$ (automa)

[numero nodo]	[parte che	puó avere	conflitti]	[goto part]
	a	**b**	**$**	**S**
0	S2			g1
1			ACCEPT	
2	S2	S4		g3
3		s5		
4		$R : S \to ab$	$R : S \to ab$	
5		$R : S \to aSb$	$R : S \to aSb$	

S = shift R = reduce g = goto

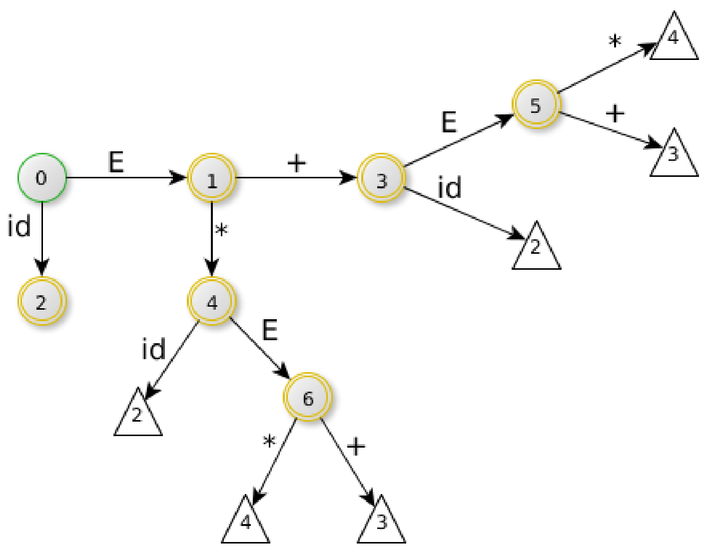
Se ho la parola $w = aabb\$$ faccio:

0	$aabb\$$	sono in 0 e vedo a. Faccio [0, a] nella tabella
0a2	$abb\$$	sono in 2 e vedo a. Faccio [2, a] nella tabella
0a2a2	$bb\$$	sono in 2 e vedo b. Faccio [2, b] nella tabella
aa2a2b4	$b\$$	sono in 4 e vedo b. Faccio il reduce (cella 4b) visto che crea 2S, e [2, S] = g3, aggiungo 3
0a2S3	$b\$$	sono in 3 e vedo b. Faccio [3, b] nella tabella
0a2S3b5	$	sono in 5 e vedo $. Faccio il reduce (cella 5$ visto che crea 0S, e [0, S] = g1, aggiungo 1
0S1	$	ok perché [S, 1] é accept

11.3.2 Esempio Conflitti Shift/Reduce

(Esempio con una grammatica ambigua)

$E \to E + E | E * E | id$
automa LR(0)



//Se va a triangoli, non parte dal kernel, altrimenti si(?)

$$0 = \begin{array}{l} E' \to .E \\ E \to .E + E \\ E \to .E * E \\ E \to .id \end{array}$$

$$1 = \begin{array}{l} E' \to E. \\ E \to E. + E \\ E \to E. * E \end{array}$$

$$2 = \begin{array}{l} E \to id. \end{array}$$

$$3 = \begin{array}{l} E \to E + .E \\ E \to .E * E \\ E \to .id \end{array}$$

$$4 = \begin{array}{l} E \to E * .E \\ E \to .E + E \\ E \to .E * E \\ E \to .id \end{array}$$

$$5 = \begin{array}{l} E \to E + E. \\ E \to E. + E \\ E \to E. * E \end{array}$$

$$6 = \begin{array}{l} E \to E * E. \\ E \to E. + E \\ E \to E. * E \end{array}$$

	id	+	*	**$**
0	S2			
1		S3	S4	ACCEPT
2		$R:\ E \to id$	$R:\ E \to id$	$R:\ E \to id$
3	S2			
4	S2			
5		$\boldsymbol{S3,\ R:\ E \to E + E}$	$\boldsymbol{S4,\ R:\ E \to E + E}$	$R:\ E \to E + E$
6		$\boldsymbol{S3,\ R:\ E \to E * E}$	$\boldsymbol{S4,\ R:\ E \to E * E}$	$R:\ E \to E * E$

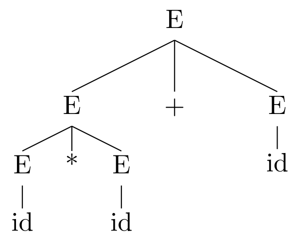
Visto che ci sono **conflitti** nella tabella, questa grammatica **non é SLR**.

In questo caso i conflitti sono di tipo Shift/Reduce. Ma possono essere anche di tipo Reduce/Reduce.

Tra le produzioni nelle celle colorate, quali sono quelle da tenere per avere una grammatica che associa a sinistra?

Nella cella [5, +] devo tenere R “$E \to E + E$ ”
Nella cella [5, *] devo tenereS4
Nella cella [6, +] devo tenere R “$E \to E * E$ ”
Nella cella [6, *] devo tenere R “$E \to E * E$ ”

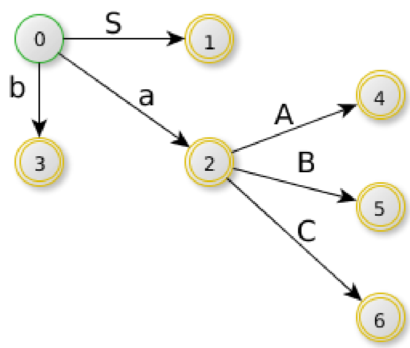
$W = id * id + id\$$
0
$0id2$
$0E1$
$0E1 * 4id2$
$0E1 * 4E6$
$0E1$
$0E1 + 3id2$
$0E1 + 3E5$
$0E1$
ok



11.3.3 Esempio Conflitti Reduce/Reduce

$S \rightarrow aAd|bBd|aBe|bAe$
$A \rightarrow c$
$B \rightarrow c$

Questa grammatica produce 4 stringhe, ogniuna in un modo, e quindi ovviamente non é ambigua.



$$0 = \begin{array}{l} S' \rightarrow .S \\ S \rightarrow .aAd \\ S \rightarrow .bBd \\ S \rightarrow .aBe \\ S \rightarrow .bAe \end{array}$$

$$1 = \quad S' \rightarrow S.$$

$$2 = \begin{array}{l} S \rightarrow a.Ad \\ S \rightarrow a.Be \\ A \rightarrow .c \\ B \rightarrow .c \end{array}$$

$$3 = \begin{array}{l} S \rightarrow b.Bd \\ S \rightarrow b.Ae \\ B \rightarrow .c \\ A \rightarrow .c \end{array}$$

$$4 = \quad S \rightarrow aA.d$$

$$5 = \quad S \rightarrow aB.e$$

$$6 = \begin{array}{l} A \rightarrow c. \\ B \rightarrow c. \end{array}$$

	a	b	d	e	\$
6			$R: \ A \rightarrow c$ $R: \ B \rightarrow c$	$R: \ A \rightarrow c$ $R: \ B \rightarrow c$	

Anche se non é una grammatica ambigua ci sono multiple entries. Questo dipende dal modo in cui scegliamo i follow. Nel parsing canonico infatti non si usano gli item LR(0) ma gli item LR(1) perché ci portiamo dietro informazione direttamente dagli item. Questo riduce/rimuove i problemi di questo tipo.

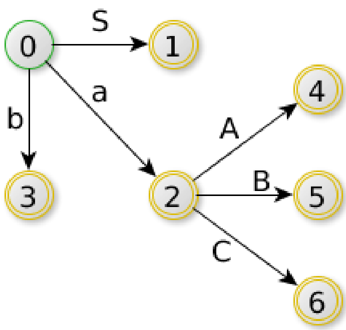
Un item LR(1) canonico é del tipo $A \rightarrow \alpha.\beta, \delta$ con $\delta \subset T \cup \{\$\}$.

11.4 LR(1)

Negli automi caratteristici LR(1) si considera come stato iniziale lo stato che si ottiene facendo: $closure_1(\{[S' \rightarrow .S, \{\$\}]\})$

11.4.1 Esempio grammatica LR(1)

$S \rightarrow aAd|aBe|bBd|bAe$
$A \rightarrow c$
$B \rightarrow c$



$$0 = \begin{array}{l} S' \rightarrow .S,\ \{\$\} \\ S \rightarrow .aAd,\ \{\$\} \\ S \rightarrow .aBe,\ \{\$\} \\ S \rightarrow .bBd,\ \{\$\} \\ S \rightarrow .bAe,\ \{\$\} \end{array}$$

$$1 = \quad S' \rightarrow S.,\ \{\$\}$$

$$2 = \begin{array}{l} S \rightarrow a.Ad,\ \{\$\} \\ S \rightarrow a.Be,\ \{\$\} \\ A \rightarrow .c,\ \{d\} \\ B \rightarrow .c,\ \{e\} \end{array}$$

$$3 = \begin{array}{l} S \rightarrow b.Bd,\ \{?\} \\ S \rightarrow b.Ae,\ \{?\} \\ A \rightarrow .c,\ \{?\} \\ B \rightarrow .c,\ \{?\} \end{array}$$

$$4 = \quad S \rightarrow aA.d,\ \{?\}$$

$$5 = \quad S \rightarrow aB.e,\ \{?\}$$

$$6 = \begin{array}{l} A \rightarrow c.,\ \{d\} \\ B \rightarrow c.,\ \{e\} \end{array}$$

11.5 Algoritmo $Closure_1(P)$

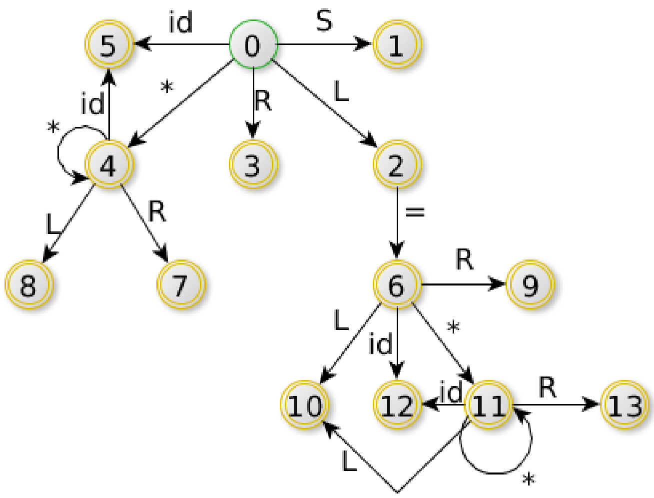
```
every item in P is unmarked
while(exists an item I in P unmarked){
  mark I;
  if([ A -> alpha. B beta, delta ] in I){
    delta1 = Union(d in delta) first(beta d)
    foreach([ B -> Y ] in P){
      if([ B -> .Y ] not in proj(P)){ //projection is the first component of an item
          yet collected
        P.add([ B -> .Y, delta1] as unmarked)
      } else {
```

```
            if([ B -> .Y, gamma ] in P && delta1 not subset gamma ){
              update [ B -> .Y, gamma ] in [Bb -> .Y, gamma U delta1] as unmarked;
            }
          }
        }
      }
    }
    return P;
```

11.6 Esempio grammatica LALR

$S \rightarrow L = R|R$
$L \rightarrow *R|id$
$R \rightarrow L$

Computare la $closure_1(\{[S' \rightarrow S,\ \{\$\}]\})$ (nodo 0 e disegnare l'automa LR(1). Capire se é SLR(1)).



$$0 = \begin{array}{l} S' \rightarrow .S,\ \{\$\} \\ S \rightarrow .L = R,\ \{\$\} \\ S \rightarrow .R,\ \{\$\} \\ L \rightarrow . * R,\ \{\$\} \\ L \rightarrow .id,\ \{\$\} \\ R \rightarrow .L,\ \{\$\} \end{array}$$

$$1 = \quad S' \rightarrow S.,\ \{\$\}$$

$$2 = \begin{array}{l} S \rightarrow L. = R,\ \{\$\} \\ R \rightarrow L.,\ \{\$\} \end{array}$$

$$3 = \quad S \rightarrow R.,\ \{\$\}$$

$$4 = \begin{array}{l} L \rightarrow *.R,\ \{=, \$\} \\ R \rightarrow .L,\ \{=, \$\} \\ L \rightarrow .id,\ \{=, \$\} \\ L \rightarrow . * R,\ \{=, \$\} \end{array}$$

$$5 = \quad L \rightarrow id.,\ \{=, \$\}$$

$$6 = \begin{array}{l} S \rightarrow L = .R,\ \{\$\} \\ R \rightarrow .L,\ \{\$\} \\ L \rightarrow . * R,\ \{\$\} \\ L \rightarrow .id,\ \{\$\} \end{array}$$

$$7 = \quad L \rightarrow *R.,\ \{=, \$\}$$

$8 = \quad R \rightarrow L., \ \{=, \$\}$

$9 = \quad S \rightarrow L = R., \ \{\$\}$

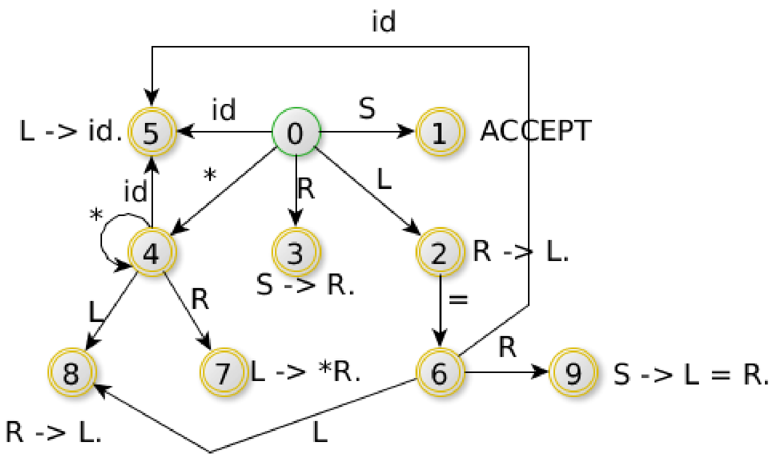
$10 = \quad R \rightarrow L., \ \{\$\}$

$$11 = \begin{array}{l} L \rightarrow *.R, \ \{\$\} \\ R \rightarrow .L, \ \{\$\} \\ L \rightarrow .*R, \ \{\$\} \\ L \rightarrow .id, \ \{\$\} \end{array}$$

$12 = \quad L \rightarrow id., \ \{\$\}$

$13 = \quad L \rightarrow *R., \ \{\$\}$

Notare che i nodi 8 e 10 sembrano uguali ma non lo sono perché hanno lookahead diversi; anche nel gravo saranno nodi distinti mentre se avessimo fatto un automa LR(0) sarebbero stati uniti. Lo stesso automa LR(0):



Quest'ultimo automa ha almeno uno shift/reduce conflict mentre l'automa LR(1) non ne avrá. Guardiamo se é SLR(1)

	id	$*$	$*$	$*$	**S**	**L**	**R**
0	S5	S4			1	2	3
1				ACC			
2			S6	r: $R \rightarrow L$			
3				r: $S \rightarrow R$			
4	S5	S4				8	7
5			r: $L \rightarrow id$	r: $L \rightarrow id$			
6	S12	S11				10	9
7			r: $L \rightarrow *R$	r: $L \rightarrow *R$			
8			r: $R \rightarrow L$	r: $R \rightarrow L$			
9				r: $S \rightarrow L = R$			
10				r: $R \rightarrow L$			
11	S12	S11				10	13
12				r: $L \rightarrow id$			
13				r: $L \rightarrow *R$			

visto che non ci sono conglitti questa grammatica é LR(1). Gli stati 4 e 11 hanno la stessa proiezione; possiamo quindi fare: $P_1 \cup P_2 \quad A \rightarrow \alpha.\beta, \ \delta_1 \cup \delta_2$.

Lo posso fare per 4 e 11, 5 e 12, 8 e 10, 7 e 13

	id	*	*	*	**S**	**L**	**R**
411	S512	S411			810	2	713
512			r: $L \to id$	r: $L \to id$			
713			r: $L \to *R$	r: $L \to *R$			
810			r: $r \to L$	r: $R \to L$			

Dovró anche aggiornare i vari shift e goto delle altre righe per farli andare ai nodi nuovi. S5 ed S12 diventeranno S512.

La grammatica cos'ıgenerata é LALR perché nella tabella non ci sono conflitti.

11.6.1 Ricorda

$SLR(1) \subset LALR(1) \subset LR(1)$

Grammatica SLR(1)

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

Grammatica LR(1)

$S \to aAd|aBe|bBd|bAe$
$A \to c$
$B \to c$

Grammatica LALR

$S \to L = R|R$
$L \to *R|id$
$R \to L$

11.7 Algoritmo LALR(1) (inefficiente)

1. Costruzione di A_1 (LR(1) automaton)

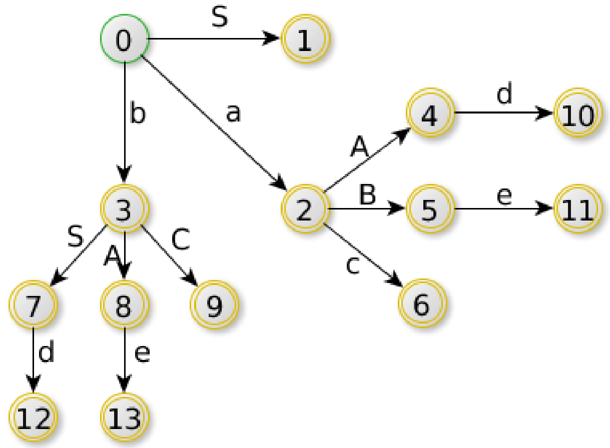
2. Creo LR(1) Merged Automa A_2 (mergio gli stati che hanno la stessa proiezione). Se lo stato $P \in A_2$ e" il merging di $Q_1, ..., Q_k$ di A_1 e Q_1 ha una Y-transizione a Q'_1 allora P ha una Y-transizione allo stato P' / $proj(P') = proj(Q')$

3. $\forall$ P finale, $\forall$ $[A \to \beta.,\ \delta_i] \in P$ $LA(P, A \to \beta) = \cup_{\delta_i}[A \to \beta.,\ \delta_i] \in P$

11.7.1 Esempio (Vedi esempio conflitti reduce/reduce)

$S \to aAd|aBe|bBd|bAe$
$A \to c$
$B \to c$

Faccio LR(1) automa



$$0 = \begin{array}{l} S' \to .S,\ \{\$\} \\ S \to .aAd,\ \{\$\} \\ S \to .bBd,\ \{\$\} \\ S \to .aBe,\ \{\$\} \\ S \to .bAe,\ \{\$\} \end{array}$$

$$1 = \quad S' \to S.,\ \{\$\}$$

$$2 = \begin{array}{l} S \to a.Ad,\ \{\$\} \\ S \to a.Be,\ \{\$\} \\ A \to .c,\ \{d\} \\ B \to .c,\ \{e\} \end{array}$$

$$3 = \begin{array}{l} S \to b.Bd,\ \{\$\} \\ S \to b.Ae,\ \{\$\} \\ B \to .c,\ \{d\} \\ A \to .c,\ \{e\} \end{array}$$

$$4 = \quad S \to aA.d,\ \{\$\}$$

$$5 = \quad S \to aB.e,\ \{\$\}$$

$$6 = \begin{array}{l} A \to c.,\ \{d\} \\ B \to c.,\ \{e\} \end{array}$$

$$7 = \quad S \to bB.d,\ \{\$\}$$

$$8 = \quad S \to bA.e,\ \{\$\}$$

$$9 = \begin{array}{l} B \to c.,\ \{d\} \\ A \to c.,\ \{e\} \end{array}$$

$$10 = \quad S \to aAd.,\ \{\$\}$$

$$11 = \quad S \to aBe.,\ \{\$\}$$

$$12 = \quad S \to bBd.,\ \{\$\}$$

$$13 = \quad S \to bAe.,\ \{\$\}$$

Gli stati 6 e 9 hanno la stessa proiezione quindi li mergio in

$$14 = \begin{array}{l} A \to c.,\ \{d\} \\ B \to c.,\ \{e\} \\ A \to c.,\ \{e\} \\ B \to c.,\ \{d\} \end{array}$$

Sostituisco 14 a 6 e 9, faccio una transizione da 3 a 14

	a	**b**	**c**	**d**	**e**	**$**	**S**	**A**	**B**
0	S2	S3					g1		
1						ACC			
2			S14					g4	g5
3			S14					g8	g7
4				S10					
5					S11				
7				S12					
8					S13				
10						$S \to aAd$			
11						$S \to aBe$			
12						$S \to bBd$			
13						$S \to bAe$			
14					$A \to c$ $B \to c$	$B \to c$ $A \to c$			

In 14 ho ancora dei conflitti reduce/reduce quindi la grammatica non é LALR.

11.7.2 Esempio

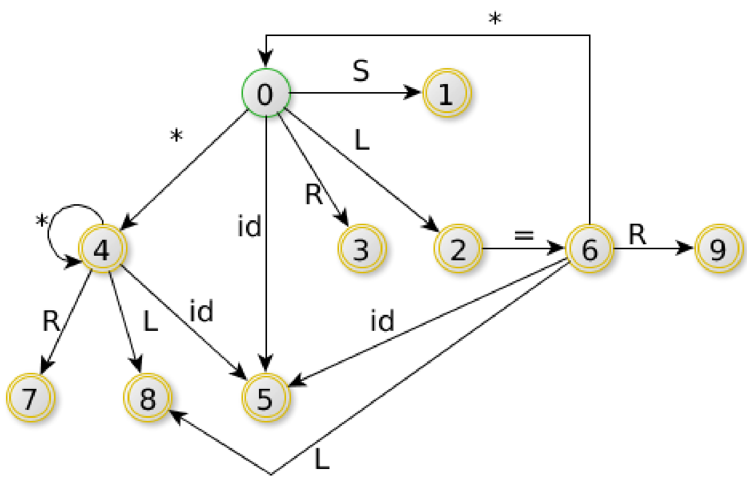
$S \to Ma|bMc|dc|bda$
$M \to d$

Non dovrebbe essere LALR perché avrei $M \to d$ con lookahead set $\{a\}$ $e\{d\}$...

11.8 Algoritmo LALR (efficiente)

11.8.1 Esempio

$S \to L = R|R$
$L \to *R|id$
$R \to L$



$$0 = \begin{array}{l} S' \to .S, \{x_0\} \\ S \to .L, \{x_0\} \\ S \to .R, \{x_0\} \\ L \to . * R, \{x_0\} \\ L \to .id, \{x_0\} \\ R \to .L, \{x_0\} \end{array}$$

$$1 = \quad S' \to S., \{x_1\}$$

$$2 = \begin{array}{l} S \to L. = R, \{x_2\} \\ R \to L., \{x_3\} \end{array}$$

$$3 = \quad S \to R., \{x_4\}$$

$$4 = \quad L \to *.R, \{x_5\}$$

$$5 = \quad L \to id., \{x_6\}$$

$$6 = \begin{array}{l} S \to L = .R, \{x_7\} \\ R \to .L, \{x_7\} \\ L \to .*R, \{x_7\} \\ L \to .id, \{x_7\} \end{array}$$

$$7 = \quad L \to *R., \{x_8\}$$

$$8 = \quad R \to L., \{x_9\}$$

$$9 = \quad S \to L = R., \{x_10\}$$

$$\begin{array}{l} x_0 = \{\$\} \\ x_1 = \{x_0\} \\ x_2 = \{x_0\} \\ x_3 = \{x_0\} \\ x_4 = \{x_0\} \\ x_5 = \{\{=, x_0\} \cup \{x_5\} \cup \{x_7\}\} \\ x_6 = \{x_5\} \\ x_7 = \{x_2\} \\ x_8 = \{x_5\} \\ x_9 = \{\{x_5\} \cup \{x_7\}\} \\ x_10 = \{x_7\} \end{array}$$

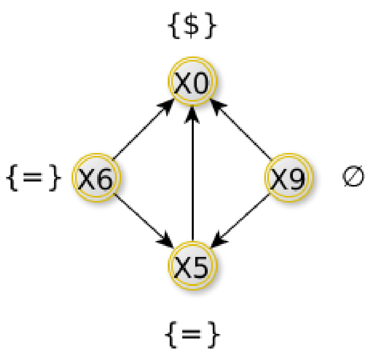
Ora devo risolvere il sistema

	class
$x_0 = \{\$\}$	x_0
x_1	x_0
x_2	x_0
x_3	x_0
x_4	x_0
$x_5 = \{=, x_0, x_5, x_7\}$	x_5
$x_6 = \{=, x_0, x_5, x_7\}$	x_6
x_7	x_0
x_8	x_5
$x_9 = \{x_5, x_7\}$	x_9
x_10	x_10

Mi rimangono quindi solo 4 variabili x_0, x_5, x_6, x_9

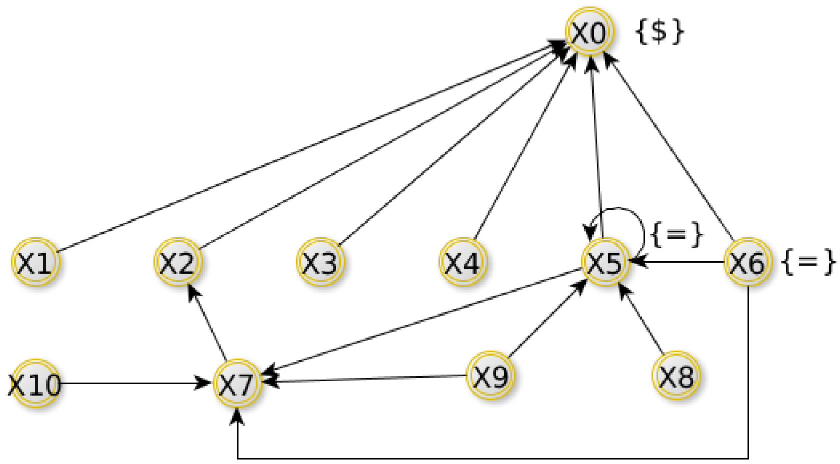
$$\begin{array}{ll} x_0 = \{\$\} & \\ x_5 = \{=, x_0, x_5, x_7\$\} & = \{=, x_0\} \\ x_6 = \{=, x_0, x_5, x_7\$\} & = \{=, x_0, x_5\} \\ x_9 = \{x_5, x_7\$\} & = \{x_5, x_0\} \end{array}$$

Creo un grafo mettendo sui nodi le variabili, con solamente i terminali di quella variablie (non quelli presi dalle altre. Solo quelli che non sono x_k insomma).



Ottengo: $x_0 = \{\$\}$
$x_5 = \{=, \$\}$
$x_6 = \{=, \$\}$
$x_9 = \{=, \$\}$

Questi sono il lookahead set che stavo cercando.



11.9 Costruzione automa simbolico

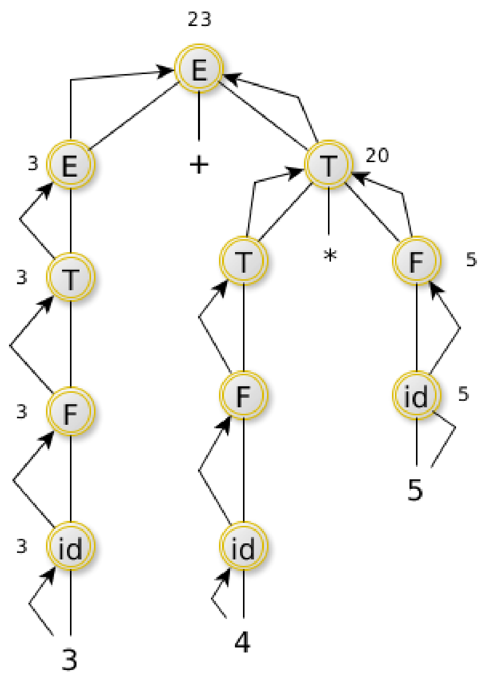
```
x_0 = new Var();
Vars = {x_0};
P_0 = closure1({[S' -> .S, {x_0}]});
States = {P_0};
P_0 unmarked;
while(exists unmarked state P in States){
   mark P;
   foreach(Y in V){
      tmp = emptySet;
      foreach([A -> alpha . Y beta, delta] in P){
         tmp.Add(([A -> alpha . Y beta, delta]);
      }
      if(tmp is not empty){
         if(lo stato target non e'' ancora stato collezionato){
            States.Add(versione simbolica del target);
            Queue.Add(equazione per kernel item in tmp);
         } else {
            raffinare le equazioni delle variabili associate ai kernel item del target;
         }
      }
   }
}
```

11.10 Grammatiche Attribuite

SDD	Syntax Directed Definition
SDT	Syntax Directed Translation

Una grammatica attribuita é una grammatica a cui sono aggiunti attributi e regole.

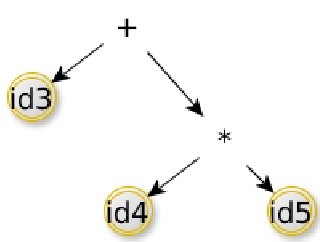
$E \to E + T|T$
$T \to T * F|F$
$F \to (E)|id$
$w = 3 + 4 * 5$



Il valore di propaga dal basso verso l'alto nell'albero e computate le operazioni man mano che si presentano.

$E_1 \to E_2 + T$	$\{E_1.val = E_2.val + T.val\}$	
$E \to T$	$\{E.val = T.val\}$	
$T_1 \to T_2 * F$	$\{T_1.val = T_2.val * F.val\}$	$\leftarrow$ Azione semantica
$T \to F$	$\{T.val = F.val\}$	
$F \to E$	$\{F.valE = E.val\}$	
$F \to id$	$\{F.val = lexval(id)\}$	

L'abstract syntax tree di questa grammatica é:

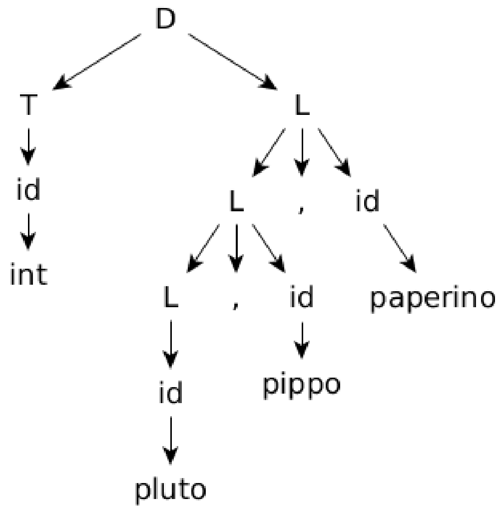


11.11 Attributi Sintetizzati ed Ereditati

$A \to \alpha$ A.a definito come una funzione degli attributi dei terminali e non terminali in α. Gli attributi dei terminali sono informazioni ottenute durante l'analisi lessicale.

11.11.1 Esempio

$D \rightarrow TL$ $\{L.i = T.t\}$
$T \rightarrow int$ $\{T.t = integer\}$
$T \rightarrow float$ $\{T.t = float\}$
$L_1 \rightarrow L_2,\ id$ $\{L_2.i = L_1.i; addType(lexval(id), L_1.i)\}$
$L \rightarrow id$ $\{addType(lexval(id), L_i)\}$
w = int pluto, pippo, paperino



Facendo bottom-up la prima cosa che faremo sará $T \rightarrow int$. A questo punto T conosce “int” e lo dice alla L piú in alto (quella subito sotto a D). Quella L lo dice alla L sotto di lui, e questo succede anche per il livello dopo. Adesso L va ad “id” che diventa “pluto” che L viene a conoscere (poiché viene passato in alto). Visto che alcuni attributi dipendono dai padri e dagli attributi dei fratelli, questi sono attributi ereditati.

Gli attributi ereditati sono funzione degli attributi di siblings e del padre.

11.11.2 Esempio

$S \rightarrow N$
$N \rightarrow o\ Digist$
$N \rightarrow Digits$
$Digits \rightarrow Digits\ d$
$Digits \rightarrow d$

Questo albero é simile a quello di prima: cé solo “o” a sinistra e potenzialmente infinite “Digits” a destra, una sotto l’altra. Come fare per essere in grado di distinguere se cé la “o” oppure no?

Posso ribaltare l’albero, riscrivendo la grammatica come:

$S \rightarrow Digits$ $\{Digits.qualcosa\}$
$Digits_1 \rightarrow Digits_2\ d$ $\{Digits_1.val = Digits_2.val * Digits_2.base +' d'\}$
$Digits \rightarrow d$ $\{Digits.base = 8;\ Digits.val =' d'\}$
$Digits \rightarrow od$ $\{Digits.base = 10;\ Digits.val =' d'\}$