Linguaggi formali e compilatori

Marini Mattia

1^o semestre 3^o anno

Indice

1 Grammatiche libere **3**
1.1 Proprietà grammatiche libere . . . 3
1.2 Chomsky normal form . . . 4
1.2.1 Riduzione in forma di chomsky . . . 4
1.3 Pumping lemma . . . 5
1.3.1 Dimostrazione . . . 5

2 Top down parsing **6**
2.0.1 Esempio 1 first follow . . . 7
2.0.2 Esempio 2 first follow . . . 8
2.1 Costruzione tabella di parsing . . . 8
2.2 Parsing . . . 9
2.3 Left recursion . . . 10
2.3.1 Grafo dei riferimenti . . . 10
2.3.2 Eliminazione immediate left recursion . . . 11
2.3.3 Eliminazione left recursion . . . 12
2.3.4 Esempio 1 . . . 14
2.3.5 Esempio 2 . . . 14
2.4 Fattorizzazione sinistra . . . 15
2.5 Conclusioni . . . 15

3 Linguaggi regolari **16**
3.1 Automa non deteministico a stati finiti . . . 16
3.2 Construzione di Thompson . . . 17
3.3 Simulazione . . . 18
3.4 Automa deterministico a stati finiti . . . 18
3.4.1 Completamento funzione DFA . . . 19
3.4.2 Conversione da NFA a DFA . . . 19
3.4.3 Minimizzazione DFA . . . 20
3.4.4 Numero stati di un DFA . . . 21
3.5 Pumping lemma per linguaggi regolari . . . 21

4 Bottom up parsing **22**
4.1 LR(0) items . . . 23
4.1.1 Chiusura di un LR(0) item . . . 24

4.2 Costruzione automa LR(0) . . . 24
4.3 Costruzione tabella di parsing . . . 26
4.4 Algoritmo di parsing . . . 26

5 Competenze richieste **27**
5.1 Linguaggi liberi . . . 27
5.1.1 Preliminari e lemmas . . . 27
5.1.2 Pumping Lemma . . . 27
5.1.3 Altro . . . 27
5.2 Top down parsing . . . 28
5.3 Linguaggi regolari . . . 28
5.4 Bottom up parsing . . . 28
5.5 Syntax Directed Definitions . . . 29
5.6 Generazione codice intermedio . . . 29

Definizioni

1 Forma normale di chomsky . . . 4
2 First(α) . . . 6
3 Follow(A) . . . 7
4 Left recursive grammar . . . 10
5 Immediately left recursive grammar . . . 10
6 Grafo dei riferimenti . . . 11
7 Fattorizzazione a sinistra . . . 15
8 Linguaggio regolare . . . 16
9 Espressione regolare . . . 16
10 Automa non deteministico a stati finiti . . . 17
11 ϵ - chiusura . . . 18
12 Automa deterministico a stati finiti . . . 19
13 Equivalenza di due stati . . . 20
14 LR(0) item . . . 24

Teoremi e Assiomi

1 Chiusura linguaggi liberi rispetto a unione . . . 3
2 Chiusura linguaggi liberi rispetto a concatenazione . . . 3
3 NON chiusura dei linguaggi liberi rispetto all'intersezione . . . 3
4 Pumping lemma . . . 5
5 Left recursion e LL(1) parsing . . . 10
6 Eliminazione ricorsione sinistra immediata . . . 12
7 Condizioni LL(1) . . . 15
8 Numero stati di un DFA . . . 21
9 Pumping lemma per linguaggi regolari . . . 22

1 Grammatiche libere

1.1 Proprietà grammatiche libere

Teorema 1: *Chiusura linguaggi liberi rispetto a unione*

I linguaggi liberi sono chiusi relativamente all'unione. In altre parole dato $\mathcal{L}_1$ e $\mathcal{L}_2$, allora

$$\mathcal{L}_1 \cup \mathcal{L}_2$$

è libero

Per dimostrare che ciò è vero, basta creare la seguente grammatica

$$\mathcal{G} = (V_1' \cup V_2' \cup \{S\}, T_1 \cup T_2, S, \mathcal{P}_1' \cup \mathcal{P}_2' \cup \{S \to S_1' | S_2'\})$$

Teorema 2: *Chiusura linguaggi liberi rispetto a concatenazione*

I linguaggi liberi sono chiusi relativamente alla concatenazione. In altre parole dato $\mathcal{L}_1$ e $\mathcal{L}_2$, allora

$$\mathcal{L}_1\mathcal{L}_2$$

è libero

Per dimostrare che ciò è vero, basta creare la seguente grammatica

$$\mathcal{G} = (V_1' \cup V_2' \cup \{S\}, T_1 \cup T_2, S, \mathcal{P}_1' \cup \mathcal{P}_2' \cup \{S \to S_1' S_2'\})$$

In ambi i casi è importante che non ci sia sovrapposizione dei non terminali. Per evitare ciò basta usare un refresh dei simboli non terminali

Teorema 3: *NON chiusura dei linguaggi liberi rispetto all'intersezione*

I linguaggi liberi NON sono chiusi relativamente all'intersezione. In altre parole dato $\mathcal{L}_1$ e $\mathcal{L}_2$, allora

$$\mathcal{L}_1 \cap \mathcal{L}_2$$

NON è libero

Per dimostrare si può considerare

- $\mathcal{L}_\infty = \{a^n b^n\}$ libero
- $\mathcal{L}_\in = \{c^n\}$ libero
- $\mathcal{L}_3 = \mathcal{L}_1\mathcal{L}_2 = \{a^n b^n c^j\}$ libero per teorema 2

Se ora considero:

- $\mathcal{L}_1 = \{a^n b^n c^j\}$
- $\mathcal{L}_2 = \{a^j b^n c^n\}$

la cui intersezione è $\{a^n b^n c^n\}$, NON è libero. Quest'ultimo fatto è dimostrabile con il pumping lemma

1.2 Chomsky normal form

Definizione 1: *Forma normale di chomsky*

Una grammatica è in forma normale di Chomsky se ogni produzione è di una delle seguenti forme:

- $A \to BC$
- $A \to b$

Dove A, B, C sono non terminali e b è terminale

1.2.1 Riduzione in forma di chomsky

E' sempre possibile ridurre una grammatica libera in forma di chomsky seguendo i seguenti passaggi:

- Eliminazione delle ϵ- produzioni
  - Per ogni produzione $A \to Y_1 Y_2 \dots Y_N$ considero tutti gli Y_i annullabili
  - Aggiungo tutte le possibili combinazioni di body che includano o meno i non terminali annullabili
  - Elimino ogni produzione $A \to \epsilon$

  Ad esempio, con X_1, X_2, X_3 annullabili, la produzione $A \to aX_1BX_2X_3cD$ va spezzata in:

a	X_1	B	X_2	X_3	c	D
a	X_1	B	X_2	X_3	c	D
a	X_1	B	X_2		c	D
a	X_1	B		X_3	c	D
a	X_1	B			c	D
a		B	X_2	X_3	c	D
a		B	X_2		c	D
a		B		X_3	c	D
a		B			c	D

  dando origine ai seguenti body:

$aX_1BX_2X_3cD$	aX_1BX_2cD	aX_1BX_3cD	aX_1BcD
aBX_2X_3cD	aBX_2cD	aBX_3cD	$aBcD$

- Eliminazione dei non terminali inutili (ossia dei non terminali che *non* compaiono in nessuna produzione)
- Eliminazione delle unity productions

- Dato $B \to \alpha \mid \beta$, sostituisco ogni produzione $A \to B$ con

$$A \to \alpha \mid \beta$$

- Spezzettamento body troppo lunghi
  - Ad esempio prendo $S \to aSb \mid ab$
  - Ogni terminale lo sostituisco con una nuova produzione:

$$S \to ASB \mid AB \quad A \to b \quad B \to b$$

  - Ogni stringa di non terminali con lunghezza ≥ 3 la spezzo inserendo un nuovo non terminale:

$$S \to AC \mid AB \quad C \to SB \quad A \to b \quad B \to b$$

1.3 Pumping lemma

Teorema 4: *Pumping lemma*

Dato un linguaggio libero $\mathcal{L}(\mathcal{G})$, allora:

- Dato $p \geq 0$
- $\forall z$ t.c. $|z| > p$ valgono le seguenti condizioni:
- Allora esiste un *"unpacking"* di z, ossia $z = uvwxy$ tali per cui
  - $|vwx| \leq p$
  - $|uv| > 0$
  - $\forall i \in \mathbb{N} \quad uv^i wx^i y \in \mathcal{L}(\mathcal{G})$

Informalmente, posso "pompare" all'infinito la lunghezza di parole abbastanza lunghe se il linguaggio è libero o, al contrario, le parole abbastanza lunghe possono essere ottenute solo tramite "pumping"

1.3.1 Dimostrazione

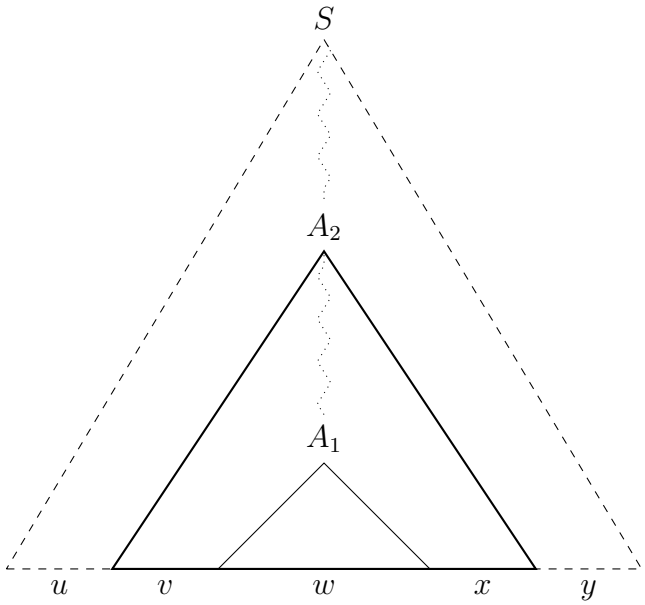
Considero la grammatica in forma normale di Chomski. Questo è sempre possibile come descritto in sezione 1.2. Questo è utile in quanto un albero di derivazione di una una parola in una grammatica in Chomsky normal form è un *albero binario*

- Considero una parola z con $|z| \geq 2^{k+1}$ dove k è il numero di non terminali distinti della grammatica

$$p = 2^{k+1}$$

- L'albero di derivazione di z ha altezza $\geq k+1$, dato che il caso in cui l'altezza è minima a parità di numero di foglie è quello in cui l'albero è completo
- Un path dalla root alla foglia di lunghezza massima ha $k+2$ nodi, di cui $k+1$ sono *non terminali*, dunque ho la certezza che *esista* almeno un non terminale ripetuto

- Fra i non terminali ripetuti, chiamo A quello la cui seconda occorrenza a partire dal basso sia più bassa
- Ora considero il seguente unpacking:



  - $|uwx| \leq p$ in quanto:
    - Ho selezionato A_2 in maniera tale che fosse il primo carattere ripetuto partendo dal basso.
    - Per questo il sottoalbero radicato in A_2 ha al più altezza k, quindi al più avrà 2^k foglie
  - $|vx| > 0$ in quanto non esistono produzioni unitarie nella forma normale di Chomsky

2 Top down parsing

Prendiamo ora come esempio una grammatica libera:

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \text{id}
\end{aligned}
$$

Definizione 2: *First(α)*

Data una grammatica e una qualsiasi stringa α, first (α) sono i caratteri terminali che possono comparire all'inizio della stringa dopo una sua qualsiasi espansione

Operativamente, per trovare i first:

- Prendo una derivazione $A \to XYZ$
- Se X è terminale allora lo aggiungo a first(α)
- Se X è non terminale allora aggiundo first (X) a first (α)
- Se X è *annullabile*, allora devo controllare anche i first(Y) (se $X \to *\varepsilon$ allora la stringa inizia per Y)

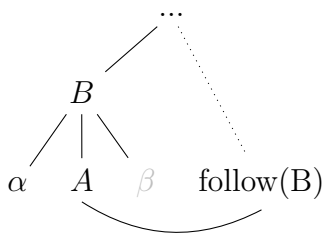
Definizione 3: *Follow(A)*

Data una grammatica e un qualsiasi non terminale A, follow (A) sono i caratteri che possono seguire A dopo una qualsiasi espansione

Operativamente, per trovare i follow(A):

- Prendo ogni derivazione $B \to \alpha A \beta$
- Aggiungo i first (β) a follow (A)
- Se β è *annullabile*, allora aggiungo follow (B) a follow (A)

Quest'ultima condizione è data dal fatto che



2.0.1 Esempio 1 first follow

Sempre in riferimento a questa grammatica, vediamo come calcolare i first e follow:

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \text{id}
\end{aligned}
$$

	first()	follow()
E	id, (	\$,)
E'	ϵ, +	\$,)
T	id, (	\$,), +
T'	ϵ, *	\$,), +
F	id, (	\$,), +, *

2.0.2 Esempio 2 first follow

$$
\begin{aligned}
&S \to aABb \\
&A \to Ac \mid d \\
&B \to CD \\
&C \to e \mid \epsilon \\
&D \to f \mid \epsilon
\end{aligned}
$$

	first()	follow()
S	a	\$
A	d	b, c, e, f
B	e, f, ϵ	b
C	e, ϵ	b, f
D	f, ϵ	b

Nota che nel calcolo di follow(A), quando considero $S \to aABb$, so che

$$\text{follow}\,(\text{A}) = \text{first}\,(\text{Bb}) + \text{follow}\,(S)$$

quindi non devo limiratmi ad aggiungere solo first(B)=e, f, ϵ, bensì first(Ab) = e, f, ϵ, b

2.0.2 Esempio 3 first follow

$$
\begin{aligned}
&S \to aA \mid bBc \\
&A \to Bd \mid Cc \\
&B \to e \mid \epsilon \\
&C \to f \mid \epsilon
\end{aligned}
$$

	first()	follow()
S	a, b	\$
A	e, d, f, c	\$
B	e, ϵ	c, d
C	f, ϵ	c

2.1 Costruzione tabella di parsing

La tabella di parsing può essere costruita secondo la seguente intuizione:

Prendo ogni non terminale e lo assegno ad ogni riga, prendo ogni terminale e lo assegno ad ogni colonna. `m[i][j]` è riempita se e solo se esiste qualche produzione che trasformi il non terminale i in una stringa che incomincia per j.

Operativamente per ogni produzione $A \to \alpha$:

- Aggiungo tutti i first(α)
- Se α è annullabile ($\epsilon \in \text{first}\,(\alpha)$) allora aggiungo tutti i follow(A)

Con aggiungo indico il fatto che la cella $m\,[A]\,[c]$ viene riempita con la produzione $A \to \alpha$

In altri termini

- Per ogni riga della tabella avrò una o più produzioni
- Per ogni produzione controllo quali caratteri C posso ottenere all'inizio di una qualsiasi stringa
- Aggiungo questa produzione nelle celle della riga corrente in corrispondenza di ogni carattere c trovato

Ad esempio, una tabella di parsing per la grammatica d'esempio

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

Tabella 1: Tabella di parsing LL(1) per il questa grammatica

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE' \mid \epsilon \\
&T \to FT' \\
&T' \to *FT' \mid \epsilon \\
&F \to (E) \mid \text{id}
\end{aligned}
$$

	first()	follow()
E	id, (	\$,)
E'	ϵ, +	\$,)
T	id, (	\$,), +
T'	ϵ, *	\$,), +
F	id, (	\$,), +, *

Nota che se il linguaggio è LL(1) allora non avrò celle con due produzioni

2.2 Parsing

Una volta costruita la tabella di parsing come descritto qui, posso verificare se una stringa appartiene ad una grammatica eseguiendone il parsing. L'algoritmo di appoggia su uno *stack* e procedere come segue:

- Lo stack contiene l'espansione parziale della parola. Inizialmente contiene solo S, ossia *lo starting symbol*
- Tengo un puntatore al carattere della parola che sto cercando di ottenere. Inizialmente questo è settato al primo carattere
- Scorro e considero `top` e `c`, rispettivamente il primo carattere dello stack e il carattere correntemente puntato:
  - Se `top` è terminale e `c == top` allora significa che sono riuscito ad espandere qualcosa e ottenere il carattere desiderato. Dunque avanzo il puntatore al carattere successivo
  - Se `c != top` devo trovare un modo per espandere `top` in `c`. Controllo la tabella di parsing in `m[top][c]`. Se ho una produzione la seguo, ossia rimuovo `top` dallo stack e inserisco il body della produzione
  - In tutti gli altri casi non posso procedere perchè la parola non appartiene al linguaggio. In particolare può succedere che:
    * `top` è terminale ma diverso da `b`
    * `top` è non terminale ma non ho una produzione in `m[top][c]` significa che la parola non appartiene al linguaggio

2.3 Left recursion

Definizione 4: *Left recursive grammar*

Una grammatica è left recursive se per qualche A ho che $A \to^* A\alpha$

Ad esempio, la grammatica

$$S \to B \mid a$$
$$B \to Sa \mid b$$

è *left recursive*. Se espando $S \to B \to Sa \to Ba \to Saa \dots$ e così via

Definizione 5: *Immediately left recursive grammar*

Una grammatica è imeddiatamente left recursive se per qualche A ho che $A \to A\alpha$

Ad esempio,la grammatica:

$$S \to Sa|b$$

è *immediately left recursive*. Infatti, espandendo $S \to Sa \to Saa \dots$ e così via

Teorema 5: *Left recursion e LL(1) parsing*

Se una grammatica è left recursive allora non può essere parsata con un parser LL(1)

Intuitivamente, supponendo di avere $S \to S\alpha|a$, in un parser come descritto in sezione 2.2 e una stringa $a\alpha\alpha$

	first	follow
S	a	$\$, \alpha$

	α	a	\$
S		$S \to a$ **and** $S \to S\alpha$	

L'idea è che se ho una ricorsione a sinistra del tipo $A \to A\alpha \mid \beta$, allora avrò sempre due produzioni che mi permettono di arrivare a β:

- $A \to A\alpha \to \beta\alpha$
- $A \to \beta$

quindi avrò sempre un conflitto nella tabella di parsing

2.3.1 Grafo dei riferimenti

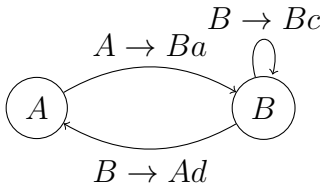
Sta roba la ho inventata a cazzo di cane ma aiuta con la ricorsione.

Definizione 6: *Grafo dei riferimenti*

Data una grammatica G, un grafo dei riferimenti è un grafo che ha come nodi i driver delle produzioni. Fra il nodo A e B esiste un arco se e solo se nel corpo di almeno una produzione con driver A compare B

Se esiste un ciclo in questo grafo c'è qualche tipo di ricorsione. Nel ciclio ogni arco (A, B) corrisponde a una produzione di tipo $A \to B\alpha$, allora la grammatica è *left recursive*. Per esempio

$$A \to Ba \mid b$$
$$B \to Bc \mid Ad \mid b$$



2.3.2 Eliminazione immediate left recursion

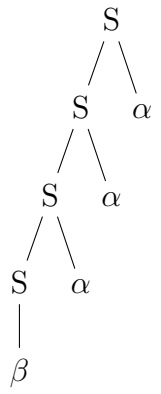
La ricorsione sinistra immediata può essere eliminata in modo abbastanza semplice. Supponiamo di avere:

$$S \to S\alpha \mid \beta$$

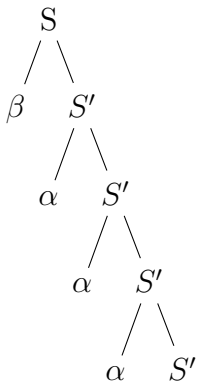
che è un linguaggio che genera stringhe di tipo $\beta\alpha^n$. Posso introdurre un fresh non terminal per eseguire una ricorsione a destra anzichè a sinistra:

$$S \to \beta \mid \beta S'$$
$$S' \to \alpha S' \mid \epsilon$$

Ad esempio, prendendo $\beta\alpha\alpha\alpha$



Grammatica 1



Grammatica 2

Più in generale possiamo procedere così:

Teorema 6: *Eliminazione ricorsione sinistra immediata*

Se abbiamo una grammatica *left recursive*, possiamo eliminare la ricorsione sinistra ottenendo una grammatica equivalente. In particolare, possiamo sostituire delle produzioni di questa forma

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_k$$

Dove $\alpha_j \neq \epsilon$ per ogni $j = 1 \ldots n$ e $\beta_i \neq A\gamma_i$ per ogni $i = 1 \ldots k$

con produzioni di quest'altra:

$$A \to \beta_1 A' \mid \ldots \mid \beta_k A'$$
$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$$

L'idea è sempre la stessa. La grammatica genera una stringa nella forma:

$$\{w_1 w_2 \mid w_1 \in \{\beta_1, \ldots, \beta_k\}, w_2 \in \{\alpha_1, \ldots, \alpha_n\}^*\}$$

quindi posso

- Dare la possibilità di espandere A in tutti i possibili β (A)
- Pushare dopo il β scelto un numero arbitrario di α (A')

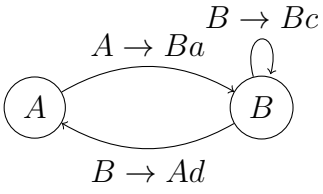
2.3.3 Eliminazione left recursion

Per eliminare in generale la *left recursion* posso

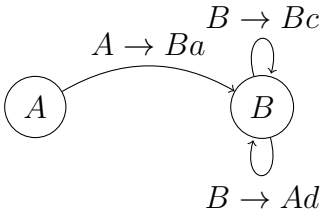
- Riporto la grammatica ad una situazione in cui esiste solo *immediate* left recursion
- Elimino la *immediate left recursion come descritto* in sezione 2.3.2

Per effettuare il primo punto posso utilizzare il grafo dei riferimenti (sez 2.3.1). Se esiste un ciclo devo eliminarlo modificando un arco (A, B) a (A, A) Prendiamo per esempio:

$$A \to Ba \mid b$$
$$B \to Bc \mid Ad \mid b$$



Posso prendere l'arco (B, A) e farlo puntare a B stesso



Per "reindirizzare" la produzione $B \to Ad$, ed eliminare il riferimento di A da parte di B, "includo" direttamente tutte le produzioni di A nella produzione che sto "reindirizzando":

$$B \to Ad$$

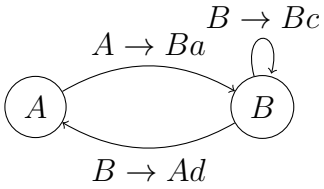
diventa

$$B \to Bad \mid bd$$

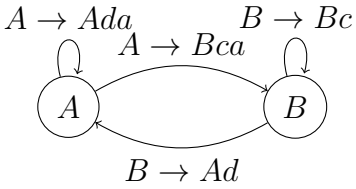
Di fatto, graficamente, nel nel grafo delle dipendenze prendo un arco (A, B) che voglio reindizizzare e

- Ottengo un nuovo grafo in cui ho eliminato (A, B)
- Ho un nuovo arco uscente da A che punta nella posizione di ogni arco uscente da B

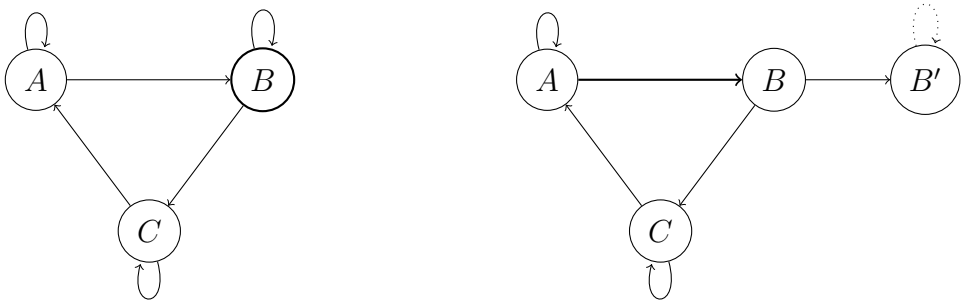
Nota che se esiste un self loop nel nodo di arrivo, allora non posso "reindirizzare" la produzione. Ad esempio



se reindirizzassi (A, B) non riuscirei ad eliminare il riferimento a B in A, in quanto otterrei:



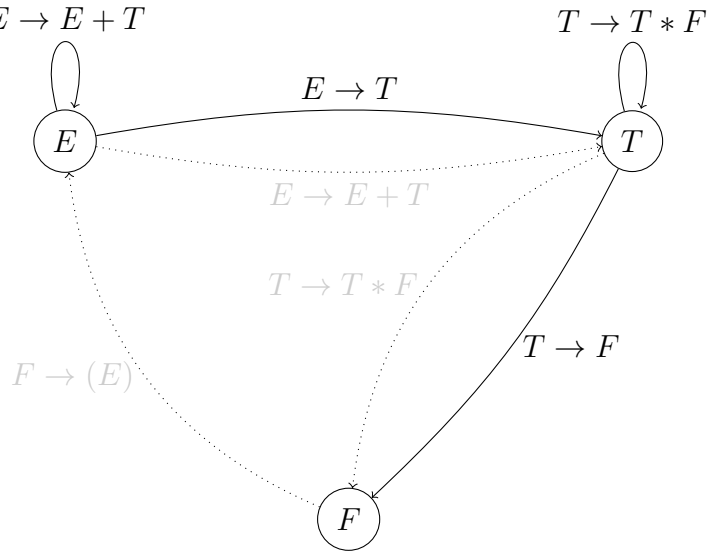
Quindi nel caso vi fosse un *self loop* sinistro su qualsiasi non terminale coinvolto nel ciclo principale, allora dovrei prima rimuoverlo su un nodo X come mostrato in sezione 2.3.2, e poi "reindirizzare" l'arco entrante in X:



Il self loop di B' è tratteggiato in quanto è vero che B' referenzia B' per forza, però è anche vero che non sarà mai una ricorsione sinistra, quindi non va considerata

2.3.4 Esempio 1

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \text{id}$$



Come si vede chiaramente nel grafico, ci sono solo left recursione *immediate*, dunque posso direttamente semplificarle:

- $E \to E + T \mid T$ diventa:

$$E \to TE'$$
$$E' \to +TE' | \epsilon$$

- $T \to T * F | F$ diventa:

$$T \to FT'$$
$$T' \to *FT'$$

Quindi ottengo:

$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \epsilon$$
$$F \to (E) \mid \text{id}$$

2.3.5 Esempio 2

$$E \to E + E \mid E * E \mid (E) \mid \text{id}$$

Qui chiaramente non posso avere cicli *non* elementari, dunque procediamo direttamente con l'eliminazione, ottenendo:

$$E \to (E)E' \mid \text{id}E'$$
$$E' \to +EE' \mid *EE' \mid \epsilon$$

Si noti però che non è comunque una grammatica LL(1), in quanto partendo da E', posso ottenere $+$ tramite

1. $E' \to +EE'$
2. $E' \to \epsilon$

abbiamo un risultato importante:

Una grammatica non left recursive non è necessariamente LL(1). Allo stesso modo non è garantito che la grammatica non sia ambigua

2.4 Fattorizzazione sinistra

Se una grammatica contiene due produzioni i cui body hanno lo stesso prefisso (iniziano per una serie di caratteri uguali), allora per forza di cose la grammatica non e LL(1). Posso però fattorizzare

Definizione 7: *Fattorizzazione a sinistra*

La fattorizzazione a sinistra consiste nel delayare quanto possibile la scelta della parte che contraddistingue due produzioni che condividono un prefisso nel body. Ad esempio:

$$S \to \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$$

può essere fattorizzata a sinistra in:

$$S \to \alpha S' \mid \gamma$$
$$S' \to \beta_1 \mid \beta_2$$

2.5 Conclusioni

- No left-recursive grammar is LL(1)
- No grammar that can be left-factorized is LL(1)
- No ambiguous grammar is LL(1)

Teorema 7: *Condizioni LL(1)*

$\mathcal{G}$ è LL(1) se e solo se per le produzioni $A \to \alpha \mid \beta$ si ha:

- $\text{first}(\alpha) \cap \text{first}(\beta) = \emptyset$
- Se $\epsilon \in \text{first}(\alpha)$ allora $\text{first}(\beta) \cap \text{follow}(A) = \emptyset$
- Se $\epsilon \in \text{first}(\beta)$ allora $\text{first}(\alpha) \cap \text{follow}(A) = \emptyset$

Intuitivamente, basta costruire la tabella come in sezione 2.1 e verificare che non ci siano celle con più produzioni canidate

3 Linguaggi regolari

Un linguaggio regolare ha una definizione molto semplice:

Definizione 8: *Linguaggio regolare*

Un linguaggio regolare è un linguaggio libero in cui ogni produzione ha la forma:

- $A \to aA$
- $A \to a$
- $A \to \epsilon$

Un'espressione regolare è definita per induzione:

Definizione 9: *Espressione regolare*

Dato un alfabeto $\mathcal{A}$, un'espressione regolare è definita per induzione:

- Caso *base*
  - Ogni $a \in \mathcal{A} \cup \epsilon$ è un'espressione regolare
- Passo *induttivo*. Siano r_1 e r_2 due espresioni regolari, allora:
  - *Alternamento* $r_1 \mid r_2$ è un'espressione regolare
  - *Concatenazione* $r_1 r_2$ è un'espressione regolare
  - *Kleene start* r^* (r) ripetuta 0 o più volte è un'espressione regolare
  - *Parentesi* (r_1) è un'espressione regolare

Nota che la precedenza degli operatori è, partendo dalla più alta

- Kleen star
- Concatenazione
- Alternamento

Quindi $(a \mid bc^*)$ è equivalente a $(a \mid (b\,(c^*)))$

3.1 Automa non deteministico a stati finiti

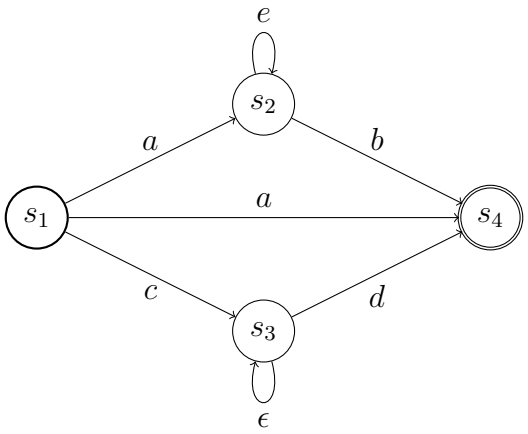
Definizione 10: *Automa non deteministico a stati finiti*

Un *automa non deterministico a stati finiti* è una tupla $(\mathcal{S}, \mathcal{A}, \text{move}_n, \mathcal{F})$, dove

- S è un insieme di stati.
- A è un alfabeto con $\epsilon \notin A$.
- $s_0 \in S$ è lo stato iniziale.
- $F \subseteq S$ è l'insieme degli stati finali (o accettanti).
- $\text{move}_n : S \times (A \cup \{\epsilon\}) \to 2^S$ è la funzione di transizione.

Questo automa può essere rappresentato in maniera conveniente con un grafo dove:

- $\mathcal{S}$ sono i nodi
- move_n sono gli archi, i cui "pesi" sono dati da lettere $\in \mathcal{A}$

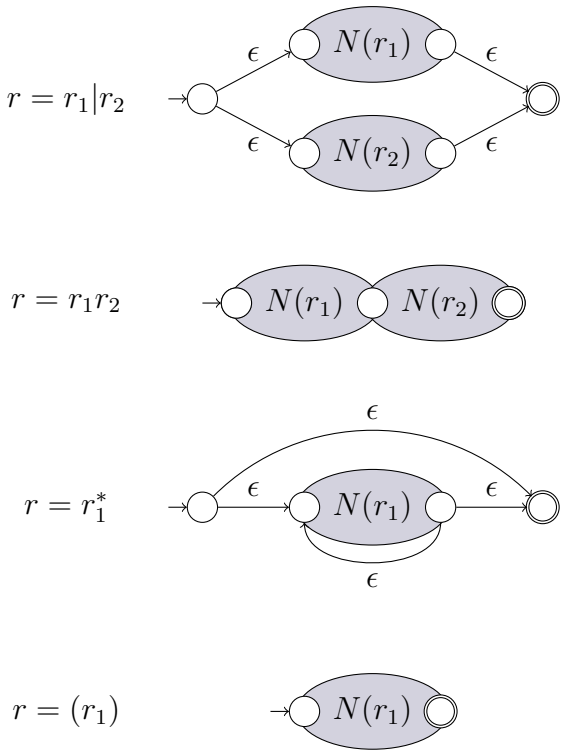


Nota che la funzione di transizione non è proprio ciò che ci si aspetta. Non è una matrice $n \times n$ in cui se esiste un arco di peso a fra i e j è rappresentato mettendo $m\,[i]\,[j] = a$. Qui la matrice ha questa forma:

	ϵ	a	b	c	d	e
s_1	$\emptyset$	$\{S_2, S_4\}$	$\emptyset$	$\{S_3\}$	$\emptyset$	$\emptyset$
s_2	$\emptyset$	$\emptyset$	$\{S_4\}$	$\emptyset$	$\emptyset$	$\{S_2\}$
s_3	$\{S_3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{s_4\}$	$\emptyset$
s_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

3.2 Construzione di Thompson

Dato un linguaggio regolare, possono costruire un automata che riconosce il suddetto linguaggio. Posso crearlo sfruttando la definizione induttiva di un linguaggio regolare. Ho componenti base per rapprdentare gli operatori:



3.3 Simulazione

La simulazione è il processo che dato un automa ci permette di verificare se una parola appartiene al linguaggio generato da esso

Definizione 11: *ϵ - chiusura*

Dato un insieme di stati di un automa, la ϵ chiusura si uno stato S è l'insieme di stati raggiungibili da S tramite soli archi ϵ

Possiamo sfruttare il concetto di ϵ- chiusura per verificare se una parola appartiene al linguaggio generato da un automa.

- Teniamo traccia in una struttura `states` gli stati "buoni" in questo momento
- Inizialmente, `states` contiene la ϵ- chiusura dello stato iniziale
- Per ogni carattere c, `states` diventa la ϵ- chiusura di *tutti gli stati raggiungibili da states tramite soli archi c*
- Si itera finchè non si arriva a fine parola
- Se in `states` ho almeno uno stato finale significa che $w \in L$, altrimenti no

3.4 Automa deterministico a stati finiti

Definizione 12: *Automa deterministico a stati finiti*

Un *automa deterministico a stati finiti* è una tupla $(\mathcal{S}, \mathcal{A}, \text{move}_n, \mathcal{F})$, dove

- S è un insieme di stati.
- A è un alfabeto con $\epsilon \notin A$.
- $s_0 \in S$ è lo stato iniziale.
- $F \subseteq S$ è l'insieme degli stati finali (o accettanti).
- $\text{move}_n : S \times A \to S$ è la funzione di transizione.

La differenza rispetto all'automa deterministico, sta nel fatto che

- Non vi sono ϵ- transizioni
- Ogni nodo ha *al più* un arco uscente pero ogni $a \in \mathcal{A}$

Questo perché

- NFA : $(S, A, \text{move}_n, s_0, F)$

  dove $\text{move}_n : S \times (A \cup \{\epsilon\}) \to 2^S$
- DFA : $(S, A, \text{move}_d, s_0, F)$

  dove $\text{move}_d : S \times A \to S$

Nota che dato un nodo S avendo un solo arco per ogni $a \in \mathcal{A}$, è molto facile controllare se $w \in L$. Basta *seguire* il perscorso, che è unico

3.4.1 Completamento funzione DFA

Spesso torna utile avere una funzione completa in un DFA. Per fortuna è sempre possibile prendere una funzione di transizione *parziale* e completarla come segue:

- Introduco nuovo nodo `sink`
- Per ogni nodo n a cui manca un arco uscente per una lerrea c dell'alfabeto, aggiungo arco $(n, \texttt{sink})_c$
- Per ogni lettera dell'alfabeto, aggiungo arco $(\texttt{sink}, \texttt{sink})_c$

3.4.2 Conversione da NFA a DFA

L'algoritmo è lento in culo e tedioso, però funziona come segue:

- Prendi la ϵ-chiusura di S_0 e la chiamo T
- Per ogni lettera dell'alfabeto x, devo vedere in quali nodi mi possa portare. Dunque considero l'insieme di nodi raggiungibili da un qualsiasi nodo di T con una x-transizione

- Calcolo la ϵ-chiusura di questo insieme. Se l'insieme è gia stato ottenuto, aggiungo un arco x da T all'insieme già ottenuto. Altrimenti ne creo un'altro e aggiungo l'arco
- Nota che gli insiemi non devono essere per forza disgiunti, anzi, è raro che lo siano

3.4.3 Minimizzazione DFA

L'idea di base si basa sul concetto di equivalenza di stati:

Definizione 13: *Equivalenza di due stati*

Dadi due stati s e t di un DFA D, questi si dicono equivalenti se importando il nodo di partenza su di uno o sull'altro ottengo lo stesso linguaggio. Formalmente:

$$\mathcal{L}_D(s) = \mathcal{L}_D(t)$$

L'idea è quindi quella di partizionare il grafo in *gruppi di stati equivalenti*. L'algoritmo si basa sulla seguente proprietà:

Se due stati s e t sono equivalenti, allora non possono esistere α-archi che partano da s e t e vadano in due gruppi diversi

Questo perché per come costruiamo i gruppi, sicuramente se due nodi stanno in due gruppi diversi non sono equivalenti. Dunque per forza se faccio

- Parto da s e vado con un arco a in B_1:

$$s \to_a B_1 \quad \text{ottengo } a\mathcal{L}(B_1)$$

- Parto da t e vado con un arco a in B_2:

$$t \to_a B_2 \quad \text{ottengo } a\mathcal{L}(B_2)$$

Siccome, per come costruiamo i gruppi $\mathcal{L}(B_1) \neq \mathcal{L}(B_2)$, per forza si cose gli stati s e t *non* sono equivalenti La procedura di minimizzazione procede così:

- Completo la funzione di transizione del DFA, come descritto in sezione 3.4.1
- Partiziono i nodi dell'automata in due insiemi:
  - F: gli stati finali
  - $S - F$: gli stati non finali
- Se esistono due nodi s e t che hanno delle α - transizioni in gruppi diversi, allora posso spezzare il gruppo corrente
- Supponendo di aver trovare s e t che puntano in due gruppi diversi B_1 e B_2 tramite α- transizione, allora posso dividere il gruppo corrente B in due
  - I nodi che puntano in B_1
  - I nodi che puntano in B_2
- Procedo ricorsivamente finchè possibile. Posso raggruppare in un singolo nodi i gruppi che non possono essere splittati ulteriormente

3.4.4 Numero stati di un DFA

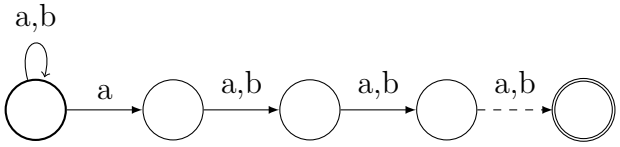
Sappiamo che fare parsin tramite un NFA è meno efficiente rispetto che tramite DFA. Tuttavia, non è tutto oro ciò che luccica

Teorema 8: *Numero stati di un DFA*

Per ogni NFA con $n+1$ stati esiste un DFA corrispondente che ha almeno 2^n stati e una funzione di transizione totale

Questo equivale a dire che nel caso pessimo, convertendo un NFA con n nodi otteniamo un DFA con 2^n nodi Dimostrazione: Considero linguaggio dato da

$$\mathcal{L} = \mathcal{L}\left((a \mid b)^* \, a \, (a|b)^{n-1}\right)$$



Ora consideriamo due parole diverse di lunghezza n: w_1 e w_2. Dato che le parole sono diverse, allora esiste un indice k al quale una sarà a e l'altra b. Senza perdita di generalità assumiamo che

$$w_1 = \underbrace{\ldots}_{k-1} a \underbrace{\ldots}_{n-k} \qquad\qquad w_1 = \underbrace{\ldots}_{k-1} b \underbrace{\ldots}_{n-k}$$

Ossia che in posizione k esima $w_1 = a$ e $w_2 = b$. Ora notiamo che

$$w_1 b^k \in \mathcal{L} \qquad\qquad w_2 b^k \notin \mathcal{L}$$

Questo perchè

$$w_1 b^k = \underbrace{\ldots}_{k-1} a \overbrace{\underbrace{\ldots}_{n-k} b^k}^{n} \qquad\qquad w_2 b^k = \underbrace{\ldots}_{k-1} b \overbrace{\underbrace{\ldots}_{n-k} b^k}^{n}$$

Quindi la seconda non è il $\mathcal{L}$ in quanto non finisce per a seguita da n caratteri. Ora ragioniamo così:

- Due parole distinte non possono terminare nello stesso stato. Questo sarebbe assurdo in quanto abbiamo dimostrato che una è accettata dal linguaggio e una non lo è. Se finissero sullo stesso stato dovrebbero essere entrambe accettate oppure non accettate
- Siccome su $\{a, b\}$ esistono 2^k parole distinte, e per ogni parola distinta necessito di almeno uno stato, allora gli stati del DFA sono almeno 2^n

3.5 Pumping lemma per linguaggi regolari

Teorema 9: *Pumping lemma per linguaggi regolari*

Dato un linguaggio regolare $\mathcal{L}(\mathcal{G})$, allora:

- Dato $p \geq 0$
- $\forall z$ t.c. $|z| > p$ valgono le seguenti condizioni:
- Allora esiste un *"unpacking"* di z, ossia $z = uvw$ tale per cui
  - $|vw| \leq p$
  - $|v| > 0$
  - $\forall i \in \mathbb{N} \quad uv^i w \in \mathcal{L}(\mathcal{G})$

Dimostrazione: considero l'automa DFA relativo a $\mathcal{L}$

- Scelgo $p = |S|$ dove $|S|$ è il numero di stati dell'automa
- Noto che ogni parola w con $|w| > p$ è formata da un path nell'automa che ha almeno un nodo che viene visitato *due volte*
- Questo garantisce che il path che genera la parola w convenga un ciclo. Chiamando il nodo che viene visitato due volte s, allora il path che genera w è

$$a_1 \dots s\ a_i\ \dots a_j\ s \dots a_z$$

dunque posso eseguire l'unpacking come segue:

$$\begin{aligned} u &= a_1 \dots s \\ v &= s\ a_i \dots a_j\ s \\ w &= s \dots a_z \end{aligned}$$

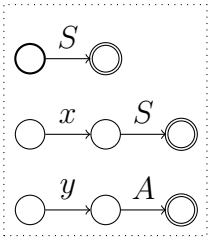
Dunque posso eseguire il pumping percorrendo il ciclo all'infinito

4 Bottom up parsing

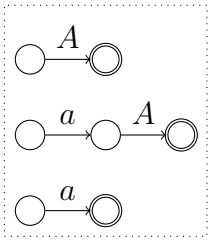
$$\begin{aligned} S &\to xS \mid yA \\ A &\to a \mid aA \end{aligned}$$

L'idea è di costruire un automa che matchi una qualsiasi espansione parziale questo alfabeto. Ad esempio, creiamo un automa che matchi sia $xxyaa$, ma anche $xxyA$.

Innanzitutto, costruiamo un automa che matchi ogni produzione base della grammatica. Questo è facile da ottenere

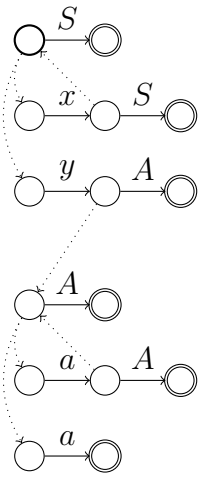


$S \to xS \mid yA$



$A \to a \mid aA$

Dopodichè bisogna ragionare nel seguente modo. Se siamo in uno stato qualsiasi, possiamo matchare come prossima cosa sia un simbolo già matchato (es S), sia una sua qualsiasi espansione (es xS o yA). Dunque devo aggiungere archi secondo questa precisa logica



L'idea è proprio che se sono su un nodo qualsiasi, posso aspettarmi che dopo venga il simbolo già ridotto (S, A), oppure qualsiasi stringa derivante da una sua espansione. Collego quindi ogni arco $S_1 \to_{\text{non terminale}} S_2$ ad ogni "sotto automa" che rappresenta una produzione basilare. Nota che si generano archi inutili

4.1 LR(0) items

Per capire bene come definire un automa di parsing lr(0), è necessario chiarire il concetto di *LR(0)-item*.

L'idea di base è che nel parsing bottom up, un parsers cerca di matchare una porzione di una stringa come espansione di un driver. Le informazioni riguardo cosa sta cercando di matchare con cosa sono contenute in un LR(0)-item.

Definizione 14: *LR(0) item*

Un item LR(0) è una produzione nella forma

$$A \to \alpha \cdot \beta$$

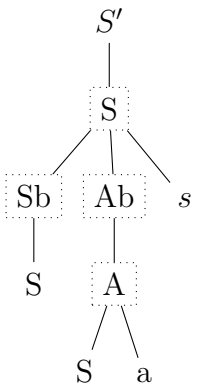
L'item indica che il parser sta cercando di matchare con A una stringa con forma $\alpha\beta$. In particolare α è già stato matchato, mentre β no

4.1.1 Chiusura di un LR(0) item

Per la costruzione di un automa LR(0), è necessario definire la chiusura di un LR(0) item. Intuitivamente, la chiusura di un item LR(0) $A \to \alpha \cdot \beta$ è l'insieme di LR(0) items che potrebbe essere necessario matchare *transitivamente* per matchare A Supponendo di avere

$$S \to Sb \mid Ab \mid s$$
$$A \to S \mid a$$

Se sono in uno stato in cui sto cercando di matchare l'espansione $S' \to \cdot S$, allora posso transitivamente star cercando di matchare ogni sottoalbero sinistro:



Operativamente per calcolare la chiusura devo:

- Per ogni nella forma $A \to \alpha \cdot B\beta$, ossia ogni LR(0) item con un *non terminale* immediatamente dopo il pallino
- Prendo ogni produzione $B \to \gamma$ e aggiungo l'item $B \to \cdot\gamma$ alla chiusura se non è già presente
- Ripeto fino a saturazione

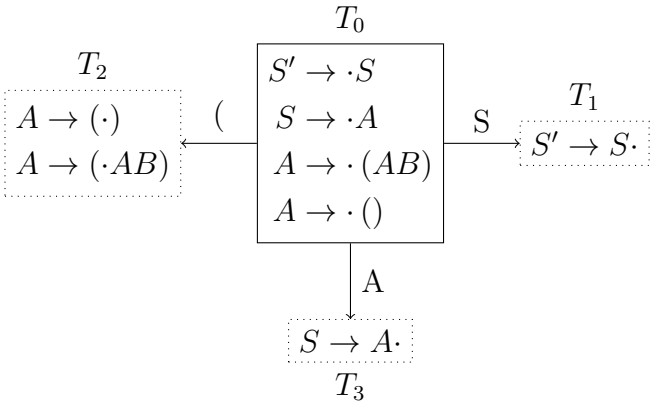
4.2 Costruzione automa LR(0)

L'idea nella costruzione dell'automa LR(0) è quella mettere in ciascun stato un insieme di $LR\,(0)$ items, che indicano tutte le possibili espansioni che il parser sta cercando di matchare nel momento in cui si trova nel nodo corrente. Dunque, formalmente

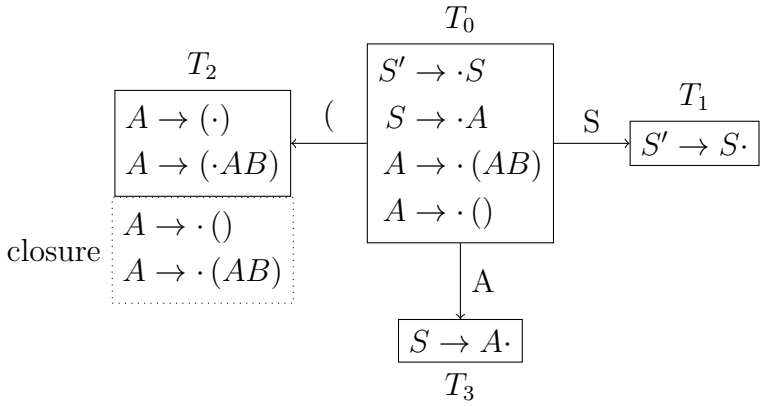
- Partiano calcolando la LR(0)-closure del LR(0) nuovo $S' \to \cdot S$. Questo contiene tutto ciò che il parser può star cercando di matchare senza ancora aver matchato nulla
- Aggiungo un arco uscente per ogni carattere presente dopo il pallino in qualsiasi LR(0) item del nodo corrente, ad es:

$$\begin{aligned} S' &\to \cdot S \\ S &\to \cdot A \\ A &\to \cdot (AB) \\ A &\to \cdot () \end{aligned}$$

  a sinistra del pallino avrò $\{S, A, (\}$. Aggiungo un arco uscrente per ciascun dei seguenti simboli
- L'idea è che con un'A-transizione posso arrivare ai nuovi stati, matchando il carattere immediatamente a sinistra del pallino.
- Gli LR(0) items tramite cui arrivo al nuovo stato costituiscono il *kernel* del nodo.



- Ora calcolo la chiusura di ogni nuovo nodo aggiunto (T_1, T_2, T_3)



4.3 Costruzione tabella di parsing

Una volta costruito un automa come in 4.2, è necessario costruire la tabella di parsing. Intuitivamente, mettiamo

- Su ogni riga: stati dell'automa
- Su ogni colonna: lettere alfabeto

Riempiamo poi la tabella come segue. `tab[P][Y]` indica che sono nello stato `P` e sto cercando di percorrere l'arco `Y`. Dunque `tab[P][Y]` =

- *Shift* se `Y` è terminale, ossia percorro arco Y
- *Goto* se `Y` è non terminale, ossia percorro arco Y
- *Reduce* in `tab[P][Y]` è contenuto un *reducing item*
- *Accept* se in `tab[P][Y]` è contenuto l'*accepting item* ($S' \to S\cdot$)

Affinchè la tabella possa essere costruita, non devono esserci celle con entrambi

- *Shift-reduce*
- *Reduce-reduce*

Questo dipende dalla grammatica. Appunto, per definizione, una grammatica è SLR(1) se ciò *NON accade*

4.4 Algoritmo di parsing

Una volta costruito un automa come in 4.2

- Inizialmente parto sul nodo T_0
- Tengo traccia del path di nodi visitato in `state_stack`
- Tengo traccia del path di nodi visitato in `state_stack`
- Tengo traccia della riduzione parziale della parola in `symbol_stack`
- Controllo se posso fare un'operazione di *shift/goto*. Se possibile la eseguo
- Se approdo in un nodo reduce con forma: $A \to \alpha\cdot$, significa che ho matchato un'espansione. Dunque
  - Rimuovo $|\alpha|$ caratteri da `symbol_stack` e inserisco la loro riduzione A
  - Rimuovo $|\alpha|$ carattedi da `state_stack`. Avendo matchato A, allora devo ricominciare a matchare dallo stato in cui ho iniziato a matchare A stesso
- Se approdo in un nodo con *accept*, so di aver matchato l'intera parola

Nota che questo algoritmo è leggermente diverso da quello proposto dalla professoressa, in quanto

- Nella costruzione della tabella di parsing, la prof inserisce *reduce* $A \to \beta$ solo se il carattere che viene dopo appartiene ai follow(A)
- La riduzione non avviene direttamente se poi non esiste un arco con il carattere seguente
- L'algoritmo non gestisce separatamente i *goto*. Effettivamente i *goto* possono solo avvenire nel momento in cui c'è una riduzione
- Inoltre, ogni qualvolta avvenga una riduzione, l'algoritmo "riavvolge" il path (esegue il pop da `symbol_stack` e `state_stack`) viene in automatico eseguito il *goto*. Questo è possibile in quanto:
  - Se una cella contiene *reduce* $A \to \alpha\cdot$, allora, riavvolgendo approdo necessariamente su una cella con un item $A \to \cdot\alpha$
  - La produzione $A \to \cdot\alpha$ sicuramente non è nel kernel dello stato. Questo perché è nel kernel se è derivata da una transizione (ci sono arrivato tramite arco), e in questo caso le stellina è almeno in seconda posizione.
  - Se il nodo è il primo (T_0), allora c'è la produzione aggiuntiva $S' \to S$ che fa da cuscinetto. Non riavvolgerò mai ad un'espansione di S' un quando l'unico modo sarebbe di riavvolgere dall'*accepting node* ma li l'algoritmo termina

5 Competenze richieste

5.1 Linguaggi liberi

5.1.1 Preliminari e lemmas

- Chiusura linguaggi liberi rispetto all'*unione*
- Chiusura linguaggi liberi rispetto all'*concatenazione*
- Non chiusura linguaggi liberi rispetto all'*intersezione*
- Forma normale di *Chomsky* e algoritmo per trasformare linguaggio libero in CNF

5.1.2 Pumping Lemma

- Dimostrazione
- Applicazione per dimostrare che un linguaggio *non* è libero

5.1.3 Altro

- Alcuni linguaggi liberi noti (es $\{a^n b^n \mid n > 0\}$)

5.2 Top down parsing

5.2.0 Parsing

- Calcolo $first(\alpha)$
- Calcolo $follow(\alpha)$
- Costruzione della tabella di parsing LL(1)
- Algoritmo di parsing LL(1)

5.2.0 Caratteristiche linguaggi LL(1)

- Left recursion e metodi di eliminazione
- Fattorizzazione a sinistra
- Condizione per cui un linguaggio è LL(1)

5.3 Linguaggi regolari

- Definizione espressione regolare e linguaggio regolare
- Definizione automa a stati finiti non deterministico (NFA)
- Definizione automa a stati finiti deterministico (DFA)
- Costruzione di *Thompson* (analisi complessità e spazio)
- Simulazione di un NFA (analisi complessità e spazio)
- Completamento funzione di transizione
- Subset construction (*convertsione* da NFA a DFA)
- Minimizzazione di un DFA (metodo del partizionamento)
- *Pumping lemma* per linguaggi regolari

5.4 Bottom up parsing

- LR(0) items e chiusura
- Costruzione automa LR(0)
- Costruzione tabella di parsing LR(0)
- Algoritmo di parsing LR(0)
- Risoluzione conflitti

5.5 Syntax Directed Definitions

- Valutazione di un SDD
- Grammatiche S attibuite e grammatiche L attributite
- Valutazione espressioni arimetiche grammatiche LARL(1) (S *attribuite*)
- Valutazione espressioni arimetiche grammatiche LL(1) (L *atributite*)
- Valutazione SDD durante il parsing SLR(1)
- Casi d'esempio:
  - Valutazione numeri (binario, ottale, decimale)
  - Conversione numeri (`float`, `integers`)
  - Creazione *abstract syntax tree*

5.6 Generazione codice intermedio

- Traduzione di *espressioni aritmetiche*
- Traduzione di *control flow* (while, if, not, booleani)
- Tecnica del *fall*
- Tipizzazione arrays
- Accesso ad arrays