Contents

1 Introduzione **5**
1.1 Descrivere un linguaggio . . . 5
1.1.1 Sintassi . . . 5
1.1.2 Semantica . . . 5
1.1.3 Implementazione . . . 5
1.2 Linguaggio formale . . . 6
1.2.1 Operazioni e definizioni aggiuntive . . . 6
1.2.2 Operazioni su linguaggi . . . 6
1.2.3 Rappresentazione finita dei linguaggi . . . 7

2 Grammatica **9**
2.1 Derivazioni . . . 9
2.1.1 Alberi di derivazione . . . 9
2.1.2 Alberi sintattici . . . 10
2.1.3 Ambiguità . . . 10

3 Vincoli sintattici contestuali **13**
3.1 Semantica dinamica . . . 13

4 Struttura di un compilatore **15**
4.0.1 Analisi lessicale (scanner) . . . 15
4.0.2 Analisi sintattica (parser) . . . 16
4.0.3 Analisi semantica . . . 16
4.0.4 Generazione della forma intermedia . . . 16
4.0.5 Ottimizzazione . . . 16
4.0.6 Generazione del codice . . . 16
4.0.7 Tabella dei simboli . . . 16

5 Semantica operazionale strutturata **17**
5.1 Cosa serve per dare la semantica . . . 17
5.1.1 Problemi . . . 17
5.2 Semantica di un ”linguaggio di programmazione”: semantica espressioni . . . 17
5.2.1 Punti da notare . . . 18
5.2.2 Determinatezza relazione di transizione . . . 19
5.2.3 Funzione di valutazione . . . 20
5.2.4 Regola di valutazione . . . 20
5.3 Semantica espressioni booleane . . . 20
5.3.1 Regole di valutazioni alternative . . . 21
5.4 Semantica dei comandi . . . 21
5.4.1 Errori non ben categorizzati . . . 22
5.5 Errori dinamici . . . 23
5.5.1 Semantiche aggiuntive . . . 23
5.6 Non determinismo e parallelismo . . . 24

6 Linguaggi regolari e espressioni regolari 25
6.1 Analisi lessicale . . . 25
6.1.1 Che cosa sono i token . . . 25
6.2 Espressioni regolari . . . 26
6.2.1 Linguaggio denotato da un espressione regolare . . . 26
6.3 Linguaggio regolare . . . 27
6.3.1 Esempi di espressioni regolari . . . 27
6.3.2 Operatori ausiliari delle espressioni regolari . . . 27
6.3.3 Definizioni regolari . . . 28
6.3.4 Equivalenza fra espressioni regolari . . . 28

7 Automi a stati finiti 29
7.1 Introduzione . . . 29
7.1.1 Funzionamento degli automi . . . 29
7.1.2 Diagramma di transizione . . . 29
7.2 Automi finiti non deterministici NFA . . . 30
7.2.1 Linguaggio riconosciuto . . . 30
7.2.2 Problemi degli NFA . . . 31
7.3 Automi finiti deterministici . . . 32
7.3.1 Epsilon-closure . . . 32
7.3.2 Utilizzo dell'epsilon-closure per definire linguaggi riconosciuti . . . 33
7.3.3 Funzione mossa . . . 33
7.3.4 Costruzione per sottoinsiemi . . . 33
7.4 Equivalenza NFA e DFA . . . 35
7.5 Da espressione regolare a NFA equivalente . . . 35

8 Grammatiche regolari 39
8.1 Da grammatica regolare a NFA equivalente . . . 39
8.2 Da DFA a grammatiche regolari . . . 41
8.3 Grammatiche regolari ed espressioni regolari . . . 42
8.4 Riassunto equivalenza NFA, regex, DFA, grammatiche regolari . . . 43
8.4.1 Costruire uno scanner . . . 43

9 Minimizzazione DFA 45
9.1 Equivalenza (o indistinguibilità) . . . 46
9.2 Relazione di equivalenza . . . 47
9.2.1 Osservazioni . . . 47
9.2.2 Tabella di equivalenza . . . 48
9.3 Algoritmo per minimizzazione . . . 49
9.4 Automa minimo . . . 50

10 Lex e Yacc 51
10.1 Lex generatore di analizzatori lessicali . . . 51
10.1.1 Come è fatto un file di input Lex . . . 51
10.1.2 Funzionamento del programma dato in output . . . 52
10.2 Utilizzo di Yacc . . . 53

11 Proprietà algoritmiche dei linguaggi regolari 55
11.1 Pumping lemma . . . 56
11.1.1 Come usare il pumping lemma per dire se un linguaggio non è regolare . . . 56
11.2 Altre proprietà dei linguaggi regolari . . . 57

12 Automi a pila 59
12.1 Analisi sintattica . . . 59
12.2 Automi a pila . . . 59
12.2.1 Problemi che risolvono i PDA rispetto ai DFA/NFA . . . 60
12.2.2 Transizioni di un PDA . . . 60
12.3 Linguaggio accettato da un PDA . . . 61
12.3.1 Esempi di PDA . . . 61

12.3.2 Classe di linguaggi riconosciuta per pila vuota e per stato finale 62
12.4 Come ottenere un PDA da una grammatica libera 63

13 Proprietà linguaggi liberi 65
13.1 Proprietà di chiusura 65
13.1.1 I linguaggi liberi non sono chiusi per intersezione 65
13.1.2 I linguaggi liberi intersecato a linguaggio regolare 66
13.2 Pumping theorem 67
13.2.1 Negazione del pumping theorem 68
13.3 Classificazione di chomsky 69

14 DPDA e linguaggi deterministici 71
14.1 Linguaggi liberi deterministici 71
14.1.1 Linguaggi regolari e DPDA 72
14.1.2 Prefix property 72
14.1.3 Assicurarsi la prefix property 72
14.2 Proprietà dei linguaggi liberi deterministici 73
14.2.1 Non ambiguità 73
14.2.2 Chiusura 73

15 Analisi sintattica: Parser 75
15.1 Introduzione parser top-down 75
15.1.1 Grammatiche non adatte 76
15.2 Introduzione parser bottom up 77
15.2.1 Non determinismo e conflitti 78
15.2.2 Problema con produzioni epsilon 78

16 Semplificazione di grammatiche 79
16.1 Eliminare le produzioni epsilon 79
16.1.1 Simboli annullabili 79
16.1.2 Algoritmo 79
16.2 Eliminare le produzioni unitarie 80
16.2.1 Coppie unitarie 80
16.2.2 Algoritmo 81
16.3 Rimuovere i simboli inutili 82
16.3.1 Come calcolare i generatori 82
16.3.2 Come calcolare i raggiungibili 82
16.3.3 Algoritmo 82
16.4 Mettere tutto insieme 84
16.5 Forme normali 85
16.6 Eliminare la ricorsione sinistra 85
16.6.1 Come rimuovere la ricorsione immediata 85
16.6.2 Come rimuovere la ricorsione non immediata 86
16.7 Fattorizzazione a sinistra 87

17 Parser top down 89
17.1 Parser a discesa ricorsiva 89
17.2 First e follow 90
17.2.1 First 90
17.2.2 Follow 91
17.3 Tabelle di parsing LL(1) 92
17.3.1 Grammatiche LL(1) 92
17.3.2 Parser non ricorsivo che fa uso della pila 93
17.3.3 Linguaggi regolari LL(1) 95
17.4 Grammatiche LL(K) 95
17.4.1 Tabella di parsing LL(K) 96
17.5 Quando una grammatica è LL(K) 96

18 Parser bottom up 97
18.1 Parser shift reduce non deterministico . . . 97
18.1.1 Come risolvere i conflitti? . . . 98
18.2 Come è fatto un parser LR . . . 98
18.2.1 Mosse del parser LR . . . 99
18.2.2 DFA dei prefissi viabili . . . 99
18.2.3 Item LR(0) . . . 99
18.2.4 Come costruire l'NFA dei prefissi viabili . . . 100
18.3 Automa canonico LR(0) . . . 101
18.3.1 Costruzione diretta dell'automa canonico LR(0) . . . 101
18.4 Tabella di parsing LR . . . 102
18.4.1 Come riempirla . . . 102
18.4.2 Grammatiche LR(0) . . . 103
18.5 Algoritmo del parser . . . 104

19 SLR(1), LR(1), LALR(1) 105
19.1 Quando bastano i parser LR(0) . . . 105
19.2 Tabella di parsing SLR(1) . . . 105
19.3 Parser LR(1) . . . 105
19.3.1 Item LR(1) . . . 105
19.3.2 NFA LR(1) . . . 106
19.4 Automa canonico LR(1) . . . 106
19.4.1 Clos e Goto . . . 107
19.5 Tabella di parsing LR(1) . . . 107
19.6 Parser LALR(1) . . . 107
19.6.1 Come ottenere un parser LALR(1) . . . 107
19.6.2 Problemi di LALR(1) . . . 107

20 Grammatiche LL(K), LR(K), SLR(K) e LALR(K) 109
20.1 Grammatiche LL(K), LR(K), SLR(K) e LALR(K) . . . 109
20.1.1 Grammatiche LR(K) . . . 109
20.1.2 Grammatiche SLR(K) . . . 109
20.1.3 Grammatiche lALR(K) . . . 109
20.2 Relazione tra grammatiche . . . 109
20.3 Proprietà di grammatiche LL(K) e LR(K) . . . 109
20.4 Classificazione dei linguaggi . . . 110
20.4.1 Teoremi utili per la classificazione di linguaggi . . . 110

Chapter 1

Introduzione

1.1 Descrivere un linguaggio

Un linguaggi viene descritto attraverso tre elementi:

- **La sintassi** che sono le regole che ci dicono quando una frase è corretta.
- **La semantica** che assegna un significato a ogni frase corretta.
- **La pragmatica** (o "buon senso") che definisce come frasi corrette e di senso compiute vengono utilizzate.

Inoltre se un linguaggio è un linguaggi eseguibile va data anche una sua **implementazione** che esprime come eseguire una frase corretta rispettando la semantica.

1.1.1 Sintassi

La sintassi si divide inoltre in due sotto-aspetti:

- **Aspetto lessicale** che è la descrizione del lessico (parole) che si può usare. Questo può essere immagazzinato in un dizionario o una struttura dati.
- **Aspetto grammaticale** che è la descrizione di come si può mettere insieme il lessico per creare frasi corrette. Si può implementare attraverso regole grammaticali (finite) e ciò generano un numero infinito di frasi possibili.

Quindi avremo che l'alfabeto è l'**insieme di caratteri** del linguaggio, il lessico l'**insieme di sequenze finite** (parole) e le frasi sono degli **insiemi di lessico**.

1.1.2 Semantica

Per poter descrivere la semantica ho bisogno di:

- Un dizionario che descriva la semantica di ogni lessico.
- Per le frasi devo sapere il linguaggio di appartenenza, un linguaggio in cui descrivere il significato della frase (ad esempio devo dare la semantica di una frase inglese ad un italiano la posso dare in italiano perché non deve essere spiegato).

1.1.3 Implementazione

L'implementazione di un linguaggio eseguito è l'esecuzione di una frase sintatticamente corretta rispettandone la semantica.

1.2 Linguaggio formale

Prima di poter definire cosa sia un linguaggio formale dobbiamo definire due elementi:

- **Un alfabeto** A che è un insieme finito di elementi detti **simboli**.
- **Una parola** su alfabeto A che è una sequenza finita di simboli in A.

Possiamo quindi dire che un linguaggio formale L su alfabeto A è **un insieme di parole su** A. Definendo inoltre un insieme A^* come:

$$A^* = \bigcup_{n \geq 0} A^n \qquad \text{dove:} \quad A^0 = \{\epsilon\} \quad \text{e} \quad A^{n+1} = A \cdot A^n$$

$$\text{dove l'operazione } \cdot \text{ è definita come:} \quad A \cdot A^n = \{aw \mid a \in A \text{ e } w \in A^n\}$$

avremo che un linguaggio formale L su alfabeto A è un sottoinsieme di A^* (cioè $L \subseteq A^*$). Notiamo che l'insieme A^* è un insieme infinito contabile (cioè possiamo enumerarlo).

1.2.1 Operazioni e definizioni aggiuntive

Altre definizioni e operazioni importanti per questo corso:

- **Lunghezza** di una parola. Definita in modo induttivo come:

$$|\epsilon| = 0 \qquad |aw| = 1 + |w|$$

- **Concatenazione** di due parole x e y è la stringa w ottenuta giustapponendo x a y. Valgono le seguenti **leggi della concatenazione**:
  - **Associatività** $x(yz) = (xy)z$.
  - **Elemento neutro** $x\epsilon = x = \epsilon x$.
- **Sottostringa**. La definizione di sottostringa è la seguente:

$$v \text{ si dice sottostringa di } w \Leftrightarrow \exists x, y \in A^* \text{ tale che } w = xvy$$

  Notiamo che ogni stringa è sottostringa di se stesso e che la stringa ϵ è sottostringa di ogni stringa.
- **Suffisso e prefisso** una stringa v si dice suffisso di w se e solo se $\exists x \in A^*$ tale che $w = xv$, nel caso in cui v sia posta prima di x si dice prefisso.
- **Potenza n-esima** di una stringa w è definita come:

$$w^0 = \epsilon \qquad w^{n+1} = ww^n \qquad \forall n \geq 0$$

  e' definita come la concatenazione di w n volte (es: $w = ab$ allora $w^3 = ababab$).

1.2.2 Operazioni su linguaggi

Ci sono diverse possibili operazioni applicabile ad un linguaggi L:

- **Complemento** definito come $\bar{L} = \{w \in A^* \mid w \notin L\} = A^* \setminus L$
- **Unione e intersezione** definiti come:

$$L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$$

$$L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$$

- **Concatenazione** definita come:

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

  cioè ogni parola del nuovo linguaggio deve essere composta da una parola del primo seguita da una del secondo.

- **Prodotto cartesiano** definito come:

$$L_1 \times L_2 = \{(w_1, w_2) \mid w_1 \in L_1 \land w_2 \in L_2\}$$

  Notiamo che non è sempre vero che $L_1 \cdot L_2 = L_1 \times L_2$ poiché nel caso della concatenazione valgono anche le regole della concatenazione, in particolare $x \cdot \epsilon = x \neq (x, \epsilon)$ e anche per il fatto che nella concatenazione due stringhe uguali vengono contante come una, nelle prodotto cartesiano no.

- **Potenza di un linguaggio** è definita come:

$$L^0 = \{\epsilon\} \qquad L^{n+1} = L \cdot L^n \qquad \forall n \geq 0$$

- **Stella di Kleene o chiusura** definita come:

$$L^* = \bigcup_{n \geq 0} L^n \quad \text{(chiusura)} \qquad L^+ = \bigcup_{n \geq 1} L^n \quad \text{(chiusura positiva)}$$

1.2.3 Rappresentazione finita dei linguaggi

Se abbiamo che L **è infinito come viene rappresentato**? Non possiamo memorizzare ogni singola stringa del linguaggio visto che è infinito, ma quello che possiamo fare è trovare una **rappresentazione finita** per rappresentarlo. Possiamo fare ciò in due modi:

- In modo **generativo** in cui il linguaggio è l'**insieme delle stringhe generate da una struttura finita, la grammatica**.
- In modo **riconoscitivo** in cui il linguaggio è l'**insieme delle stringhe riconosciute da una struttura finita detta automa**.

Chapter 2

Grammatica

Esistono **varie classi di grammatiche**: regolari, libere (dal contesto), dipendenti dal contesto, monotone o generali. Tutte queste alla fine **seguono lo stesso pattern**, ma **si differenziano solo per la caratterizzazione delle produzioni**

Definizione grammatiche libere: *Una grammatica libera da contesto è definita come una quadrupla del tipo* (NT, N, R, S) *dove:*

- NT *è un* ***insieme finito di simboli non terminali****, di solito indicati con lettere maiuscole.*
- T *è un* ***insieme finito di simboli terminali****.*
- $S \in NT$ *è detto* ***simbolo iniziale****.*
- R *è un* ***insieme finito di produzione*** *tutti con la seguente forma:*

$$V \rightarrow w \quad \textit{dove } V \in NT \textit{ e } w \in (T \cup NT)^*$$

2.1 Derivazioni

Esistono vari tipi di derivazione, le più importanti sono la rightmost e la leftmost:

- **Rightmost** in cui ad ogni passo si riscrive il non terminale più a destra.
- **Leftmost** in cui ad ogni passo si riscrive il non terminale più a sinistra.
- **Ne esistono ulteriori**.(DA AGGIUNGERE)

Definizione derivazione immediata: *Data una grammatica libera da contesto* $G = (NT, N, R, S)$ *diciamo che da* $v \in (T \cup NT)^*$ *si* ***deriva immediatamente*** $w \in (T \cup NT)^*$ *e lo denotiamo con* $v \Rightarrow w$*, se:*

$$\frac{v = xAy \quad (A \rightarrow z) \in R \quad w = xzy}{v \Rightarrow w} \quad \text{con } x, y, z \in (T \cup NT)^*$$

Definizione derivazione in n passi: *Diciamo che da* v ***si deriva*** *in n passi* w *e lo denotiamo con* $v \Rightarrow^* w$ *se esiste una sequenza finita di derivazioni immediate:* $v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w$*, cioè:*

$$\frac{}{v \Rightarrow^* v} \quad (\textit{Riflessività}) \qquad \frac{v \Rightarrow^* w \quad w \Rightarrow z}{v \Rightarrow^* z} \quad (\textit{Transività})$$

2.1.1 Alberi di derivazione

Definizione albero di derivazione: *Data una grammatica libera $G = (NT, T, S, R)$ un albero di derivazione (o di parsing) è un albero ordinato in cui:*

- *La radice è etichettata con S (il simbolo iniziale).*
- ***Ogni nodo è etichettato con un simbolo in** $NT \cup \{\epsilon\} \cup T$. Notiamo però che:*
  - *Ogni **nodo interno** è etichettato con un simbolo in NT.*
  - *Se un nodo ha etichetta $A \in NT$, i suoi figli in ordine sono $m_1, ..., m_k$ con etichetta $x_1, ..., x_k \in NT \cup T$ allora $(A \to x_1...x_k) \in R$ (cioè è una produzione della grammatica). Ci cioè sta dicendo che **il legame fra nodo padre e nodi figlio è dato dalle regole della grammatica**.*
  - *Se un nodo ha etichetta ϵ allora è una foglia, è figlio unico e, detto A suo padre, $(A \to \epsilon) \in R$. Se non fosse cosi potrebbero essere ammesse produzioni del tipo $A \to (NT \cup T)\epsilon(NT \cup T)$, ma non ha senso avere ϵ in mezzo.*
  - *Se **ogni nodo foglia è etichettato con** $T \cup \{\epsilon\}$ allora l'albero corrisponde ad una **derivazione completa**.*

Questi alberi forniscono informazioni semantiche come l'ordine delle operazioni. Ognuno di questi **alberi di derivazione riassume tante derivazioni diverse ma equivalenti** che generano lo stesso albero (ad esempio la leftmost e la rightmost).

Teorema appartenenza al linguaggio di una stringa: *Una stringa $w \in T^*$ appartiene a $L(G)$ se e solo se ammette un albero di derivazione completo in cui le foglie, lette da sinistra verso destra, diano la stringa w.*

2.1.2 Alberi sintattici

Da l'albero di derivazione posso poi ricavare un **albero sintattico** che è l'albero di derivazione senza i nodi interni, e quindi composto solo da simboli terminali. Nel caso di una espressione si otterrà che i nodi interni sono gli operatori e le foglie gli operandi.

2.1.3 Ambiguità

Definizione grammatica ambigua: *Una grammatica libera G è ambigua se $\exists w \in L(G)$ che ammette più di un albero di derivazione.*

Definizione linguaggio ambigua: *Un linguaggio L è ambiguo se e solo se tutte le grammatiche G, tali che $L(G) = L$, sono ambigue.*

Esempio 2.1.1 (linguaggio ambiguo). *Consideriamo i linguaggi $L_1 = \{a^n b^n c^m d^m \mid n, m \geq 1\}$ e $L_2 = \{a^n b^m c^m d^n \mid n, m \geq 1\}$ tutti e due liberi e non ambigui, consideriamo inoltre il linguaggio $L = L_1 \cup L_2$. Possiamo dire sicuramente che L è ambiguo visto che per ogni stringa del tipo $w = a^n b^n c^n d^n$ (notiamo che $w \in L$ e $w \in L_1 \cap L_2$) possiamo scrivere w con due derivazioni, quella con la grammatica di L_1 e quella con la grammatica di L_2.*

Una **grammatica che è ambigua è inutilizzabile** perché la semantica di ogni stringa non è unica visto che ci posso essere più alberi semantici associata ad essa con semantiche diverse. E' per questo che si cerca sempre grammatiche non ambigue e, nel caso lo fossero, si cerca di eliminare l'ambiguità. Può succedere che delle grammatiche ambigue non possano essere disambiguate, in questo caso il linguaggio generato da queste è sempre ambiguo.
Notiamo inoltre che **l'ambiguità non dipende dal tipo di derivazione**.

Esempio 2.1.2 (Rimuovere l'ambiguità). *Consideriamo la grammatica:*

$$S \to a \mid b \mid c \mid S + S \mid S \cdot S$$

questa grammatica è ambigua, ciò deriva da due motivi: non c'è nessuna regola di precedenza fra operatori, l'associatività degli operatori. Per risolvere ciò possiamo costruire una nuova grammatica per lo stesso linguaggio:

$$E \to E + T \mid T$$

$$T \to A \cdot T \mid A$$

$$A \to a \mid b \mid c \mid (E)$$

con questa nuova grammatica abbiamo che il $\cdot$ ha precedenza rispetto al $+$ perchè è più interno nell'albero generato. Nel albero semantico finale le parentesi saranno poi rimosse perché sono solo zucchero sintattico.

Da quest'ultimo esempio notiamo quindi che ci sono due tipi di sintassi:

- **La sintassi concreta** che è la grammatica non ambigua che fa uso di zucchero sintattico (le parentesi ad esempio). Questo è quello che viene dato al compilatore.
- **La sintassi astratta** che è una grammatica semplice senza zucchero sintattico ma ambigua, dall'albero di derivazione quando estraiamo l'albero sintattico sarà sotto forma di sintassi astratta.

Chapter 3

Vincoli sintattici contestuali

I vincoli sintattici contestuali sono vincoli sintattici che **non sono esprimibili per mezzo di grammatiche libere** (o BNF) perché non in grado di descrivere vincoli che dipendono dal contesto.

Esempio 3.0.1 (Vincoli sintattici contestuali). *Un paio di esempi di questi vincoli possono essere:*

- *Una variabile che deve essere dichiarata prima di poterla usare.*
- *Compatibilità di tipo in un assegnamento, $x = e$ sia x sia e devono avere lo stesso tipo.*

Questi vincoli, nonostante appartengano alla sintassi, vengono detti di **semantica statica** poiché si segue la visione in cui la sintassi riguarda solo ciò descrivibile con BNF.

Definizione semantica statica: *Per semantica statica indichiamo l'insieme dei controlli che possono essere eseguiti sul testo del programma, senza quindi eseguirlo.*

Quali possono essere **le soluzioni**? troviamo due alternative:

- Usare **grammatiche dipendenti dal contesto**, il problema di ciò è che è poco pratico visto che verificare se $w \in L(G)$ dove G è una grammatica contestuale è esponenziale rispetto a $|w|$.
- Usare **controlli ad hoc**, in particolare questo viene fatto nei compilatori nella fase di **analisi semantica**.

3.1 Semantica dinamica

Definizione semantica dinamica: *Per semantica dinamica intendiamo una rappresentazione formale dell'esecuzione del programma, ciò può mostrare errori "dinamici". Con rappresentazione formale dell'esecuzione si intende un modello matematico che descrive, indipendentemente dall'architettura, il comportamento del programma*

Esempio 3.1.1 (Semantica dinamica). *Supponiamo di voler rappresentare una possibile semantica dinamica dell'espressione:*

$$x := x + 1$$

Un possibile modo per realizzare ciò è tramite il ***modello a grafo****:*

$$\langle x := x + 1, \sigma \rangle \to \sigma \left[\frac{\sigma(x) + 1}{x} \right]$$

ciò indica che valuto il comando $x := x + 1$ utilizzando lo store σ, il risultato di ciò ($\to$) è uno store aggiornato in cui a x associo il valore $\sigma(x) + 1$.

Esempio 3.1.2 (Errori dinamica). *Supponiamo di aver del codice di questo tipo:*

$$\textit{int } A, B;$$

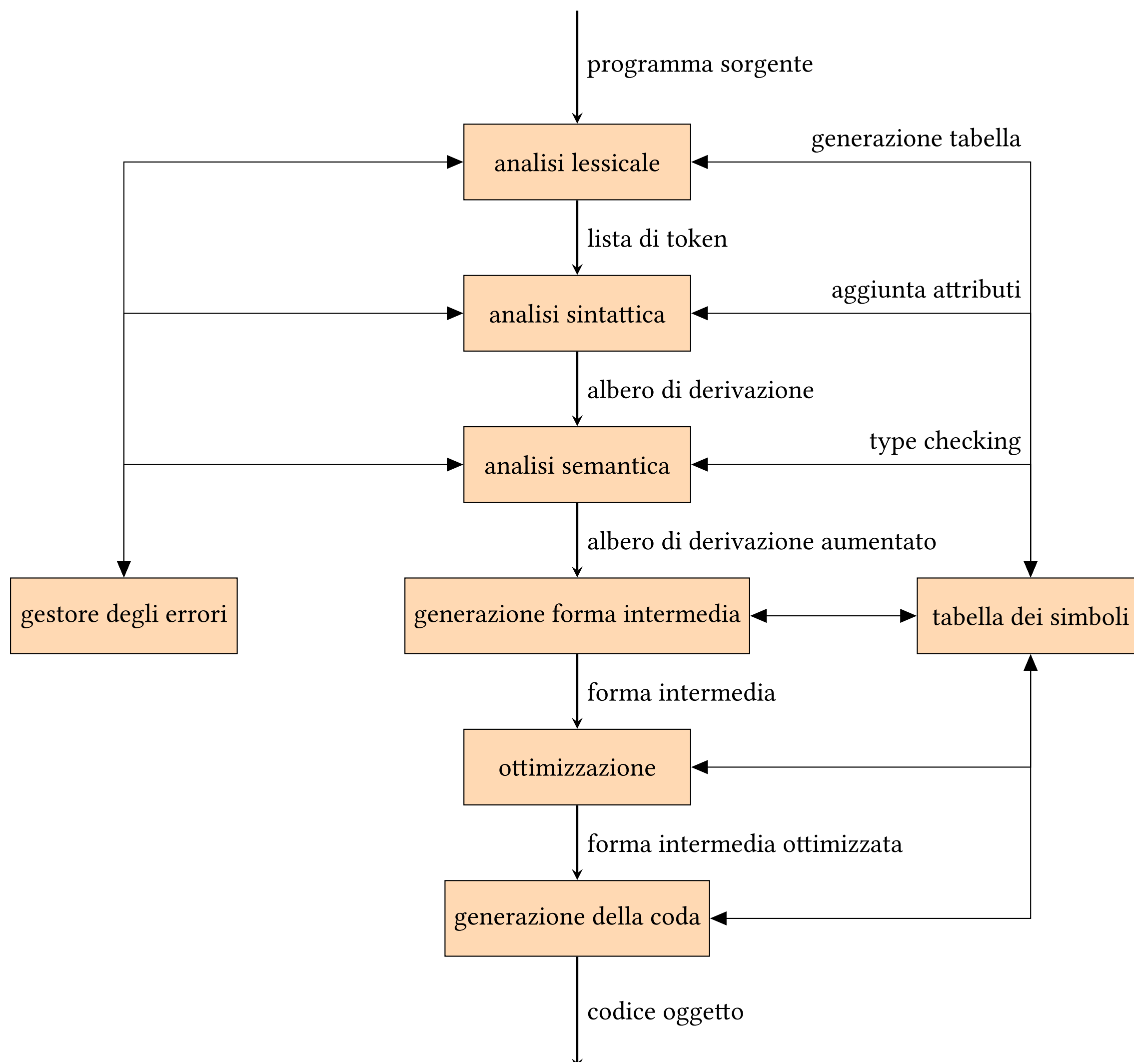
$$\textit{read}(A);$$

$$B := 10/A$$

Non possiamo dire se sono presenti errori fino alla esecuzione, visto che la condizione di errore dipende da un input in fase di esecuzione.

Chapter 4

Struttura di un compilatore



Notiamo quindi come un compilatore non sia composto da una unica fase che "fa tutto" ma sia diviso in tante fasi diverse ognuna che risolve un problema specifico. Nel caso in cui le prime tre fasi rilevino errori non bloccano la compilazione ma generano un opportuno messaggio d'errore, gestito poi dal gestore degli errori.

4.0.1 Analisi lessicale (scanner)

L'analisi lessicale, effettuata dallo scanner, si occupa di **spezzare il programma sorgente in token**, cioè in componenti sintattici primitivi. Questi token possono essere di vario tipo come: identificatori, numeri, operatori, parentesi e parole riservate. Lo scanner si occupa solo di **controllare che il lessico sia ammissibile** (evitare la presenza di simboli strani), e di **riempire parzialmente la tabella dei simboli**.

4.0.2 Analisi sintattica (parser)

L'analisi sintattica, effettuata dal parser, si occupa di **generare l'albero di derivazione del programma** partendo dalla lista di token generata dallo scanner. Si occupa inoltre di **riconoscere eventuali errori sintattici**.

Esempio 4.0.1 (Errore sintattico). *Un esempio di errore sintattico potrebbe essere un'espressione del tipo:*

$$(a(+b)))$$

In quest'espressione troviamo due errori: il numero di parentesi che non è bilanciato e l'operatore $+$ *che non ha un argomento sinistro.*

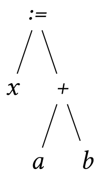
4.0.3 Analisi semantica

L'analisi semantica si occupa di **eseguire controlli di semantica statici** per rilevare eventuali errori (ad esempio verifica se i tipi sono corretti in un assegnamento). Inoltre **arricchisce l'albero di derivazione** con informazioni sui tipi.

4.0.4 Generazione della forma intermedia

Da questa fase in poi il compilatore non si occuperà più di rilevare errori. Questa fase si occupa di **generare codice scritto in un linguaggio intermedio**, facilmente traducibile nel linguaggio macchina di varie macchine e **indipendente dall'architettura**. Segue il principio *easy to produce - easy to translate* ed utilizza operazioni molto semplici, tipicamente delle **three-address code**. Per generare questo codice intermedio viene seguita la struttura dell'albero sintattico.

Esempio 4.0.2 (Generazione codice). *Supponiamo di avere un albero di derivazione ricavato da un espressione del tipo* $x := a + b$ *come il seguente:*



da qua ogni operazione verrà scritta come un'operazione a se stante, generando il seguente codice:

$$temp_1 := a + b$$

$$x := temp_1$$

4.0.5 Ottimizzazione

In questa fase si effettuano delle ottimizzazioni sul codice intermedio per renderlo più efficiente come:

- Rimozione del codice inutile.
- Espansione in linea di chiamate di funzioni.
- Mettere fuori da cicli espressioni che non cambiano.

4.0.6 Generazione del codice

In questa fase viene **generato codice specifico per un architettura**, andando magari ad aggiungere ottimizzazioni possibili su quella architettura (posso ad esempio usare i registri per ottimizzare i tempi).

4.0.7 Tabella dei simboli

Questa tabella memorizza informazioni sui nomi presenti nel programma, questi nomi possono essere associati ad identificatori di variabili, funzioni, etc... .

Chapter 5

Semantica operazionale strutturata

5.1 Cosa serve per dare la semantica

La semantica di un determinato linguaggio con un opportuna sintassi la si fornisce **tramite dei modelli detti sistema di transizioni**.

Definizione sistema di transizione: *Un sistema di transizione è un tripla del tipo* $\langle \Gamma, T, \rightarrow \rangle$ *dove:*

- Γ *è l'**insieme di stati** (o configurazioni). Questo insieme può essere infinito.*
- $T \subseteq \Gamma$ *è l'**insieme degli stati terminali**.*
- $\rightarrow \subseteq \Gamma \times \Gamma$ *è la **relazione di transizione**.*

Definizione computazione: *Una computazione a partire dallo stato* γ_0 *è una **sequenza*** $\gamma_0 \rightarrow \ldots \rightarrow \gamma_n$ *che può essere finita o infinita.*

Definizione chiusura riflessiva e transitiva: *Con* $\rightarrow^*$ *indichiamo la chiusura riflessiva e transitiva di* $\rightarrow$*, ovvero quando:*

$$\frac{}{\gamma \rightarrow^* \gamma}\text{(riflessiva)} \qquad \frac{\gamma \rightarrow^* \gamma' \quad \gamma' \rightarrow \gamma''}{\gamma \rightarrow^* \gamma''}\text{(transitiva)}$$

Quindi quando abbiamo due stati γ e γ' viene indicato con $\gamma \rightarrow \gamma'$ il singolo passo mentre con $\gamma \rightarrow^* \gamma'$ n passi.

5.1.1 Problemi

Con le definizione date fino ad ora incontriamo però dei problemi:

- L'insieme Γ è spesso un **insieme infinito contabile** (perché il numero di stati è proporzionale alle stringhe del linguaggio), e quindi **necessita una rappresentazione finita** implicita attraverso grammatiche e/o BNF.
- L'insieme $\rightarrow \subseteq \Gamma \times \Gamma$ è una relazione costituita spesso da infinite coppie, e quindi anche lei **necessita una rappresentazione finita** implicita . Tale rappresentazione è data da **insieme finito finito di assiomi** che rappresentano la minima relazione che li soddisfa.

5.2 Semantica di un "linguaggio di programmazione": semantica espressioni

Proviamo ad dare un esempio di un sistema di transizione di un linguaggio di programmazione basico. Prima diamo la sintassi del linguaggio

Definizione sintassi linguaggio di programmazione: *La sintassi del nostro linguaggio di programmazione sarà composta da tre insiemi base:*

- *Booleani:* $\{tt, ff\}$.
- *Numeri naturali:* $\{0, 1, 2, ...\}$.
- *Variabili:* $\{a, b, c, ..., z\}$.

Tre insiemi derivati:

- *Espressioni aritmetiche:* $e ::= m \mid v \mid e + e \mid e - e \mid e * e$
- *Espressioni booleane:* $b ::= t \mid e = e \mid b \; or \; b \mid \; \sim b$
- *Comandi:* $c ::= skip \mid v := e \mid c; c \mid while \; b \; do \; c \mid if \; b \; then \; c \; else \; c$

Diamo ora la semantica delle varie espressioni e comandi in modo separato per semplicità:

Definizione Semantica delle espressioni aritmetiche: *Vogliamo ora dare la semantica delle espressioni aritmetiche, cioè il sistema di transizione* $\langle \Gamma_e, T_e, \rightarrow_e \rangle$ *dove avremo che:*

- $\Gamma_e = \{< e, \sigma > \mid e \in Exp, \; \sigma \in Store\}$.
- $T_e = \{< n, \sigma > \mid n \in \mathbb{N}, \; \sigma \in Store\}$. *Notiamo che per come definiamo l'insieme* T_e*, gli stati finali sono solo quelli in cui non abbiamo da valutare ancora un espressione ma abbiamo già un risultato numerico* (n).

Definiti questi due insiemi ci manca da definire la relazione $\rightarrow_e$*, per far ciò andremo ad usare un insieme di assiomi:*

$$(Var)\frac{}{< v, \sigma > \rightarrow_e < \sigma(v), \sigma >}$$

notiamo che in questo caso lo stato successivo alla valutazione è terminale perché $\sigma(v) \in \mathbb{N}$.

$$(Sum_1)\frac{< e_0, \sigma > \rightarrow_e < e_0', \sigma' >}{< e_0 + e_1, \sigma > \rightarrow_e < e_0' + e_1, \sigma' >}$$

In questo passo stiamo valutando parte dell'espressione sinistra e_0.

$$(Sum_2)\frac{< e_1, \sigma > \rightarrow_e < e_1', \sigma' >}{< m + e_1, \sigma > \rightarrow_e < m + e_1', \sigma' >}$$

In questo passo avendo già valutato e_0 *completamente stiamo valutando parte dell'espressione destra* e_1.

$$(Sum_3)\frac{}{< m + m', \sigma > \rightarrow_e < p, \sigma >} \text{con } p = m + m'$$

Ultimo passo di valutazione di un'espressione di somma.

$$(Sub_1)\frac{< e_0, \sigma > \rightarrow_e < e_0', \sigma' >}{< e_0 - e_1, \sigma > \rightarrow_e < e_0' - e_1, \sigma' >}$$

$$(Sub_2)\frac{< e_1, \sigma > \rightarrow_e < e_1', \sigma' >}{< m + -e_1, \sigma > \rightarrow_e < m - e_1', \sigma' >}$$

$$(Sub_3)\frac{}{< m - m', \sigma > \rightarrow_e < p, \sigma >} \text{con } p = m - m' \; ep \geq 0$$

5.2.1 Punti da notare

Notiamo che in tutte queste regole di transizione possiamo dire che **lo store σ non viene mai modificato** durante la valutazione. Per dimostrare ciò possiamo farlo in maniera induttiva:

- Nessuno dei tre assiomi (Var, Sum_3, Sub_3) va a modificarlo (quei 3 sono assiomi perché non hanno premesse).
- Nelle altre regole l'unico punto in cui si potrebbe modificare σ è nella premessa, perché diciamo che σ diventa σ' dopo la valutazione di e_0 o e_1. Però la valutazione di e_0 o e_1 può avvenire solo in due modi:
  - Attraverso un assioma (nel caso in cui e sia del tipo $m + m'$ o $m - m'$), in questo caso dal punto 1 sappiamo che lo store non si modifica.
  - Attraverso una delle altre regole. Di nuovo in questo caso sappiamo che non si modifica perché le altre regole $(Sum_1, Sum_2, Sub_1, Sub_2)$ non modificano direttamente lo store.

Un'altra cosa da notare è che l'assioma Sub_3 è valido solo nel caso in cui $m \geq m'$ perché altrimenti avremo numeri negativi non contemplati, quindi equivarebbe ad uno stato di errore.

5.2.2 Determinatezza relazione di transizione

Teorema determinatezza relazione di transizione: *La relazione di transizione $\rightarrow_e$ è deterministica, ciò vale che:*

$$\gamma \rightarrow_e \gamma' \wedge \gamma \rightarrow_e \gamma'' \implies \gamma' = \gamma'' \qquad \forall \gamma, \gamma', \gamma''$$

Cioè dato uno stato γ esiste un solo stato a cui può arrivare attraverso un unico passo.

Per dimostrare questo teorema useremo la **tecnica dell'induzione strutturale**, cioè se vogliamo dimostrare una proprietà P per ogni $e \in Expr$ con $e ::= m \mid v \mid e+e \mid e-e \mid e*e$ se dimostriamo $P(m), P(v), P(e+e), P(e-e), P(e*e)$ abbiamo dimostrato $P(e)$ per un generico e.

Dimostrazione: *Vogliamo dimostrare la proprietà seguente per ogni $e \in Expr$:*

$$P(e) = \forall \sigma, \gamma', \gamma'' \quad (< e, \sigma > \rightarrow_e \gamma' \wedge < e, \sigma > \rightarrow_e \sigma'') \implies \gamma' = \gamma''$$

Dimostriamola strutturalmente:

- *Caso $e = m \in \mathbb{N}$. In questo caso non dobbiamo dimostrare niente visto che siamo in uno stato terminale non esiste nessun γ' tale che $< e, \sigma > \rightarrow_e \gamma'$.*
- *Caso $e = v \in Var$. Per la regola Var abbiamo solo la transizione $< v, \sigma > \rightarrow_e < \sigma(v), \sigma >$. Visto che il valore di $\sigma(v)$ è univoco e solo la regola Var è applicabile in questo caso vale il teorema.*
- *Caso $e = e_0 + e_1$. Supponiamo che esistano due γ' e γ'' come risultato della transizione, vogliamo dimostrare che $\gamma' = \gamma''$. Visto che per l'espressione $e = e_0 + e_1$ abbiamo tre possibili regole dividiamo la dimostrazione per casi:*
  - *Caso (1), questo è il caso in cui dobbiamo usare Sum_1, cioè in cui $e_0 \notin \mathbb{N}$. Avremo quindi che $< e_0, \sigma > \rightarrow < e_0', \sigma' >$ e $\gamma' =< e_0' + e_1, \sigma' >$. Visto che esiste un secondo stato γ'' deve valere che $< e_0, \sigma > \rightarrow < e_0'', \sigma'' >$ e $\gamma'' =< e_0'' + e_1, \sigma'' >$. Per ipotesi induttiva però sappiamo che $P(e_0)$ deve valere, per cui deve essere che $e_0' = e_0''$ (cosa vuole dire? visto che la valutazione di Sum_1 dipende dalla valutazione di e_0 che sappiamo essere deterministica, allora anche quella di Sum_1 lo deve essere).*
  - *Caso(2) è uguale a il caso (1) solo che applicato per e_1.*
  - *Caso(3), questo è il caso di Sum_3. In questo caso qui l'unica transizione possibile è quella $< e_0 + e_1, \sigma > \rightarrow < p, \sigma >$ con $p = e_0 + e_1$, quindi la tesi vale.*
- *Caso $e = e_0 - e_1$ e $e = e_0 * e_1$ sono uguali.*

□

5.2.3 Funzione di valutazione

Teorema funzione eval di valutazione: *Visto che la relazione $\rightarrow_e$ è deterministica partendo da $< e, \sigma >$ arriveremo sempre su una sola configurazione terminale $< n, \sigma >$. E' quindi possibile definire una **funzione parziale**:*

$$eval : Expr \times store \nrightarrow \mathbb{N}$$

dove:

$$eval(e, \sigma) = \begin{cases} m, & \text{se} < e, \sigma > \rightarrow^* < m, \sigma >. \\ \text{indefinita}, & \text{altrimenti.} \end{cases} \tag{5.1}$$

La funzione è parziale perché per certe espressioni non si raggiunge uno stato terminale (pensa un'espressione con valori negativi). Nota questa funzione è riferita ad una regola di valutazione (vedi dopo) quindi per regole diverse le funzioni potrebbero essere diverse.

Definita la funzione eval si può anche definire l'**equivalenza di espressione**:

$$e \equiv e' \Leftrightarrow eval(e, \sigma) = eval(e', \sigma)$$

5.2.4 Regola di valutazione

Le regole di $Sum_1, Sum_2, Sub_1, Sub_2$ seguono una **regola di valutazione interna sinistra** (IS), in cui prima si valuta l'argomento più a sinistra, cioè e_0, e poi, una volta terminata la valutazione di e_0, si passa a e_1.

Esempio 5.2.1 (Regola interna destra). *Proviamo a dare la valutazione interna destra alle regole Sum_1 e Sum_2 (è uguale per quelle della sottrazione):*

- $$(Sum'_1)\frac{< e_1, \sigma > \rightarrow_e < e'_1, \sigma' >}{< e_0 + e_1, \sigma > \rightarrow_e < e_0 + e'_1, \sigma' >}$$
- $$(Sum'_2)\frac{< e_0, \sigma > \rightarrow_e < e'_0, \sigma' >}{< e_0 + m, \sigma > \rightarrow_e < e'_0 + m, \sigma' >}$$

Si può inoltre dimostrare che $eval_{IS} = eval_{ID}$ cioè **le due regole di valutazione non influiscono sulla valutazione**. Esistono però altre tecniche di valutazione come ES (esterna sinistra), ED, EP (esterna parallela (esterna indica che se posso valutare solo uno dei due argomenti lo faccio) e IP (interna parallela) ma per queste vale che $eval_{ES} \neq eval_{IS}$. Attenzione non è che la loro valutazione è "diversa", semplicemente ES, ED, EP restituiscono valori anche quando IS o ID non lo fanno.

Esempio 5.2.2 (Regola interna parallela). *La regola interna parallela è semplicemente un mix di quella interna destra e sinistra, infatti le sue tre regole per la somma sono Sum_1 (IS), Sum'_1 (ID), Sum_3. Questa regola è però **nondeterministica** ma **confluente**, infatti per espressione $e_0 + e_1$ posso valutare prima e_0 o e_1 (nondeterministica) però il risultato finale non cambia (confluente).*

5.3 Semantica espressioni booleane

Definizione Semantica delle espressioni booleane: *Vogliamo ora dare la semantica delle espressioni booleane, cioè il sistema di transizione $\langle \Gamma_b, T_b, \rightarrow_b \rangle$ dove avremo che:*

- $\Gamma_b = \{< b, \sigma > \mid b \in Bexp, \ \sigma \in Store\}$.
- $T_b = \{< tt, \sigma >, < ff, \sigma > \mid \sigma \in Store\}$. *Gli stati finali T_b sono quindi quelle dell'espressione booleane completamente valutate in tt o ff.*

Definiamo ora gli assiomi che definiscono $\rightarrow_b$:

$$(Eq_1)\frac{\langle e_0, \sigma\rangle \rightarrow_e \langle e_0', \sigma'\rangle}{\langle e_0 = e_1, \sigma\rangle \rightarrow_b \langle e_0' = e_1, \sigma'\rangle}$$

Passo di valutazione IS di una uguaglianza.

$$(Eq_2)\frac{\langle e_1, \sigma\rangle \rightarrow_e \langle e_1', \sigma'\rangle}{\langle m = e_1, \sigma\rangle \rightarrow_b \langle m = e_1', \sigma'\rangle}$$

$$(Eq_3)\frac{}{\langle m = m', \sigma\rangle \rightarrow_b \langle t, \sigma'\rangle}$$

Dove t = tt se $m = n$ senno è ff. Questa è la fine di valutazione di una uguaglianza.

$$(Or_1)\frac{\langle b_0, \sigma\rangle \rightarrow_e \langle b_0', \sigma'\rangle}{\langle b_0 \text{ or } b_1, \sigma\rangle \rightarrow_b \langle b_0' \text{ or } b_1, \sigma'\rangle}$$

$$(Or_2)\frac{}{\langle tt \text{ or } b_1, \sigma\rangle \rightarrow_b \langle tt, \sigma\rangle}$$

La valutazione di Or avviene attraverso una tecnica ES, cioè valutiamo da sinistra e se possiamo valutare solo un argomento lo facciamo, infatti in questo caso facciamo "cortocircuito".

$$(Or_3)\frac{}{\langle ff \text{ or } b_1, \sigma\rangle \rightarrow_b \langle b_1, \sigma\rangle}$$

$$(Neg_1)\frac{\langle b, \sigma\rangle \rightarrow_b \langle b', \sigma'\rangle}{\langle \sim b, \sigma\rangle \rightarrow_b \langle \sim b_1', \sigma\rangle}$$

$$(Neg_1)\frac{}{\langle \sim t, \sigma\rangle \rightarrow_b \langle t', \sigma\rangle}$$

dove t^{prime} è tt se $t = ff$ sennò è ff.

Si potrebbero dimostrare due proprietà:

- Lo **store non viene mai modificato**.
- La **relazione di transizione è deterministica**, per cui si può definire:

$$eval_b(b, \sigma) = \begin{cases} t, & \text{se } < b, \sigma > \rightarrow_b^* < t, \sigma >. \\ \text{indefinita}, & \text{altrimenti.} \end{cases} \tag{5.2}$$

5.3.1 Regole di valutazioni alternative

Anche in questo caso possiamo definire regole di valutazione alternative, come ED o IS

Esempio 5.3.1 (Valutazione ED). *Valutazione ED, cioè valutiamo da destra e facciamo cortocircuito sul valore destro.*

$$(Or_1')\frac{\langle b_1, \sigma\rangle \rightarrow_e \langle b_1', \sigma'\rangle}{\langle b_0 \text{ or } b_1, \sigma\rangle \rightarrow_b \langle b_0 \text{ or } b_1', \sigma'\rangle}$$

$$(Or_2')\frac{}{\langle b_0 \text{ or } tt, \sigma\rangle \rightarrow_b \langle tt, \sigma\rangle}$$

$$(Or_3')\frac{}{\langle b_0 \text{ or } ff, \sigma\rangle \rightarrow_b \langle b_0, \sigma\rangle}$$

5.4 Semantica dei comandi

L'ultima semantica che ci manca da dare è quella dei comandi:

Definizione Semantica dei comandi: *Vogliamo ora dare la semantica dei comandi, cioè il sistema di transizione* $\langle \Gamma_c, T_c, \rightarrow_c \rangle$ *dove avremo che:*

- $\Gamma_c = \{< c, \sigma > \ | \ c \in Com, \ \sigma \in Store\} \cup \{\sigma \mid \sigma \in store\}$. *Agli stati "normali" aggiungiamo anche lo stato solo* σ *perché è ciò che un comando completato restituisce*
- $T_c = \{\sigma \mid \sigma \in Store\}$. *Gli stati finali* T_c *sono i comandi che vengono valutati completamente in uno store.*

Definiamo ora gli assiomi che definiscono $\rightarrow_c$*:*

$$(Skip)\frac{}{\langle skip, \sigma \rangle \rightarrow_c \langle \sigma \rangle}$$

Il comando skip è un comando che non fa nulla e non modifica nulla.

$$(Ass)\frac{\langle e, \sigma \rangle \rightarrow_e^* \langle m, \sigma \rangle}{\langle v := e, \sigma \rangle \rightarrow_c \sigma[\frac{m}{v}]}$$

L'assegnamento valuta prima l'espressione e poi va a ***modificare il valore dello store*** σ *in corrispondenza di* v*.*

$$(Seq_1)\frac{\langle c_0, \sigma \rangle \rightarrow_c \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow_c \langle c_0'; c_1, \sigma' \rangle}$$

Valutazione di due comandi in sequenza, parto valutando il primo.

$$(Seq_2)\frac{\langle c_0, \sigma \rangle \rightarrow_c \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow_c \langle c_1, \sigma' \rangle}$$

Finita la valutazione di uno dei due comandi. Nota lo store può veramente cambiare adesso!

$$(If_1)\frac{\langle b, \sigma \rangle \rightarrow_b^* \langle tt, \sigma \rangle}{\langle if \ b \ then \ c_0 \ else \ c_1, \sigma \rangle \rightarrow_c \langle c_0, \sigma \rangle}$$

$$(If_2)\frac{\langle b, \sigma \rangle \rightarrow_b^* \langle ff, \sigma \rangle}{\langle if \ b \ then \ c_0 \ else \ c_1, \sigma \rangle \rightarrow_c \langle c_1, \sigma \rangle}$$

Se la valutazione della condizione viene valutata a tt allora eseguo il "then" sennò "else".

$$(While_1)\frac{\langle b, \sigma \rangle \rightarrow_b^* \langle tt, \sigma \rangle}{\langle while \ b \ do \ c, \sigma \rangle \rightarrow_c \langle c; \ while \ b \ do \ c, \sigma \rangle}$$

Se l'espressione nella guardia del while è vera allora eseguo il comando dentro il while e poi di nuovo il while.

$$(While_2)\frac{\langle b, \sigma \rangle \rightarrow_b^* \langle ff, \sigma \rangle}{\langle while \ b \ do \ c, \sigma \rangle \rightarrow_c \sigma}$$

Nel caso in la guardia sia falsa esco dal ciclo.

Come le altre relazioni anche $\rightarrow_c$ è una **relazione deterministica**. Anche in questo caso possiamo definire una funzione parziale di valutazione:

$$exec : Com \times store \nrightarrow store$$

$$exec(c, \sigma) = \begin{cases} \sigma', & \text{se } < c, \sigma > \rightarrow_c^* \sigma'. \\ \text{indefinita}, & \text{altrimenti.} \end{cases} \tag{5.3}$$

5.4.1 Errori non ben categorizzati

La funzione può essere indefinita tipo se il ciclo while è un ciclo infinito o se espressioni sono negative, come possiamo vedere nel seguente esempio:

Esempio 5.4.1 (Funzione indefinita, divergenza e deadlock). *Ci possono essere due casi di funzione che non è definita:*

- $c_1 =$ *while tt do skip, in questo caso siamo in presenza di* ***divergenza***.
- $c_2 = x := (3 - 5)$ *in questo caso siamo in presenza di deadlock(?).*

in ogni caso vale che $exec(c_1, \sigma) = exec(c_2, \sigma)$.

Ciò può creare problemi visto che errori di "tipo" diversi vengono valutati ugualmente, per far ciò dobbiamo aggiungere altri elementi alla nostra sintassi e semantica.

5.5 Errori dinamici

Per raffinare la nostra sintassi andremo ad aggiungere un terminale che indica l'errore err, ottenendo cioè che:

$$\Gamma' = \Gamma \cup \{err\} \qquad T' = T \cup \{err\} \quad \text{nota per tutti } T_e, T_b, T_c, \text{ e per tutti } \Gamma_e, \Gamma_b, \Gamma_c$$

Le funzioni diventeranno di questo tipo ora:

$$eval : Expr \times store \to \mathbb{N} \cup err$$

$$eval_b : Bexp \times store \to \{tt, ff\} \cup err$$

$$exec : Com \times store \not\hookrightarrow store \cup err$$

Notiamo che $eval$ e $eval_b$ diventano **funzioni totali** perché in ogni caso restituiscono un valore (nel caso in cui prima la funzione era indefinita adesso restituisce err). La funzione $exec$ non è invece totale perché il while infinito è ancora caso di "indefinitezza".

5.5.1 Semantiche aggiuntive

Per completare gli errori dinamici dobbiamo aggiungere regole di transizione anche per essi, in particolare per la generazione dell'errore da parte della sottrazione, e per la sua propagazione:

- **Generazione errore**:

$$(Sub_4)\frac{}{< m - m', \sigma > \to_e err} \text{Se } m' > m$$

- **Propagazione dell'errore**:

$$(Sum_4)\frac{\langle e_0, \sigma \rangle \to_e err}{< e_0 + e_1, \sigma > \to_e err}$$

$$(Sum_5)\frac{\langle e_1, \sigma \rangle \to_e err}{< m + e_1, \sigma > \to_e err}$$

$$(Or_4)\frac{\langle b, \sigma \rangle \to_b err}{< b_0 \text{ or } b_1, \sigma > \to_e err}$$

$$(Neg_3)\frac{\langle b, \sigma \rangle \to_b err}{<\sim b, \sigma > \to_e err}$$

$$(Ass_2)\frac{\langle e, \sigma \rangle \to_e^* err}{< v := e, \sigma > \to_e err}$$

$$(Seq_3)\frac{\langle c_0, \sigma \rangle \to_c err}{< c_0; c_1, \sigma > \to_e err}$$

$$(If_3)\frac{\langle b, \sigma \rangle \to_b^* err}{< \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma > \to_e err}$$

$$(While_3)\frac{\langle b, \sigma \rangle \to_b^* err}{< \text{while } b \text{ do } c, \sigma > \to_e err}$$

Tutto sto casino solo per aggiungere un tipo di errore dinamico Sub_4, se ne volessimo altri dovremmo fare altre regole.

5.6 Non determinismo e parallelismo

Esistono altri due costrutti per comandi non deterministici e paralleli, per aggiungerli dobbiamo estendere la BNF dei comandi nel seguente modo:

$$c ::= skip \mid ... \mid c \text{ or } \mid c \text{ par } c$$

e andare aggiungere le seguenti regole di transizione:

- Per il **non determinismo**:

$$(Nd_1)\frac{}{< c_0 \text{ or } c_1, \sigma > \rightarrow_c \langle c_0, \sigma \rangle}$$

$$(Nd_1)\frac{}{< c_0 \text{ or } c_1, \sigma > \rightarrow_c \langle c_1, \sigma \rangle}$$

  Questo costrutto serve per aggiungere appunto non determinismo, perché non sai quale dei due viene eseguito.

- Per il **parallelismo**:

$$(Par_1)\frac{\langle c_0, \sigma \rangle \rightarrow_c \langle c_0', \sigma' \rangle}{< c_0 \text{ par } c_1, \sigma > \rightarrow_c \langle c_0' \text{ par } c_1, \sigma' \rangle}$$

$$(Par_2)\frac{\langle c_0, \sigma \rangle \rightarrow_c \sigma'}{< c_0 \text{ par } c_1, \sigma > \rightarrow_c \langle c_1, \sigma' \rangle}$$

$$(Par_3)\frac{\langle c_1, \sigma \rangle \rightarrow_c \langle c_1', \sigma' \rangle}{< c_0 \text{ par } c_1, \sigma > \rightarrow_c \langle c_0 \text{ par } c_1', \sigma' \rangle}$$

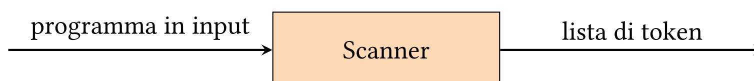
$$(Par_4)\frac{\langle c_1, \sigma \rangle \rightarrow_c \sigma'}{< c_0 \text{ par } c_1, \sigma > \rightarrow_c \langle c_0, \sigma' \rangle}$$

Chapter 6

Linguaggi regolari e espressioni regolari

6.1 Analisi lessicale

L'analisi lessicale, effettuata dallo scanner in un compilatore, è la sezione che si occupa di **riconoscere nella stringa token**, cioè sequenze di simboli che corrispondono a determinate categorie sintattiche (come identificatori, parole riservate o numeri...).
Lo scanner prenderà in input un programma e come output avrà una lista di token.



6.1.1 Che cosa sono i token

Un token è una coppia (nome,valore). Il token è identificato da 4 elementi diversi:

- Il **nome** è un **simbolo che identifica la classe del token** ("ide", "const" etc...).
- Il **valore** è la sequenza di simboli del testo in ingresso.
- Il **pattern** è la **descrizione della forma dei valori** di una determinata classe di token.
- Il **lessema** è un'**instanza di un pattern**.

Esempio 6.1.1 (token semplice). *Un possibile token potrebbe essere quello di un identificatore:*

$$\langle ide, x_1 \rangle$$

Dove abbiamo che "ide" è il nome del token (cioè il "tipo"), x_1 è il valore e lessema di un particolare pattern.

Esempio 6.1.2 (Da codice a token). *Supponiamo di dare in input al nostro scanner il seguente codice:*

if$(x == 0)$ *printf("zero");*

L'output del nostro scanner sarà una lista di token. Lo scanner prendere il testo in input e lo andrà a dividere in tutti i token che trova ottenendo:

<IF>, <(>, <ide,x>, <OPREL, ==>, <CONST-NUM, 0>, <)>

<ide, printf>, <(>, <¿<CONST-STRING, zero> <)>

In realtà non è proprio questo che fa lo scanner, infatti per gli identificatore invece di metterci il valore associa un indirizzo nella tabella dei simboli:

<ide, puntatore$_1$ >

6.2 Espressioni regolari

Definizione Espressioni regolari: *Fissato un alfabeto $A = \{a_1, a_2, ..., a_n\}$, definiamo le espressioni regolari su A con la seguente BNF:*

$$r ::= \emptyset \mid \epsilon \mid a \mid r \cdot r \mid r|r \mid r^*$$

Questa **sintassi è ambigua** e per disambiguarla si potrebbero usare le parentesi ma, per semplicità, si assume che:

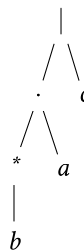
- La concatenazione, la disgiunzione e la ripetizione **associno a sinistra**.
- La **precedente degli operatori** sia la seguente: $* > \cdot > |$

Le espressioni regolare sono utili per **definire il pattern**, ma non sono in grado di riconoscere se un lessema rispetti o no questo pattern, per questo in seguito definiamo strutture chiamate autonomi che avranno proprio questo ruolo.

Esempio 6.2.1 (Espressione regolare e albero sintattico). *Dalle assunzioni di prima possiamo dire che la seguente espressione regolare:*

$$b^*a|c$$

corrisponda al seguente albero:



6.2.1 Linguaggio denotato da un espressione regolare

Definizione Funzione da regex a alfabeto: *Dato l'alfabeto A definiamo la funzione:*

$$\mathcal{L} : Exp - Reg \rightarrow \wp(A^*)$$

notiamo che $\wp(A^*)$ rappresenta l'insieme delle parti di A^* che però rappresenta tutte le possibili stringhe generabili con l'alfabeto A, quindi $\wp(A^*)$ ***rappresenta tutti i possibili linguaggi su alfabeto*** *A. Definiamo inoltre i valori della funzione $\mathcal{L}$ come:*

$$\begin{aligned}
\mathcal{L}[\emptyset] &= \emptyset \\
\mathcal{L}[\epsilon] &= \{\epsilon\} \\
\mathcal{L}[a] &= \{a\} \\
\mathcal{L}[r_1 \cdot r_2] &= \mathcal{L}[r_1] \cdot \mathcal{L}[r_2] \quad \text{dove } \cdot \text{ sta per la concatenazione di linguaggi} \\
\mathcal{L}[r_1|r_2] &= \mathcal{L}[r_1] \cup \mathcal{L}[r_2] \\
\mathcal{L}[r^*] &= (\mathcal{L}[r])^* \quad \text{dove } ^* \text{ sta per la stella di kleene di un linguaggo}
\end{aligned}$$

La funzione $\mathcal{L}$ quindi è una funzione che **associa un linguaggio ad un'espressione regolare**.

6.3 Linguaggio regolare

Definizione Linguaggio regolare: *Un linguaggio $L \subseteq A^*$ è detto regolare se e solo se esiste un'espressione regolare r tale che:*

$$L = \mathcal{L}[r]$$

Possiamo notare come **ogni linguaggio finito è anche regolare**, infatti basta prendere un'espressione regolare che copre tutte le stringhe dell'alfabeto.

Esempio 6.3.1 (Linguaggio regolare finito). *Sia dato $L = \{a, bc\}$, possiamo dire che L è regolare perché con $r = a|bc$ abbiamo che $L = \mathcal{L}[r]$, infatti:*

$$\mathcal{L}[a|bc] = \mathcal{L}[a] \cup \mathcal{L}[bc] = \{a\} \cup \mathcal{L}[b] \cdot \mathcal{L}[c] = \{a\} \cup \{b\} \cdot \{c\} = L$$

Esempio 6.3.2 (Linguaggio regolare infinito). *Sia dato $r = a^*b$ il linguaggio regolare ad essa associata è un linguaggio infinito ed è:*

$$L = \mathcal{L}[r] = \{a\}^* \cdot \{b\} = \bigcup_{n \geq 0} \{a\}^n \cdot \{b\} =$$
$$= \{\epsilon, a, aa...\} \cdot \{b\} = \{a^n b \mid n \geq 0\}$$

6.3.1 Esempi di espressioni regolari

Esempio 6.3.3 (Alcune espressioni regolari). *Sia dato l'alfabeto $A = \{0, 1\}$. Definiamo una serie di espressioni regolari:*

- $r = 0^*10^*$ *denota il linguaggio regolare* $L = \{w \in A^* \mid$ *w contiene un solo 1*$\} = \{0^n10^m \mid n, m \geq 0\}$.
- $r = (0|1)^*1(0|1)^*$ *denota il linguaggio* $L = \{w \in A^* \mid$ *w contiene almeno un 1*$\}$. *Le stringhe contengono almeno un 1 perché l'unico simbolo dell'alfabeto che c'è sicuramente è sempre quel 1 centrale, che può essere preceduto e seguito da stringhe qualsiasi di 0 e 1.*
- $r = 1^*(011^*)^*$ *denota il linguaggio* $L = \{w \in A^* \mid$ *ogni occorenza di 0 è seguita subito da almeno un 1*$\}$. *Notiamo che la stella di kleene di* $(011^*)^*$ *è applicata a tutta la stringa* (011^*), *mentre quella del 1 dentro è applicata solo a lui.*
- $r = ((0|1)(0|1))^*$ *denota* $L = \{w \in A^* \mid$ *w è di lunghezza pari*$\}$. *Questo è vero perché l'espressione regolare* $(0|1)(0|1)$ *rappresenta una qualunque stringa di lunghezza due, infatti:*

  $$\mathcal{L}[(0|1)(0|1)] = \{00, 01, 10, 11\}$$

  Quindi applichiamo la stella di kleene ad una qualunque stringa di lunghezza 2, ottenedo una stringa di lunghezza $2 * n$ *e quindi pari.*

6.3.2 Operatori ausiliari delle espressioni regolari

Possiamo aggiungere anche altri operatori per le nostre espressioni regolari come:

- **Ripetizione positiva** indicata con r^+. Notiamo che $r^+ \equiv rr^*$.
- **Possibilità** indicata con $r?$, che può valere r o ϵ. Notiamo che $r? \equiv (r|\epsilon)$.
- **Elenco** indicato con $[a_1, ..., a_n]$ con $a_1, ..., a_n \in A$ che equivale ad $a_1|...|a_n$. Molto comodo se gli a_i sono ordinati, in questo caso possiamo indicare $[a_1 - a_n]$.

Esempio 6.3.4 (numeri decimali senza segno). *Vogliamo ora costruire una espressione regolare che denota il linguaggio dei numeri decimali senza segno. Dato un alfabeto $A = \{0, 1, ..., 9, .\}$ possiamo scrivere:*

$$r = [0-9]^+(.[0-9]^+)?$$

che indica che avremo sempre almeno un cifra prima della virgola, ed essa può essere seguita da un punto seguito da un altro numero con almeno una cifra.

6.3.3 Definizioni regolari

Definizione definizioni regolari: *Una definizione regolare su alfabeto A è costituita da una lista di definizioni:*

$$\alpha_1 := r_1$$
$$\vdots$$
$$\alpha_n := r_n$$

Dove α_i sono simboli non "usati" e ogni r_i è un espressione regolare su alfabeto esteso $A \cup \{\alpha_1, ..., \alpha_n\}$.

Le definizioni regolari sono come la creazione di variabili (o di define di c) che rendono scrivere espressioni regolari lunghe più veloce e più leggibile.

Esempio 6.3.5 (numeri decimali senza segno con definizioni regolari). *Se diamo queste definizioni regolari:*

$$segno := -|+$$
$$cifre := cifra^+$$
$$cifra := [0-9]$$

Possiamo poi definire un numero decimale con segno come:

$$r = segno\ cifre\ (.cifre)?$$

6.3.4 Equivalenza fra espressioni regolari

Definizione Equivalenza: *Due espressioni regolari r e s si dicono equivalenti e lo denotiamo con $r \cong s$ se e solo se $\mathcal{L}[r] = \mathcal{L}[s]$, cioè se denotano lo stesso linguaggio.*

Teorema : *Dalle definizioni date si possono dedurre una serie di congruenze fra espressioni regolari:*

$$r|s \cong s|r \qquad \text{(commutatività)}$$
$$r|(s|t) \cong (r|s)|t \quad r \cdot (st) \cong (rs) \cdot t \qquad \text{(associatività)}$$
$$r|r \cong r \quad (r^*)^* \cong r^* \qquad \text{(idempotenza)}$$
$$r \cdot (s|t) \cong rs|rt \quad (r|s)t \cong rt|st \qquad \text{(distributività)}$$
$$\epsilon \cdot r \cong r \cong r \cdot \epsilon \qquad \text{(elemento neutro)}$$
$$\emptyset^* \cong \epsilon \quad r \cdot \emptyset \cong \emptyset \quad r|\emptyset \cong r \quad (\epsilon|r)^* \cong r^* \quad (r^*s^*)^* \cong (r|s)^* \qquad \text{(generali)}$$

Chapter 7

Automi a stati finiti

7.1 Introduzione

Gli automi a stati finiti sono particolare strutture dati che ci permettono di **determinare se una stringa rispetta o no un pattern**. Essi sono composti da una **memoria finita** usata per definire gli stati dell'automa. Essi prendono in input una stringa e restituiscono un bit, indicante se la stringa in input è stata riconosciuta.

7.1.1 Funzionamento degli automi

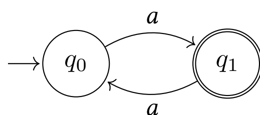
Per funzionare gli automi a stati finiti eseguono il seguente ciclo:

- Leggi un carattere in input e sposta la testina (che legge l'input) avanti.
- In base all'input e allo stato spostati su un altro stato
- Se ho finito di leggere la stringa e sono in uno stato finale allora la stringa è riconosciuta, altrimenti no. Inoltre alla lettura di un carattere mi blocco perché non c'è nessun altro stato in cui saltare allora di nuovo non è riconosciuta.

7.1.2 Diagramma di transizione

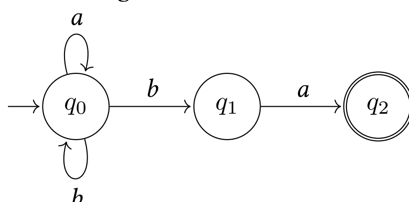
La struttura degli stati di un automa è ben rappresentato tramite un diagramma chiamato appunto diagramma di transizione.

Esempio 7.1.1 (Diagramma di transizione). *Facciamo ora un esempio di un diagramma di transizione del automa finito che riconosce il linguaggio regolare* $a(aa)^* = \{a^{2n+1} \mid n \geq 0\}$.



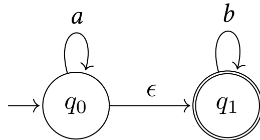
Visto che solo lo stato q_1 *è finale per poter essere riconosciuta una stringa alla fine della sua "lettura" dobbiamo essere in* q_1*. Detto ciò possiamo dire che riconosce il linguaggio* $a(aa)^*$ *perché deve per forza riconoscere una* a *iniziale (cosi finisce in* q_1*) e nel caso nel riconosca un'altra (tornando in* q_0*) ne deve riconoscere un'altra ancora.*

Esempio 7.1.2 (Automa più complesso). *Costruiamo ora un'automa per l'espressione regolare* $(a|b)^*ba$*. L'automa seguente è corretto:*

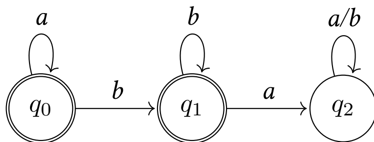


Quest'automa funziona, perché? Il riconoscimento della porzione $(a|b)^*$ è affidato al primo nodo, infatti finché riconosce delle a o delle b resta su q_0 (nel caso riconosca una b può anche andare su q_1, ma supponiamo che faccia look-ahead (vedi tanto dopo) e sa cosa gli aspetta). Il riconoscimento della porzione ba è affidata agli ultimi due nodi. Questo automa è detto ***non deterministico*** *poiché ci sono due mosse b da q_0.*

Esempio 7.1.3 (Automa con mossa epsilon). *Nell'automa che stiamo andando a vedere andiamo ad aggiungere una transizione ϵ.*



Notiamo come questo automa riconosca a^*b^*, poiché possiamo mettere n a, poi m b. Nota anche questo è ***non deterministico****, poiché "quando vuole lui" può "far finta di leggere" un carattere ϵ e quindi ha sempre 2 mosse disponibili nonostante il carattere in input sia 1. Vedremo poi in seguito che le mosse epsilon sono uno dei motivi per cui un automa è non deterministico. Questo automa si può* ***trasformare in deterministico*** *rimodellando un po, ma mantenendo il linguaggio riconosciuto uguale:*



I primi due nodi sono abbastanza ovvi, il nodo strano invece è l'ultimo da cui non si può uscire e non è finale. Questo nodo infatti è un "nodo di errore" che viene raggiunto con stringhe del tipo $a^*b^*a(a|b)^*$

7.2 Automi finiti non deterministici NFA

Definizione NFA: *Un automa finito non deterministico è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove:*

- Σ *è un* ***alfabeto finito*** *di simboli in input.*
- Q *è un* ***insieme finito di stati****.*
- $q_0 \in Q$ *è lo* ***stato iniziale****.*
- $F \subseteq Q$ *è l'****insieme degli stati finali****.*
- δ *è la* ***funzione di transizione*** *del tipo:*

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \wp(Q)$$

Rappresenta la funzione che dato uno stato e un simbolo (quello che si sta leggendo in input) ti dice quali sono gli stati possibili a cui puoi andare. Infatti vale che:

$$\delta(q, \sigma) = Q' \subseteq Q$$

7.2.1 Linguaggio riconosciuto

Definizione informale: *Un NFA $N = (\Sigma, Q, \delta, q_0, F)$ accetta $w = a_1...a_n$ se e solo se nel diagramma di transizione esiste un cammino da q_0 ad uno stato in F nel quale la stringa ottenuta concatenando le etichette degli archi percorsi sia uguale a w.*

Questa era una definizione informale che però fa capire quando una stringa è accettata da un NFA. Per dare definizione formale di linguaggio riconosciuto dobbiamo prima introdurre tre nuovi elementi:

- **Descrizione istantanea** che è una coppia (q, w) dove q è lo stato attuale e w la stringa in input ancora da leggere.
- **Mossa** una mossa è descritta da due elementi, una premessa e una regola (?), e viene scritta nel seguente modo:
$$\frac{q' \in \delta(q, \sigma)}{(q, \sigma w) \vdash_N (q', w)} \quad \sigma \in \Sigma \cup \{\epsilon\}, \quad w \in \Sigma^*$$
Questa scrittura mi dice: se q' fa parte degli stati raggiungibili da q con mossa σ allora dalla descrizione istantanea $(q, \sigma w)$ passo a (q', w) consumando quindi σ. Vale anche in questo caso la **chiusura riflessiva e transitiva**:
$$\frac{}{(q, w) \vdash_N^* (q, w)} \quad \frac{(q, w) \vdash_N^* (q', w') \quad (q', w') \vdash_N (q'', w'')}{(q, \sigma w) \vdash_N^* (q'', w'')}$$
Nel caso di più mosse lo chiamiamo **cammino**.

Definizione riconoscimento di stringa: *Dato un NFA N diciamo che una stringa w è riconosciuta dall'automa se e solo se:*

$$\exists q \in F \mid (q_0, w) \vdash_N^* (q, \epsilon)$$

Cioè se consumando tutta la stringa in input arrivo ad uno stato finale allora la stringa è riconosciuta.

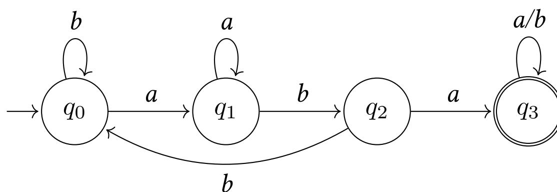
Definizione linguaggio riconosciuto: *Dato un NFA N definiamo con $L[N]$ il linguaggio riconosciuto (o accettato) dal NFA N. Questo linguaggio è:*

$$L[N] = \{w \in \Sigma^* \mid \exists q \in F.\ (q_0, w) \vdash_N^* (q, \epsilon)\}$$

Il linguaggio riconosciuto dal NFA N è quindi l'insieme di tutte le stringhe riconosciute (abbastanza ovvio).

Esempio 7.2.1 (Da linguaggio a NFA). *Vogliamo trovare l'NFA che riconosca il linguaggio:*

$$L = \{w \in (a|b)^* \mid w \text{ contiene la sottostringa } aba\}$$



Funziona questo automa? Possiamo leggere quante "b" vogliamo all'inizio e anche quante "a" vogliamo ma, in quest'ultimo caso, ci spostiamo sullo stato q_1 perché? perché visto che la "a" è il primo carattere della sottostringa che dobbiamo avere, quando leggiamo una "a" possiamo pensare di stare iniziando a leggere la stringa "aba". Se continuiamo a leggere "a" restiamo dove siamo (perché tanto non influisce avere a^*aba o aba e basta), alla prima "b" ci spostiamo di nuovo perché abbiamo appena letto una stringa "ab" e quindi ci dobbiamo mettere in attesa del carattere "a". Una volta in q_2 se leggiamo un'altra "b" vuol dire che abbiamo letto abb non ci va bene e torniamo indietro, se invece leggiamo "a" vuol dire che abbiamo, almeno, letto "aba" e quindi siamo a posto. Dopo possiamo leggere quello che vogliamo.

7.2.2 Problemi degli NFA

Gli NFA nonostante siano relativamente comodi da costruire poiché non hanno tante restrizioni sono molto **inefficienti**, infatti, per verificare che una stringa w venga accettata, bisogna seguire cammini sull'automa che però, essendo non deterministico, ammette tante strade diverse e quindi necessita di backtracking.

7.3 Automi finiti deterministici

Definizione DFA: *Un'automa finito deterministico (DFA) è una quintupla* $(\Sigma, Q, \delta, q_0, F)$*, dove* Σ, Q, q_0 *e* F *sono uguali agli NFA mentre l'unica cosa che cambia è la funzione di transizione che diventa del tipo:*

$$\delta : Q \times \Sigma \rightarrow Q$$

e quindi di conseguenza:

$$\delta(q, \sigma) = q'$$

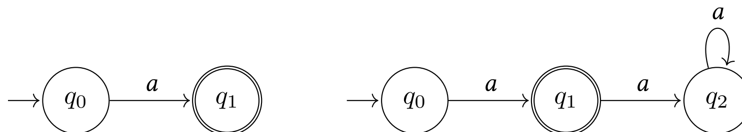
Cioè la differenza fra gli NFA e i DFA è la funzione di transizione per due ragioni:

- *Non esistono più le mosse epsilon. Infatti avevamo visto che causavano non determinismo.*
- *Per ogni stato e per ogni carattere esiste (**sempre!**) un solo stato d'arrivo. Nei NFA potevamo essere più di uno o zero.*

Si possono quindi vedere i DFA come degli NFA in cui valgono le seguenti proprietà:

$$\forall q \in Q \quad \delta(q, \epsilon) = \emptyset$$
$$\forall \sigma \in \Sigma, \forall q \in Q \; \exists q' \in Q. \quad \delta(q, \sigma) = \{q'\}$$

Esempio 7.3.1 (DFA o NFA). *Vogliamo dire se i seguenti automi sono DFA o NFA, come fare? (L'alfabeto è* $\Sigma = \{a\}$*)*



Il primo automa è un DFA? No perché nonostante non ci siano mosse epsilon e non ci siano "doppie scelte" (cioè un $\delta(q, \sigma)$ *con più di un valore) il nodo* q_1 *non ha una mossa* a *e quindi non è un DFA.*
Il secondo invece lo è, visto che ogni nodo ha una e una sola mossa per ogni carattere e non ci sono mosse epsilon.

Teorema Espressiva DFA: *Per ogni NFA* N *è possibile costruire un DFA ad esso equivalente. Ciò implica che **NFA e DFA hanno lo stesso potere espressivo**.*

Non dimostriamo questo teorema ma diamo però un algoritmo in grado di ottenere un DFA equivalente da un NFA qualsiasi, prima di ciò però alcune definizioni.

7.3.1 Epsilon-closure

Prima di scrivere l'algoritmo dobbiamo definire un insieme chiamato ϵ-**closure**.

Definizione ϵ-closure: *Sia* q *uno stato di un NFA, definiamo la* ϵ*-closure di* q *come l'**insieme degli stati raggiungibili con sole mosse*** ϵ*. In alternativa si può definire come il minimo insieme che rispetta le seguenti regole:*

$$\frac{}{\{q\} \subseteq \epsilon\text{-closure}(q)} \qquad \frac{p \in \epsilon\text{-closure}(q)}{\delta(p, \epsilon) \subseteq \epsilon\text{-closure}(q)}$$

La prima regola è ovvia, il nodo stesso è raggiungibile con solo mosse ϵ *(non facendone). La seconda regola invece ci dice che se un nodo* p *è dentro la* ϵ*-closure allora tutti i nodi raggiungibili da* q *con mosse* ϵ *sono dentro la* ϵ*-closure .*

Nel caso abbiamo un insieme P di nodi allarghiamo la definizione di ϵ-closure a quella ovvia di:

$$\epsilon\text{-closure}(P) = \bigcup_{p \in P} \epsilon\text{-closure}(p)$$

Per il **calcolo della ϵ-closure** si può usare un algoritmo banale come il seguente:

Algorithm 1: Calcolo della ϵ-closure (P)

```
1  T = P;
2  ϵ-closure(P) = P;
3  while: T ≠ ∅ do {
4     "Scegli un r ∈ T e rimuovilo da T";
5     for each s ∈ δ(r, ϵ) do:
6        if s ∉ ϵ-closure(P) {
7           add s to ϵ-closure (P);
8           add s to T;
9        }
10 }
```

Nota che questo algoritmo è come una **visita di un grafo** in cui andiamo a visitare tutti i nodi che sono raggiungibili con una ϵ-closure .

7.3.2 Utilizzo dell'epsilon-closure per definire linguaggi riconosciuti

In realtà questa parte non fa parte dei DFA ma degli NFA, comunque **usando l'epsilon closure si può definire in un'altra maniera il linguaggio riconosciuto da un NFA**. Sia data la seguente funzione:

$$\hat{\delta} : Q \times \Sigma^* \to \wp(Q)$$

$$\hat{\delta}(q, \epsilon) = \epsilon\text{-closure}(q)$$

$$\hat{\delta}(q, xa) = \epsilon\text{-closure}(P) \quad \text{dove } P = \left\{p \in Q \mid \exists r \in \hat{\delta}(q, x) \; e \; p \in \delta(r, a)\right\}$$

Notiamo come la funzione $\hat{\delta}(q, w)$ rappresenta "dove si può arrivare svuotando completamente la stringa w partendo da q". Possiamo allora dire che:

$$w \in L[N] \text{ sse } \exists p \in F \text{ tale che } p \in \hat{\delta}(q_o, w)$$

Una stringa w quindi apparterrà al linguaggio riconosciuto se uno degli stati a cui si arriva svuotando completamente w è finale.

7.3.3 Funzione mossa

Definizione mossa: *Definiamo inoltre una nuova funzione mossa come estensione della funzione di transizione δ di un NFA come:*

$$mossa : \wp(Q) \times \Sigma \to \wp(Q)$$

$$mossa(P, a) = \bigcup_{p \in P} \delta(p, a)$$

Cioè l'insieme delle mosse "a" da un insieme di nodi P è l'unione di tutte le mosse "a" di ogni nodo dell'insieme.

7.3.4 Costruzione per sottoinsiemi

Qua di seguito l'algoritmo per la costruzione di sottoinsiemi, che serve per **passare da un NFA a un DFA equivalente**.

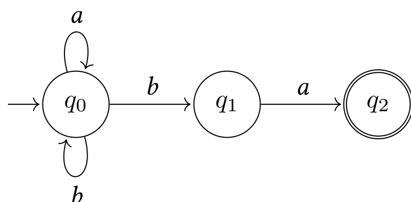
Algorithm 2: Costruzioni per sottoinsiemi

1 Dato un NFA $N = (\Sigma, Q, \delta, q_0, F)$;
2 $S = \epsilon$-closure(q_0);
3 $T = \{S\}$;
4 Finché c'è un $P \in T$ non marcato {
5 Marca P;
6 **for each** $a \in \Sigma$ {
7 $R = \epsilon$-closure(mossa(P, a));
8 **if** $R \notin T$ {
9 add R to T;
10 }
11 definisci $\Delta(P, a) = R$;
12 }
13 }

Il DFA equivalente sarà $M_N = (\Sigma, T, \Delta, \epsilon\text{-closure}(q_o), F)$. Due cose importanti:

- L'alfabeto non varia mentre gli stati, lo stato iniziale e gli stati finali si.
- Abbiamo una nuova funzione di transizione Δ definita come $\Delta(A, b) = \epsilon\text{-closure}(mossa(A, b))$ e che rispetta le limitazioni dei DFA: non ci sono mosse epsilon e per ogni carattere dell'alfabeto esiste una ed una sola mossa in qualunque stato.

Esempio 7.3.2 (Semplice). *Consideriamo l'NFA seguente (relativo all'espressione regolare $(a|b)^*ab$):*



Vogliamo ora trovare un DFA equivalente ad esso, applichiamo l'algoritmo di costruzioni per sottoinsiemi:

- *Per prima cosa calcoliamo lo stato iniziale $S = \epsilon$-closure(q_0), notiamo che $S = q_0$. Creiamo T e entriamo dentro il blocco marcando S.*
- *Per ogni simbolo adesso dobbiamo calcolare l'ϵ-closure$(mossa(P, a))$. Visto che l'alfabeto è $\{a, b\}$ dobbiamo farlo due volte:*
  - *Per il carattere "a" calcoliamo con $P = S = \{q_0\}$:*

$$R = \epsilon\text{-closure}(mossa(P, a)) = \epsilon\text{-closure}(\bigcup_{p \in \{q_0\}} \delta(p, a))$$
$$= \epsilon\text{-closure}(\delta(q_0, a)) = \epsilon\text{-closure}(\{q_0\}) = \{q_0\}$$

    Definiamo quindi $\Delta(S, a) = \{q_0\}$. Visto che $\{q_0\} \in T$ non lo andiamo a ri-aggiungere.
  - *Per il carattere "b" calcoiamo:*

$$R = \epsilon\text{-closure}(mossa(P, b)) = \epsilon\text{-closure}(\bigcup_{p \in \{q_0\}} \delta(p, b))$$
$$= \epsilon\text{-closure}(\delta(q_0, b)) = \epsilon\text{-closure}(\{q_0, q_1\}) = \{q_0, q_1\}$$

    Visto che $\{q_0, q_1\} \notin T$ lo andiamo ad aggiungere.
- *Ripetiamo con $P = \{q_0, q_1\}$ andando a calcolare la stessa cosa:*

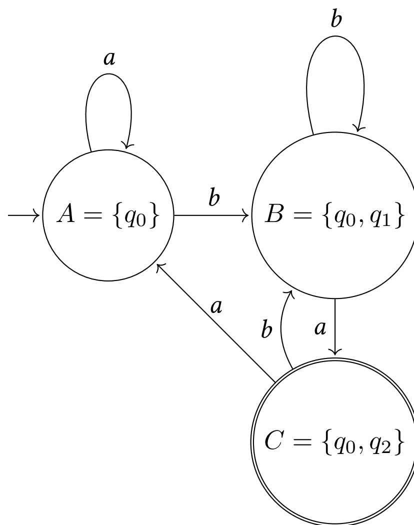
   - *Per il carattere "a":*

$$R = \epsilon\text{-}closure(mossa(P,a)) = \epsilon\text{-}closure(\bigcup_{p \in \{q_0,q_1\}} \delta(p,a))$$
$$= \epsilon\text{-}closure(\delta(q_0,a) \cup \delta(q_1,a)) = \epsilon\text{-}closure(\{q_0\} \cup \{q_2\}) = \{q_0,q_2\}$$

   Quindi $\Delta(\{q_0,q_1\},a) = \{q_0,q_2\}$. *Visto che* $\{q_0,q_2\} \notin T$ *lo aggiungiamo.*

- *Per il carattere "b", salto i calcoli, viene* $R = \{q_0,q_1\}$. *Quindi* $\Delta(\{q_0,q_1\},b) = \{q_0,q_1\}$ *e non lo riaggiungiamo.*
- *Ripetiamo per* $P = \{q_0,q_2\}$ *(salto i calcoli)*

Alla fine otteniamo il DFA $N' = (\Sigma, \{\{q_0\},\{q_0,q_1\},\{q_0,q_2\}\}, \Delta, \{q_0\}, \{q_0,q_2\})$, *che in forma di diagramma di transizione è:*



7.4 Equivalenza NFA e DFA

Teorema : *Sia* $N = (\Sigma, Q, \delta, q_0, F)$ *un NFA e sia* M_N *l'automa ottenuto per costruzione per sottoinsiemi. Allora* M_N *è un DFA e si ha che:*

$$L[N] = L[M_N]$$

Ciò implica anche che ***la classe di linguaggi riconosciuta da NFA e DFA è la stessa****.*

7.5 Da espressione regolare a NFA equivalente

Teorema : *Data un'espressione regolare* s *possiamo costruire un NFA* $N[s]$ *tale che:*

$$L[s] = L[N[s]]$$

Cioè un NFA che riconosce il linguaggio denotato dall'espressione regolare. Ciò vuole inoltre dire che gli ***NFA riconoscono tutti i linguaggio regolari*** *(ricorda, linguaggio regolare = denotato da un'espressione regolare).*

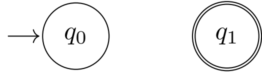
Dimostrazione: *Faremo una dimostrazione d'induzione strutturale sulla struttura dell'espressione regolare* s*. Nel far ciò, per semplificarci, cercheremo sempre un NFA che mantenga le seguenti invarianti (nota di NFA*

ce ne possono essere tanti tali che $L[s] = L[N[s]]$, *noi ne troviamo uno):*

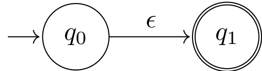
- *Lo stato iniziale non ha archi entranti.*
- *Ci sia solo uno stato finale senza archi uscenti.*

Andiamo ora ad esaminare caso per caso sulla struttura di s*:*

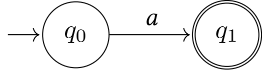
- *Caso* $s = \emptyset$. *In questo caso possiamo avere un NFA con due stati, uno iniziale e uno finale, che non sono collegati:*



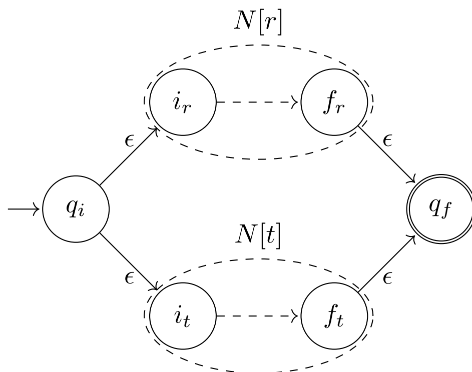
- *Caso* $s = \epsilon$. *In questo caso la regex riconosce solo la stringa "*ϵ*", quindi anche l'NFA dovrà riconoscere solo essa:*



- *Caso* $s = a$. *Come prima dobbiamo riconoscere solo la stringa "a", e quindi avremo che il nostro NFA è:*

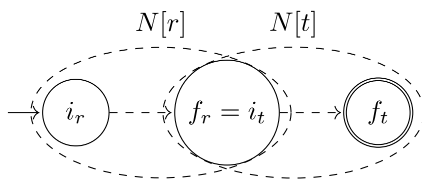


- *Caso* $s = r|t$. *Qua, procedendo per induzione strutturale, supponiamo di avere già due NFA per le due regex* r *e* t *chiamati* $N[r]$ *e* $N[t]$. *Per trovare l'NFA corrispondente ad* s *possiamo fare cosi:*



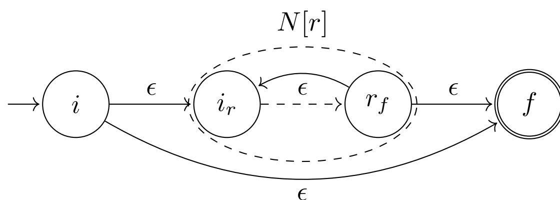
Notiamo che è corretto visto che riusciamo a riconoscere stringhe delle due regex semplicemente usando la mossa ϵ *che ci porta nel "sotto automa" corretto.*

- *Caso* $s = r \cdot t$. *Come prima per ipotesi induttiva abbiamo già gli automi* $N[r]$ *e* $N[t]$. *In questo caso andranno semplicemente concatenanti uno dopo l'altro.*

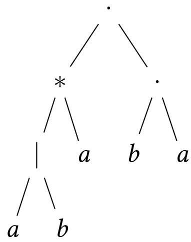


La correttezza deriva dal fatto che una stringa è riconosciuta da $N[s]$ *se la prima parte è riconosciuta da* $N[r]$ *e la seconda da* $N[t]$.

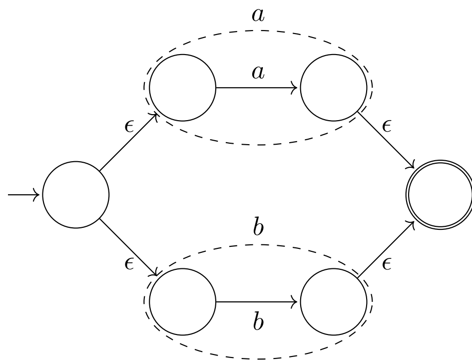
- *Caso* $s = r^*$. *Per ipotesi induttiva abbiamo l'NFA* $N[r]$ *e dobbiamo costruire* $N[s]$. *L'idea è che dobbiamo rendere* $N[r]$ ***ripetibile e saltabile*** *(per il caso* r^0*):*



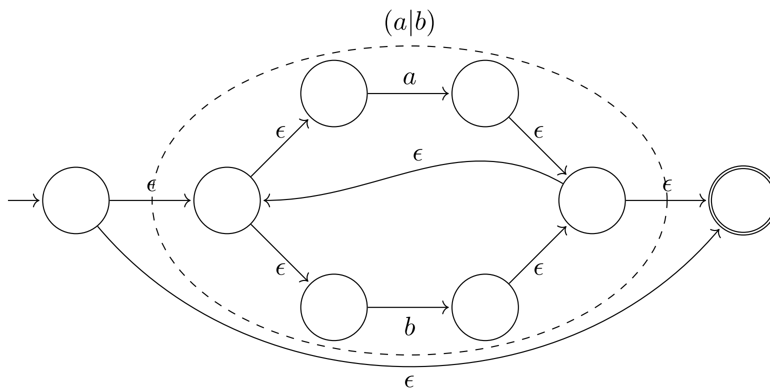
Esempio 7.5.1 (). *Vogliamo trovare un NFA equivalente alla regex* $s = (a|b)^*ba$*, per far ciò costruiamo l'albero sintattico dell'espressione:*



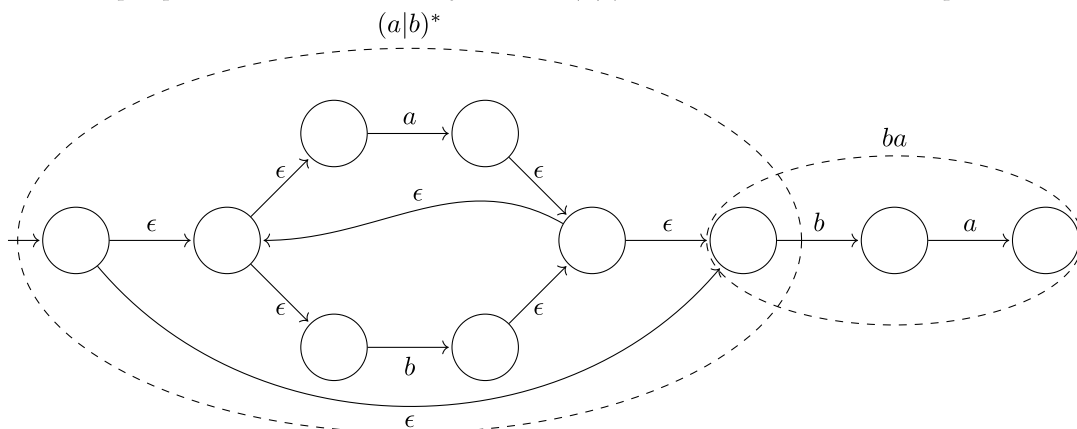
Per costruire l'NFA equivalente procediamo dall'alto verso il basso dell'albero sintattico. Costruiamo prima l'NFA associato all'espressione regolare $a|b$*. Per far ciò possiamo guardare la dimostrazione di prima trovando:*



Costruiamo poi quello di $(a|b)^*$ *così:*



Costruiamo poi quello della concatenazione finale, cioè $(a|b)^*ba$ *(nota l'automa* ba *lo salto perché ovvio).*



Nel caso poi volessimo un DFA associato a questo NFA si usa la tecnica della costruzione per insiemi.

Chapter 8

Grammatiche regolari

Definizione grammatica regolare: *Una grammatica libera è regolare se e solo se ogni produzione è della forma:*

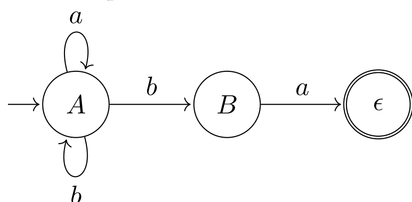
$$V \to aW$$
$$V \to a$$

dove $V, W \in NT$ e $a \in T$. Per il simbolo iniziale, e solo per lui, è ammessa anche la produzione $S \to \epsilon$ (certe volte useremo però una definizione che permette produzioni epsilon anche per simboli diversi da S).

Esempio 8.0.1 (Grammatica regolare, nfa e regex). *La seguente grammatica:*

$$A \to aA \mid bB \mid bA$$
$$B \to a$$

è una grammatica regolare. Notiamo inoltre che $L(G) = (a|b)^*ba$, infatti finché si usano le transizioni $A \to aA|bA$ possiamo scrivere quante "a" o "b" vogliamo, cioè $(a|b)^*$, poi quando scegliamo la transizione $A \to bB$ scriviamo "ba". Se volessimo associare un NFA a questa grammatica potremmo procedere nel seguente modo: per ogni non terminale creiamo un nodo e per ogni produzione creiamo una transizione. Facendo ciò otteniamo qualcosa come:



8.1 Da grammatica regolare a NFA equivalente

Teorema da grammatica regolare a NFA equivalente: *Data una grammatica regolare G si può costruire un NFA N_G equivalente*

Le dimostrazioni di questi teoremi (questo e quello dopo) non sono vere e proprie dimostrazioni ma servono per capire come fare gli esercizi

Dimostrazione: *Non è una vera dimostrazione è semplicemente illustrato* ***come costruire l'NFA equivalente****. Sia $G = (NT, T, R, S)$ una grammatica regolare, allora l'NFA $N_G = (T, Q, \delta, S, \{\epsilon\})$ (dove T è*

l'alfabeto, Q l'insieme di stati, δ la funzione di transizione, S lo stato iniziale e $\{\epsilon\}$ lo stato finale) è definito come segue:

- *$Q = NT \cup \{\epsilon\}$, cioè gli stati dell'NFA sono i non terminali e lo stato ϵ finale.*
- *Lo stato finale è lo stato ϵ mentre lo stato iniziale è dato dal simbolo (non terminale) iniziale S della grammatica.*
- *La funzione di transizione è definita come segue:*
  - *$Z \in \delta(V, a)$ se $V \to aZ \in R$. Cioè esiste la transizione da nodo V a nodo Z con carattere a se esiste una produzione grammaticale dove V si espande in aZ.*
  - *$\epsilon \in \delta(V, a)$ se $V \to a \in R$. Cioè esiste la transizione da nodo V a nodo finale ϵ se V ha una produzione che non contiene non terminali, cioè termina lì.*
  - *$\epsilon \in \delta(V, \epsilon)$ se $S \to \epsilon \in R$.*

Si può poi dimostrare che:

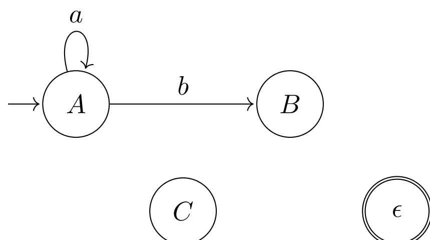
$$S \Rightarrow_G^* w \iff (S, w) \vdash^* (\epsilon, \epsilon)$$

Cioè che la grammatica in n passi produce la stringa w se e solo se l'NFA N_G in n passi finisce nello stato finale ϵ con la stringa in input vuota.

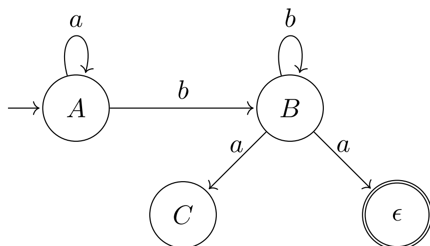
Esempio 8.1.1 (Da grammatica regolare a NFA). *Consideriamo la seguente grammatica regolare:*

$$A \to aA \mid bB$$
$$B \to bB \mid aC \mid a$$
$$C \to aA \mid bB$$

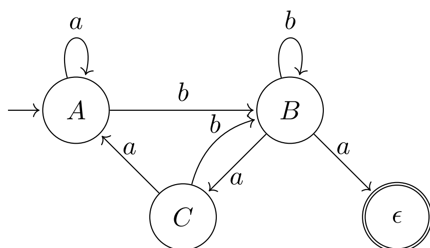
con la tecnica data nella dimostrazione possiamo trovare l'NFA equivalente. Partiamo dal non terminale A:



Passiamo ora alle transizioni di B. Notiamo che B ha una transizione "finale" cioè che non contiene non terminali ($B \to a$), per implementarla faremo una transizione a verso un nodo finale ottenendo:



Passiamo poi alle transizioni di C:



Questo NFA è equivalente alla grammatica regolare data inizialmente.

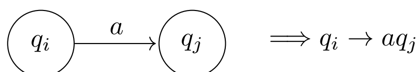
8.2 Da DFA a grammatiche regolari

Teorema : *Dato un DFA M possiamo definire una grammatica regolare G_M tale che:*

$$L[M] = L(G_M)$$

Dimostrazione: *La dimostrazione è un po il contrario di quella di prima visto che dobbiamo partire dall'automa e generare la grammatica. Sia $M = (\Sigma, Q, \delta, q_0, F)$ il DFA, allora la grammatica $G_M(Q, \Sigma, R, q_0)$ definita nel seguente modo:*

- *Come non terminali ha gli stati del DFA.*
- *Per terminali ha l'alfabeto del DFA.*
- *Come simbolo iniziale ha lo stato iniziale del DFA.*
- *Le produzioni invece sono le seguenti:*
  - *Per ogni $\delta(q_i, a) = q_j$ abbiamo che $q_i \to aq_j \in R$, nel caso in cui $q_j \in F$ avremo anche una transizione $q_i \to a \in R$.*

    Nel caso in cui q_j sia finale aggiungiamo anche la transizione $q_i \to a$ perché la transizione da q_i a q_j può essere "l'ultima".
  - *Se $q_o \in F$ allora $q_0 \to \epsilon \in R$.*

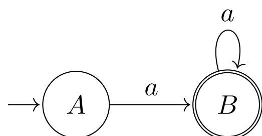
*Viene spesso usata una **versione alternativa** che usa semplicemente due regolare per creare la grammatica:*

- *Per ogni $\delta(q_i, a) = q_j$ abbiamo che $q_i \to aq_j \in R$*
- *Se $q \in F$ allora $q \to \epsilon \in R$.*

Si può poi dimostrare che:

$$w \in L[M] \iff w \in L(G_M)$$

Esempio 8.2.1 (Da DFA a grammatica regolare). *Consideriamo il seguente DFA:*



Usando la tecnica data nella dimostrazione otteniamo una grammatica regolare $G_M = (\{A, B\}, \{a\}, R, A)$ dove R è dato da:

- *Per la transizione $\delta(A, a) = B$ aggiungiamo la produzione $A \to aB$ in R. Inoltre visto che B è stato finale dobbiamo aggiungere anche la transizione $A \to a$.*
- *Per la transizione $\delta(B, a) = B$ aggiungiamo la produzione $B \to aB$ in R. Visto che B è finale aggiungiamo anche $B \to a$.*

Otteniamo quindi che le nostre produzioni sono:

$$A \to aB \mid a$$
$$B \to aB \mid a$$

8.3 Grammatiche regolari ed espressioni regolari

Teorema : *Il linguaggio definito da una grammatica regolare G è un linguaggio regolare, cioè è possibile costruire un'espressione regolare S_G tale che:*

$$L(G) = \mathcal{L}[S_G]$$

La dimostrazione di questo teorema non va fatta, ma il prof ha fatto vedere un idea di come dovrebbe essere fatta. Da questa idea si capisce un po come vanno fatti gli esercizi in cui data una grammatica va trovata l'espressione regolare "equivalente" (o anche dagli esercizi stessi si capisce).

Esempio 8.3.1 (Da grammatica regolare a regex). *Sia data la seguente grammatica:*

$$A \to aB \mid \epsilon$$
$$B \to bA \mid \epsilon$$

vogliamo trovare una espressione regolare S_G tale che $L[G] = \mathcal{S}_G$. Per far ciò facciamo come se le produzione della grammatica fossero delle equazioni in un sistema e quindi, come nei sistemi, partiamo ricavando una variabile (in questo caso partiamo da B):

$$B \approx bA \mid \epsilon$$

In questo caso non conoscendo il "valore" di A riscriviamo semplicemente la produzione. Adesso cerchiamo il "valore" di A sapendo già quello di B:

$$A \approx aB \mid \epsilon$$
$$A \approx a(bA|\epsilon) \mid \epsilon \qquad \text{applicando una sorta di proprietà distributiva otteniamo:}$$
$$A \approx abA \mid a \mid \epsilon$$

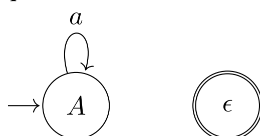
Notiamo che nella prima produzione è presente A (il non terminale) ed è come se avessimo un ciclo, quindi una sorta di stella di kleene, inoltre le ultime due produzioni sono finali, quindi nella regex finale saranno concatenate, in alternativa l'una all'altra, alla fine:

$$A \approx (ab)^*(a|\epsilon)$$

Esempio 8.3.2 (Linguaggio vuoto). *Una grammatica regolare del tipo:*

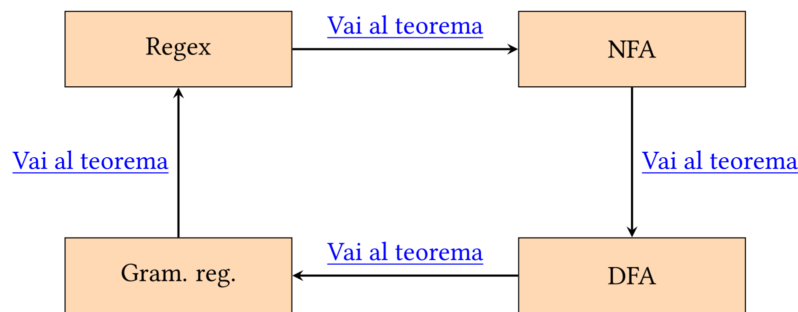
$$A \to aA$$

non è equivalente all'espressione regolare $(a)^*$ poiché il non terminale A non può essere mai eliminato completamente, e quindi non esiste nessuna stringa definita da quella grammatica. Ciò implica che la regex equivalente è $S_G = \emptyset$. Possiamo anche vedere ciò con l'NFA equivalente costruito tramite l'algoritmo dato un po sopra, ottenendo:



Non essendoci cammini fra il nodo iniziale e uno finale riconosce anche esso il linguaggio vuoto.

8.4 Riassunto equivalenza NFA, regex, DFA, grammatiche regolari



Tutto ciò che abbiamo detto implica che tutti questi 4 formalismi diversi sono in realtà equivalenti, cioè **tutti generano/riconoscono la stessa classe di linguaggi, i linguaggi regolari**.

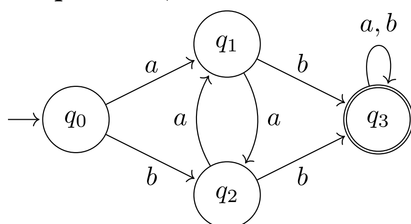
8.4.1 Costruire uno scanner

Detta questa equivalenza, per costruire uno scanner si parte da un'espressione regolare che viene trasformata in NFA che a sua volta viene trasformato in un DFA. Il DFA finale sarà poi ridotto di dimensioni cercando il DFA minimo.

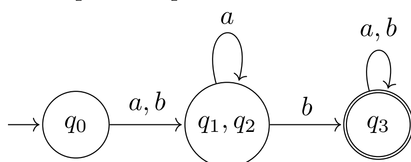
Chapter 9

Minimizzazione DFA

Esempio 9.0.1 (Che cosa è la minimizzazione). *Supponiamo di partire con il seguente DFA:*



Questo DFA è equivalente all'espressione regolare $(a|b)a^*b(a|b)^*$. *Indicheremo con* $L[N, q_0] = (a|b)a^*b(a|b)^*$ *il fatto che usando il nodo* q_0 *come nodo iniziale il nostro DFA è equivalente a quell'espressione. Con* $L[N, q_1]$ *indicheremo quindi il linguaggio riconosciuto dal DFA partendo da* q_1.
Possiamo dire che $L[N, q_0] = L[N, q_1]$*? no perché la transizione da* q_0 *al resto del DFA ci garantisce che le nostre stringhe abbiano il prefisso* $(a|b)$. *Possiamo però dire che* $L[N, q_1] = L[N, q_2] = a^*b(a|b)^*$, *vedremo poi che questo implica che* q_1 *e* q_2 *sono nodi "equivalenti" e che possono essere fusi:*



Questo DFA è equivalente a quello sopra ma si può dimostrare essere minimo, vedremo bene poi come, ma l'idea è quella di verificare che $L[N', q_0] \neq L[N', (q_1, q_2)] \neq L[N', q_3]$ *implicando che non ci sono nodi equivalenti e quindi raggruppabili.*

In questo capitolo vedremo appunto come andare a minimizzare un DFA qualsiasi, però prima di ciò va data la seguente notazione:

Definizione : *Dato un DFA* $N = (\Sigma, Q, \delta, q_0, F)$ *è definita la seguente funzione:*

$$\hat{\delta} : Q \times \Sigma^* \to Q$$
$$\hat{\delta}(q, \epsilon) = q$$
$$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$$

e quindi vale che:

$$w \in L[n] \iff \hat{\delta}(q_0, w) \in F$$

A differenza della funzione $\hat{\delta}$ *degli NFA che restituiva un insieme di stati, la funzione* $\hat{\delta}$ *per gli DFA, essendo essi deterministici, restituirà sempre un solo stato. Questa funzione* ***rappresenta lo stato raggiunto consumando tutta la stringa***.

9.1 Equivalenza (o indistinguibilità)

Definizione Equivalenza fra stati: *Due stati q_1 e q_2 di un DFA N **sono equivalenti** se:*

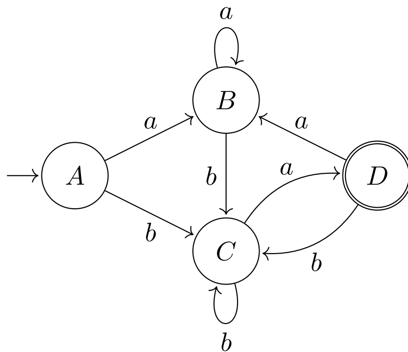
$$\forall x \in \Sigma^*. \quad \hat{\delta}(q_1, x) \in F \iff \hat{\delta}(q_2, x) \in F$$

Cioè se, ripresa la notazione dell'esercizio sopra, $L[N, q_1] = L[N, q_2]$ (cioè il linguaggio riconosciuto da N usando q_1 come stato iniziale è uguale a quello riconosciuto da N usando q_2 come stato iniziale).
*In modo simile due stati q_1 e q_2 **non sono equivalenti** se:*

$$\forall x \in \Sigma^*. \quad \hat{\delta}(q_1, x) \in F \wedge \hat{\delta}(q_2, x) \notin F$$
$$o$$
$$\hat{\delta}(q_1, x) \notin F \wedge \hat{\delta}(q_2, x) \in F$$

L'idea che si usa per dire se due stati sono equivalenti è vedere se rispettano il requisito con stringhe $x \in \Sigma^*$ sempre più lunghe, partendo dalla stringa più corta (ϵ). In questo esempio useremo appunto questa tecnica.

Esempio 9.1.1 (). *Considero il seguente DFA:*

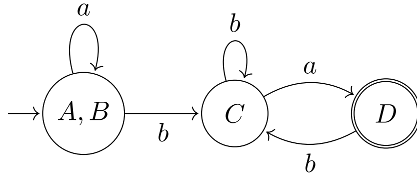


Partendo dalla stringa più corta cerchiamo di andare a distinguere gli stati del DFA:

- *Caso "ϵ". La stringa ϵ distingue tutti gli stati finali dai non finali, infatti visto che con ϵ "non ci si muove" se parto da uno stato finale finisco in uno finale, se parto in uno stato non finale finisco in un non finale. Scrivo le coppie di stati distinti:*
$$(A, D), (B, D), (C, D)$$
- *Caso lunghezza 1. In questo caso ci sono due sotto casi possibili, la stringa a e b:*
  - *Caso "a". La stringa "a" distingue sicuramente C e B perché da C con "a" finisco in uno stato finale, mentre da B no. Stesso ragionamento per A e C, otteniamo quindi:*
  $$(B, C), (C, A)$$
  - *Caso "b". "b" non distingue nulla infatti sia da A, sia da B sia da C resto sempre in uno stato non finale.*
- *Caso lunghezza 2. Qua ci sono 4 casi distinti, "aa", "ab", "ba", "bb" ne faccio solo due di questi perché si vedrà che non distinguono nessuno loro.*
  - *Caso "aa". Da A con "aa" finisco in B, da B con "aa" finisco in B (quindi A e B sono non sono distinti), da C con "aa" finisco in B. Quindi con "aa" non distinguo nulla.*
  - *Caso "ab". Da A con "ab" finisco in C, da B con "ab" finisco in C, da C con "ab" finisco in C. Nessuno è stato distinto.*

  Vedremo poi dopo che quando in un passo nessuno è stato distinto si può terminare, quindi terminiamo qua.

Dalle nostre coppie possiamo dire che solo la coppia (A, B) non è mai stata distinta, quindi loro due sono equivalenti e li possiamo fondere insieme ottenendo:



9.2 Relazione di equivalenza

Definizione : *Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ definiamo una famiglia di relazioni:*

$$\sim_i \subseteq Q \times Q$$

(cioè dove ogni relazione è un insieme di coppie di stati) e la definiamo nel seguente modo:

$$\sim_0 = F \times F \cup (Q \setminus F) \times (Q \setminus F)$$
$$q_1 \sim_{i+1} q_2 \iff \forall a \in \Sigma \quad \delta(q_1, a) \sim_i \delta(q_2, a)$$

Notiamo che il secondo punto è la stessa cosa che dire questo:

$$q_1 \sim_{i+1} q_2 \iff \forall x \in \Sigma^* \text{ con } |x| \leq i+1, \;\; \hat{\delta}(q_1, x) \in F \text{ sse } \hat{\delta}(q2, x) \in F$$

La **relazione di equivalenza è quindi una approssimazione dell'equivalenza fra stati** cioè $\sim_i$ indica gli stati che sono equivalenti fra loro con stringhe lunghe massimo i. Idealmente quindi due stati sono equivalenti se:

$$\lim_{i \to \infty} q_1 \sim_i q_2$$

Ragionando invece sulla costruzione della relazioni notiamo questo:

- La relazione $\sim_0$ è definita come tutte le coppie di stati finali più tutte le coppie di stati non finali, questo perché $\sim_0$ rappresenta gli stati che sono equivalenti con stringhe di lunghezza zero (cioè solo ϵ), e come avevamo visto nell'esempio sopra erano solo quelle coppie.
- La relazione $\sim_{i+1}$ è definita come le coppie di stati che facendo un qualsiasi (quindi per ogni carattere dell'alfabeto) passo di uno arrivano ad due stati che sono equivalenti $\sim_i$. Nota diventa i invece che $i+1$ perché 1 carattere dell'input è stato consumato col primo salto.

9.2.1 Osservazioni

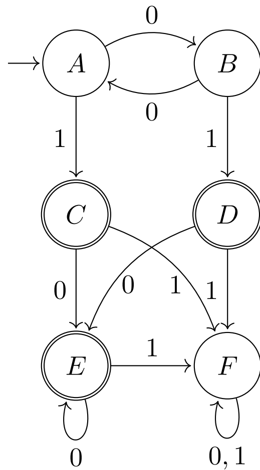
Oltre alle considerazioni sopra se ne possono fare altre:

- Ogni coppia identica (q, q) fa parte di $\sim_0$, cioè uno stato è sempre equivalente a se stesso.
- Valgono le proprietà di **simmetria, riflessività e transitività**. La transitività vuol dire che se $q_1 \sim_i q_2$ e $q_2 \sim_i q_3$ allora $q_1 \sim_i q_3$.
- E' sempre vera la relazione:

  $$\sim_{i+1} \subseteq \sim_i$$

  Questo perché la relazione $\sim_{i+1}$ è più restrittiva di quella $\sim_i$, infatti se $q_1 \sim_{i+1} q_2$ vuol dire che q_1 e q_2 sono equivalenti per stringhe di lunghezza massimo $i+1$, ma questo include anche quelle i, quindi deve rispettare i requisiti di $\sim_i$ più quello per le stringhe di lunghezza $i+1$.
- Se esiste un k tale per cui $\sim_k = \sim_{k+1}$ allora vale anche che $\forall j > k \quad \sim_j = \sim_k$, cioè dopo un certo k la relazione non cambia più. Inoltre questo k sarà sempre minore di $| \sim_0 |$ perché il caso pessimo è quello in cui elimino una coppia ad ogni passaggio, e il numero di coppie iniziali è dato appunto da $| \sim_0 |$.

Esempio 9.2.1 (). *Proviamo a minimizzare il DFA seguente:*



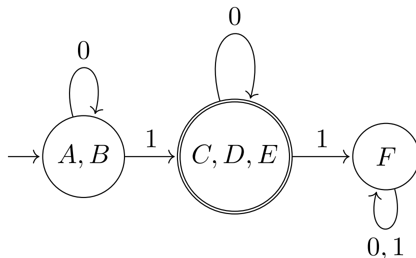
Costruiamo ora l'insieme $\sim_0$ come per definizione:

$$\sim_0 = \{(A,A),(A,B),(A,F),(B,B),(B,A),(B,F),(F,F),(F,A),(F,B)\}$$
$$\cup$$
$$\{(C,C),(C,D),(C,E),(D,D),(D,C),(D,E),(E,E),(E,C),(E,D)\}$$

Procediamo ora per costruire l'insieme $\sim_0$, per velocizzarci controlliamo solo le coppie in $\sim_0$ e ovviamente non quelle identiche (tipo (A,A)) (con controllare intendo fare quello fatto nell'esercizio di sopra, per adesso solo con stringhe di lunghezza 1 "0" e "1"). Otteniamo:

$$\sim_1 = \{(A,A),(A,B),(B,B),(B,A),(F,F),(C,C),(C,D),(C,E),(D,D),(D,C),(D,E),(E,E),(E,C),(E,D)\}$$

Non ho voglia di scriverlo ma $\sim_2 = \sim_1$ quindi possiamo terminare qua e ridisegnare il DFA. Gli stati equivalenti sono quindi quelli che non compaiono nelle coppie quindi (A,B), (F), (C,D,E). Otteniamo quindi:



Qua è semplice anche dire la regex associata che è 0^*10^*, inoltre lo stato F rappresenta uno stato di errore a cui si arriva con stringhe del tipo $0^*10^*1(0|1)^*$.

9.2.2 Tabella di equivalenza

Le tabelle di equivalenza sono uno strumento per rendere più veloce e algoritmico la creazione della relazione di equivalenza di un DFA. Ripetiamo l'esercizio di prima:

Esempio 9.2.2 (). *Prima di tutto costruiamo una tabella con solo delle coppie che non sono identiche (quindi niente (A,A)), per far ciò la costruiamo a scaletta (dopo vedremo che sia l'ordine degli stati è importante).*

B					
C					
D					
E					
F					
	A	B	C	D	E

Adesso, come prima, andiamo a controllare quali coppie di stati sono distinti con stringhe di lunghezza i, mettendo una x_i nella cella corrispondete alla coppia.

B					
C	x_0	x_0			
D	x_0	x_0			
E	x_0	x_0			
F			x_0	x_0	x_0
	A	B	C	D	E

Poi marchiamo con x_1 le coppie di stati distinti da stringhe di lunghezza 1. Ovviamente non dobbiamo ricontrollare le coppie marcate x_0. Otteniamo:

B					
C	x_0	x_0			
D	x_0	x_0			
E	x_0	x_0			
F	x_1	x_1	x_0	x_0	x_0
	A	B	C	D	E

Come prima ci si può fermare all'iterazione 1. Gli stati equivalenti saranno quelli non ancora marcati, quindi $(A, B), (C, E, D)$.

9.3 Algoritmo per minimizzazione

L'algoritmo si di minimizzazione di un DFA attraverso la tabella di equivalenza.

Algorithm 3: Minimizzazione di un DFA

```
1  Costruire la tabella a scala;
2  Marcare con x_0 ogni coppia (q_1, q_2) tale che uno sia finale e l'altro no;
3  i := 1;
4  b := true;
5  while b do:{
6     b := false;
7     per ogni coppia non marcata (q_1, q_2) {
8        if (∃a ∈ Σ con (δ(q_1, a), δ(q_2, a)) già marcati) {
9           marca (q_1, q_2) con x_i ;
10          b := true;
11       }
12       i := i + 1;
13    }
14 }
15 Chiamiamo J l'insieme delle coppie non marcate;
16 La relazione di equivalenza ~ è data da ~= J ∪ {(q_2, q_1) | (q_1, q_2) ∈ J} ∪ {(q, q) | q ∈ Q};
```

Vediamo un po cosa fa l'algoritmo:

- Il valore di b ci indica se c'è stata almeno una modifica al passo i-esimo, se non c'è stata possiamo fermarci (in base a quello che avevamo visto sopra).
- Ad ogni iterazioni controlliamo ogni coppia non ancora marcata. Se facendo un passo da ogni stato della coppia arriviamo ad una coppia di stati che sono marcati (cioè che sappiamo già essere distinti) allora anche la coppia iniziale è distinta e quindi la marchiamo.
- La relazione di equivalenza finale è data dalle coppie non marcate unito all'insieme delle coppie non marcate ma al contrario (perché marchiamo (q_2, q_1) e non (q_1, q_2) ma li vogliamo entrambi) e le coppie identiche.

Teorema correttezza: *Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ l'algoritmo della tabella a scala termina, inoltre due stati p e q sono distinguibili se e solo se la casella (p, q) (o la casella (q, p)) è marcata.*

Dimostrazione: *La dimostrazione della terminazione è ovvia, infatti esiste sempre un k tale che $\sim_k = \sim_{k+1}$ (l'algoritmo termina proprio quando trova quel k). Per il resto del teorema dividiamo il se e solo se:*

- $\Rightarrow$*) (cioè se p e q sono distinguibili allora la casella è marcata). Se p e q sono distinguibili allora $\exists x \in \Sigma^* \mid \hat{\delta}(p,x) \in F \wedge \hat{\delta}(q,x) \notin F$ (o anche il contrario, cioè $\hat{\delta}(p,x) \notin F$ e $\hat{\delta}(q,x) \in F$). Se prendo $k = |x|$ allora abbiamo di sicuro che $(p,q) \notin \sim_k$ cioè (p,q) viene marcata entro l'iterazione k (non mi sembra una vera e propria dimostrazione ma vabbe).*
- $\Leftarrow$*). Non so cosa dica in questo punto.*

9.4 Automa minimo

Definizione : *Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'automa minimo equivalente $M_{min} = (\Sigma, Q_{min}, \delta_{min}, [q_0], F_{min})$ è dato da:*

- *Gli stati $Q_{min} = \{[q] \mid q \in Q\}$ dove $[q] = \{q' \in Q \mid q \sim q'\}$, cioè sono tutte le classi di equivalenza degli stati di M.*
- *La funzione di transizione è $\delta_{min}([q], a) = [\delta(q, a)]$, cioè la mossa di carattere "a" partendo dal nodo $[q]$ (classe di equivalenza di q) è data dalla classe di equivalenza del nodo a cui arriverei con mossa "a" partendo da q.*
- *Lo stato iniziale è la classe di equivalenza dello stato iniziale del DFA.*
- *Lo stato finale è definito come $F_{fin} = \{[q] \mid q \in F\}$ cioè l'insieme delle classi di equivalenza di tutti gli stati che sono finali.*

Teorema correttezza: *Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'automa $M_{min} = (\Sigma, Q_{min}, \delta_{min}, [q_0], F_{min})$ riconosce lo stesso linguaggio di M ed ha il numero minimo di stati fra gli automi deterministici che riconoscono questo linguaggio.*

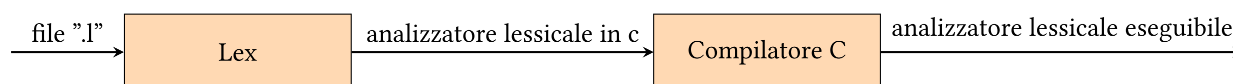
Chapter 10

Lex e Yacc

10.1 Lex generatore di analizzatori lessicali

Lex è un software generatore di analizzatori lessicali (un analizzatore lessicale prende in input un testo e genera una lista di token). Lex è un programma che:

- Prende in **input** un file di tipo ".l" contenete un'insieme di definizioni regolari, e delle espressioni regolari con delle corrispondenti azioni.
- In **output** restituisce un programma in C che realizza l'automa riconoscitore e che associa ad ogni espressione regolare la sua relativa azione.



10.1.1 Come è fatto un file di input Lex

Un file input in Lex è diviso in 3 parti:

- **Dichiarazioni**, cioè delle definizioni regolari.
- **Regole**. Le regole sono una coppia di espressione regolare e azioni. Quando un lessema è riconosciuto da un'espressione regolare eseguo l'azione corrispondente.
- **Funzioni ausiliarie**. Servono per evitare di scrivere funzioni complesse nel campo azione e magari per riusabilità.

Esempio 10.1.1 (Dichiarazioni e regole). *Degli esempi di dichiarazioni possono essere:*

$$cifra\ [0-9]$$
$$cifre\ [0-9]^+$$
$$ide\ [a-zA-Z]([a-zA-Z] \mid [0-9])^*$$

dove ad esempio ide sta per identificatori è la regex indica che devono iniziare con una lettera (sia minuscola sia maiuscola) e poi dopo possono contenere anche numeri. Esempi di regole invece possono essere:

$$\{cifre\}\ \{printf("<NUM,\%s>", yytext);\}$$
$$\{ide\}\ \{printf("<IDE,\%s>", yytext);\}$$

Notiamo che la variabile "yytext" contiene il testo che ha fatto match con la regex. Queste due regole fanno scrivere il token relativo ad un numero se leggiamo un numero o relativo all'ide nel caso di un identificatore.

10.1.2 Funzionamento del programma dato in output

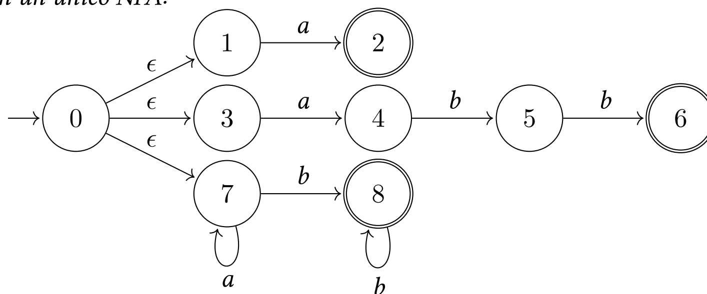
Il programma dato in output da Lex implementa semplicemente un DFA che riconosce l'insieme delle espressioni regolari contenute nelle regole. La sua esecuzione si divide principalmente in 3 passi:

- Scorre il testo sorgente cercando una stringa che matcha con un'espressione regolare. Nel far ciò cerca sempre di :
  - Fare il **match più lungo possibile** (perché sennò se avessimo un pattern che ha un altro pattern come prefisso il primo non verrebbe mai matchato perché l'altro fa match più velocemente).
  - Nel caso invece di pattern che sono casi particolari di altri si **matcha il primo in elenco** (guarda l'esempio sotto per capire) (questo fatto fa in modo che nei file .l le parole riservate vadano prima di tutto, senno invece di rilevare parole riservate rilevo tipo identificatori).
- Quando riconosce un lessema esegue l'azione della regola che ha fatto match.
- Se l'input non corrisponde a nessun pattern allora lo lascia inalterato e segnale errore al gestore degli errori.

Esempio 10.1.2 (). *Prendiamo come insieme di regole le seguenti:*

$$\begin{array}{ll} P_1 & \{a\}\,\{printf("Z");\} \\ P_2 & \{abb\}\,\{printf("K");\} \\ P_3 & \{a^*b^+\}\,\{printf("Y");\} \end{array}$$

Notiamo che P_1 è prefisso di P_2 perché la stringa "a" fa match con P_1 ma è anche l'inizio di P_2. Abbiamo inoltre che P_3 è un caso particolare di P_2 quando abbiamo a^1b^2, in questo caso si riconosce sempre il primo in elenco.
Con un input di "babba" il programma stamperà "Y" "K" "Z" perché b viene riconosciuta da P_3, "abb" da P_2 e "a" da P_1.
Ma come implementa l'automa per riconoscerlo? Prende i tre automi separati di P_1, P_2, P_3 e li mette insieme in un unico NFA:



Quello che fa ora è una sorta di simulazione di un DFA andando a creare una tabella che appunto simula il comportamento del DFA. A differenza di un DFA questa tabella non rispetta i vincoli che ogni stato deve avere ogni possibile mossa, quindi nel caso in cui si arrivi ad un punto morto (cioè senza mosse) si fa backtracking.

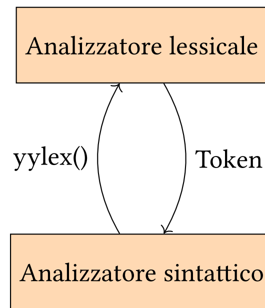
Stato	a	b	pattern riconosciuto	azione
0137	247	8	nessuno	nessuna
247	7	58	P_1	printf("Z");
8	-	8	P_3	printf("Y");
7	7	8	nessuno	nessuna
58	-	68	P_3	printf("Z");
68	-	8	P_2	printf("K");

Quindi nel caso del input "babba" partendo dallo stato 0137 facciamo la transizione "b" e finiamo nello stato 8. Visto che lo stato 8 non ha transizioni "a" mi fermo e vedo che il pattern riconosciuto è P_3 e quindi stampo "Y". Ora l'input è "abba" partendo ad 0137 leggo "a" vado nello stato 247, da 247 leggo "b" e vado nello stato

58, da 58 leggo "b" e vado nello stato 68, da 68 leggo "a" e mi fermo e quindi stampo "K". In input resta solo "a" quindi da 0137 leggo "a" e vado in "247" e mi fermo stampando "Z".

10.2 Utilizzo di Yacc

In realtà **l'analizzatore lessicale** generato da Lex **non viene utilizzato in modo indipendente** ma viene utilizzata come **sub routine dell'analizzatore sintattico** generato da Yacc (che è appunto un generatore di analizzatori sintattici), come nel seguente schema:

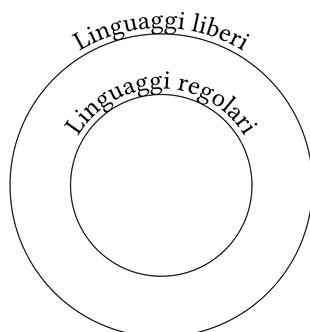


L'analizzatore lessicale viene usato dall'analizzatore sintattico quando ha necessità di un nuovo token attraverso la funzione "yylex()" che restituisce il token successivo (in realtà restituisce il nome del token, mentre il token vero è in una variabile condivisa). Quindi non l'analizzatore lessicale non viene eseguito una volta per tutte, ma la sua esecuzioni è "on-demand".

Chapter 11

Proprietà algoritmiche dei linguaggi regolari

Vedremo in questo capitolo una proprietà dei linguaggi regolari che ci permetterà di dire se un linguaggio libero è anche regolare o no. Infatti un linguaggio regolare è sicuramente anche un linguaggio libero, ma non vale il viceversa, se infatti "disegniamo i due insiemi" otteniamo:



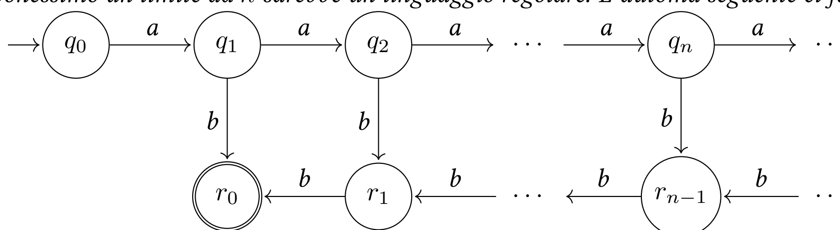
Esempio 11.0.1 (Linguaggio libero non regolare). *Prendiamo come esempio il seguente linguaggio:*

$$L = \{a^n b^n \mid n \geq 0\}$$

Questo linguaggio è sicuramente libero, infatti è generato dalla grammatica:

$$S \to \epsilon \mid aSb$$

ma si può dimostrare non essere regolare. L'idea dietro il perché non è regolare è che i linguaggi regolari non sono in grado di contare infinitamente (infatti ci possiamo aspettare un numero infinito di "a" e "b"), se invece ponessimo un limite ad n *sarebbe un linguaggio regolare. L'automa seguente ci fa capire ciò:*



Questo automa ci fa capire infatti come noi dovremo avere tanti stati q *e* r *quanto è il numero massimo che* n *può assumere, ma visto che è infinito ne dovremmo avere infiniti. Invece un linguaggio come* $L' = \{a^n b^m \mid n, m \geq 0\}$ *è sia libero sia regolare perché qua* n *e* m *sono indipendenti fra di loro e quindi non ci bisogno di contare, infatti corrisponde alla regex* $a^* b^*$*.*

11.1 Pumping lemma

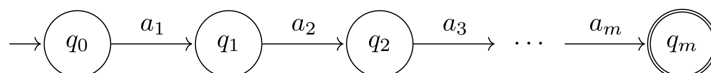
Teorema Pumping lemma: *Se L è un linguaggio regolare, allora $\exists N > 0$ tale che $\forall z \in L$ con $|z| \geq N$, $\exists u, v, w$ tali che:*

$$z = uvw$$
$$|uv| \leq N$$
$$|v| \geq 1$$
$$\forall k \geq 0 \quad uv^k w \in L$$

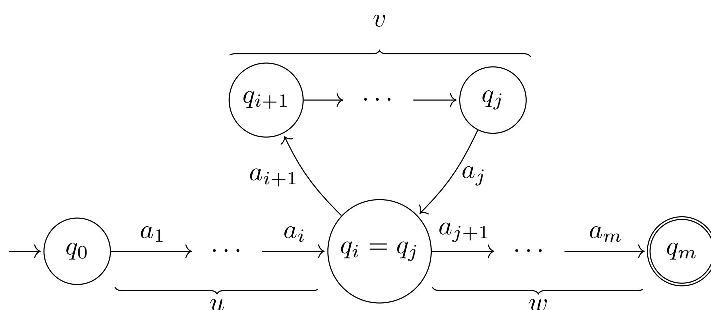
Inoltre N è minore o uguale al numero di stati dell'automa minimo che accetta L.

Quello che ci dice questo teorema è che per i linguaggi regolari esiste sempre un valore N tale che ogni stringa del linguaggio più lunga di N è composta da tre elementi, di cui uno, quello centrale, ripetibile quante volte si vuole (rimanendo dentro il linguaggio). Dalla dimostrazione si capisce anche meglio l'intuizione dietro a ciò.

Dimostrazione: *Sia $N = |Q_M|$ dove M è il DFA minimo che accetta il linguaggio L. Sia $z = a_1 a_2 ... a_m \in L$ con $m \geq N$. Avremo quindi che il nostro DFA M ha un cammino del genere:*



questo perché la stringa z è accettata dal DFA e quindi deve esistere un cammino da stato iniziale a stato finale con transizioni nell'ordine $a_1, a_2, ..., a_m$. Notiamo che il cammino è dato da $m + 1 > N$ stati ma ciò implica che $\exists i, j (i \neq j)$ tale che $q_i = q_j$. Quest'ultima cosa è vera perché abbiamo detto che l'automa ha N stati e il nostro cammino $m + 1 > N$ quindi almeno uno si deve ripetere per forza. Possiamo quindi vedere il nostro cammino come:



Visto che $i \neq j$ abbiamo che $v = a_{i+1} ... a_j$ è tale che $|v| \geq 1$. Notiamo che la condizione $|uv| \leq N$ è valida visto che il ciclo inizia al nodo i-esimo ma sappiamo che $i < N$, inoltre questa condizione ci dice che, nonostante ci possano essere più cicli nel nostro cammino, il ciclo di v sarà sempre il primo che incontriamo. Notiamo anche come la condizione finale $\forall k \geq 0 \quad uv^k w \in L$ è valida, perché possiamo percorrere k volte il ciclo di v e finché poi viene seguito dalla stringa w arriveremo sempre al nodo finale.

Si possono fare varie osservazioni su questo teorema:

- Se L è finito allora non esiste nessuna $z \in L$ con $|z| \geq N$ e quindi deve valere che la premessa è falsa.
- Se $\exists z \in L$ con $|z| \geq N$ allora l'automa che riconosce il linguaggio riconosce un linguaggio infinito.

11.1.1 Come usare il pumping lemma per dire se un linguaggio non è regolare

Prendo il pumping lemma otteniamo che se un linguaggio è regolare allora il pumping lemma vale. Ma se avessimo che per un linguaggio vale la negazione del pumping lemma allora il linguaggio sarà sicuramente non regolare. Dobbiamo però sapere la negazione del pumping lemma:

Teorema negazione pumping lemma: *Se* $\forall N > 0 \ \exists z \in L$ *con* $|z| \geq N$ *tale che:*

$$\forall u, v, w \text{ se :}$$
$$z = uvw$$
$$|uv| \leq N$$
$$|v| \geq 1$$
$$\text{allora } \exists k \geq 0 \text{ tale che } uv^k w \notin L$$

allora L *non è regolare.*

Esempio 11.1.1 (). *Vogliamo dimostrare che* $L = \{a^n b^n \mid n \geq 1\}$ *non è regolare. Per farlo fissiamo un* $N > 0$ *generico (infatti dobbiamo dimostrarlo* $\forall N > 0$*). Scegliamo poi* $z = a^N b^N$*, notando che* $|z| \geq N$ *e quindi va bene per il teorema.*
Scegliamo tre u, v, w *generici tali che:*

$$z = uvw$$
$$|uv| \leq N$$
$$|v| \geq 1$$

Visto che $|uv| \leq N$ *abbiamo che sia* u *sia* v *siano sole "a" (perché la stringa* z *ha* N *"a" iniziali) e quindi* $v = a^j$ *con* $j \geq 1$ *e* $u = a^{N-j}$ *. Dobbiamo trovare adesso un* $k \geq 0$ *tale che* $uv^k w \notin L$*. Prendiamo* $k = 2$ *(si può prendere in questo caso qualunque* $k \geq 2$*) e vediamo che la stringa* $uv^2 w \notin L$ *perché* $uv^2 w = a^{N-j} a^{2j} b^N = a^{N+j} b^N$ *ma sappiamo che* $a^{N+j} b^N \notin L$ *perché dovremmo avere che* $N + j = N$ *ma* $j \geq 1$ *e quindi* L *non è regolare.*

Esempio 11.1.2 (). *Il linguaggio* $L = \left\{a^{n^2} \mid n \geq 0\right\}$ *(cioè n è un quadrato perfetto) è regolare? no e per dimostrarlo facciamo lo stesso ragionamento di prima. Non riscrivo tutta la cosa iniziale. Prendendo* $z = a^{N^2}$ *possiamo dire che* $|z| \geq N$*.*
Prendiamo u, v, w *con le solite proprietà e prendiamo* $k = 2$*. Dobbiamo dimostrare che* $uv^2 w \notin L$ *cioè che* $|uv^2 w|$ *non è un quadrato perfetto. Sappiamo che* $|uv^2 w| = |uvw| + |v| = N^2 + |v|$ *per trovare il valore di* v *usiamo le due proprietà del teorema (*$|uv| \leq N$ *e che* $|v| \geq 1$*) ottenendo che* $1 \leq |v| \leq N$*. Ma quindi otteniamo che:*

$$N^2 + |v| < N^2 + N < N^2 + 2N + 1 = (N + 1)^2$$

Ma quindi $|uv^2 w|$ *è in mezzo a due quadrati perfetti (*$N^2 < |uv^2 w| < (N + 1)^2$*) e quindi non può essere lui stesso un quadrato perfetto e quindi* $uv^2 w \notin L$ *e quindi* L *non è regolare.*

11.2 Altre proprietà dei linguaggi regolari

Teorema : *La classe dei linguaggi regolare è chiusa (chiusa vuol dire che se prendo due linguaggi regolari e applico una di queste 5 operazioni il risultante è un linguaggio regolare) per:*

- ***Unione.***
- ***Concatenazione.***
- ***Stella di kleene.***
- ***Complementazione.***
- ***Intersezione.***

Dimostrazione: *Dividiamo in fasi la dimostrazione:*

- *La prima, seconda e terza operazione sono ovvie che sono chiuse, perché deriva dal fatto che quelle tre operazioni (unione, concatenazione e stella di kleene) hanno delle operazioni identiche nelle espressioni regolari (cioè siano L_1, L_2 regolari allora esistono s_1, s_2 regex tali che $L_1 = \mathcal{L}[s_1]$ e $L_2 = \mathcal{L}[s_2]$ e sia $L_3 = L_1 \cup L_2$ si può dire che è regolare perché $L_3 = \mathcal{L}[s_1|s_2]$).*
- *La complementazione invece è chiusa perché dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ tale che $L = L[M]$ possiamo costruire il DFA $\overline{M} = (\Sigma, Q, \delta, q_0, Q \setminus F)$ tale che $\overline{L} = L[\overline{M}]$. infatti $w \in L[M]$ se e solo se $w \notin L[\overline{M}]$. Nota infatti che vale:*
$$L[\overline{M}] = \Sigma^* \setminus L[M]$$
- *L'intersezione deriva dalla legge di De Morgan:*
$$L[M_1] \cap L[M_2] = \overline{\overline{L[M_1]} \cup \overline{L[M_2]}}$$
e visto che sia la negazione sia l'unione sono chiuse anche l'intersezione lo è.

La chiusura per intersezione può essere usata per dimostrare che un linguaggio non è regolare in questo modo:

$$L \cap L_{reg} = L_{non-reg} \implies L \text{ non è regolare}$$

Esempio 11.2.1 (). *$L = \left\{w \in \{a, b\}^* \mid \text{in } w \text{ occorrono tante "a" quante "b"}\right\}$ è regolare? se lo fosse allora il linguaggio:*

$$L' = L \cap a^*b^* = \{a^n b^n \mid n \geq 0\}$$

dovrebbe essere regolare, ma abbiamo dimostrato non esserlo e quindi L non è un linguaggio regolare

Chapter 12

Automi a pila

I linguaggi liberi sono una classe più generale di quella dei linguaggi regolari, infatti nella grammatica di un **linguaggio libero** sono ammesse tutte le produzioni del tipo:

$$V \to \alpha \quad \alpha \in (T \cup NT)^*$$

Mentre nei **linguaggi regolari** erano ammesse solo le produzioni del tipo:

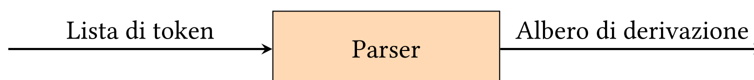
$$\begin{aligned} &V \to aW \quad W \in NT,\ a \in T \\ &V \to a \quad a \in T \\ &S \to \epsilon \end{aligned}$$

Per riconoscere i linguaggi liberi si utilizzano degli automi particolari chiamati **automi a pila** (PDA), di essi se ne possono trovare di due tipi:

- **Non deterministici** che riconoscono i linguaggi liberi ma sono poco efficienti e quindi poco utili per realizzare compilatori.
- **Deterministici** che invece riconoscono solo una classe di linguaggi liberi detti **linguaggi liberi deterministici**. Questi sono però più efficienti e quindi più usati.

12.1 Analisi sintattica

L'analisi sintattica si occupa di **generare un albero di derivazione** data una lista di token.



Il parser sarà creato a partire da una grammatica libera.

12.2 Automi a pila

Definizione : *Un automa a pila non deterministico (PDA) è una 7-upla del tipo* $(\Sigma, Q, \Gamma, \delta, q_0, \bot, F)$ *dove:*

- Σ *è un alfabeto finito.*
- Q *è un insieme finito di stati.*
- Γ *è un insieme finito di* ***simboli sulla pila****.*
- $\bot \in \Gamma$ *è il* ***simbolo iniziale sulla pila****.*
- $F \subseteq Q$ *è l'insieme degli stati finali.*

- *δ è la funzione di transizione del tipo:*

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \wp(Q \times \Gamma^*)$$

Notiamo quindi che la funzione di transizione prende in input una tripla: stato, simbolo letto (che viene consumato o, nel caso di ϵ, no), simbolo in cima alla pila. Restituisce poi un'insieme (e quindi non è deterministico!) di possibili stati di arrivo. Questi stati di arrivo sono caratterizzati dallo stato in cui arrivo più lo stato, dopo la transizione, della pila. Sulla pila può essere scritta una stringa qualsiasi.

E' non deterministico perché, come già detto sopra, abbiamo che la funzione di transizione punta a più stati cioè:

$$|\delta(q, \sigma, A)| \text{ può essere } > 1$$

ma non solo: anche il fatto che mosse epsilon sono accettate mantiene l'automa non deterministico, infatti possiamo avere, dato uno stato un simbolo $b \in \Sigma$ e una pila, che sia possibile sia una mossa epsilon sia una mossa b.
Gli automi a pila introducono quindi una **pila che ha le seguenti proprietà**:

- Può crescere senza limiti.
- Si può leggere solo l'elemento top della pila.
- Si può rimuovere solo l'elemento top della pila (in realtà sia questa sia quella di prima sono restrizioni che poi non si usano sempre).
- Può inserire un elemento in testa.

12.2.1 Problemi che risolvono i PDA rispetto ai DFA/NFA

Quali sono i problemi dei DFA/NFA che non permette il riconoscimento di un linguaggio libero? Il problema principale è che non hanno memoria e quindi sono incapaci di tener traccia di elementi.

Esempio 12.2.1 (Problema dei DFA/NFA). *Prendiamo il linguaggio $L = \{ww^R \mid w \in \{a,b\}^*\}$, L è libero poiché è generato dalla grammatica:*

$$S \to \epsilon \mid aSa \mid bSb$$

notiamo infatti che non rispetta il vincolo dei linguaggi regolari (nelle produzioni possibili di essi infatti non c'è nessuna con un non terminale seguito da un terminale). In questo caso un DFA/NFA dovrebbe essere in grado di ricordarsi la prima parte w per poi poterla confrontare con la seconda parte.
L'automa a pila è invece in grado di riconoscerlo perché, grazie alla pila, è in grado di memorizzare la porzione iniziale dell'input.

12.2.2 Transizioni di un PDA

Come per gli NFA/DFA andiamo a dare le definizioni di tre elementi: configurazione, mossa e cammino.

Definizione transizione istantanea o configurazione: *Viene detta transizione istantanea o configurazione una tripla del tipo:*

$$(q, w, \beta) \text{ dove } q \in Q, w \in \Sigma^*, \beta \in \Gamma^*$$

Questa configurazione mi dice quindi lo stato completo del PDA cioè sia lo stato attuale (q), sia l'input non ancora letto (w) e sia la stringa presente sulla pila (β).

Definizione mossa: *Una mossa è una coppia (premessa, azione) (?) del tipo:*

$$\frac{(q', \alpha) \in \delta(q, a, X)}{(q, aw, X\beta) \vdash_N (q', w, \alpha\beta)} \alpha \in \Sigma$$
$$\frac{(q', \alpha) \in \delta(q, \epsilon, X)}{(q, w, X\beta) \vdash_N (q', w, \alpha\beta)} \alpha \in \Sigma$$

"Leggendo la prima mossa a parole" ci dice: se abbiamo che (partendo da q, leggendo a e nella pila X arriviamo allo stato q' con α sulla pila) allora se siamo nella configurazione con stato q, input da leggere aw e sulla pila abbiamo $X\beta$ allora "transitiamo" sulla configurazione con stato q', con input da leggere w (abbiamo consumato "a") e sulla pila $\alpha\beta$ (cioè abbiamo sostituito X con α).

Definizione computazione/cammino: *Una computazione o cammino è semplicemente la concatenazione di n mosse, cioè la chiusura riflessiva (caso 0 mosse) e transitiva (caso n mosse) della mossa.*

$$\frac{}{(q, w, \beta) \vdash_N^* (q, w, \beta)}$$
$$\frac{(q, w, \beta) \vdash_N^* (q', w', \beta') \vdash_N (q'', w'', \beta'')}{(q, w, \beta) \vdash_N^* (q'', w'', \beta'')}$$

12.3 Linguaggio accettato da un PDA

Ci sono due modi diversi per definire quando una stringa è riconosciuta da un PDA:

- **Per stato finale**, cioè una stringa è riconosciuta se c'è un cammino che porta a stato finale (della pila finale non ci interessa):

$$L[N] = \{w \in \Sigma^* \mid (q_0, w, \bot) \vdash_N^* (q, \epsilon, \alpha) \text{ con } q \in F\}$$

- **Per pila vuota**, cioè una stringa è riconosciuta se c'è un cammino che porta ad una configurazione con la pila vuota:

$$P[N] = \{w \in \Sigma^* \mid (q_0, w, \bot) \vdash_N^* (q, \epsilon, \epsilon)\}$$

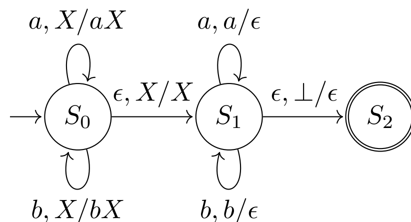
Notiamo che, per un generico PDA N, vale che:

$$L[N] \neq P[N]$$

ciò non vuol dire che siano sempre diversi, ma che generalmente lo sono.

12.3.1 Esempi di PDA

Esempio 12.3.1 (). *Dato il linguaggio $L = \{ww^R \mid w \in \{a,b\}^*\}$ vogliamo costruire un PDA che lo riconosca (non scrivo la costruzione perché non sappiamo ancora farla). Il PDA seguente funziona:*

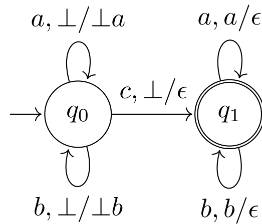


Dove una transizione etichettata come $a, X/aX$ si legge come: leggo un carattere "a", lo stato della pila pre mossa e X e dopo la mossa aX (quindi metto il carattere nella pila). Questo PDA funziona perché nello stato S_0 riceviamo tutta la parte dell'input relativa ad w e ce la salviamo nella pila poi, in modo non deterministico (perché le mosse epsilon sono non deterministiche e quindi implicano il back tracking), ci spostiamo nello stato S_1 che andrà a matchare caratteri solo se sono quelli che stanno in w^R (infatti accetta "a" solo se c'è anche

nella pila). Quando la pila è vuota va nello stato finale, quindi avremo anche che:

$$L[N] = P[N]$$

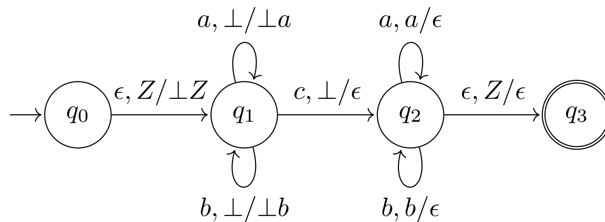
Esempio 12.3.2 (). *Dato il linguaggio* $L = \{wcw^R \mid w \in \{a,b\}^*\}$ *costruiamo il PDA associato.*



Questo PDA per funzionare fa in modo che il simbolo $\bot$ *venga tenuto in cima nello stato* q_0 *mentre sarà proprio la transizione verso lo stato* q_1 *che consumerà questo* $\bot$ *quando in input leggiamo* c.
Notiamo che in questo caso:

$$L[N] \neq P[N]$$

poiché l'automa N *per stato finale riconosce il linguaggio* wcy *dove* y *è un prefisso di* w^R *(infatti quando siamo in* q_1 *niente vieta di fermarci) mentre per pila vuota riconosciamo correttamente il linguaggio* L*. Possiamo comunque costruire un PDA alternativo che funziona anche per stato finale:*



Che agisce quasi identicamente a prima solo che aggiungiamo un simbolo iniziale della pila Z *che indica quando abbiamo finito di leggere la stringa* w^R*. In questo caso:*

$$L[N] = P[N] = L$$

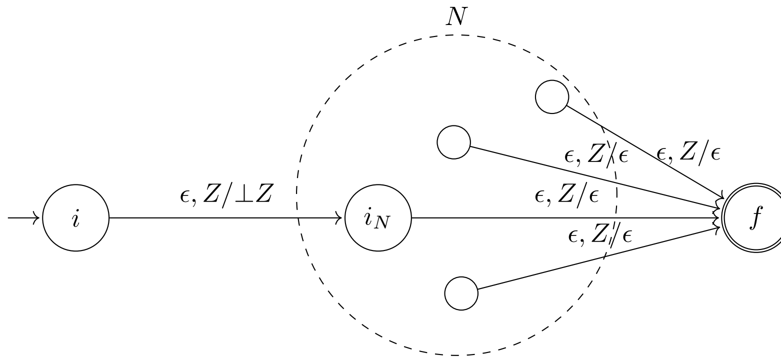
12.3.2 Classe di linguaggi riconosciuta per pila vuota e per stato finale

Teorema : *La classe dei linguaggi riconosciuti per stato finale è uguale a quella per pila vuota. Infatti vale che:*

- *Se* $L = P[N]$ *allora possiamo costruire* N' *tale che* $L = L[N']$.
- *Se* $L = L[N]$ *allora possiamo costruire* N' *tale che* $L = P[N']$.

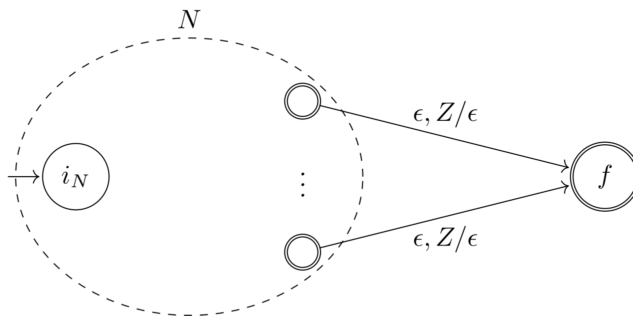
Dimostrazione: *Dimostriamo punto per punto:*

- *Abbiamo un PDA* N *che riconosce per pila vuota possiamo costruire il seguente PDA* N' *che riconosce per stato finale:*



dove i nodi "bianchi" sono i nodi in cui ci aspettiamo che il nostro PDA riconosca per pila vuota, li rendiamo quindi "finali" facendoli fare, nel caso di pila vuota, una transizione verso uno stato finale.

- *Abbiamo un PDA N che riconosce per stato finale possiamo costruire il seguente PDA N' che riconosce per pila vuota:*



dove i nodi "bianchi" sono i nodi in cui ci aspettiamo che il nostro PDA riconosca per pila vuota, li rendiamo quindi "finali" facendoli fare, nel caso di pila vuota, una transizione verso uno stato finale.

12.4 Come ottenere un PDA da una grammatica libera

Prima di dare l'idea di come costruire un PDA enunciamo il seguente teorema che, nella dimostrazione, conterrà anche il metodo per ottenere un PDA da una grammatica libera.

Teorema : *Un linguaggio L è libero da contesto se e solo se è accettato da un qualche PDA.*

Dimostrazione: *Se L è libero allora $\exists G = (NT, T, R, S)$ libera tale che $L = L(G)$. Costruiamo poi il PDA $N = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$ (dove T è l'alfabeto dei simboli in input, q è l'unico stato, $T \cup NT$ sono i simboli che possono stare sulla pila, $\emptyset$ ci dice che non ci sono stati finali, infatti riconosciamo per pila vuota). Costruiamo poi la funzione di transizione δ in questo modo:*

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \in R\} \quad \forall A \in NT$$
$$\delta(q, a, a) = \{(q, \epsilon)\} \quad \forall a \in T$$

Queste transizioni hanno il compito di simulare una derivazione canonica sinistra (infatti espando sempre il non terminale sul top che corrisponde a quello più a sinistra). Si può poi dimostrare che:

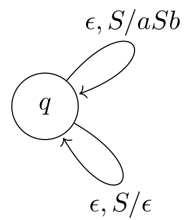
$$S \Longrightarrow^* w \quad \text{sse} \quad (q, w, s) \vdash_N^* (q, \epsilon, \epsilon)$$

L'idea dietro alla costruzione del PDA è che per ogni produzione della grammatica faccio delle transizioni epsilon che sostituiscono un non terminale con una sua produzione. Poi dopo faccio delle transizione per consumare la pila in cui consumo i terminali che si sono accumulati sulla pila. Mentre lo stato è solo uno e il PDA riconoscerà per pila vuota.
Questo tipo di metodo viene detto **top-down** perché partendo dal simbolo iniziale (che si deve trovare sulla pila all'inizio) andiamo a ricreare l'albero di derivazione.

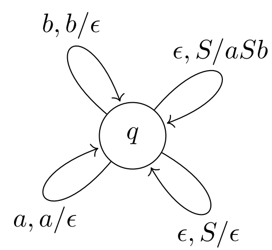
Esempio 12.4.1 (Da grammatica a PDA). *Supponiamo di avere la seguente grammatica:*

$$S \to aSb \mid \epsilon$$

e di volere trovare un PDA che la riconosca, seguendo la costruzione scritta sopra creiamo prima le transizioni per le produzioni della grammatica:



Poi aggiungendo le transizioni di "consumo" (o di match penso le chiami il prof):



Chapter 13

Proprietà linguaggi liberi

13.1 Proprietà di chiusura

Teorema : *I linguaggi liberi sono chiusi per:*

- ***Unione***. *Cioè siano L_1, L_2 due linguaggi liberi allora il linguaggio $L_3 = L_1 \cup L_2$ è ancora libero.*
- ***Concatenazione***. *Cioè siano L_1, L_2 due linguaggi liberi allora il linguaggio $L_3 = L_1 \cdot L_2$ è ancora libero.*
- ***Ripetizione***. *Cioè sia L_1 un linguaggio libero allora $L_2 = (L_1)^*$ è esso stesso linguaggio libero.*

Dimostrazione: *Sia $L_1 = L(G_1)$ con $G_1 = (NT_1, T_1, R_1, S_1)$ e sia $L_2 = L(G_2)$ con $G_2 = (NT_2, T_2, R_2, S_2)$ (assiumiamo poi che $NT_1 \cap NT_2 = \emptyset$ cioè non hanno non terminali in comune). Dimostriamo che le operazioni elencate sopra sono chiuse:*

- *Unione. Sia $L(G) = L_1 \cup L_2$ possiamo dire quindi che:*

$$G = (NT_1 \cup NT_2 \cup \{S\}, T_1 \cup T_2, S, R_1 \cup R_2 \cup \{S \to S_1 | S_2\})$$

cioè la nuova grammatica libera è una grammatica in cui i non terminali sono tutti i non terminali di G_1 più quelli di G_2 insieme ad un nuovo non terminale S che sarà anche il simbolo iniziale, questo carattere iniziale avrà solo una produzione $S \to S_1 | S_2$.

- *Concatenazione. Sia $L(G) = L_1 \cdot L_2$ dove:*

$$G = (NT_1 \cup NT_2 \cup \{S\}, T_1 \cup T_2, S, R_1 \cup R_2 \cup \{S \to S_1 S_2\})$$

Come prima semplicemente aggiungiamo un simbolo iniziale con un'unica produzione di tipo concatenazione.

- *Stella di kleene. Sia $L(G) = (L_1)^*$ allora:*

$$G = (NT_1 \cup \{S\}, T_1, S, R_1 \cup \{S \to S_1 S | \epsilon\})$$

dove di nuovo aggiungiamo un simbolo iniziale con una produzione che ci effettua la stella di kleene.

13.1.1 I linguaggi liberi non sono chiusi per intersezione

Teorema : *I linguaggi liberi* ***non sono chiusi per intersezione****. Cioè dati L_1, L_2 linguaggi liberi il linguaggio:*

$$L = L_1 \cap L_2$$

*è **non libero**.*

Come conseguenza di questo teorema c'è che i linguaggi liberi **non sono chiusi per complementazione**. Questo deve essere vero perché sennò se lo fossero allora anche l'intersezione dovrebbe essere visto che:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

13.1.2 I linguaggi liberi intersecato a linguaggio regolare

Teorema : *Siano L_1 un linguaggio **libero** e L_2 un linguaggio **regolare**. Allora il linguaggio:*

$$L = L_1 \cap L_2$$

*è un linguaggio **libero**.*

Dimostrazione: *Siano:*

- $L_1 = L[N_1]$ *con* $N_1 = (\Sigma, Q_{N_1}, \Gamma, \delta_{N_1}, q_{N_1}, z, F_{N_1})$ *(quindi N_1 è un PDA) che riconosce per stato finale.*
- $L_2 = L[N_2]$ *con* $N_2 = (\Sigma, Q_{N_2}, \delta_{N_2}, q_{N_2}, F_{N_2})$ *(quindi N_2 è un DFA/NFA).*

Costruiamo poi un PDA N che "simula" il comportamento dei due automi in un unico automa. Definiamo quindi:

$$N = (\Sigma, Q_{N_1} \times Q_{N_2}, \Gamma, \delta, (q_{N_1}, q_{N_2}), Z, F_{N_1} \times F_{N_2})$$

notiamo quindi che gli stati di questo PDA sono coppie di uno stato di N_1 e uno di N_2, mentre gli stati finali sono le coppie che hanno tutti e due gli stati finali. Definiamo poi la funzione di transizione:

$$\delta((q,p), a, X) = \{((r,s), \gamma) \mid s = \delta_{N_2}(p, a) \wedge (r, \gamma) \in \delta_{N_1}(q, a, X)\}$$

Cioè partendo dallo stato (q,p) con pila X e mossa "a" io mi posso muovere sulla configurazione di stato (r,s) e pila γ dove:

- *(r, γ) è uno dei possibili passi successivi nel PDA partendo dalla configurazione (q, a, X).*
- *s è invece lo stato raggiunto partendo dallo stato p con mossa "a" nel DFA/NFA.*

Sto quindi sviluppando in "parallelo" (ma nello stesso automa) i due automi diversi, infatti lo stato + pila mi rappresenta lo stato che avrei nel PDA, mentre l'altro stato rappresenta quello nel DFA/NFA. Diventa ovvio dire che:

$$((q_{N_1}, w, z) \vdash^*_{N_1} (q, \epsilon, \gamma) \text{ e } \hat{\delta}(q_{N_2}, w) = q') \Leftrightarrow ((q_{N_1}, q_{N_2}), w, z) \vdash^*_{N} ((q, q'), \epsilon, \gamma)$$

tutta questa scrittura verifica la correttezza di questo sviluppo in "parallelo". Per capirla meglio dividiamo in parte sinistra e destra del se e solo se:

- *La prima parte è divisa in due, una per il PDA e una per il DFA:*
  - *La parte relativa al PDA $((q_{N_1}, w, z) \vdash^*_{N_1})$ ci dice che una stringa in n passi arriva allo stato q con pila γ e con input da leggere finito.*
  - *La parte relativa al DFA/NFA $(\hat{\delta}(q_{N_2}, w) = q')$ ci dice che in n passi arriviamo allo stato q' con stringa vuota da leggere.*
- *La seconda parte ci dice che nel nuovo PDA costruito in in passi arriviamo allo stato $(q, q'$ con pial γ e stringa da leggere nulla. Quindi questo nuovo PDA ci porta ad una coppia di stati uguali a quella del PDA e DFA/NFA separati.*

Avremo poi quindi che se $q \in F_{N_1}$ e $q' \in F_{N_2}$ la stringa è riconosciuta sia dal PDA N_1 sia dal DFA/NFA N_2 sia dal PDA N che quindi riconosce $L_1 \cap L_2$.

Dalla dimostrazione possiamo anche capire perché la chiusura dell'intersezione vale solo fra linguaggio libero e regolare. Perché se avessimo due linguaggi liberi e provassimo a costruire l'automa N questo si incasinerebbe perché i due automi N_1 e N_2 andrebbero tutti e due a lavorare sulla pila e quindi non funzionerebbero correttamente.

Esempio 13.1.1 (). *Dimostriamo che $L = L_1 \cap L_2$ con:*

$$L_1 = \{a^n b^n \mid n \geq 0\}$$
$$L_2 = \left\{w \in \{a,b\}^* \mid \exists k \in \mathbb{N} \text{ con } |w| = 4k\right\}$$

sia un linguaggio libero. Sapendo che L_1 è un linguaggio libero (poiché corrisponde alla grammatica $S \to aSb|\epsilon$), mentre L_2 è regolare (corrisponde alla regex $s = [(a|b)(a|b)(a|b)(a|b)]^*$) quindi per il teorema sopra possiamo dire che L è libero.

Come conseguenza del teorema sopra vale che se $L_1 \cap L_2$ non è libero ma L_2 è regolare allora L_1 non è libero, questo si può usare negli esercizi.

13.2 Pumping theorem

Teorema : *Se L è un linguaggio libero, allora $\exists N > 0$ tale che $\forall z \in L$ con $|z| \geq N$, $\exists u, v, w, x, y$ tali che:*

$$z = uvwxy$$
$$|vwx| \leq N$$
$$|vx| \geq 1$$
$$\forall k \geq 0 \quad uv^k wx^k y \in L$$

Cioè similmente al pumping theorem relativo ai linguaggi regolari il pumping theorem dei linguaggi liberi ci dice che, per stringhe sufficientemente lunghe, due sottostringhe di questa stringa possono essere pompate, cioè ripetute quante volte si vuole rimanendo all'interno del linguaggio.

Dimostrazione: *Sia $G = (NT, T, R, S)$ una grammatica libera tale che $L = L(G)$.*

- *Sia b il massimo fattore di ramificazione (cioè il massimo numero di simboli che compaiono nella parte destra di una produzione), ovvero:*

  $$b = \max\{|\alpha| \mid A \to \alpha \in R\}$$

  vale che $B \geq 2$ senno la grammatica sarebbe banale (perché sarebbe composta tutta da produzioni di un simbolo).

- *Un albero di altezza h (dove si parte contando da 0 la radice) e fattore di ramificazione b ha al più b^h foglie. Questo perché nel caso "pessimo" (dove abbiamo più foglie) avremo che ogni nodo si ramifica in b sotto nodi e quindi dopo h ramificazioni avremo in totale b^h foglie.*

- *Il valore b^h mi limita quindi la massima lunghezza di una stringa che ha albero di derivazione alto h, perché una stringa sarà composta solo dai caratteri (sempre terminali) delle foglie, e quindi il suo massimo numero di caratteri sarà dato da b^h.*

Detto questo possiamo fissare un valore $N = b^{|NT|+1}$. Presa una stringa z tale che $|z| \geq N$ avremo che quindi ogni albero di derivazione deve avere altezza almeno:

$$b^h \geq N = b^{|NT|+1} \Rightarrow \log_b(b^h) \geq \log_b(b^{|NT|+|}) \Rightarrow h \geq |NT| + 1$$

quindi l'altezza di ogni albero di derivazione per z deve essere almeno $|NT| + 1$. Consideriamo quindi gli alberi di derivazione di z (se ce ne sono più di uno, possibile nel caso in cui G sia ambigua, prendiamo quello

con il minor numero di nodi), allora:

$|z| \geq N \Rightarrow$ *albero con altezza* $\geq |NT| + 1$
$\Rightarrow \exists$ *un cammino da radice* S *ad una foglia con almeno* $|NT| + 2$ *nodi*
$\Rightarrow$ *Quel cammino attraversa* $|NT| + 1$ *nodi interni (perchè dei* $|NT| + 2$ *uno è la foglia) eticettati con un non terminale*
$\Rightarrow$ *Un non terminale si deve ripetere perché passiamo* $|NT| + 1|$ *non terminali ma i non terminale in totale sono* $|NT|$.

Allora la derivazione $S \Rightarrow^* z$ *può essere divisa come:*

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy$$

dove il non terminale A *è quello che si ripete. Vale anche che (non sono sicuro di questo)* $A \Rightarrow^* vAx$ *e quindi per questo possiamo ripetere il la parte* vAx *quante volte vogliamo e rimaniamo dentro il linguaggio. Bisogna adesso verificare i due vincoli:*

- $|vx| \geq 1$. *Questo è ovvio perché se fossero entrambi* ϵ *allora la stringa* uv^0wx^0y *sarebbe uguale a* z *avendo però meno nodi nell'albero di derivazione contraddicendo l'ipotesi che abbiamo preso quello più piccolo (ha meno nodi perché non "passa" sui nodi che espandevano* A *in* vAx*).*
- $|vwx| \leq N$. *Anche questo è vero perché il cammino da* A *alla foglia è di lunghezza minore di* $|NT|+1$ *e quindi non può generare parole più lunghe di* $b^{|NT|+1} = N$ *(perché se lo vediamo come un sotto albero di altezza* $h \leq |NT| + 1$ *al massimo genera stringhe lunghe* $b^h \leq N$*)*

13.2.1 Negazione del pumping theorem

Come per i linguaggi regolari possiamo usare la versione negata del pumping theorem per andare a verificare se un linguaggio è libero o no.

Teorema negazione pumping theorem: *Se* $\forall N > 0 \; \exists z \in L$ *con* $|z| \geq N$ *tale che:*

$$\begin{aligned} \forall u, v, w, x, y \text{ se}: & \\ & z = uvwxy \\ & |vwx| \leq N \\ & |vx| \geq 1 \\ \text{allora} \quad & \exists k \geq 0 \; uv^kwx^ky \notin L \end{aligned}$$

allora L *non è libero*

Esempio 13.2.1 (). *Dimostriamo che* $L = \{a^nb^nc^n \mid n \geq 0\}$ *è non libero. Fissiamo un generico* $N > 0$ *(dobbiamo dimostrarlo* $\forall N > 0$*) e scegliamo* $z = a^Nb^Ncx^N$ *(notiamo* $|z| \geq N$*) per ogni* u, v, w, x, y *che rispettano le seguenti proprietà:*

$$\begin{aligned} & z = uvwxy \\ & |vwx| \leq N \\ & |vx| \geq 1 \end{aligned}$$

Notiamo che la sottostringa vwx *non può contenere sia delle "a" sia delle "c", infatti essendo meno lunga di* N *non è abbastanza lunga per superare le* N *"b" in mezzo fra le "a" e le "c". Dividiamo ora in casi:*

- *Caso in cui* vwx *non contiene "c". La stringa* uv^2wx^2y *non appartiene al linguaggio perché il numero di "c" adesso non è pari al numero di "a" e "b".*
- *Caso in cui* vwx *non contiene "a". Uguale a prima solo che il numero di "a" adesso è minore degli altri (in* uv^2wx^2y*).*

quindi L non è libero (perché abbiamo trovato un $k \geq 0$ tale per cui $uv^kwx^ky \notin L$).

Esempio 13.2.2 (). *Dimostriamo che $L = \{a^n \mid n \text{ è primo}\}$ non è libero. Fissiamo un generico $N > 0$ e scelgo $z = a^p$ con p primo primo tale che $p \geq N + 2$. Per ogni u, v, w, x, y che rispettano le tre proprietà:*

$$z = uvwxy$$
$$|vwx| \leq N$$
$$|vx| \geq 1$$

Dalla seconda e terza possiamo dire che $1 \leq |vx| \leq N$, chiamiamo poi $|vx| = m$. Notiamo che la stringa $|uv^0wx^0y| = |uwy| = p - m$, quindi deve valere che:

$$|uv^{p-m}wx^{p-m}y| = |uwy| + (p-m)|vx| = p - m + (p-m) \cdot m$$

possiamo poi raccogliere il termine comune $(p - m)$ ottenendo:

$$= (p-m)(1+m)$$

per essere primo quel numero deve essere divisibile solo per 1 e per se stesso e quindi deve valere che $(p-m) = 1$ o che $(1 + m) = 1$ (perché se non fosse cosi e fossero dei numeri diversi da 1 allora quei numeri li sarebbero un terzo divisore del numero completo che quindi non sarebbe primo):

- *$(p-m) > 1$, infatti abbiamo che $(p-m) > 1$ visto che $m < N$ (perché $|vwx| \leq N$) e che $p \geq N+2$.*
- *$(1 + m) > 1$ perché $m \geq 1$.*

quindi $(p - m)(1 - m)$ non è primo e quindi la lunghezza di $|uv^{p-m}wx^{p-m}y|$ non è un numero primo e quindi:

$$uv^{p-m}wx^{p-m}y \notin L$$

13.3 Classificazione di chomsky

Esiste una classificazione delle grammatiche di vario tipo:

- Le **grammatiche regolari** sono tutte le grammatiche che usano solo queste produzioni:

  $$A \to aB \quad A \to a \quad S \to \epsilon$$

- Le **grammatiche libere da contesto** sono quelle che usano produzioni di tipo:

  $$A \to \alpha \quad \text{con } \gamma \in (NT \cup T)^+ \quad \S \to \epsilon$$

  Notiamo che la definizione è più restrittiva della nostra perché da noi $\gamma \in (NT \cup T)^*$ e quindi possiamo avere produzioni epsilon anche per non terminali diversi dal simbolo iniziale.

- **Grammatiche dipendenti dal contesto** usando produzione del tipo:

  $$\gamma A \delta \to \gamma w \delta \quad \gamma, \delta \in (NT \cup T)^* \quad w \in (NT \cup T)^+$$

- Le **grammatiche generali** che usano produzione del tipo:

  $$\gamma \to \delta \quad \text{(senza vincoli)}$$

Chapter 14

DPDA e linguaggi deterministici

Definizione DPDA: *Un PDA $N = (\Sigma, Q, \Gamma, \delta, q_0, \bot, F)$ è deterministico se e solo se:*

$$\forall q \in Q, \forall z \in \Gamma \text{ se } \delta(q, \epsilon, z) = \emptyset \text{ allora } \delta(q, a, z) = \emptyset \quad \forall a \in \Sigma$$
$$\forall q \in Q, \forall z \in \Gamma, \forall a \in \Sigma \cup \{\epsilon\} \quad |\delta(q, a, z)| \leq 1$$

Cioè i DPDA sono PDA che dato uno stato hanno massimo una mossa per simbolo e stato della pila. Inoltre nel caso avessi una mossa epsilon in uno stato non ci possono essere nessun'altra tipo di mossa.

14.1 Linguaggi liberi deterministici

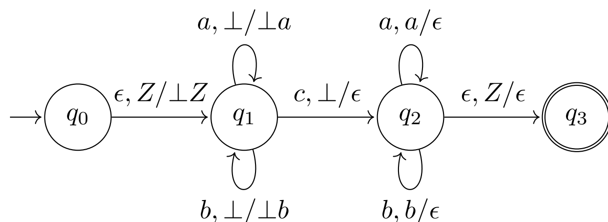
Definizione : *Un linguaggio è libero deterministico se è accettato per **stato finale** da un DPDA.*

Notiamo che a differenza dei linguaggi liberi che potevano essere accettati sia per stato finale sia per pila vuota, nel caso di linguaggi liberi deterministici è per forza per stato finale.

Teorema : *La classe dei linguaggi liberi deterministici è inclusa propriamente nella classe dei linguaggi liberi, cioè:*

$$\textit{Ling liberi deterministici} \subset \textit{Ling liberi}$$

Esempio 14.1.1 (). .
Il linguaggio $L_1 = \{ww^R \mid w \in \{a,b\}^*\}$ è libero ma, si può dimostrare, non essere deterministico.
Il linguaggio $L_2 = \{wcw^R \mid w \in \{a,b\}^*\}$ si può dimostrare essere libero deterministico e quindi di conseguenza libero, infatti il seguente DPDA riconosce L_2:



14.1.1 Linguaggi regolari e DPDA

Teorema : *Se L è un linguaggio regolare allora $\exists N$ dove N è un DPDA tale che:*

$$L = L[N]$$

quindi L è riconosciuto da N per stato finale.

Dimostrazione: *I DFA possono essere visti come DPDA dove non si tiene conto della pila, quindi sapendo che $\exists M$ DFA tale che $L = L[M]$ posso costruire N DPDA che si comporta come M senza modificare la pila.*

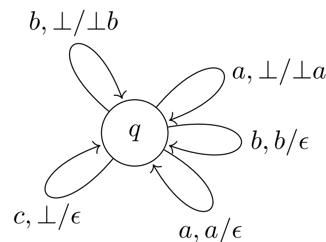
14.1.2 Prefix property

Definizione : *Un linguaggio L si dice che gode della prefix property se :*

$$\nexists x, y \in L \quad \text{tale che } x \text{ è prefisso di } y$$

Teorema : *Un linguaggio libero deterministico L è riconosciuto da un DPDA* ***per pila vuota*** *se e solo se L gode della prefix property.*

Esempio 14.1.2 (). *Il linguaggio $L = \left\{wcw^R \mid w \in \{a, b\}^*\right\}$ gode della prefix property è può quindi essere riconosciuto per pila vuota. Infatti il DPDA seguente riconosce per pila vuota L:*



14.1.3 Assicurarsi la prefix property

Teorema : *Sia L un linguaggio libero deterministico, allora il linguaggio:*

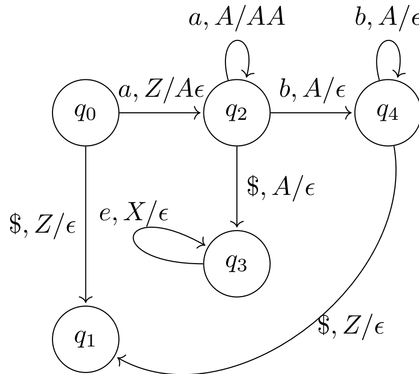
$$L' = L\$ = \{w\$ \mid w \in L\}$$

dove \$ non fa parte dell'alfabeto del linguaggio, gode della prefix property.

Dimostrazione: *Dimostrazione fatta da me (non studiarla).*
Siano date due generiche stringhe $x, y \in L'$ tali che x è prefisso di y. Vogliamo dimostrare, per assurdo, che y non appartiene al linguaggio. Questo è vero infatti y non può appartenere ad L' perché il fatto che x sia prefisso suo implica che y ha all'interno di se due caratteri \$ (uno dato dalla fine della stringa e uno dato dalla fine di x) ma ciò implica che y a cui togliamo il carattere \$ non sta in L (perché il carattere \$ rimasto da x non è parte dell'alfabeto di L) ma quindi y non può appartenere neanche ad L'.

E' quindi diciamo sempre possibile riconoscere un linguaggio tramite un DPDA per pila vuota, perché basta crearsi questo "nuovo" linguaggio che siamo sicuri goda della prefix property. Questo verrà usato sempre nei parser, che riconosceranno sempre linguaggi del tipo $L\$$.

Esempio 14.1.3 (). *Il linguaggio* $L = a^* \cup \{a^n b^n \mid n \geq 1\}$ *non gode dalla prefix property, ma possiamo quindi creare il linguaggio* $L' = L\$$ *che ne gode. Possiamo adesso quindi creare un DPDA che riconosca* L' *per pila vuota:*



Notiamo che il nodo q_2 *è lo "switch" fra riconoscere* a^* *e* $a^n b^n$*, infatti se legge "b" va a leggere tante "b" quante "a" ha letto, sennò se invece di leggere "b" ad un certo punto legge "\$" allora finisce in* q_3 *che è uno stato che svuota la pila.*

14.2 Proprietà dei linguaggi liberi deterministici

14.2.1 Non ambiguità

Teorema : *Se* L *è un linguaggio libero deterministico allora* L *è generabile da una grammatica libera* ***non ambigua****, ciò implica che i* ***linguaggi liberi deterministici non sono ambigui****.*

14.2.2 Chiusura

Teorema : *I linguaggi liberi deterministici* ***sono chiusi*** *per:*

- ***Complementazione****. Cioè sia L un linguaggio libero deterministico allora il linguaggio* $L_1 = \overline{L}$ *(notiamo che* $L_1 = \Sigma^* \setminus L$*) è ancora libero deterministico. E' equivalente dire che se sia* N *il DPDA tale che* $L = L[N]$ *allora esiste un DPDA* N' *tale che* $\overline{L} = L[N']$*.*

A differenza dei linguaggi liberi i linguaggi liberi deterministici sono chiusi per meno cose, infatti:

Teorema : *I linguaggi liberi deterministici* ***non sono chiusi*** *per:*

- ***Intersezione****.*
- ***Unione****.*

Esempio 14.2.1 (). *Prendiamo due linguaggi liberi deterministici* $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ *e* $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$*. Il linguaggio:*

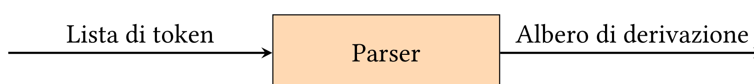
$$L_3 = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$$

si può dimostrare non essere libero.

Chapter 15

Analisi sintattica: Parser

I parser sono il il secondo componente dei compilatori e si pongono dopo lo scanner. Essi prendono in input una lista di token e ne restituiscono un albero di derivazione. I parser **sono implementanti attraverso un DPDA** che viene creato da programmi come YACC partendo da una grammatica libera.



I parser possono essere di due tipi diversi:

- **Parser non deterministici**. Loro sono quelli meno efficienti perché durante la ricerca di una derivazione potrebbero seguire cammini che non portano a niente e quindi **necessitano di backtracking**.
- **Parser deterministici**. Loro sono più efficienti perché non hanno "possibilità di scegliere", il cammino che possono seguire dato una stringa in input è infatti deterministico e solo 1, non necessitano quindi di backtracking.

Entrambi possono usare informazioni aggiuntive sull'input (come look ahead) per guidare la ricerca. I parser poi i dividono ulteriormente in due categorie diverse:

- **Parser top-down**. Questi ricostruiscono una **derivazione leftmost** partendo dal simbolo iniziale S (cioè lui è il primo simbolo della pila). Vengono chiamati top down perché creano l'albero di derivazione partendo dall'alto (appunto dal simbolo iniziale).
- **Parser bottom-up**. Questi ricostruiscono una **derivazione rightmost** partendo dalla stringa w e cercando di ridurla al simbolo iniziale S (che sarà alla fine sulla pila). Sono chiamati bottom-up perché partendo dalle foglie (la stringa w) la riducono cercando di creare S.

15.1 Introduzione parser top-down

Definizione : *Data una grammatica libera* $G = (NT, T, S, R)$*, costruiamo il PDA* $M = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$ *che riconosce per pila vuota (infatti non ha stati finali) e dove* δ *è definita così:*

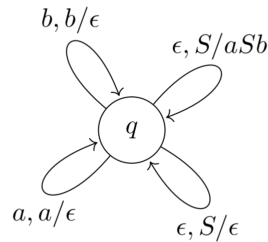
$$(q, \beta) \in \delta(q, \epsilon, A) \quad \text{se } A \to \beta \in R \qquad \text{(espandi)}$$
$$(q, \epsilon) \in \delta(q, a, A) \quad \forall a \in T \qquad \text{(consuma)}$$

tale che $L(g) = P[M]$.

Notiamo che abbiamo due tipi di transizioni:

- Le transizioni "espandi" che possiamo attuare quando abbiamo un non terminale A in cima la pila che si espande in β, e quindi, non consumando input, andiamo a sostituire il non terminale sulla pila con β.
- Le transizioni "consuma" che le usiamo quando abbiamo in cima la pila un simbolo "a" e il prossimo carattere in input è proprio "a", l'input e il simbolo sulla pila vengono consumati.

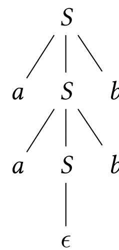
Esempio 15.1.1 (). *Partiamo con la grammatica $S \to aSb \mid \epsilon$ e costruiamo il PDA come dato dalla definizione:*



Prendiamo ora la stringa $w = aabb$ in input, vogliamo adesso farla passare attraverso il parser (il nostro PDA) per ricavare l'albero di derivazione (le configurazioni che vediamo adesso sono del tipo (stato, input, stack)):

$$(q, aabb, S) \overset{espando\ S\to aSb}{\vdash} (q, aabb, aSb) \overset{consumo"a"}{\vdash} (q, abb, Sb)$$
$$\overset{espando\ S\to aSb}{\vdash} (q, abb, aSbb) \overset{consumo"a"}{\vdash} (q, bb, Sbb) \overset{espando\ S\to\epsilon}{\vdash} (q, bb, bb)$$
$$\overset{consumo"b"}{\vdash} (q, b, b) \overset{espando\ S\to aSb}{\vdash} (q, \epsilon, \epsilon)$$

Se adesso faccio questo procedimento: leggo ogni mossa fatta e, se è un espando il mio nodo si dirama nell'espansione, se invece è consumo è una foglia e quindi non faccio niente. Ottengo quindi il seguente albero di derivazione canonico sinistro:



*Notiamo che questo parser è **molto non deterministico**, infatti al passaggio in cui passo dalla configurazione $(q, bb, Sbb) \vdash (q, bb, bb)$ sarei anche potuto passare alla configurazione $(q, bb, aSbb)$ che ovviamente è sbagliata. Per risolvere ciò si usa il look ahead, cioè guardo il prossimo simbolo in input e, notando che è "b", posso capire che l'espansione di S in aSb sarebbe errata, per questo scelgo $S \to \epsilon$.*

La grammatica dell'esercizio di sopra vedremo che sarà di classe $LL(1)$, significando che senza look ahead di un simbolo il nostro parser sarà sempre non deterministico. La doppia L sta ad indicare che è un parser che legge da sinistra verso destra ("Left to right") e produce una derivazione leftmost.

Esempio 15.1.2 (). *Una grammatica $S \to aSb \mid ab$ è invece di classe $LL(2)$ infatti un singolo simbolo di look ahead non basta per decidere quale delle due produzioni usare, invece con due simboli:*

- *Se leggo "aa" espando in aSb, perché è l'unica che può dare altre "a" oltre la prima.*
- *Se leggo "ab" espando in ab.*

15.1.1 Grammatiche non adatte

Esistono grammatiche che non sono per nulla adatte al top down parsing, questo perché possiamo aggiungere tantissimo look ahead ma non basterà mai per rendere il parsing non deterministico. Un esempio di questo problema sono le **grammatiche left recursive**, cioè in cui un non terminale si espande in lui stesso seguito da altro (in realtà dopo daremo una definizione migliore).

Esempio 15.1.3 (). *Una grammatica del tipo:*

$$S \to Sb \mid a$$

che produce il linguaggio $L(G) = ab^*$ è left recursive (S si espande infatti in Sb). Vedremo in seguito tecniche per manipolare la grammatica togliendoli la ricorsione sinistra, in questo caso otterremo:

$$S \to aA$$
$$A \to bA \mid \epsilon$$

15.2 Introduzione parser bottom up

Definizione : *Data una grammatica libera $G = (NT, T, R, S)$ costruiamo un PDA M che riconosce $L(G)\$$ con $M = (T, \{q\}, T \cup NT \cup \{z\}, \delta, q, z, \emptyset)$., dove la funzione δ è cosi definita:*

$$(q, aX) \in \delta(q, a, X) \quad \forall a \in T, \forall x \in T \cup NT \cup \{z\} \qquad \textit{(shift)}$$
$$(q, A) \in \delta(q, \epsilon, \alpha^R) \quad \textit{se } A \to \alpha \in R \qquad \textit{(reduce)}$$
$$(q, \epsilon) \in \delta(q, \$, SZ) \qquad \textit{(accept)}$$

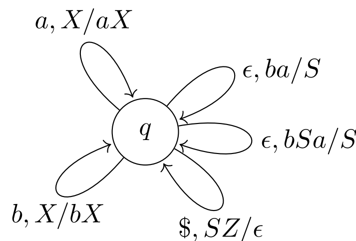
Notiamo quindi che abbiamo tre tipi di transizioni:

- Le transizioni shift che corrispondono allo spostamento di un simbolo dall'input sulla pila.
- Le transizione reduce che corrispondono alla riduzione di una sequenza nel non terminale che si espande in essa (cioè noi abbiamo α^R nello stack e una produzione $A \to \alpha$ quindi sostituiamo α^R con A). Il fatto che sia α^R è perché nello stack è tutto al contrario.
- Le transizioni accept che corrispondono alla fine della lettura.

Esempio 15.2.1 (). *Data la grammatica:*

$$S \to aSb \mid ab$$

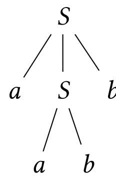
seguendo il procedimento di sopra costruiremo un PDA fatto così:



Nel caso di un input del tipo $aabb\$$ il parser si comporterà cosi:

Stack	*Input*	*Azione*
Z	*aabb\$*	*shift*
Za	*abb\$*	*shift*
Zaa	*bb\$*	*shift*
Zaab	*b\$*	*reduce* $S \to ab$
ZaS	*b\$*	*shift*
ZaSb	*\$*	*reduce* $S \to aSb$
ZS	*\$*	*accept*
ϵ	ϵ	

Guardando le azioni reduce possiamo trovare il seguente albero di derivazione canonico destro:



15.2.1 Non determinismo e conflitti

In questo parser c'è ancora molto non determinismo, infatti possiamo trovare spesso conflitti:

- Conflitti shift-reduce, in cui possiamo fare sia shift sia reduce.
- Conflitti reduce reduce, in cui possiamo fare più reduce diverse.

Per risolvere ciò ci servirà avere un DFA particolare chiamato **DFA dei prefissi viabili** e **look ahead**.

15.2.2 Problema con produzioni epsilon

Le produzioni epsilon hanno il problema che rendono una reduce sempre applicabile, e per questo rendono non deterministico il parser. Esistono metodi (che vediamo nel capitolo dopo) che eliminano le produzioni epsilon dalla grammatica.

Chapter 16

Semplificazione di grammatiche

16.1 Eliminare le produzioni epsilon

In questa sezione vogliamo, partendo da una grammatica G libera con produzioni ϵ (quindi del tipo $A \rightarrow \epsilon$), ottenere una grammatica G' senza produzioni ϵ tale che:

$$L(G') = L(G) \setminus \{\epsilon\}$$

Notiamo che se vogliamo mantenere lo stesso linguaggio iniziale (quindi con anche ϵ dentro) basta creare una grammatica G'' uguale a G' con un simbolo extra S' e una produzione extra $S' \rightarrow \epsilon|S$.

16.1.1 Simboli annullabili

Definizione : *Un non terminale $A \in NT$ è detto annullabile se:*

$$A \Rightarrow^{+} \epsilon$$

cioè se in n passi si può riscrivere in ϵ. Indichiamo con $N(G) = \{A \in NT \mid A \Rightarrow^{+} \epsilon\}$ tutti i simboli annullabili di una grammatica G.

Viene calcolato induttivamente in questo modo:

$$N_0(G) = \{A \in NT \mid A \rightarrow \epsilon \in R\}$$
$$N_{i+1}(G) = N_i(G) \cup \{B \in NT \mid B \rightarrow C_1...C_k \in R \wedge C_1, .., C_k \in N_i(G)\}$$

cioè al passo $i+1$ aggiungiamo il simbolo non terminale B se esiste una produzione composta da soli simboli annullabili (in N_i). Notiamo che:

$$\exists i_c \text{ tale che } N_{i_c}(G) = N_{i_c+1}(G)$$

vale inoltre che:

$$N_{i_c}(G) = N(G)$$

quindi ci basta trovare un valore i_c per cui per un iterazione l'insieme non cambia che abbiamo finito di calcolare i simboli annullabili.

16.1.2 Algoritmo

Calcoliamo prima i simboli annullabili $N(G)$ della grammatica $G = (NT, T, S, R)$ e costruiamo poi la grammatica $G = (NT, T, S, R')$, dove l'unica cosa diversa sono le produzioni. Per costruire le produzioni diciamo che per ogni produzione $A \rightarrow \alpha \in R$ con $\alpha \neq \epsilon$, in cui occorrono i simboli annullabili $Z_1, ..., Z_k$ mettiamo in R' tutte le produzioni $A \rightarrow \alpha'$ dove α' si ottiene da α cancellando tutti i possibili sottoinsiemi di $Z_1, ..., Z_k$ (incluso $\emptyset$, quindi non cancello nulla in quel caso), tranne nel caso in cui, dopo aver cancellato il sottoinsieme, risulta che $\alpha' = \alpha$.

Quello che stiamo facendo è quindi "simulare" l'annullamento dei non terminali dentro ad α, infatti stiamo prendendo ogni possibile forma di α in cui parte dei non terminali annullabili si sono annullati. Dobbiamo appunto "simularlo" perché i non terminali non si potranno più annullare per conto loro perché le produzioni ϵ sono state tolte.

Teorema correttezza: *Data una grammatica libera G, la grammatica G' ottenuta dall'algoritmo di sopra non ha produzioni ϵ e vale che:*

$$L(G') = L(G) \setminus \{\epsilon\}$$

Esempio 16.1.1 (). *Sia data la grammatica G seguente:*

$$\begin{aligned} S &\to AB \\ A &\to aAA \mid \epsilon \\ B &\to bBB \mid \epsilon \end{aligned}$$

I simboli annullabili di questa grammatica sono:

$$N(G) = \{A, B, S\}$$

infatti $N_0(G) = \{A, B\}$ e $N_1(G) = \{A, B, S\}$. Consideriamo adesso la produzione $S \to AB$ e "simuliamo" l'annullamento dei sottoinsiemi di simboli annullabili:

$$\begin{aligned} &(\emptyset) \Rightarrow S \to AB \\ &(A) \Rightarrow S \to B \\ &(B) \Rightarrow S \to A \\ &\cancel{(A, B) \Rightarrow S \to \epsilon} \end{aligned}$$

L'ultima produzione non la prendiamo perché abbiamo $\alpha' = \epsilon$. vediamo ora la produzione $A \to aAA \in R$:

$$\begin{aligned} &(\emptyset) \Rightarrow A \to aAA \\ &(A) \Rightarrow A \to aA \\ &(A) \Rightarrow A \to aA \\ &(A, A) \Rightarrow A \to a \end{aligned}$$

In questo caso abbiamo che la seconda e terza produzione sono un duplicato e quindi le prendiamo solo una volta. Vediamo quelle di B:

$$\begin{aligned} &(\emptyset) \Rightarrow B \to bBB \\ &(B) \Rightarrow B \to bB \\ &(B) \Rightarrow B \to bB \\ &(B, B) \Rightarrow B \to b \end{aligned}$$

Ottenendo quindi una grammatica G' con queste produzioni:

$$\begin{aligned} S &\to AB \mid B \mid A \\ A &\to aAA \mid aA \mid a \\ B &\to bBB \mid bB \mid b \end{aligned}$$

16.2 Eliminare le produzioni unitarie

In questa sezione vogliamo, partendo da una grammatica G libera con produzioni unitarie del tipo $A \to B$ con $A, B \in NT$, ottenere una grammatica G' senza esse.

16.2.1 Coppie unitarie

Definizione : *Una coppia* (A, B) *con* $A, B \in NT$ *è detta coppia unitaria se:*

$$A \Rightarrow^* B$$

cioè se in n passi A *si riscrive solo in* B*. Definiamo poi l'insieme* $U(G) = \{(A, B) \mid A, B \in NT \wedge A \Rightarrow^* B\}$.

Dalla definizione data abbiamo che le coppie identiche (A, A) sono coppie unitarie. Definiamo in modo induttivo l'insieme:

$$U_0(G) = \{(A, A) \mid A, A \in NT\}$$
$$U_{i+1}(G) = U_i(G) \cup \{(A, C) \mid (A, B) \in U_i(G) \wedge B \Rightarrow C \in R\}$$

Cioè al passo $i + 1$-esimo aggiungiamo la coppia (A, C) se (A, B) è coppia unitaria in U_i e in un passo B si può riscrivere totalmente in C. Valgono le stesse osservazioni di prima sul i_c.

16.2.2 Algoritmo

Data la grammatica $G = (NT, T, R, S)$ libera, definiamo la grammatica $G' = (NT, T, R', S)$ dove:

$$\forall (A, B) \in U(G) \quad A \to \alpha \in R' \quad \text{dove } B \to \alpha \in R$$

cioè invece di fare riscrivere A in B in n passi e poi B in α riscriviamo direttamente A in α. Notiamo che R' contiene tutte le produzioni non unitarie perché $(A, A) \in U(G)$ e quindi:

$$\text{se } A \to \alpha \in R \text{ allora } A \to \alpha \in R'$$

Teorema correttezza: *Sia* $G = (NT, T, R, S)$ *libera e sia* $U(G)$ *l'insieme delle sue coppie unitarie allora la grammatica* G' *ottenuta con l'algoritmo di sopra non ha produzioni unitarie e:*

$$L(G) = L(G')$$

Esempio 16.2.1 (). *Sia* G *la grammatica con le seguenti produzioni:*

$$E \to E + T \mid T$$
$$T \to T \cdot A \mid A$$
$$A \to a \mid b \mid (E)$$

calcoliamo ora l'insieme $U(G)$ *come segue:*

$$U_0(G) = \{(E, E), (T, T), (A, A)\}$$
$$U_1(G) = U_0 \cup \{(E, T), (T, A)\}$$
$$U_2(G) = U_1 \cup \{(E, A)\} = U(G)$$

Seguendo l'algoritmo sopra andiamo a creare le produzioni date da U_0, U_1, U_2*, iniziando da* U_0*:*

$$E \to E + T$$
$$T \to T \cdot A$$
$$A \to a \mid b \mid (E)$$

notiamo che queste sono tutte le produzioni che non sono unitarie, però a queste mancano ancora alcuni infatti con solo queste produzioni non produrremo lo stesso linguaggio. Aggiungiamo adesso quelle date da U_1*:*

$$E \to E + T \mid T \cdot A$$
$$T \to T \cdot A \mid a \mid b \mid (E)$$
$$A \to a \mid b \mid (E)$$

aggiungiamo poi quelle di U_2:

$$
\begin{aligned}
&E \to E + T \mid T \cdot A \mid a \mid b \mid (E) \\
&T \to T \cdot A \mid a \mid b \mid (E) \\
&A \to a \mid b \mid (E)
\end{aligned}
$$

16.3 Rimuovere i simboli inutili

Definizione : *Un simbolo $X \in T \cup NT$ è:*

- *Un **generatore** se e solo se $\exists w \in T^*$ con $X \Rightarrow^* w$. Cioè se in n passi X si riscrivere in una stringa.*
- ***Raggiungibile** se e solo se $S \Rightarrow^* \alpha X \beta$ per qualche $\alpha, \beta \in (T \cup NT)^*$. Cioè se posso raggiungere X partendo da S.*
- ***Utile** se e solo se è sia generatore sia raggiungibile, cioè se $S \Rightarrow^* \alpha X \beta \Rightarrow^* z \in L(G)$, cioè X comparare in almeno una derivazione di una stringa del linguaggio.*

16.3.1 Come calcolare i generatori

Li si calcola induttivamente nel seguente modo:

$$
\begin{aligned}
&G_0(G) = T \\
&G_{i+1}(G) = G_i(G) \cup \{B \in NT \mid B \to C_1...C_k \in R \wedge C_1, ..., C_k \in G_i(G)\}
\end{aligned}
$$

Notiamo che nel punto due è contemplato anche la produzione $B \to \epsilon$ (nel caso di $k = 0$). Si possono fare gli stessi ragionamenti di i_c.

16.3.2 Come calcolare i raggiungibili

Di nuovo li si calcola induttivamente:

$$
\begin{aligned}
&R_0(G) = \{S\} \\
&R_{i+1}(G) = R_i(G) \cup \{x_1, ..., x_k \mid B \to x_1...x_k \in R \wedge B \in R_i(G)\}
\end{aligned}
$$

16.3.3 Algoritmo

Per potere eliminare tutti i simboli inutili quello che faccio è:

- Cancello tutti i simboli che non sono generatori e tutte le produzioni che usavano uno di questi simboli.
- Cancello tutti i simboli che non sono raggiungibili e tutte le produzioni che usavano uno di questi simboli.

L'ordine delle operazioni è importante sennò può capitare che non elimino tutti i simboli inutili.

Teorema correttezza: *Sia $G = (NT, T, R, S)$ una grammatica libera tale che $L(G) \neq \emptyset$ allora:*

- *Sia G_1 la grammatica ottenuta da G eliminando tutti i simboli che non sono generatori e tutte le produzioni che ne avevano almeno uno all'interno.*
- *Sia G_2 la grammatica ottenuta da G_1 eliminando tutti i simboli non raggiungibili e tutte le produzioni che ne avevano almeno uno all'interno.*

Allora G_2 non ha simboli inutili e $L(G) = L(G_2)$.

Dimostrazione: *Dobbiamo dimostrare che $L(G) \subseteq L(G_2)$ e che $L(G_2) \subseteq L(G)$, dimostriamolo in due parti:*

- $L(G_2) \subseteq L(G)$ *è ovvio perché G_2 contiene meno produzioni di G.*
- $L(G) \subseteq L(G_2)$*, per dimostrarlo dobbiamo dimostrare che se $S \Rightarrow^*_G w$ allora $S \Rightarrow^*_{G_2} w$, ma è sicuramente vero perché ogni simbolo usato nella derivazione $S \Rightarrow^*_G w$ è sia raggiungibile e sia generatore e quindi non eliminato in G_2.*

Esempio 16.3.1 (). *Nella grammatica seguente:*

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

Abbiamo che i generatori sono:

$$G(G) = \{S, B, a, b\}$$

quindi prima eliminiamo i non generatori (A) e le produzioni ad essa legate:

$$S \to a$$
$$B \to b$$

adesso abbiamo che i raggiungibili sono:

$$R(G) = \{a\}$$

Quindi andiamo a togliere tutti i simboli non raggiungibili e le produzioni in cui conmpaiono essi:

$$S \to a$$

Esempio 16.3.2 (). *Consideriamo la grammatica seguente:*

$$S \to AB \mid aC$$
$$A \to a$$
$$B \to bB$$
$$C \to b \mid AC$$
$$D \to a \mid aS$$

Abbiamo che i generatori sono:

$$G(G) = \{a, b, A, C, S, D\}$$

quindi eliminiamo tutti i non generatori (B) e le produzioni ad essa legate:

$$S \to aC$$
$$A \to a$$
$$C \to b \mid AC$$
$$D \to a \mid aS$$

adesso abbiamo che i raggiungibili sono:

$$R(G) = \{a, b, C, S, A\}$$

Quindi andiamo a togliere tutti i simboli non raggiungibili (D) e le produzioni in cui conmpaiono essi:

$$\begin{aligned} &S \to aC \\ &A \to a \\ &C \to b \mid AC \end{aligned}$$

16.4 Mettere tutto insieme

Per mettere insieme questi primi tre passaggi va seguito un ordine ben preciso, sennò si rischia di non ottenere il risultato giusto:

- Eliminare le produzioni ϵ.
- Eliminare le produzioni unitarie.
- Eliminare i simboli inutili.

Esempio 16.4.1 (). *Partiamo dalla grammatica seguente:*

$$\begin{aligned} &S \to aAa \\ &A \to C \\ &C \to S \mid \epsilon \end{aligned}$$

Partiamo togliendo le produzioni ϵ. Calcoliamo prima gli annullabili:

$$N(G) = \{C, A\}$$

andiamo quindi a fare l'algoritmo di rimozione delle produzioni epsilon ottenendo:

$$\begin{aligned} &S \to aAa \mid aa \\ &A \to C \\ &C \to S \end{aligned}$$

Passiamo ora a rimuovere le produzioni unitarie, calcoliamo prima le coppie unitarie:

$$U(G') = \{(S, S), (A, A), (C, C), (A, C), (C, S), (A, S)\}$$

applichiamo l'algoritmo ottenendo:

$$\begin{aligned} &S \to aAa \mid aa \\ &A \to aAa \mid aa \\ &C \to aAa \mid aa \end{aligned}$$

rimuoviamo ora i simboli inutili, calcoliamo generatori:

$$G(G'') = \{S, A, C, a\}$$

sono tutti generatori quindi calcoliamo i raggiungibili:

$$R(G'') = \{a, A, S\}$$

cancelliamo quindi C:

$$\begin{aligned} &S \to aAa \mid aa \\ &A \to aAa \mid aa \end{aligned}$$

16.5 Forme normali

Esistono delle forme di grammatica che sono dimostrate garantire proprietà particolare alla grammatica. Di queste ne troviamo due:

- **Forma normale di Chomsky** in cui tutte le produzioni sono del tipo:

  $$A \rightarrow BC \qquad A \rightarrow a$$

  con il simbolo iniziale che può avere anche una produzione ϵ, inoltre S non compare mai a destra in una produzione. Se abbiamo che una grammatica libera è nella forma normale id Chomsky allora non ha ne produzioni epsilon ne produzioni unitarie.

- **Forma normale di Graibach** in cui tutte le produzioni sono della forma:

  $$A \rightarrow aBC \qquad A \rightarrow aB \qquad A \rightarrow a$$

  (con S che può avere produzioni epsilon). IN questo caso una grammatica libera non ha ne produzioni epsilon, ne produzioni unitarie ne ricorsività sinistra.

16.6 Eliminare la ricorsione sinistra

Definizione : *Una produzione del tipo:*

$$A \rightarrow A\alpha \quad \alpha \in (T \cup NT)^*$$

è detta ricorsiva sinistra perché il non terminale si riscrive in se stesso con a destra qualcosa. Questo tipo di ricorsione è detto ***ricorsione immediata***

Definizione : *Una grammatica si dice ricorsiva sinistra se:*

$$A \Rightarrow^+ A\alpha \in R \quad A \in NT \ \alpha \in (T \cup NT)^*$$

Questo tipo di ricorsione è detto ***ricorsione non immediata****.*

La seconda definizione è più generale infatti basta che A si riscriva in n passi in $A\alpha$ (mentre la prima era solo per un passo).

16.6.1 Come rimuovere la ricorsione immediata

La produzione immediata è facile da risolvere, infatti se abbiamo che:

$$A \rightarrow A\alpha_1 \mid .. \mid A\alpha_n \mid \beta_1 \mid .. \mid \beta_m \in R$$

dove le stringhe β_i non incominciano con A, allora possiamo rimuovere la ricorsione immediata rimpiazzando quelle produzioni con:

$$A \rightarrow \beta A' \mid .. \mid \beta_m A'$$
$$A' \rightarrow \alpha_1 A' \mid .. \mid \alpha_n A' \mid \epsilon$$

Notiamo che la produzione $A' \rightarrow \epsilon$ va aggiunta perché simula il caso in cui scegliamo subito $A \rightarrow \beta_i$.

Esempio 16.6.1 (). *Partendo dalla grammatica:*

$$A \rightarrow Ab \mid Ac \mid d$$

applicando l'algoritmo otteniamo:

$$A \to dA'$$
$$A' \to bA' \mid cA' \mid \epsilon$$

Nel caso di una grammatica con solo una produzione del tipo:

$$A \to Aa$$

Non possiamo applicare l'algoritmo perché ci mancano le produzioni di "base" (quelle β).

16.6.2 Come rimuovere la ricorsione non immediata

L'algoritmo che siamo per dare prende in input una grammatica senza ϵ-produzioni, senza produzioni unitarie, ma con ricorsione sinistra non-immediata. L'output di tale algoritmo invece è una grammatica senza ricorsione sinistra ma con possibilità di avere ϵ-produzioni.

Algorithm 4: Eliminazione ricorsione sx non immediata

1. Sia $NT = \{A_1, A_2, ..., A_n\}$ in un ordine fissato ;
2. **for** i **from** 1 **to** n{
3.   **for** j **from** 1 **to** $i-1${
4.   sostituisci ogni produzione della forma $A_i \to A_j\alpha$
5.   con le produzioni $A_i \to \beta_1\alpha \mid .. \mid \beta_k\alpha$ dove $A_j \to \beta_1 \mid .. \mid \beta_k$;
6.   }
7.   Elimina la ricorsione immediata su A_i ;
8. }

Quello che sta facendo l'algoritmo è un po' un mix della cosa delle produzioni unitarie e della ricorsione immediata:

- Prima di tutto, similmente a quello delle produzioni unitarie, faccio una espansioni dei non terminali a destra delle produzioni, cercando di ricondurmi ad un caso di ricorsione immediata.
- Nel caso si sia creata ricorsione immediata vado a risolverla sul singolo non terminale.

Esempio 16.6.2 (). *Data la grammatica:*

$$S \to Ba \mid b$$
$$B \to Bc \mid Sc \mid d$$

che possiamo dire essere ricorsiva sx sia immediata ($B \to Bc$) sia non ($B \Rightarrow Sc \Rightarrow Bac$). Siano i non terminali in ordine $NT = \{S, B\}$ eseguiamo l'algoritmo:

- *Per $i = 1$ e con $A_i = S$ non eseguo il ciclo interno e non risolvo la ricorsione immediata che non c'è su S.*
- *Per $i = 2$ e quindi con $A_i = B$ il ciclo interno viene eseguito solo per $A_j = S$. Quindi tutte le produzioni della forma $B \to S\alpha$ vengono rimpiazzate. In particolare l'unica che va cambiata è $B \to Sc$ e viene rimpiazzata con:*

  $$B \to Bac \mid bc$$

  Va poi risolta la ricorsione immediata:

  $$B \to bcB' \mid dB'$$
  $$B' \to cB' \mid acB' \mid \epsilon$$

16.7 Fattorizzazione a sinistra

Questo tipo di tecnica è importante per il top down parsing, visto che ci permette di ridurre il numero di simboli di look ahead necessario per il parser. L'idea dietro è di raccogliere le parti comuni di più produzioni.

Algorithm 5: Fattorizzazione a sinistra

1 Inizializza $N = NT$;
2 Sia $NT = \{A_1, A_2, ..., A_n\}$ in un ordine fissato ;
3 **while** si può modificare N o modificare l'insieme delle produzione{
4   **foreach** $A \in N${
5   Sia α il prefisso più lungo comune alle parti destre di alcune produzioni di A ;
6   **if** $\alpha \neq \epsilon$ {
7   Sia A' un nuovo non terminale ;
8       $N = N \cup \{A'\}$;
9   Rimpiazza tutte le produzioni per A, $A \to \alpha\beta_1 \mid .. \mid \alpha\beta_k \mid \gamma_1 \mid .. \mid \gamma_h$ con le produzioni:;
10         $A \to \alpha A' \mid \gamma_1 \mid .. \mid \gamma_h$
11         $A' \to \beta_1 \mid .. \mid \beta_k$
12   }
13   }
14 }

Esempio 16.7.1 (). *Data la grammatica*

$$
\begin{aligned}
&E \to T \mid T + E \mid T - E \\
&T \to A \mid A \cdot T \\
&A \to a \mid b \mid (E)
\end{aligned}
$$

seguendo l'algoritmo otteniamo:

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

Chapter 17

Parser top down

Esistono diversi tipi di parser top down, il primo che vediamo è quello a discesa ricorsiva che, implicitamente, utilizza una pila per gestire le chiamate ricorsive.

17.1 Parser a discesa ricorsiva

Data una grammatica libera $G = (NT, T, S, R)$ per ogni non terminale A con produzioni:

$$A \to X_1^1..X_{n_1}^1 \mid .. \mid X_1^k..X_{n_k}^k$$

(cioè con k produzioni diverse) definisco la funzione:

Algorithm 6: Parser a discesa ricorsiva

1 **function** A() {
2 Scegli non deterministicamente h tra 1 e k, ovvero una delle k produzioni;
3 **for** $i = 1$ **to** n_h{
4 **if** $x_i^h \in NT$
5 X_i^h();
6 **else if** $x_i^h =$ simbolo corrente dell'input
7 avanza di un simbolo sull'input ;
8 **else**
9 fail / backtracking;
10 }
11 **return** ;
12 }

Quello che sto facendo quindi è partendo da una produzione andare a ricercare, dentro una delle produzioni scelte a caso del simbolo, se c'è:

- Un non terminale, allora in questo caso, ricorsivamente, rieseguo la ricerca sulle sue produzioni.
- Se ho un simbolo uguale a quello corrente dell'input allora lo consumo.
- Se ho un simbolo diverso da quello dell'input faccio backtracking perché ho sbagliato produzione.

Questo tipo di algoritmo è **fortemente non deterministico**, infatti nel caso peggiore deve guardare tutte le possibili combinazioni. L'idea per migliorarlo e guardare anche il prossimo carattere in input.

Esempio 17.1.1 (). *Data la grammatica $S \to ac \mid aSb$ e input "aacb", iniziamo invocando la funzione sul simbolo iniziale:*

- *Guardiamo prima la produzione $S \to ac$, visto che x_i^h con $i = 0$ è uguale al nostro simbolo in input allora consumiamo esso. Andando avanti vediamo che $x_i^h \neq a$ e quindi facciamo backtracking.*
- *Proviamo ora con la produzione $S \to aSb$, consumiamo "a" e, visto che troviamo S ricorsivamente richiamiamo la funzione:*

- *Guardiamo prima la produzione $S \to aSb$, consumiamo "a" (adesso ci rimane "cb"), e ricorsivamente richiamerebbe di nuovo la funzione. Non lo faccio perché porta, in ogni caso, a backtrackin.*
- *Guardiamo la produzione $S \to ac$, consumiamo "a" (ci rimane adesso "cb") consumiamo "b" e facciamo return, perché abbiamo già letto tutta la produzione.*

Adesso come input ci rimane "b" e possiamo consumarlo visto che $X_3^h =$ "b".

17.2 First e follow

Per poter andare ad analizzare il simbolo successivo per rendere più deterministico il parser, andiamo a creare due nuove funzioni. Prima di ciò però diciamo che ogni linguaggio che andremmo a vedere sarà in realtà $L \cdot \$$, perché è necessario che goda della prefix property.

17.2.1 First

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

$$a \in T,\ a \in First(\alpha) \quad sse \quad \alpha \Rightarrow^* a\beta \qquad per\ \beta \in (T \cup NT)^*,\ a \in T$$
$$inoltre\ se\ \alpha \Rightarrow^* \epsilon\ allora\ \epsilon \in First(\alpha)$$

Algorithm 7: Calcolo di first di una grammatica

1 Per ogni $x \in T$, $First(x) = \{x\}$;
2 Per ogni $X \in NT$, inizializza $First(X) = \emptyset$;
3 **While** qualche $First(X)$ viene modificato: {
4 Per ogni produzione $X \to Y_1..Y_k$;
5 **for** $i = 1$ **to** n{
6 **if** $y_1, .., y_{i-1} \in N(G)$
7 $First(X) = First(X) \cup (First(y_i) \setminus \{\epsilon\})$;
8 }
9 Per ogni $X \in N(G)$, $First(X) = First(X) \cup \{\epsilon\}$;
10 }

Notiamo che prima di tutto l'insieme $N(G)$ è l'insieme dei simboli annullabili della grammatica. Quello che stiamo facendo in questo algoritmo è :

- Dentro il for è che stiamo guardando da che simbolo Y_i può iniziare la produzione, infatti se i primi i simboli si possono annullare, allora il first è dato anche dal first di Y_i (perché appunto possiamo avere $X \Rightarrow^* Y_i...Y_k$, ma comunque ci saranno anche derivazioni $X \Rightarrow^* Y_h...Y_k$ con $h < i$, cioè non è per forza detto che i primi i simboli si annullino!).
- Inoltre nel caso in cui X stesso si può annullare ($A \Rightarrow^* \epsilon$) per definizione $\epsilon \in First(A)$.

Un **modo più veloce ancora per calcolarlo** ed esteso per stringhe $\alpha \in (T \cup NT)^*$ è il seguente:

$$First(\epsilon) = \{\epsilon\}$$
$$First(X\beta) = First(X) \qquad \text{se } X \notin N(G)$$
$$First(X\beta) = (First(X) \setminus \{\epsilon\}) \cup First(\beta) \qquad \text{se } X \in N(G)$$

Ottenendo quindi che se $A \to \alpha_1 \mid .. \mid \alpha_k$ allora:

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

Esempio 17.2.1 (). *Data la grammatica:*

$$S \to Ab \mid C$$
$$A \to aA \mid \epsilon$$

Possiamo calcolare:

$$First(S) = First(Ab) \cup First(c) = (First(A) \setminus \{\epsilon\}) \cup First(b) \cup First(c) =$$
$$= \{a\} \cup \{b\} \cup \{c\} = \{a, b, c\}$$
$$First(A) = First(aA) \cup First(\epsilon) = \{a\} \cup \{\epsilon\} = \{a, \epsilon\}$$

Esempio 17.2.2 (). *Data la grammatica:*

$$SABC \mid CB$$
$$A \rightarrow a \mid \epsilon$$
$$B \rightarrow b \mid \epsilon$$
$$C \rightarrow c$$

Abbiamo che $N(G) = \{A, B\}$, possiamo quindi calcolare: Possiamo calcolare:

$$First(S) = First(ABC) \cup First(CB) = (First(A) \setminus \{\epsilon\}) \cup First(BC) \cup First(C) =$$
$$= \{a\} \cup (First(B) \setminus \{\epsilon\}) \cup First(C) \cup \{c\} = \{a\} \cup \{b\} \cup \{c\} = \{a, b, c\}$$

17.2.2 Follow

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

$$a \in Follow(A) \quad sse \quad S \Rightarrow^* \alpha A a \beta \qquad per\ \alpha, \beta \in (T \cup NT)^*,\ a \in T$$
$$\$ \in Follow(A) \quad se \quad S \Rightarrow^* \alpha A$$

(nota che visto che $S \Rightarrow^* S$ abbiamo che $\$ \in Follow(S)$).

Esempio 17.2.3 (). *Data la grammatica:*

$$S \rightarrow Ab \mid c$$
$$A \rightarrow aA \mid \epsilon$$

dalla definizione data sopra abbiamo che:

$$Follow(S) = \{\$\}$$
$$Follow(A) = \{b\} \quad (perchè\ S \Rightarrow^* Ab)$$

Algorithm 8: Calcolo di $follow(X)$ con $X \in NT$

1. Per ogni $X \in NT$, inizializza $Follow(X) = \emptyset$;
2. $Follow(S) = \{\$\}$;
3. **While** qualche $Follow(X)$ viene modificato: {
4. Per ogni produzione $X \rightarrow \alpha Y \beta$ fai:
5. $Follow(Y) = Follow(Y) \cup (First(\beta) \setminus \{\epsilon\})$;
6. Per ogni produzione $X \rightarrow \alpha Y$ e per ogni produzione $X \rightarrow \alpha Y \beta$ con $\epsilon \in First(\beta)$ fai:
7. $Follow(Y) = Follow(Y) \cup Follow(X)$;
8. }

Esempio 17.2.4 (). *Data la grammatica:*

$$\begin{aligned} S &\to ABC \\ A &\to aaA \mid \epsilon \\ B &\to b \mid \epsilon \\ C &\to cC \mid \epsilon \end{aligned}$$

Creiamo una tabella con first e follow di ogni non terminale e calcoliamo i first:

	First	Follow
S	a, b, c, ϵ	
A	a, ϵ	
B	b, ϵ	
C	c, ϵ	

In questo caso io partirei calcolando il $Follow(S)$ che però, visto che S non compare a sinistra in nessuna produzione, abbiamo $Follow(S) = \{\$\}$. Il prossimo più semplice da calcolare è $Follow(C)$:

$$Follow(C) = Follow(C) \cup Follow(S) = \{\$\}$$

Calcoliamo poi il $Follow(B)$:

$$Follow(B) = Follow(B) \cup (First(C) \setminus \{\epsilon\}) \cup Follow(S) = \{c, \$\}$$

Calcoliamo poi il $Follow(A)$:

$$Follow(A) = Follow(A) \cup (First(BC) \setminus \{\epsilon\}) \cup Follow(S) = \{b, c, \$\}$$

Otteniamo quindi:

	First	Follow
S	a, b, c, ϵ	$\$$
A	a, ϵ	$b, c, \$$
B	b, ϵ	$c, \$$
C	c, ϵ	$\$$

17.3 Tabelle di parsing LL(1)

Le tabelle di parsing LL(1) sono delle matrici bidimensionali dove:

- Le righe sono i non terminali.
- Le colonne sono i terminali (incluso $).
- La casella $M[A, a]$ contiene le produzioni che possono essere scelte quando il parser tenta di espandere A e ha input corrente a.

Se ogni casella contiene al massimo una produzione allora il parser è deterministico (perché abbiamo massimo una scelta e quindi non è necessario backtracking). La **tabella viene riempita nel seguente modo**:

- Per ogni produzione $A \to \alpha$:
  - Per ogni $a \in T$ e $a \in First(\alpha)$, inserisco $A \to \alpha$ nella casella $M[A, a]$.
  - Se $\epsilon \in First(\alpha)$ inserisco $A \to \alpha$ in tutte le caselle $M[A, x]$ per $x \in Follow(A)$

E' possibile comunque che rimangano caselle vuote, in quel caso vanno considerate come caselle di errore.

17.3.1 Grammatiche LL(1)

Definizione : *Una grammatica G è LL(1) se e solo se ogni casella della tabella di parsing LL(1) contiene al massimo una produzione.*

Nel caso in cui G sia LL(1) il parser associato ad essa costruisce l'albero di derivazione per l'input w in modo top-down e predicendo quale produzione usare (e quindi evitando backtracking).

Teorema IMPORTANTE!: *Una grammatica G è LL(1) se e solo se per ogni coppia di produzioni distinte con stessa testa:*

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

si ha che:

- $First(\alpha) \cap First(\beta) = \emptyset$.
- *Se* $\epsilon \in First(\alpha)$ *allora* $First(\beta) \cap Follow(A) = \emptyset$.
- *Se* $\epsilon \in First(\beta)$ *allora* $First(\alpha) \cap Follow(A) = \emptyset$.

La prima delle condizioni è abbastanza ovvia, infatti se possiamo guardare solo un carattere in input per predirre ci serve che il primo carattere di ogni produzione sia diverso sennò non riusciremmo a distinguerle. I punti 2 e 3 dicono la stessa cosa, infatti nel caso in cui ϵ sia dentro un first il primo carattere di quella produzione può essere il carattere che viene dopo A (e quindi il $Follow(A)$) e per questo che non ci devono di nuovo essere caratteri in comune con l'altro first.

Esempio 17.3.1 (). *La grammatica:*

$$S \to A \mid B$$
$$A \to ab \mid cd$$
$$B \to ad \mid cb$$

questa grammatica non è LL(1) infatti:

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

Possiamo però manipolarla per renderla LL(1), prima espando in non terminali delle produzioni di S:

$$S \to ab \mid cd \mid ad \mid cb$$

poi fattorizzo con la tecnica della fattorizzazione:

$$S \to aT \mid cT'$$
$$T \to b \mid d$$

questa invece è LL(1).

17.3.2 Parser non ricorsivo che fa uso della pila

Abbiamo visto come il parser a discesa ricorsiva facesse uso implicitamente della pila per le chiamate ricorsive, invece adesso vediamo un tipo di parser che non utilizza chiamate ricorsive ma utilizza una pila di supporto per

gestire i simboli.

Algorithm 9: Parser LL(1) non ricorsivo

```
1  Pila = S$ ;
2  X = S;
3  input = W$);
4  i_c = primo carattere dell'input;
5  while (X ≠ $ ) {
6     if (X è un terminale){
7        if (X = i_c) {
8           pop X dalla pila;
9           avanza i_c sull'input;
10       }
11       else
12          errore()        // match errato;
13    }
14    else {        // stiamo guardando un non terminale
15       if (M[X, i_c] = X → y_1..y_n) {
16          pop X dalla pila;
17          push y_1, .., y_n sulla pila;
18          output la produzione X → y_1..y_n;
19       }
20       else {
21          errore()        // casella vuota;
22       }
23    }
24    X = top della pila;
25 }
26 if) (i_c ≠ $) errore()        // ho svuotato la pila ma non ho finito l'input;
```

Esempio 17.3.2 (). *Data la grammatica:*

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

Facciamo la tabella dei first e follow:

	First	*Follow*
S	a, b	$
A	a	b
B	b	$

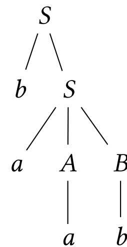
Attraverso l'algoritmo dato sopra costruisco la tabella LL(1):

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

Prendiamo poi in input una stringa "baab\$" e vediamo l'evoluzione della pila.

Input	*Stack*
baab$	*S$*
baab$	*bS$*
aab$	*S$*
aab$	*aAB$*
ab$	*AB$*
ab$	*aB$*
b$	*B$*
b$	*b$*
$	*$*

Ottenendo un albero di derivazione:



17.3.3 Linguaggi regolari LL(1)

Teorema : *Ogni linguaggio regolare è generabile da una grammatica G di classe LL(1).*

Dimostrazione: *Se L è regolare allora $\exists$ DFA $M = (Q, \Sigma, \delta, q_o, F)$ tale che $L = L[M]$. Seguendo la seconda tecnica data per ottenere una grammatica regolare da un DFA otteniamo una grammatica $G = (NT, T, S, R)$ con $NT = \{[q] \mid q \in Q\}, T = \Sigma, S = [q_0]$ e R definita come segue:*

$$\text{se } \delta(q, a) = q' \text{ allora } [q] \to a[q'] \in R$$
$$\text{se } q \in F \text{ allora } [q] \to \epsilon \in R$$

allora G è di classe LL(1), infatti visto che M è deterministico da ogni $q \in Q$ e per ogni $a \in \Sigma$ esiste un solo q' tale che $q \to^a q'$, cioè ci sarà solo una produzione $[q] \to^a [q']$ che inizia per "a". Inoltre se q è finale allora $[q] \to \epsilon$ è applicabile solo per i $Follow([q]) = \{\$\}$ e quindi non abbiamo conflitti.

17.4 Grammatiche LL(K)

Le grammatiche LL(k) sono grammatiche che richiedono più simboli di look ahead, per creare la tabella si utilizzano le funzioni uguali a prima ma in versione "k":

- $First_k(\alpha)$ definita come:

$$w \in First_k(\alpha) \text{ sse } \alpha \Rightarrow^* w\beta \text{ con } |w| = k, w \in T^*, \beta \in (T \cup NT)^*$$
$$\text{oppure } \alpha \Rightarrow^* w \text{ con } |w| \leq k, w \in T^*$$

- $Follow_k(A)$ definita come:

$$w \in Follow_k(A) \text{ sse } S \Rightarrow^* \alpha A w \beta \text{ con } |w| = k, w \in T^*, \alpha, \beta \in (T \cup NT)^*$$
$$\text{oppure } S \Rightarrow^* \alpha A w \text{ con } |w| \leq k, w \in T^*$$

17.4.1 Tabella di parsing LL(K)

La tabella di parsing LL(K) è molto simile a quella LL(1) solo che con terminali anche lunghi k nelle colonne:

- Le righe sono sempre i non terminali.
- Le colonne sono : $\{w \in T^* \mid |w| \leq k\}$ (mettiamo solo quelli utili, perché sennò ce ne sarebbero troppi, quelli utili sono solo quei w che appartengono ad un qualche $First_k$).

Viene creata nello stesso modo, infatti per ogni produzione $A \to \alpha$ abbiamo che $M[A, w]$ contiene:

- $A \to \alpha$ per ogni $w \in First_k(\alpha)$ con $w \neq \epsilon$ e per ogni $w \in Follow_k(A)$ se $\epsilon \in First_k(\alpha)$.

Come per le LL(1) se ogni casella contiene al massimo una produzione allora la grammatica è LL(K).

Esempio 17.4.1 (Es di massima difficoltà). *Consideriamo la grammatica:*

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

Calcoliamo i $First_2$ delle tre produzioni:

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

abbiamo che G è LL(2) perché:

$$First_2(aSb) \cap First_2(ab) = \emptyset$$
$$First_2(aSb) \cap First_2(c) = \emptyset$$
$$First_2(c) \cap First_2(ab) = \emptyset$$

e abbiamo una tabella LL(2):

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

17.5 Quando una grammatica è LL(K)

Teorema UTILI PER GLI ESERCIZI: *Si può dimostrare che:*

- *Una grammatica ricorsiva sinistra non è LL(K) per nessun k.*
- *Una grammatica ambigua non è LL(K) per nessun k.*
- *Se G è LL(K) per qualche K allora G non è ambigua.*
- *Se G è LL(K) allora $L(G)$ è libero deterministico (infatti i parser usano dei DPDA).*
- *Esiste L libero deterministico tale che non esiste G di classe LL(K) tale che $L = L(G)$.*

Definizione : *Un linguaggio L è di classe LL(K) se esiste G di classe LL(K) tale che $L = L(G)$.*

Teorema : *Per ogni $K \geq 0$ vale che:*

$$LL(K) \subset LL(K+1)$$

Chapter 18

Parser bottom up

I parser bottom up si basano su due operazioni fondamentali:

- **Mossa shift** in cui un simbolo non terminale viene spostato dall'input sulla pila.
- **Mossa reduce** in cui una serie di simboli sulla cima della pila corrispondono al reverse di una parte destra di una produzione allora quella serie di simboli viene sostituita con la testa della produzione.

I parser bottom up saranno dei parser LR (cioè leggo da sinistra a destra e creo una derivazione rightmost).

18.1 Parser shift reduce non deterministico

Questo tipo di parser prendono in input una grammatica libera G con simbolo iniziale S e una stringa $w \in T^*$ e restituiscono, se $w \in L(G)$, la derivazione rightmost a rovescio di essa. Funziona nel seguente modo:

- Inizializziamo la pila con "\$" e l'input con "w\$".
- Costruiamo il PDA $M = (T, \{q\}, T \cup NT \cup \{\$\}, \delta, \$, \emptyset)$ con solo uno stato e che riconosce per pila vuota (infatti gli stati finali non ci sono), dove:

$$\begin{aligned} &(q, aX) \in \delta(q, a, X) \quad \forall a \in T, \forall x \in \Gamma && \text{(SHIFT)} \\ &(q, A) \in \delta(q, \epsilon, \alpha^R) \quad \text{se } A \to \alpha \in R && \text{(REDUCE)} \\ &(q, \epsilon) \in \delta(q, \$, S\$) && \text{(ACCEPT)} \end{aligned}$$

  Nel caso si esegua un operazione di reduce in output mettiamo la produzione usata.

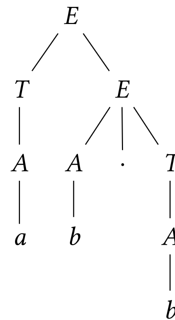
Esempio 18.1.1 (). *Data la grammatica:*

$$\begin{aligned} &E \to T \mid T + E \mid T - E \\ &T \to A \mid A \cdot T \\ &A \to a \mid b \mid (E) \end{aligned}$$

Cerchiamo la derivazione di "$a + b \cdot b\$$", seguendo le regole date sopra:

Pila	*Input*	*Azione*	*Output*
$	*a+b·b$*	*shift*	
$a	*+b·b$*	*reduce*	$A \to a$
$A	*+b·b$*	*reduce*	$T \to A$
$T	*+b·b$*	*shift*	
$T+	*b·b$*	*shift*	
$T+b	*·b$*	*reduce*	$A \to b$
$T+A	*·b$*	*shift*	
$T+A·	*b$*	*shift*	
$T+A·b	*$*	*reduce*	$A \to b$
$T+A·A	*$*	*reduce*	$T \to A$
$T+A·T	*$*	*reduce*	$T \to A \cdot T$
$T+T	*$*	*reduce*	$E \to T$
$T+E	*$*	*reduce*	$E \to T + E$
$E	*$*	*accept*	
ϵ	ϵ		

*Le scelte fatte **non sono deterministiche** infatti spesso potevamo fare sia shift sia reduce (vengono chiamati conflitti queste cose), per questo è detto parser non deterministico. Partendo dalla produzione più in basso in output ci possiamo poi ricavare un albero di derivazione ottenendo:*



18.1.1 Come risolvere i conflitti?

L'idea è che quando abbiamo la scelta fra due o più azioni dobbiamo scegliere quella più giusta, ovvero quella che fa in modo che sulla pila ci sia un prefisso viabile

Definizione : *Un prefisso viabile è una stringa $\gamma \in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom up per una computazione che accetta un input (cioè "qualcosa di corretto").*

Definizione alternativa: *Una stringa $\gamma \in (T \cup NT)^*$ è un prefisso viabile per G se e solo se esiste una derivazione rightmost:*

$$S \Rightarrow^* \delta A y \Rightarrow \delta\alpha\beta y = \gamma\beta y$$

*per qualche $y \in T^*, \delta \in (T \cup NT)^*$ e per una produzione $A \to \alpha\beta$. Inoltre S è n prefisso viabile per definizione, Un prefisso viabile si dice **completo** se $\beta = \epsilon$, in tal modo α è detta **maniglia** per γy (se abbiamo un prefisso viabile completo vuol dire che trovo sulla pila α^R e che quindi posso fare una reduce). Se invece abbiamo che $\beta \neq \epsilon$ abbiamo che β rappresenta ciò che mi manca per fare la reduce.*

Detto questo quello che veramente ci interessa è che la sequenza in cima alla pila sia un prefisso di una parte destra di una produzione (e che appunto ci manca ancora un pezzo di input per poter fare la reduce). Quindi per risolvere i conflitti bisogna fornire al parser una struttura di controllo, chiamata **tabella di parsing**, che lo renda deterministico.

18.2 Come è fatto un parser LR

Un parser LR è un parser che fa affidamento su tre elementi:

- Una **tabella di parsing** che serve per decidere che azione fare al passo successivo. Questa tabella di parsing è creata a partire da un DFA chiamato dei prefissi viabili. Contiene quattro tipi di mosse: shift, reduce, accept e goto.
- Una **pila di stati** che rappresenta in che stato siamo sul DFA dei prefissi viabili. E' necessaria sia una pila invece che un valore unico perché bisogna essere in grado di tornare indietro di n stati quando si fa la reduce.
- Una **pila dei simboli** della grammatica che è la solita pila che usiamo.

Definizione : *Una configurazione di un parser LR è una tripla:*

$$(s_0..s_n, x_1..x_m, a_i..a_k\$)$$

Dove il primo elemento è lo stack degli stati, il secondo lo stack dei simboli e il terzo è l'input rimanente.

Notiamo che la stringa "$x_1..x_m a_i..a_k\$$" è una stringa intermedia della derivazione canonica destra, infatti la parte $a_i..a_k\$$ è completamente sviluppata, mentre la parte $x_1..x_m$ contiene dei non terminali che vanno ancora espansi per ottenere una stringa del linguaggio (o magari non sono non terminali e la stringa è già una stringa del linguaggio).

18.2.1 Mosse del parser LR

Il parser LR come detto consulta la tabella di parsing per decidere che mossa fare. La cella letta della tabella dipende dallo stato attuale s_n e dall'attuale simbolo in input a_i. Consultando la cella $M[s_n, a_i]$ possiamo trovare:

- Se $M[s_n, a_i]$ = shift, allora la nuova configurazione è:

  $$(s_0..s_n s, x_1..x_m a_i, a_{i+1}..a_k\$)$$

  cioè andiamo a spostare un simbolo dall'input sullo stack dei simboli e ci spostiamo in un nuovo stato s (questo stato sarà di nuovo dato dalla tabella).
- Se $M[s_n, a_i]$ = reduce $A \to \beta$ allora la nuova configurazione è:

  $$(s_0..s_{n-r} s, x_1..x_{m-r} A, a_i..a_k\$)$$

  dove $r = |\beta|$ e s è dato dalla casella $M[s_{n-r}, A]$ = goto s (è sicuramente goto perché nelle colonne dei non terminali troviamo solo goto). Quindi quello che facciamo è eliminare (facendo pop della pila) r elementi dallo stack dei simboli sostituendoli con A e poi torniamo indietro di r stati e da quello stato ci calcoliamo il nuovo stato top (leggendo $M[s_{n-r}, A]$).

18.2.2 DFA dei prefissi viabili

Teorema : *Data G libera, i prefissi viabili di G costituiscono un linguaggio regolare e che può quindi essere descritto con un DFA (il DFA dei prefissi viabili).*

In base a come è fatto questo DFA (che eventualmente può usare informazioni di look-ahead e dei follow) il parser può risultare deterministico o meno. In teoria per determinare come sia fatto un prefisso viabile ogni volta che modifichiamo la pila dovremmo ripartire dal primo stato però, in pratica, non bisogna farlo perché ogni prefisso di un prefisso viabile è esso stesso un prefisso viabile e quindi abbiamo che:

- Se facciamo una azione di shift passiamo da avere sulla pila $\$\gamma$ a $\$\gamma x$. Ma visto che γ è prefisso di un prefisso viabile è lui stesso un prefisso viabile, quindi dando γ in input al DFA arriveremo ad uno stato e da quello stato ci basterà ripartire leggendo x invece che ripartire da capo.
- Anche per la reduce si può dire una cosa simile.

18.2.3 Item LR(0)

Definizione : *Un item LR(0) è una produzione con indicata, con un punto, una posizione della sua parte destra*

Da ogni produzione verranno prodotti più item, perché bisogna tener conto di ogni possibile posizione del ".". Inoltre item del tipo $A \to .\gamma$ vengono detti **item iniziali**

Esempio 18.2.1 (). *La produzione $A \to XYZ$ genera 4 item diversi:*

$$A \to .XYZ$$
$$A \to X.YZ$$
$$A \to XY.Z$$
$$A \to XYZ.$$

In questo caso l'item $A \to .XYZ$ è un item iniziale.

La posizione del "." indica fino a dove abbiamo già analizzato di questa produzione, per questo abbiamo che:

- Se l'item $A \to \alpha.\beta$ è nello stato attuale del DFA vuol dire che α è in cima sulla pila del simboli e che stiamo aspettando β.
- Se invece abbiamo $A \to \alpha.$ vuol dire che abbiamo letto tutta la produzione e possiamo fare una reduce.

18.2.4 Come costruire l'NFA dei prefissi viabili

Data la grammatica $G = (NT, T, S, R)$ libera, prendiamo la grammatica aumentata con un simbolo iniziale extra S' e una produzione extra:

$$S' \to S$$

L'NFA dei prefissi viabili si ottene poi cosi:

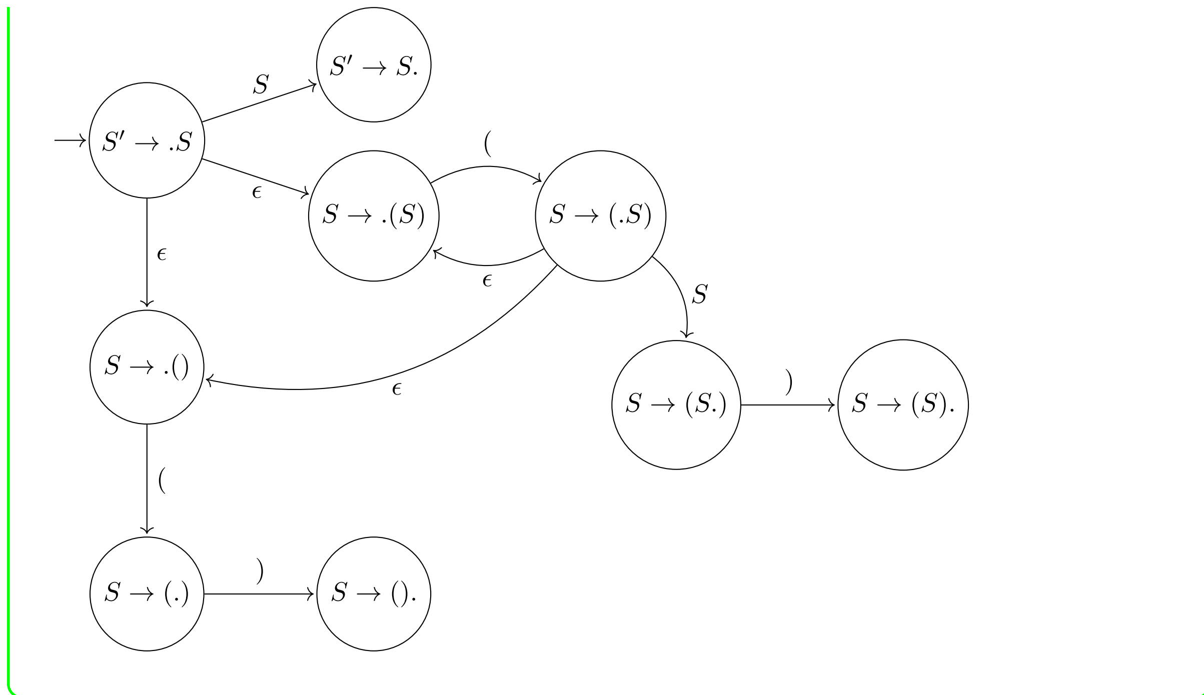
- L'item $[S' \to .S]$ è lo stato iniziale.
- Dallo stato $[A \to \alpha.X\beta]$ c'è una transizione verso lo stato $[A \to \alpha X.\beta]$ etichettata con X per $X \in T \cup NT$.
- Dallo stato $[A \to \alpha.X\beta]$ per $X \in NT$ e per ogni produzione $X \to \gamma$ c'è una transizione epsilon verso lo stato $[X \to .\gamma]$.

Gli stati finali non servono a nulla perché questo NFA serve solo come ausilio per il parser.

Esempio 18.2.2 (). *Data la grammatica aumentata:*

$$S' \to S$$
$$S \to (S)$$
$$S \to ()$$

seguendo la costruzione data sopra otteniamo un NFA dei prefissi viabili come questo:



18.3 Automa canonico LR(0)

L'automa canonico LR(0) è il DFA dei prefissi viabili. Per ottenerlo ci sono due metodi:

- Costruire prima l'NFA e poi con la costruzione di sottoinsiemi ottenere il DFA.
- In modo diretto usando due funzioni ausiliare chiamate $clos(I)$ e $Goto(I, X)$ dove I è un insieme di item e $X \in T \cup NT$.

18.3.1 Costruzione diretta dell'automa canonico LR(0)

Andiamo prima a definire le due funzioni $Clos(I)$ e $Goto(I, X)$ come:

Algorithm 10: Clos(I)

```
1 while I non è più modificato {
2   foreach item A → α.Xβ in I {
3     foreach produzione X → γ {
4       aggiungi X → .γ ad I ;
5     }
6   }
7 }
8 return I ;
```

Algorithm 11: Goto(I,X)

```
1 Inizializza J = ∅;
2 foreach item A → α.Xβ in I {
3   aggiungi A → αX.β a J;
4 }
5 return clos(J) ;
```

Possiamo notare che:

- $Clos(I)$ si comporta come una sorta di ϵ-closure rispetto all'NFA, infatti aggiungo a $Clos(I)$ tutti gli item che sarebbero stati raggiungibili nel NFA con mosse ϵ.
- $Goto(I, X)$ è la funzione che "fa scorrere i punti". Infatti crea il nuovo nodo con dove il punto si è mosso di una posizione e poi fa di nuovo la "ϵ-closure".

Date queste due funzioni adesso possiamo costruire l'automa DFA direttamente:

Algorithm 12: Costruzione DFA LR(0)

```
1  Inizializza 𝒮 = {clos({S' → .S})};
2  Inizializza δ = ∅;
3  while 𝒮 o δ non sono più modificati {
4     foreach I in 𝒮 {
5        foreach item A → α.Xβ in I {
6           Sia J = Goto(I, X);
7           Aggiungi J a 𝒮;
8           Aggiungi δ(I, X) = J a δ;
9        }
10    }
11 }
```

18.4 Tabella di parsing LR

La tabella di parsing LR è una tabella bidimensionale tale che:

- Le righe sono tutti gli stati dell'automa canonico LR(0).
- Le colonne sono tutti i simboli terminali più dollari, e inoltre ci sono colonne per ogni non terminale (loro avranno le azioni goto).

La tabella nella casella $M[s, X]$ contiene l'azione da fare con s in cima alla pila degli stati e X simbolo in input. Nel caso in cui la casella $M[s, X]$ sia vuota allora è un caso di errore, mentre se una casella ha più azioni allora c'è conflitto (e il parser non è quindi deterministico).

18.4.1 Come riempirla

La tabella di parsing si riempe in base all'automa canonico LR(0). Per ogni stato s va ripetuta la seguente cosa:

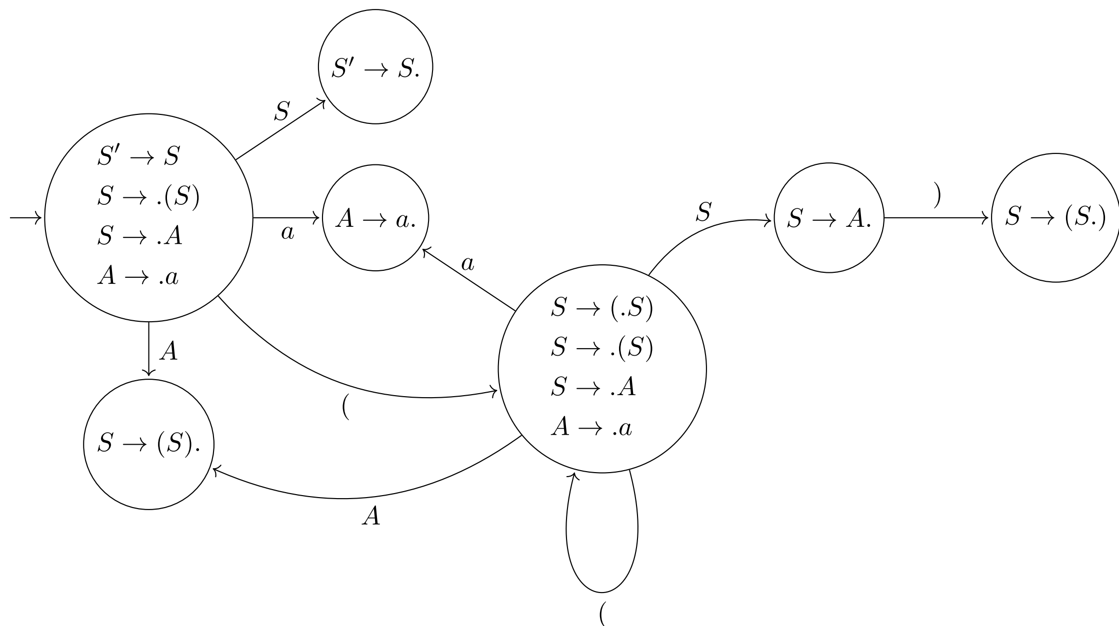
- Se $x \in T$ e $s \rightarrow^x t$ nell'automa LR(0), allora $M[s, x] =$ shift t.
- Se $A \rightarrow \alpha. \in s$ e $A \neq S'$ allora per ogni $x \in T \cup \{\$\}$, $M[s, x] =$ reduce $A \rightarrow \alpha$.
- Se $S' \rightarrow S. \in s$ allora $M[s, \$] =$ accept.
- Se $A \in NT$ e $s \rightarrow^A t$ nell'automa LR(0), allora $M[s, A] =$ goto t.

Quindi per le transizioni etichettate con terminali avremo shift, per transizioni etichettate con non terminali avremo goto. Mentre per item "finali" avremo delle reduce (o nel caso di $S.$ avremo accept).

Esempio 18.4.1 (). *Data la grammatica aumentata:*

$$S' \rightarrow S$$
$$S \rightarrow (S)$$
$$S \rightarrow ()$$

Costruiamo l'automa canonico LR(0) secondo la costruzione data sopra:



Dall'automa con le regole date sopra ci possiamo ricavare la tabella di parsing LR(1):

	a	(	)	$	S	A
1	S2	S3			G4	G5
2	R3	R3	R3	R3		
3	S2	S3			G6	G5
4				ACC		
5	R2	R2	R2	R2		
6			S7			
7	R1	R1	R1	R1		

E con un input "((a))$" otteniamo:

$$
\begin{array}{llll}
(1, & \epsilon, & ((a))\$) & \\
(13, & (, & (a))\$) & \\
(133, & ((, & a))\$) & \textit{reduce } A \to a \\
(1332, & ((a, &))\$) & \\
(1335, & ((A, &))\$) & \textit{reduce } S \to A \\
(1336, & ((S, &))\$) & \\
(13367, & ((S), &))\$) & \textit{reduce } S \to (S) \\
(136, & (S, &)\$) & \\
(1367, & (S), &)\$) & \textit{reduce } S \to (S) \\
(14, & S, & \$) & \textit{accept}
\end{array}
$$

18.4.2 Grammatiche LR(0)

Definizione : *Una grammatica è di classe LR(0) se ogni casella nella tabella di parsing LR(0) contiene al massimo un elemento.*

18.5 Algoritmo del parser

Il parser data la tabella di parsing LR esegue questo algoritmo per calcolare l'albero di derivazione. Questo algoritmo è per un generico parser LR (quindi funziona anche per LR(k)).

Algorithm 13: Parser LR

```
1  Inizializza la pila con $s_o;
2  Inizializza i_c con il primo carattere in input;
3  while (true) {
4     S = Top(pila);
5     switch M[S, i_c] {
6        case shift t: {
7           push t sulla pila;
8           avanza i_c sull'input;
9        }
10       case accept : {
11          output("accept");
12       }
13       case reduce A → α: {
14          pop |α| stati dalla pila;
15          s_1 = Top(pila);
16          s_2 = M[s_1, A];
17          push s_2 sulla pila        //s_2 è lo stato dato dal goto;
18          output la produzione A → α;
19       }
20       case vuoto : {
21          errore()                   //la casella è vuota;
22       }
23    }
24 }
```

Chapter 19

SLR(1), LR(1), LALR(1)

Può accadere che data una grammatica non si riesce a creare un parser di tipo LR(0) senza che ci siano conflitti, cioè ci possiamo trovare in una cella più di una azione. Questi conflitti sono spesso conflitti shift/reduce, cioè abbiamo una azione di shift e una di reduce, e accadono perché siamo stati poco ristrettivi sulle azioni reduce e le abbiamo messo anche dove sappiamo che non potranno mai essere usate.

19.1 Quando bastano i parser LR(0)

Si possono trovare delle proprietà dei linguaggi che garantiscono che un parser LR(0) non abbia conflitti:

- Se L è **libero deterministico** e gode della **prefix property** allora L **è LR(0)**.
- Se L è **finito** e gode della **prefix property** allora **è LR(0)**. Mentre, al contrario, se L è finito e non gode della prefix property allora non è LR(0).
- Se L è LR(0) ma è infinito può non godere della prefix property.
- Se G presenta produzioni ϵ allora, spesso, non è LR(0) (l'idea è che le produzioni ϵ permettono di applicare sempre una riduzione).

19.2 Tabella di parsing SLR(1)

La tabella è delle stesse dimensioni e della stessa forma di quella LR(0). L'unica cosa che cambia e che invece di mettere una reduce su tutta una riga la si mette solo in corrispondenza dei follow del non terminale in testa alla produzione, cioè:

- Se $A \to \alpha. \in s$ e $A \neq S'$ allora inserisci reduce $A \to \alpha$ in $M[s, x]$ **per tutti gli** $x \in$ **Follow(A)**.

L'idea è che riduciamo le azioni di reduce ai soli casi possibili (in realtà vedremo che siamo ancora troppo "larghi" e dobbiamo ridurre ancora i casi). Il termine SLR(1) sta per "Simple, left-to-right, rightmost derivation" con un simbolo di lookahead.

19.3 Parser LR(1)

L'idea dietro i parser LR(1) è quella di ridurre ancora di più i casi possibili di reduce a quelli ancora più plausibili (infatti il follow ci da troppi casi). Per questo vengono definiti nuovi tipi di item, gli item LR(1).

19.3.1 Item LR(1)

Definizione : *Un item LR(1) è una coppia formata da:*

- *Un item LR(0) che viene chiamato nucleo.*
- *Un simbolo di look-ahead in* $T \cup \{\$\}$.

Esempio 19.3.1 (). *Un possibili item LR(1) è:*

$$[S \to V. = E, \$]$$

dove abbiamo che $S \to V. = E$ *è il core e* $\$$ *è il simbolo di look ahead. Se avessi uno stato con due item dentro:*

$$[S \to V. = E, \$]$$
$$[E \to V., \$]$$

questo item mi indica che se leggo dall'input "=" allora faccio lo shift, mentre se leggo "$" faccio una reduce $E \to V$.

L'idea dietro questi item è appunto ridurre i casi di reduce. Se ho uno stato che contiene un item $[A \to \alpha.\beta, x]$ allora:

- Sto cercando di riconoscere $\alpha\beta$. Di questo α è già sulla pila mentre β no.

- In input mi aspetto una stringa derivabile da βx che cosi possono ridurre a βx e completare la maniglia $\alpha\beta$.

Mentre un item del tipo $[A \to \alpha., X]$ mi indica di fare la reduce $A \to \alpha$ solo se il prossimo input è proprio x.

19.3.2 NFA LR(1)

Per creare l'NFA LR(1) si fa abbastanza similmente al NFA LR(0) dove in più teniamo traccia dei simboli di look ahead:

- Si parte dallo stato iniziale $[S' \to .S, \$]$.

- Dallo stato $[A \to \alpha.X\beta, a]$ c'è una transizione allo stato $[A \to alphaX.\beta, a]$ etichettata "X" per $X \in T \cup NT$ (quindi qua è uguale al LR(0), il simbolo di look ahead viene "ereditato" dall'altro stato).

- Dallo stato $[A \to \alpha.X\beta, a]$ per $X \in NT$ e per ogni produzione $X \to \gamma$ c'è una transizione ϵ verso lo stato $[X \to .\gamma, b]$ **per ogni** $b \in$ **First(**βa**)**. Perché faccio questo?

  - L'obbiettivo è quello di ridurre $\alpha.X\beta$ in A. Però visto che A contiene un non terminale X (che non è ancora stato letto) allora prima cerco di ricondurmi a lui (riducendo una sua produzione) e poi dopo riduco verso A.

  - La riduzione di una produzione $X \to \gamma$ ha però senso solo se l'input successivo è parte di βa, ma visto che βa non è si è ancora iniziato a leggerlo io dopo la riduzione mi aspetto un $First(\beta a)$ e quindi io la riduzione la applico solo in quei casi li. Se la applicassi invece in altri casi io riduco γ in X però dopo non riesco a ridurre $\alpha X\beta$ in A perché ho un simbolo da leggere che non è quello di β e quindi probabilmente invece di una reduce avrei dovuto fare shift.

19.4 Automa canonico LR(1)

Si ottiene sempre in due modi, o attraverso NFA LR(1) o attraverso una costruzione diretta tramite le funzioni $Clos(I)$ e $Goto(I, x)$.

19.4.1 Clos e Goto

Algorithm 14: Clos(I)

```
1  while I non è più modificato {
2     foreach item [A → α.Xβ, a] in I {
3        foreach produzione X → γ {
4           foreach b in First(βa) {
5              aggiungi [X → .γ, b] ad I ;
6           }
7        }
8     }
9  }
10 return I ;
```

Algorithm 15: Goto(I,X)

```
1  Inizializza J = ∅;
2  foreach item [A → α.Xβ, a] in I {
3     aggiungi [A → αX.β, a] a J;
4  }
5  return clos(J) ;
```

19.5 Tabella di parsing LR(1)

E' sempre uguale a quella LR(0) e SLR(1) solo che inserisco la reduce solo per il simbolo di look ahead, cioè:

- Se $[A \rightarrow \alpha., x] \in s$ e $A \neq S'$ allora inserisci reduce $A \rightarrow \alpha$ in $M[s, x]$ **per tutti gli** x **del look ahead**.

> **Definizione :** *Una grammatica libera G è di classe LR(1) se ogni casella della sua tabella di parsing LR(1) contiene al massimo una azione.*

19.6 Parser LALR(1)

Le tabelle di parsing LR(1) risultato spesso molto grandi per questo si utilizzano più spesso tabelle LALR(1) che sono un buon compromesso tra semplicità (le sue tabelle sono grandi quanto quelle LR(0)) e selettività di LR(1) (però comunque LR(1) è "più forte" di LALR(1)).

19.6.1 Come ottenere un parser LALR(1)

Il parser LALR(1) si ottene dal automa canonico LR(1) in cui stati con lo stesso nucleo sono stati fusi insieme.

19.6.2 Problemi di LALR(1)

Ci sono un paio di problemi:

- Su input corretti LALR(1) e LR(1) si comportano uguale. Su input errati LALR(1) prima di riconoscere che sono sbagliati ci mette più mosse.
- Si possono creare nuovi conflitti reduce/reduce dati dalla fusione degli stati (lo si può dimostrare, se lo vuoi vedere vai PDF 16 pag 20).

Chapter 20

Grammatiche LL(K), LR(K), SLR(K) e LALR(K)

ù

20.1 Grammatiche LL(K), LR(K), SLR(K) e LALR(K)

20.1.1 Grammatiche LR(K)

Si costruisce l'automa in modo simile, l'unica vera differenza è che il look ahead sarà dato da sequenze di lunghezza massima k. Per costruirlo si useranno le funzioni $First_k$ e $Follow_k$ al posto di $First$ e $Follow$. Nel caso che la tabella di parsing LR(K) non contiene celle con più di una azione allora la grammatica e LR(K).

20.1.2 Grammatiche SLR(K)

Si parte sempre dall'automa LR(0) e si riempe la tabella di parsing SLR(K), dove le colonne saranno simboli in T^K, secondo la seguente legge:

- Se $[A \to \alpha.] \in S$ e $A \neq S'$ inserisci reduce $A \to \alpha$ in $M[s, w]$ dove $w \in Follow_k(A)$ (**quindi l'unica cosa che cambia è $Follow_k$ invece che $Follow$**).

20.1.3 Grammatiche lALR(K)

E' esattamente identico a LALR(1) solo che invece che usare l'automa canonico LR(1) si usa LR(K).

20.2 Relazione tra grammatiche

Le grammatiche libere si dividono in due grandi sottoinsiemi: le grammatiche ambigue e quelle non ambigue. Guardando solo le grammatiche non ambigue abbiamo che:

$$LL(K) \subset LR(K) \quad \forall K \geq 0$$

cioè le grammatiche LR(K) includono sempre anche le grammatiche LL(K). Ma vale però anche:

$$LL(K) \not\subset LR(K-1) \quad \forall K \geq 1$$

cioè le grammatiche LL(K) invece non sono incluse in LR(K-1) (quindi ad esempio LR(0) non include LL(1)). Per le grammatiche SLR(K) e LALR(K) vale invece che:

$$SLR(K) \subset LALR(K) \subset LR(K) \quad \forall K \geq 1$$

Cioè le grammatiche SLR e LALR sono sempre comprese nella loro controparte LR.

20.3 Proprietà di grammatiche LL(K) e LR(K)

Teorema : *Se G è LL(K) allora G è non ambigua e $L(G)$ è deterministico.*

Teorema : *Se G è LR(K) allora G è non ambigua e $L(G)$ è deterministico.*

Nota che comunque esistono linguaggi generati da grammatiche non ambigue ma che sono non deterministici, tipo:

$$S \to aSa \mid bSb \mid \epsilon$$
$$L(G) = \{ww^R \mid w \in \{a,b\}^*\}$$

20.4 Classificazione dei linguaggi

Definizione : *Un linguaggio L è di classe X se $\exists G$ di classe X tale che $L = L(G)$ (X sta per LR(0), SLR(1) ..).*

Esempio 20.4.1 (). *La grammatica G_1 seguente:*

$$S \to aAc$$
$$A \to bAb \mid b$$

si può dimostrare non essere LR(K) per nessun K, ma il linguaggio generato da essa:

$$L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$$

è in realtà un linguaggio LR(0), perché è generabile da una grammatica LR(0) come la seguente:

$$S \to aAc$$
$$A \to Abb \mid b$$

(che si può dimostrare, facendo la tabella di parsing, essere LR(0)) (ricordati che la ricorsione sinistra non da fastidio ai parser bottom up).

20.4.1 Teoremi utili per la classificazione di linguaggi

Teorema : *Un linguaggio è libero deterministico se e solo se è generato da una grammatica LR(K) per qualche $K \geq 0$.*

Teorema : *Un linguaggio è libero deterministico se e solo se è generato da una grammatica SLR(1).*

Notiamo che dati i due teoremi se abbiamo una grammatica che ci genera un linguaggio libero deterministico non vuol dire che quella grammatica sia SLR(1), ma ci dice solo che esisterà sempre una grammatica SLR(1) che lo genera.

Teorema : *I linguaggio generati da grammatiche LL(K) sono strettamente contenuti nei linguaggi generati da grammatiche SLR(1) (nota vale solo per i linguaggi questo e non per le grammatiche!).*

Esempio 20.4.2 (). *Il linguaggi* $L = \left\{a^i b^j \mid i \geq j \geq 0\right\} = a^* \left\{a^n b^n \mid n \geq 0\right\}$ *è libero deterministico, quindi vuol dire che esisterà una grammatica SLR(1) che lo genera (e anche grammatiche LR(K) che lo generano). Nonostante ciò non esiste nessuna grammatiche LL(K) che lo genera.*

Teorema : *L'unione di due linguaggi LL(1) non è detto che sia LL(1).*

Teorema : *L'unione di due linguaggi LR(0) non è detto che sia LR(0).*

Teorema : *La concatenazione di due linguaggi LL(1) non è detto che sia LL(1).*