Appunti di Linguaggi di Programmazione e Verifica del Software

per facoltà di informatica

di Arlind Pecmarkaj

Anno Accademico 2022/2023

23 dicembre 2022

Indice

Capitolo 1

Fondamenti dei linguaggi di programmazione

1.1 Teoria dei linguaggi formali

Definiamo come linguaggio la capacità umana di utilizzare sistemi di comunicazione complessi.
Possiamo dire anche che è l'insieme degli strumenti che permettono la comunicazione.

Un linguaggio è detto formale se è provvisto con una rappresentazione matematica rigorosa di:

- **Alfabeto** di simboli usati;
- **Regole di formazione**, ossia la grammatica che permette di unire i simboli in parole e poi di unire le parole in frasi;

Introduciamo le notazioni e le definizioni che verranno usate.
Chiamiamo **alfabeto** un insieme non vuoto e finito di simboli e lo denotiamo generalmente con la lettera Σ.
Esempi di alfabeti possone essere:

- $\Sigma_1 = \{0, 1\}$
- $\Sigma_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
- $\Sigma_4 = \{a, b, c, ..., z\}$

Chiamiamo **stringa** (oppure **parola**) in un alfabeto Σ una sequenza finita di simboli presi dall'alfabeto Σ.
Per esempio:

- 1010 può essere parola degli alfabeti $\Sigma_1, \Sigma_2, \Sigma_3$;

- 123 può essere parola degli alfabeti Σ_2, Σ_3;
- *ciao* può essere parola dell'alfabeto Σ_4;

Denotiamo con

- $a, b, c, ...$ i simboli generici e
- $v, w, x, y, z, ...$ le parole generiche.

Per denotare la stringa vuota usiamo il simbolo ϵ.
La lunghezza di una stringa viene rappresentata con il simbolo $|v|$.
Per esempio:

- $|\epsilon| = 0$;
- $|a| = 1$;
- $|ciao| = 4$;

Sono possibili diverse operazioni tra stringhe:

- vw è la **concatenazione** delle stringhe v e w;
- v è una **sottostringa** di w se e solo se $xvy = w$;
- v è **prefisso** di w se e solo se $vy = w$
- v è **sufisso** di w se e solo se $xv = w$

Come conseguenza la stringa vuota ϵ è elemento neutro per la concatenazione, infatti $\epsilon w = w\epsilon = w$.
Inoltre la concatenazione non è commutativa.

Chiamiamo **k-esima potenza** di un alfabeto Σ l'insieme di tutte le parole uniche di lunghezza k creabili tramite l'alfabeto Σ, i.e.

$$\Sigma^k \stackrel{\text{def}}{=} \{a_1...a_k | a_1, ..., a_k \in \Sigma\}$$

Deriva che $\Sigma^0 = \{\epsilon\}$ per ogni Σ.
Prendiamo per esempio $\Sigma = \{0, 1\}$ allora $\Sigma^2 = \{00, 01, 10, 11\}$.

Chiamiamo **chiusura di Kleene** di un alfabeto Σ l'insieme di tutte le parole creabili a partire dall'alfabeto Σ

$$\Sigma^* \stackrel{\text{def}}{=} \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ... = \bigcup_{i=0}^{\infty} \Sigma^i$$

Se usiamo il simbolo $+$ al posto di $*$ escludiamo la stringa vuota, i.e.

$$\Sigma^+ \stackrel{\text{def}}{=} \Sigma^* \setminus \{\epsilon\} = \bigcup_{i=1}^{\infty} \Sigma^i$$

Possiamo adesso dare una definizione formale di linguaggio: un **linguaggio** L in un determinato alfabeto Σ è un qualsiasi sottoinsieme di Σ^* ossia

$$L \subseteq \Sigma^*$$

Poichè i linguaggi sono degli insiemi è possibile fare operazioni su di esso.
Siano L_1 e L_2 due linguaggi creati rispettivamente da un alfabeto Σ_1 e Σ_2, abbiamo:

- $L_1 \cup L_2 = \{w | w \in L_1 \vee w \in L_2\}$;
- $L_1 \cap L_2 = \{w | w \in L_1 \wedge w \in L_2\}$;
- $\bar{L}_1 = \{w \in \Sigma_1^* | w \notin L_1\}$ è l'operazione di complemento ed è definito sull'alfabeto su cui è costruito L_1. Perciò $\bar{L}_1$ non indica tutte le parole tranne quelle in L_1, ma ogni parola in Σ_1^* che non è in L_1;
- $L_1 L_2 = \{w_1 w_2 | w_1 \in L_1 \wedge w_2 \in L_2\}$ è la concatenazione di ogni stringa di L_1 per ogni stringa di L_2. Si ha che $L_1 L_2 \neq L_2 L_1$;
- $L_1^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup ... = \bigcup_{i=0}^{\infty} L^i$ è la chiusura di Kleene del linguaggio L_1;

Uno dei problemi più importanti riguardanti i linguaggi formali è il **problema di appartenenza**: dato un linguaggio L e una stringa w vogliamo trovare un modo effettivo ed efficace per decidere se $w \in L$.
Esistono due approcci per la risoluzione del problema.
Il primo viene dalla linguistica (che studia i linguaggi formali per lo studio scientifico dei linguaggi umani) ed è **l'approccio generativo**: un linguaggio è l'insieme delle stringhe generate da un oggetto matematico chiamato **grammatica**.
Una grammatica definisce un insieme finito di regole su cui si possono costruire le frasi del linguaggio tramite un **processo di generazione**:

1. Si inizia dal simbolo iniziale;
2. Si espande la stringa attuale tramite regole di riscrittura;
3. Ci si ferma quando viene generata una stringa del linguaggio;

Pur essendo un approccio intuitivo e informarle che permette di creare potenzialmente infiniti linguaggi da un insieme finito di regole, in alcuni casi può essere inefficiente.
L'altro approccio è di tipo **riconoscitivo** ed è quello usato nell'informatica: in questo caso un linguaggio è l'insieme di parole accettate da una macchina astratta detta automa.
In questo caso ci basiamo sul **processo riconoscitivo**:

1. Si parte dallo stato iniziale dell'automa;

2. Ci si sposta negli altri stati dell'automa tramite i simboli della stringa;

3. Si continua fino a che si è letta tutta la stringa e si è giunti a uno stato di accettazione/rifiuto;

Questo approccio è poco intuitivo, ma è facilmente automatizzabile e dunque usabile in un calcolatore.

Nell'informatica i linguaggi formali vengono usatiper la definizione dei linguaggi di programmazione e per lo sviluppo di **compilatori**.
Un compilatore è un programma che trasforma un codice (detto sorgente) in un linguaggio di programmazione in un programma equivalente scritto in un altro linguaggio (linguaggio target). Il compilatore è formato di tanti automi che usano l'approccio riconoscitivo che date le istruzioni eseguono questi passi:

$$scanning \rightarrow parsing \rightarrow analisi\, semantica$$

Nello scanning gli automi si occupano dell'analisi lessicale, nel parsing viene fatta l'analisi della frase e viene costruito l'albero sintattico. Finita questa fase si passa alla generazione del codice.

1.2 Classificazione di Chomsky delle grammatiche

Scopo di una grammatica è quello di generare parole e frasi.
Le parole devono avere una certa struttura ragionata in maniera composizionale.
Una grammatica è una tupla $G = (V, T, S, P)$ dove:

- V è un insieme non vuoto e finito di simboli chiamati **variabili** (oppure **non-terminali** o **categorie sintattiche**);
- T è un alfabeto di simboli detti **terminali**;
- $S \in V$ è il simbolo di **partenza** (o **iniziale**) della grammatica;
- P è un insieme finito di **produzioni** $\alpha \to \beta$ dove $\alpha \in (V \cup T)^+$ e $\beta \in (V \cup T)^*$;

Usiamo la seguente notazione:

- $A, B, C, ... \in V$ indicano i simboli non terminali;
- $a, b, c, ... \in T$ indicano i simboli terminali;
- $X, Y, Z, ... \in V \cup T$ indicano simboli generici;
- $u, v, w, x, ... \in T*$ indicano le stringe create da T;
- $\alpha, \beta, \gamma, \delta, ... \in (V \cup T)^*$ indicano le stringhe create da $V \cup T$;

Prendiamo come esempio la seguente grammatica G:

- Variabili: $\{S, V, O, F\}$;
- Terminali: $\{i, y, b, e, a, p\}$;
- Simbolo iniziale: $\{F\}$
- Produzioni: $\{S \to i, S \to y, V \to b, V \to e, O \to a, O \to p, F \to SVO\}$

Una possibile derivazione può essere: $F \to SVO \to yVO \to yeO \to yep$.
Ma se y stesse per "you", e per "eat" e p per "pasta"?
E se S fosse il soggetto della frase, V il verbo e O il complemento oggetto?
Notiamo come con una grammatica relativamente semplice è possibile già creare pezzi di linguaggio naturale come l'inglese.

Una derivazione $\mu \to_G \gamma$ è detta a singolo passo se e solo se:

1. $\mu = \sigma\alpha\tau$ e
2. $\gamma = \sigma\beta\tau$ e
3. $\alpha \to \beta \in P$.

Una derivazione $\mu \rightarrow_G^* \gamma$ è detta a passi multipli se e solo se:

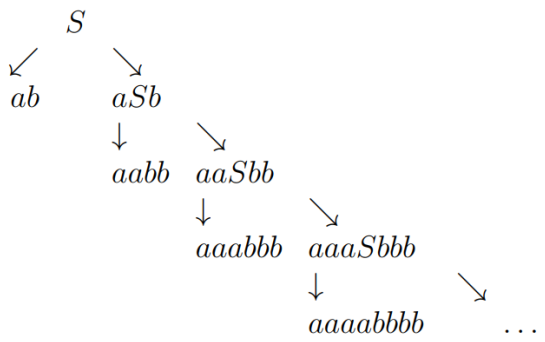
1. $\mu = \gamma$ oppure
2. $\exists \delta \, tale \, che \, \mu \rightarrow_G \delta$ e $\delta \rightarrow_G^* \gamma$.

. Come si può notare si segue un ragionamento di tipo induttivo.

Chiamiamo linguaggio generato da una grammatica l'insieme delle stringhe create da un alfabeto di terminali T che possono essere derivate dal simbolo iniziale S in qualsiasi numero di passi usando le produzioni in P. Ossia sia $G = (V, T, S, P)$ una grammatica allora il linguaggio $L(G)$ generato da G sarà:

$$L(G) \stackrel{\text{def}}{=} \{w \in T^* | S \rightarrow_G^* w\}$$

Per esempio prendiamo la grammatica
$G_1 = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow AB\})$, il linguaggio $L(G_1)$ sarà:



Ossia $L(G_1) = \{a^n b^n | n > 0\}$.

Le grammatiche vengono classificate in base alle restrizioni che si hanno sulle produzioni e si va da grammatiche più potenti ed espressive a grammatiche più limitanti e semplici:

- **Generali**: produzioni senza restrizioni;
- **Monotone**: le produzioni sono del tipo $\alpha \rightarrow \beta \implies |\alpha| \leq |\beta|$ ovvero la regola non decrementa la lunghezza della stringa che viene riscritta.
- **Dipendenti dal contesto**: le produzioni sono del tipo $\gamma A \delta \rightarrow \gamma \beta \delta$ ossia la trasformazione può essere effettuata soltante se viene mantenuto il contesto (rimangono γ e δ.
- **Libere dal contesto (o libere)**: le produzione sono del tipo $A \rightarrow \beta$ ossia il simbolo A può essere sempre trasformato in β a prescindere dal contesto in cui si trova.
- **Lineari**: le produzioni sono del tipo $A \rightarrow uBv$ ossia un non terminale può comparire solo tra due stringhe di terminali.

- **Lineari a destra**: le produzioni sono del tipo $A \to wB$ ossia un non terminale deve stare a destra di una stringa di terminali.
- **Lineari a sinistra**: le produzioni sono del tipo $A \to Bw$ ossia un non terminale deve stare a sinistra di una stringa di terminali.

Il linguista Noam Chomsky divise le grammatiche formali in classi con potere espressivo sempre più grande.
Ogni classe più in alto nella gerarchia può generare un insieme più grande di linguaggi formali rispetto a quelli sotto.
La divisione è come segue:

Classe di Chomsky	Grammatica	Linguaggio
Tipo-0	Generale	L_0 : ricorsivamente numerabile
Tipo-1	Dipendente dal contesto	L_1 : dipendente dal contesto
Tipo-2	Libera (dal contesto)	L_2 : libero (dal contesto)
Tipo 3	Lineare a destra/sinistra	L_3 : regolare

Le grammatiche generali generano linguaggi ricorsivamente numerabili ossia la cui dimensione delle stringhe è numerabile tramite un algoritmo.
La relazione di inclusione tra i linguaggi è stretta, ossia

$$L_3 \subset L_2 \subset L_1 \subset L_0 \subset \wp(T^*)$$

Notiamo come esistono linguaggi che non sono generati da grammatiche. Infatti la relazione $L_0 \subset \wp(T*)$ implica che esistono linguaggi in $\wp(T*)$, ma non in L_0 che è l'insieme dei linguaggi generati da grammatiche generali, le più potenti.
É possibile dimostrare ciò matematicamente. Assumiamo di avere un alfabeto Σ e l'insieme delle stringhe w_i generato da esso.
Adesso prendiamo l'insieme di tutte le grammatiche, costruiamo una tabella a cui associamo grammatica e parola: se la parola è generata dalla grammatica poniamo 1, altrimenti 0.

	w_1	w_2	w_3	$\cdots$
G_1	0	1	1	$\cdots$
G_2	1	1	1	$\cdots$
G_3	1	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Definiamo l'insieme delle parole presenti nelle diagonali (ossia parole $w_i \in L(G_i)$) e il suo complemento

$$D = \{w_i | w_i \in L(G_i)\} = \{w_2, \dots\}$$
$$\bar{D} = \{w_i | w_i \notin L(G_i)\} = \{w_1, w_3, \dots\}$$

Supponiamo per contraddizione che $\bar{D} = L(G_j)$ allora

$$w_j \in \bar{D} \implies w_j \notin L(G_j) \; assurdo!$$
$$w_j \notin \bar{D} \implies w_j \in L(G_j) \; assurdo!$$

Conclusione: non esiste grammatica che genera $\bar{D}$.

1.3 Teoria degli automi e delle macchine di Turing

La teoria degli automi è lo studio degli automi, macchine astratte che riconoscono linguaggi. Una di queste macchine astratte è nota ed è la **macchina di Turing**. Questi tre punti descrivono cosa sia:

- Una macchina di Turing è una macchina astratta a stati finiti che esegue calcoli tramite il supporto di un nastro infinito diviso in celle di contenuto atomico (ossia ogni cella contiene un solo simbolo);
- Un puntatore indica la cella corrente che può essere letta o scritta prima di passare a una delle celle immediatamente vicine;
- Abbiamo una contol unit, con il suo stato interno, che contiene un insieme finito di istruzioni che determinano le azioni da eseguire;

Formalmente una macchina di Turing è una tripla $\mathcal{M} = (S, A, I)$ dove:

- S è l'insieme finito degli stati, che include lo stato iniziale q_{in};
- A è l'insieme finito dei simboli chiamato alfabeto di $\mathcal{M}$ (la cella vuota è denotata con il simbolo $\square$);
- I è l'insieme finito di istruzioni in $S \times A \times A \times \{R, L\} \times S$. La notazione breve per le istruzioni è $(q, s) \to (s', X, q')$ dove $X \in \{R, L\}$; (ossia spostamento a destra 'R' o sinistra 'L' del puntatore)
  $q, q' \in S; s, s' \in A$;

Una macchina di Turing è detta **deterministica** se per ogni $(q, s) \in S \times A$ esiste al più un'unica istruzione in I della forma $(q, s) \to (\_, \_, \_)$, altrimenti è **nondeterministica**.

La **configurazione** di una macchina di Turing è descritta:

- dalla stringa di simboli scritti nel nastro;
- dalla dalla posizione del puntatore di testa;
- dallo stato della macchina di Turing.

Formalmente, una configurazione c è una sequenza $\alpha q \beta$ dove:

- $\alpha\beta$, con $\alpha, \beta \in A^*$, è la stringa nel nastro;
- Il puntatore è nella posizione dove β inizia.
- $q \in S$ è lo stato attuale.

Un **passo di computazione** (o **transizione**) è lo spostamento da una configurazione c_i a una configurazione c_j per via dell'applicazione di un'istruzione. Formalmente:

- Se $(q, b) \to (d, R, q') \in I$ allora $\alpha q b \beta \to \alpha d q' \beta$;
- Se $(q, b) \to (d, L, q') \in I$ allora $\alpha a q b \beta \to \alpha q' a d \beta$

Una configurazione si dice **terminante** se non permette nessuna transizione. Una **computazione** è una sequenza di transizioni che possono essere infinite o finite che terminano in una configurazione terminante.

Una macchina di Turing viene usata come:

- *Trasduttore* per calcolare funzioni (pensate all'origine a valore intero) per ragionare sul concetto di calcolabilità;
- *Accettatore di linguaggi* per decidere sul problema di appartenenza per un linguaggio formale;

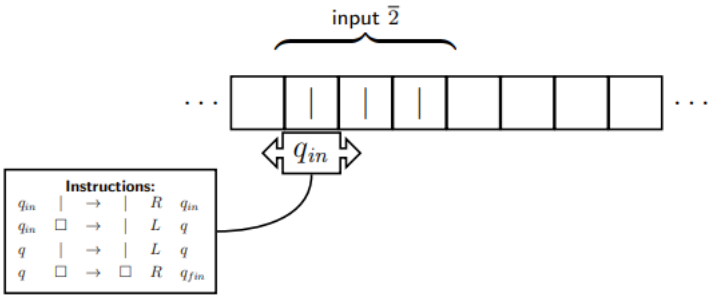
Vediamo un esempio di calcolo fatto con una macchina di Turing per calcolare il successore di un numero.
Un'alafabeto $A = \{|\}$ è detto **unario** e ogni numero naturale n può essere rappresentato da $n + 1$ consecutivi $|$, i.e.

$$\bar{0} = | \ \bar{1} = || \ \bar{2} = ||| \ \cdots$$

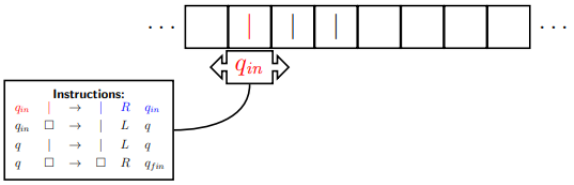
$\bar{n}$ è chiamato *numerale*.
Sia $\mathcal{M} = (S, A, I)$ dove:

- $S = \{q_{in}, q, q_{fin}\}$;
- $A = \{|, \square\}$
- $I = \left\{ \begin{matrix} q_{in} \mid \to \mid R \ q_{in} & q \mid \to \mid L \ q \\ q_{in} \square \to \mid L \ q & q_{in} \square \to \square \ R \ q_{fin} \end{matrix} \right\}$

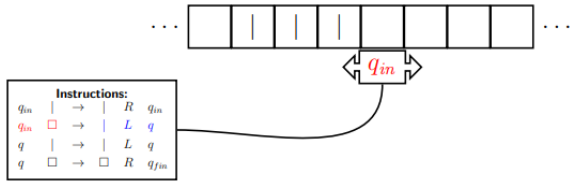
Il nastro viene inizializzato con la stringa di input, la testina è nella cella più a sinistra e lo stato iniziale è q_{in}



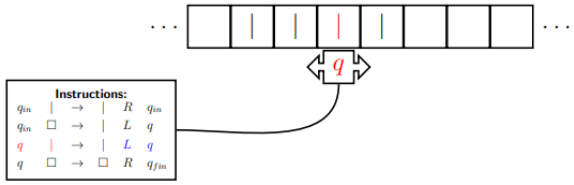
Viene eseguita la prima istruzione, ossia si legge il contenuto e se siamo nello stato q_{in} passiamo alla cella a destra.



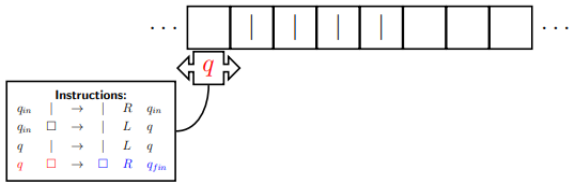
Continuiamo fino a che non capitiamo alla prima cella con carattere □. Scriviamo | e cambiamo stato a q.



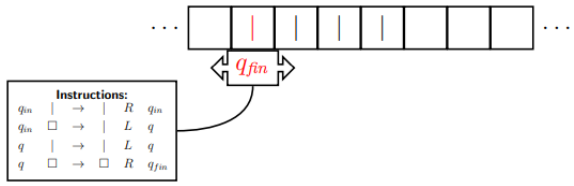
Cambiamo stato e torniamo indietro



Arrivati alla cella □ non scriviamo nulla e cambiamo stato in q_{fin} e puntiamo la testina verso destra.



Siamo arrivati a una configurazione di terminazione. Il risultato nel nastro è il nostro successore.



Sfruttiamo le macchine di Turing anche per riconoscere le parole di un linguaggio.
Facciamo ciò tramite l'automa di Turing ossia una macchina di Turing dotata di stati di accettazione: due stati speciali q_{acc} (stato di accettazione) e q_{ref} (stato di rifiuto).
Entrambi i due stati sono terminanti.
Una macchina di Turing con stati di accettazione accetta una stringa w se partendo da una configurazione iniziale $q_{in}w$ esiste una computazione finita che termina in una configurazione di accettazione.
Il linguaggio $\mathcal{L}(\mathcal{M})$ è l'insieme di stringhe accettata dalla macchina di Touring con stati di accettazione $\mathcal{M}$.
In genere data una stringa w quando attiviamo una macchina di Touring per riconoscerla, può succedere che

i. Il calcolo non termina;

ii. Il calcolo termina nello stato di rifiuto;

iii. Il calcolo termina nello stato di accettazione;

Se la macchina di Touring non termina, non possiamo assumere che la stringa viene rifiutata.
Se una macchina di Turing termina per ogni stringa che non viene accettata viene detta *decidente* perchè per ogni stringa data in ingresso riesce a prendere una decisione. In questo caso il linguaggio accettato è deciso dalla macchina.
In caso contrario se per qualche input la macchina non termina, allora essa non decide il linguaggio.
Si pone dunque la differenza tra *accettabile* e *decidibile*:

- $\mathcal{L}(\mathcal{M})$ è detta Turing-accettabile: la macchina di Turing riesce a terminare per tutte le parole presenti nel linguaggio;
- Linguaggi Turing-accettabili coincidono con linguaggi ricorsivamente numerabili;
- Se $\mathcal{M}$ termina nello stato di rifiuto per ogni stringa non accettata allora $\mathcal{M}$ *decide* $\mathcal{L}(\mathcal{M})$;
- $\mathcal{L}$ è detta Turing-decidibile se esiste una macchina di Touring con stati di accettazione che la decide;
- Linguaggi Turing-decidibili sono Turing-accettabili, ma il viceversa in generale non vale.

1.4 Calcolabilità

Nella sezione precedente abbiamo affermato come una macchina di Touring (da adesso in poi verranno indicate con TM) possa essere usata come trasduttore e parcolare una funzione di valori interi.

Data una TM $\mathcal{M}$ e $n \in \mathbb{N}$: se $\mathcal{M}$ per qualsiasi input $\bar{n}$ termina e restiuisce l'output $\bar{m}$ scriviamo $\mathcal{M}(\bar{n}) \downarrow \bar{m}$, altrimenti scriviamo $\mathcal{M}(\bar{n}) \uparrow$.
Una funzione (parziale) si dice **Turing-calcolabile** se esiste una TM $\mathcal{M} = (S, \{|, \square\}, I)$ tale che, per ogni $n \in \mathbb{N}$:

- Se $f(n)$ è definita allora $\mathcal{M}(\bar{n}) \downarrow \overline{f(n)}$;
- Altrimenti $\mathcal{M}(\bar{n}) \uparrow$;

La Turing-calcolabilità permette di catturare la nozione informale e intuitiva della *calcolbilità effettiva*: una funzione è calcolabile effettivamente se e solo se è Turing calcolabile (tesi di Church-Turing).
La tesi non è una congettura in quanto la nozione di calcolabilità effettiva è intuitiva e informale mentre quella di Turing-calcolabilità è matematica e rigorosa. Inoltre, dal lato euristico, non è stato trovato un controesempio verso la tesi. La tesi ci servirà poi successivamente nella *decidibilità* e nei problemi di decisione.

Un problema di decisione ha due elementi costituenti:

1. Un insieme di istanze;
2. Una domanda *booleana*;

I problemi di decisione sono essenzialmente dei *problemi di appartenenza* ed essi sono associati con il proprio *linguaggio caratteristico* X, formato da tutte le istanze del problema per cui la risposta alla domanda è sì.
Se pensiamo anche al complemento $\overline{X}$, ossia le istanze per cui la risposta alla domanda è no, allora il problema di decisione consiste nel determinare se l'istanza appartiene a X oppure $\overline{X}$.
Il problema può essere descritto tramite una funzione, la funzione *caratterestica* di X che per ogni istanza x restituisce 1 se l'istanza appartiene a X, altrimenti 0.
Se accettiamo la tesi di Church-Turing abbiamo che il problema è decidibile se e solo se è Turing-decidibile.
Formalmente un insieme $X \subseteq \mathbb{N}$ è **Turing-decidibile** se esiste una TM $\mathcal{M} = (S, \{|, \square\}, I)$ tale che, per ogni $n \in \mathbb{N}$:

- Se $n \in X$ allora $\mathcal{M}(\bar{n}) \downarrow \bar{1}$;
- Se $n \notin X$ $\mathcal{M}(\bar{n}) \downarrow \bar{0}$;

La **funzione caratteristica** di X è

$$f_X(n) = \begin{cases} 1 \ \ se\, n \in X \\ 0 \ \ altrimenti \end{cases}$$

Vale il seguente teorema:

L'insieme $X \subseteq \mathbb{N}$ è Turing-decidibile se e solo se f_X è Turing-calcolabile.

Se X è ricorsivamente numerabile allora diciamo che X è *Turing-accettabile*.
Se esiste una TM che decide X allora diciamo che X è **ricorsivo**.
Dunque dato un problema di decisione, esso può essere:

- *decidibile* se e solo se il suo linguaggio caratteristico è *ricorsivo*;
- *semidecidibile* se e solo se il suo linguaggio caratteristico è *ricorsivamente numerabile*;
- *indecidibile* se e solo se non è decidibile;

Elenchiamo alcune proprietà riguardanti la decidibilità:

- Se X è ricorsivo, allora $\overline{X}$ è ricorsivo;
- L'unione di insiemi ricorsivi è anch'esso ricorsivo;
- L'unione di insieme ricorsivamente numerabili è anch'esso ricorsivamente numerabile;
- **Teorema di Post**: Se X e $\overline{X}$ sono ricorsivamente numerabili allora X è ricorsivo;
- **Teorema di Rice**: Un insieme S di insiemi ricorsivamente numerabili è ricorsivo se e solo se $S = \emptyset$ o S è l'insieme di tutti i possibili insiemi ricorsivamente numerabili. Se pensiamo ad un insieme di insieme ricorsivamente numerabili come una *proprietà*, il teorema ci dice che le uniche proprietà decidibili che possiamo ricavare sono *banali*. La propietà $\emptyset$ è il problema *'è un insieme ricorsivamente numerabile non un insieme ricorsivamente numerabile?'* la cui risposta è banalmente no, mente l'altra proprietà è il problema *'è un insieme ricorsivamente numerabile un insieme ricorsivamente numerabile?'* la cui risposta è banalmente sì;

Un insieme $X \subseteq N$ è effettivamente decidibile se e solo se la sua funzione caratteristica è effettivamente calcolabile.
Esistono però molti problemi che non possono essere decisi per niente.
Uno di questi è problema della terminazione: data una TM $\mathcal{M}$ e $n \in \mathbb{N}$, decidere se $\mathcal{M}$ termina o meno sull'input $\bar{n}$.

Si dimostra matematicamente tramite argomento della diagonale (analogamente alla dimostrazione che alcuni linguaggi non sono generati da grammatiche) che il problema non è decidibile.
Ciò ha come effetto collaterale che non è psssibile scrivere un programma di debugging perfetto in quanto potrebbero potenzialmente esserci input 'scorretti' di cui non possiamo decidere l'esistenza.
Altro problema non decidibile è il problema dell'equivalenza: date due TM $\mathcal{M}_1$ e $\mathcal{M}_2$ decidere se calcolano la stessa funzione.
Anche questo problema è non decidibile e si porta come effetto il fatto di non poter stabilire se due probrammi esibiscono lo stesso comportamento.
É possibile però creare una macchina di Turing che può simularne un'altra. Se questa macchina può simulare ogni macchina di Turing allora la chiamiamo **macchina di Turing universale**.
Formalmente una TM $\mathcal{U}$ può essere costruita per ogni TM $\mathcal{M}$ e per ogni $n \in \mathbb{N}$ tale che:

- Se $\mathcal{M}(\bar{n}) \downarrow \bar{m}$ allora $\mathcal{U}(\overline{\lceil \mathcal{M} \rceil}, \overline{n}) \downarrow \bar{m}$;
- Se $\mathcal{M}(\bar{n}) \uparrow$ allora $\mathcal{U}(\overline{\lceil \mathcal{M} \rceil}, \overline{n}) \uparrow$

Una TM nondeterministica non è detta che sia più espressiva di una macchina di Turing. Anzi, è possibile simulare ogni tipo di TM nondeterministica da una macchina di Turing deterministica.

Adesso possiamo completare la divisione di Chomsky mettendo in relazione automi, grammatiche e linguaggi formali.

Automa	Grammatica	Linguaggio
Turing	Generale	L_0 : ricorsivamente numerabile
Limitato linearmente	Dipendente dal contesto	L_1 : dipendente dal contesto
Pushdown	Libera (dal contesto)	L_2 : libero (dal contesto)
A stati finiti	Lineare a destra/sinistra	L_3 : regolare

Ricordandoci della relazione $L_3 \subset L_2 \subset L_1 \subset L_0 \subset \wp(T^*)$ vediamo subito che esistono dei linguaggi formali non decidibili.
Grammatiche generali generano linguaggi ricorsivamente numerabili che come abbiamo visto sono Turing accettabili, ma ciò non implica che siano Turing decidibili: questo vuol dire che il problema di appartenenza è *semidecidibile* e che per parole non presenti nel linguaggio, l'automa che risponde al problema potrebbe entrare in un loop infinito.
Questo implica che le grammatiche generali sono inutili per un linguaggio di programmazione in quanto un eventuale compilatore non terminerebbe mai la sua esecuzione.

Per grammatiche dipendenti dal contesto l'automa che implementa il riconoscimento è un'automa *limitato linearmente*, ossia una TM con nastro finito.

I linguaggi generati da queste grammatiche sono Turing-accettabili e sono decidibili, ma la soluzione del problema di appartenenza in questo caso è di complessità *esponenziale* rendendo inutili linguaggi di programmazione generati da queste grammatiche.

Gli automi che si occupano del problema di appartenenza di linguaggi generati da grammatiche libere sono automi pushdown, TM con nastri finiti con politica di accesso LIFO.
In questo caso la soluzione del problema è di complessità quadratica ed infatti i linguaggi di programmazione usano grammatiche libere per la propria sintassi.

Per le grammatiche lineari e linguaggi regolari basta un automa a stati finiti, ossia un automa senza memoria, per risolvere il problema di appartenenza.
La complessità è lineare e i linguaggi di programmazione usano grammatiche lineari per il proprio lessico (lessico che viene controllato dallo scanner dei compilatori).

Capitolo 2

Automi a stati finiti

2.1 Automi a stati finiti deterministici

Un **automa a stati finiti deterministico** (DFA) è una macchina che prende una stringa di valori in input e risponde sì se la sequenza è stata accettata o no se è stata rifiutata.
É caratterizzato da:

- **Stato**: rappresenta il supporto di memoria basico per l'automa, tale per cui la lettura di un simbolo fa muovere l'automa da stato a stato;
- **Essere finito**: quando viene analizzata una stringa, l'automa può ricordarsi solo un insieme finito di fatti sulla sequenza;
- **Essere deterministico**: dato uno stato e un simbolo in input l'automa ha una sola via su cui procedere.

Formalmente un DFA è una tupla $A = (Q, \Sigma, \delta, q_0, F)$ dove:

- Q è l'insieme finito degli **stati**;
- Σ è l'alfabeto dei **simboli in input**;
- $\delta : Q \times E \to Q$ è la **funzione di transizione**. δ preso lo stato di sistema attuale e il simbolo in input produce uno stato nuovo.
  La funzione è parziale per definire il determinismo della macchina;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati **finali** (o **di accettazione**);

La lettura di una stringa $a_1a_2...a_n$ induce una visita nel cammino dell'automa come segue

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n$$

tale che:

- q_0 è lo stato inizale;
- $q_{i+1} = \delta(q_i, a_{i+1})$;
- Se $q_n \in F$ la stringa $a_1 a_2 ... a_n$ viene accettata, altrimenti è rifiutata;

Per esempio possiamo creare un automa che riconosce il seguente linguaggio

$$L = \{x \in \{0,1\}^* | \, 01 \; sottostringa \; di \; x\}$$

L'alfabeto è $\Sigma = \{0, 1\}$.
Usiamo tre stati:

- q_0 "non ho visto simboli oppure ho visto solo 1;
- q_1 "l'ultimo simbolo era 0";
- q_2 "ho visto la sequenza 01";

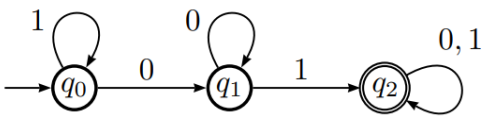
Lo stato iniziale sarà q_0 e l'insieme degli stati finali sarà $F = \{q_2\}$.
Specifichiamo le seguenti transizioni:

- Da q_0 se leggi 0 vai a q_1, ossia $\delta(q_0, 0) = q_1$;
- Da q_0 se leggi 1 rimani in q_0, ossia $\delta(q_0, 1) = q_0$;
- Da q_1 se leggi 0 rimani in q_1, ossia $\delta(q_1, 0) = q_1$;
- Da q_1 se leggi 1 vai a q_2, ossia $\delta(q_1, 1) = q_2$;
- Da q_2 se leggi 0 oppure 1 rimani in q_2, ossia $\delta(q_2, 0) = q_1$, $\delta(q_2, 1) = q_2$;

Un modo conveniente per rappresentare DFA piccoli è quello di creare un grafo dove:

- per ogni stato $q \in Q$ c'è un nodo chiamato q;
- per ogni stato $q \in Q$ e ogni simbolo $a \in \Sigma$ è presente un arco chiamato a che va dal nodo chiamato q a quello chiamato $\delta(q, a)$;
- il nodo q_0 ha una freccia entrante mentre gli stati finali $q \in F$ sono enfatizzati o hanno una freccia uscente;

Il grafo del DFA creato nell'esempio sarà



Oltre alla rappresentazione grafica, si può creare una rappresentazione sotto forma di tabella per rappresentare le transizioni. Riallacciandoci all'esempio di prima avremo:

	0	1
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_2
$^*q_2$	q_2	q_2

Estendendo la funzione di transizione alle stringhe possiamo definire i linguaggi accettati dai DFA. L'estensione sarà

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$$

La funzione si basa sul principio di induzione ed è definita come segue

$$\hat{\delta}(q, \epsilon) = q$$
$$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$$

Dove $xa = w$ è la parola in ingresso e a è l'ultimo carattere della stringa.
Il passo induttivo si basa nel rimuovere l'ultimo carattere della stringa fino ad ottenere la stringa vuota, per poi usare la funzione di transizione normale per calcolare ogni coppia stato-carattere.
I linguaggi accettati da una DFA $A = (Q, \Sigma, \delta, q_0.F)$ sono

$$L(A) \stackrel{\text{def}}{=} \{w \in \Sigma^* | \hat{\delta}(q_0, w) \in F\}$$

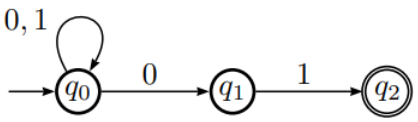
I linguaggi accettati da un automa a stati finiti sono i linguaggi regolari.

2.2 Automi a stati finiti non deterministici

Un **automa a stati finiti non deterministico** (NFA) è una tupla $A = (Q, \Sigma, \delta, q_0, F)$ dove:

- Q è l'insieme finito degli **stati**;
- Σ è l'alfabeto dei **simboli in input**;
- $\delta : Q \times \Sigma \to \wp(Q)$ è la **funzione di transizione**. In questo caso è presente il simbolo speciale ε (nessun simbolo) che permette di cambiare stati senza input;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati **finali** (o **di accettazione**);

Per vedere come funziona un NFA, prendiamo la versione nondeterministica dell'automa dell'esempio della sezione precedente



Ci son diversi cammini chiamati con la stessa stringa

$$q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0$$
$$q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$$

L'automa accetta w se esiste un cammino chiamato w che ci porta in uno stato di accettazione.
L'automa deve compiere una scelta e può seguire una di queste tre interpretazioni:

- L'automa è un **oracolo** e sa già che cammino prendere se la stringa è accettabile;
- Tramite **parallelismo** l'automa si clona e prova tutti i cammini allo stesso tempo. I cloni che non contengono il cammino muoiono e se tutti i cloni son morti allora la stringa non viene accettata;
- Viene usato il **backtracking** ossia ad ogni bivio la macchina si salva il suo stato (stato e posizione della stringa in input) e prova uno dei cammini.
  Se l'automa non può più procedere torna indietro in uno stato salvato e ne prova un altro.

Per definire matematicamente i linguaggi accettati da una macchina nondeterministica l'estensione della funzione di transizione tiene traccia di tutti i stati in cui l'automa potrebbe essere dopo aver letto una stringa in input. L'estensione sarà dunque

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

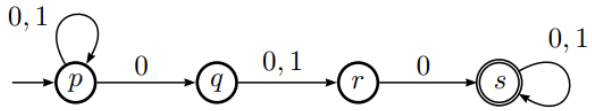
ed è definita come segue

$$\hat{d}(q, \epsilon) = \{q\}$$
$$\hat{d}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$$

Dove $xa = w$ è la parola in ingresso e a è l'ultimo carattere della stringa.
I linguaggi accettati da un NFA saranno dunque

$$L(A) \stackrel{\text{def}}{=} \{w \in \Sigma^* | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Gli automi nondeterministici **non** sono più espressivi dei deterministici. Anzi si dimostra matematicamente che una NFA può essere trasformato in un DFA. Per fare ciò costruiamo un DFA che codifica nei suoi stati un insieme di stati del NFA. C'è però il rischio nel caso pessimo di avere un automa con 2^n stati dove n è il numero di stati del NFA.
Vediamo un esempio di trasformazione e prendiamo il seguente NFA



La sua tabella di transizione è

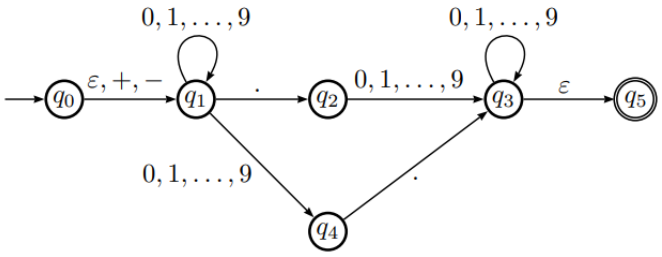
	0	1
$\to p$	$\{p, q\}$	$\{p\}$
p	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$^*s$	$\{s\}$	$\{s\}$

2.3 Automi a stati finiti con ε-transizioni

Un **automa a stati finiti non deterministico con ε-transizioni** (ε-NFA) è una tupla $A = (Q, \Sigma, \delta, q_0, F)$ dove:

- Q è l'insieme finito degli **stati**;
- Σ è l'alfabeto dei **simboli in input**;
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \to \wp(Q)$ è la **funzione di transizione**. δ preso lo stato di sistema attuale e il simbolo in input restituisce un insieme di stati possibili e questo caratterizza il nondeterminismo della macchina;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati **finali** (o **di accettazione**);

Per capire il funzionamento, vediamo il seguente automa che accetta in input qualsiasi numero reale



Notiamo che l'automa può passare direttamente a q_1 partendo da q_0 tramite una ε-transizone: questo è il caso in cui il numero non abbia il segno che per convenzione è positivo.
Un'altra ε-transizione è nell'arco da q_3 a q_5: questo ci serve per passare allo stato di accettazione quando la parte decimale del numero termina.

Come per gli automi non deterministici, cerchiamo di descrivere i linguaggi accettati da un ε-NFA.
Per prima cosa però tocca definire una funzione aggiuntiva, la **ε-chiusura**: dato una stato q, $\varepsilon - chiusura(q)$ è l'insieme degli stati raggiungili da q soltanto tramite ε-transizioni, incluso q stesso. La definizione induttiva e formale di $\varepsilon - chiusura(q)$ è come segue:

- Caso base: $q \in \varepsilon - chiusura(q)$;
- Passo induttivo: se $p \in \varepsilon - chiusura(q)$, allora $\delta(p, \varepsilon) \subseteq \varepsilon - chiusura(q)$;

Si può generalizzare la ε-chiusura anche ad insiemi di stati S

$$\varepsilon - chiusura(S) = \bigcup_{q \in S} \varepsilon - chiusura(q)$$

Prendiamo per esempio l'automa di prima

q	$\varepsilon - chiusura(q)$
q_0	$\{q_0, q_1\}$
q_1	$\{q_1\}$
q_2	$\{q_2\}$
q_3	$\{q_3, q_5\}$
q_4	$\{q_4\}$
q_5	$\{q_5\}$

Per descrivere i linguaggi accettati da una ε-NFA estendiamo la funzione di transizione alle stringhe.
L'estensione è

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

ed è definita come segue

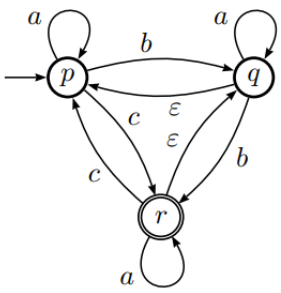
$$\hat{\delta}(q, \varepsilon) = \varepsilon - chiusura(q)$$
$$\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \varepsilon - chiusura(\delta(p, a))$$

I linguaggi accetatti sono dunque

$$L(A) \stackrel{\text{def}}{=} \{w \in \Sigma^* | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

La definizione è uguale a quella degli NFA, ma con l'aggiunta delle ε-chiusure.

Come per i NFA, è sempre possibile costruire un DFA da un ε-NFA. Prendiamo per esempio il seguente automa:



Come per i NFA codifichiamo insiemi di stati in singoli stati.
Partiamo dallo stato iniziale p e vediamo come cambia lo stato:

- Con l'input a rimaniano in p;
- Con l'input b andiamo in q, ma da q possiamo andare di nuovo in p per via della ε-transizione. Andremo perciò all'insieme di stati $\{p, q\}$;

- Con l'input c si va in r, ma per via della ε-transizione si va anche in b. Però come prima, da b si può andare in p per via della transizione speciale;

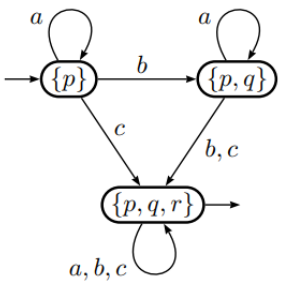
Iniziamo a comporre la tabella delle transizioni

	a	b	c
$\rightarrow \{p\}$	$\{p\}$	$\{p,q\}$	$\{p,q,r\}$

Abbiamo ottenuto due nuovi insiemi di stati, $\{p,q\}$ e $\{p,q,r\}$.
Adesso riapplichiamo lo stesso procedimento per $\{p,q\}$. Per l'input a, sappiamo come si comporta p, q invece rimane invariato e dunque si rimane nello stesso stato.
Per l'input b, q passa a r e dunque l'insieme in cui si passa è $\{p,q,r\}$.
Finiamo con l'insieme $\{p,q,r\}$ che per ogni input rimane nello stesso stato per via delle ε-transizioni presenti in r: questo vuol dire che è uno stato pozzo.
Poiché l'insieme contiene lo stato di accettazione r, esso è uno stato finale.
La tebella delle transizioni sarà perciò:

	a	b	c
$\rightarrow \{p\}$	$\{p\}$	$\{p,q\}$	$\{p,q,r\}$
$\{p,q\}$	$\{p,q\}$	$\{p,q,r\}$	$\{p,q,r\}$
$\{p,q,r\}$	$\{p,q,r\}$	$\{p,q,r\}$	$\{p,q,r\}$

Il grafico dell'automa invece:

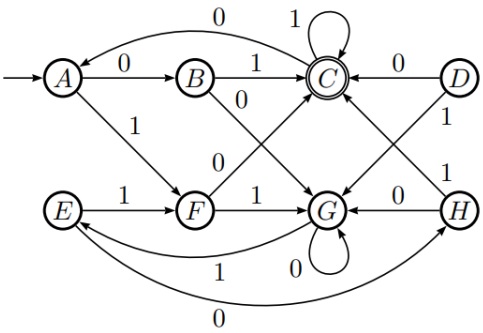


2.4 Minimizzazione ed equivalenza per automi a stati finiti

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA.
Si dice che $p, q \in Q$ sono **indistinguibili**, se per ogni stringa in input w

$$\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$$

$\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$ non sono obbligati a essere lo stesso stato finale.
Se due stati p, q non sono indistinguibili diremo che essi sono **distinguibili**.
Prendiamo per esempio il seguente automa:



- C e G sono distiguibili per ϵ;
- A e G sono distinguibili per 01;
- A e E sono indistinguibili;

Si dimostra che la relazione di indistinguibilità è una relazione di equivalenza: la relazione è ovviamente riflessiva e simmetrica.
Supponiamo per contraddizione che p sia indistinguibile da q, che q sia indistinguibile da r, ma che p sia **distinguibile** da r.
Allora esiste w tale per cui solo uno tra $\hat{\delta}(p, w)$ e $\hat{\delta}(r, w)$ è in F: supponiamo $\hat{\delta}(q, w) \in F$:

- Se $\hat{\delta}(q, w) \in F$ allora q sarebbe distinguibile da r che va contro l'indistinguibilità tra q ed r;
- Se $\hat{\delta}(q, w) \notin F$ allora q sarebbe distinguibile da p che va contro l'indistinguibilità tra q ed p;

La relazione di indistinguibilità crea una partizionamento in classi di equivalenza nell'insieme Q degli stati: se q e p appartengono alla stessa classe allora sono indistinguibili, se invece appartengono a classi diverse sono distinguibili.

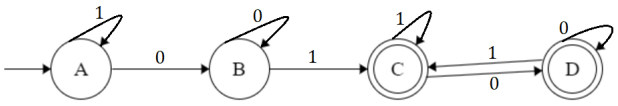
L'algoritmo per vedere se una coppia è distinguibile si basa sul principio di induzione:

- Caso base: se $p \in F$ e $q \notin F$ allora $\{p, q\}$ è distinguibile;
- Induzione: Siano $p, q \in Q$ tali che per un simbolo $a \in \Sigma$ abbiamo $r = \delta(p, a)$, $s = \delta(q, a)$ e $\{r, s\}$ è una coppia distinguibile allora $\{p, q\}$ è distinguibile a sua volta.
  Questo vuol dire che se troviamo un simbolo per cui p e q cambiano in stati distinguibili allora p e q sono distinguibili a loro volta.
  Se non sappiamo nulla degli stati nuovi riapplichiamo il procedimento fino a che non arriviamo al caso base;

L'indistinguibilità ha due applicazioni importanti:

- **Equivalenza tra gli automi**: presa l'unione di automi si verifica se gli stati iniziali di essi sono indistinguibili. Se ciò fosse allora gli automi sono equivalenti;
- **Minimizzazione di automi**: tutti gli stati indistinguibili vengono "fusi" in uno;

Vediamo con un sempio come minimizzare un automa usando l'algoritmo descritto sopra



Per prima cosa creiamo la tabella in cui associamo ogni stato per ogni altro stato dell'automa.
Poichè la distinguibilità è simmetrica basta creare solo metà delle coppie escludendo le coppie di due stati uguali.

B		-	-
C			-
D			
	A	B	C

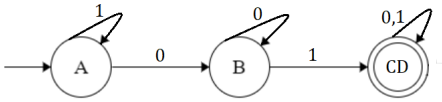
Adesso prendiamo il caso base dell'algoritmo e ci segniamo tutte le coppie formate da uno stato normale e stato di accettazione.

B		-	-
C	✓	✓	-
D	✓	✓	
	A	B	C

Procediamo in modo incrementale.
Prendiamo la coppia (A, B): per l'input 0, A diventa B e B non cambia e ciò non ci interessa; per l'input 1 invece A rimane lo stesso, B cambia invece in C che è stato di accettazione e ciò ci porta al caso base dell'induzione.
Dunque segniamo anche (A, B).

B	✓	-	-
C	✓	✓	-
D	✓	✓	
	A	B	C

Ci rimane l'ultima coppia (C, D) che però per ogni lettera in in put non cambia in una coppia stato normale e stato finito: con 0 diventa (D, D) e con 1 diventa (D, D).
Questo implica che C e D sono indistinguibili tra di loro e possono esser fusi in un unico stato.
L'automa minimizzato perciò si presenta così



Capitolo 3

Linguaggi regolari

3.1 Automi a stati finiti e grammatiche lineari destre

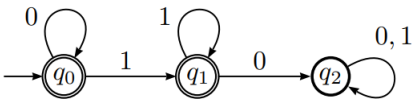
Data un DFA $M = (Q, \Sigma, \delta, q_0, F)$ è possibile defnire una grammatica $G = (Q, \Sigma, q_o, P)$ dove:

- se $\delta(q, a) = p$ allora $q \to ap \in P$;
- se $\delta(q, a) = p$ e $p \in F$ allora $q \to a \in P$;

In tal caso avremo che $L(M) = L(G)$, ossia che il linguaggio accettato dalla macchina è lo stesso generato dalla grammatica.
Ciò che facciamo è rappresentare gli stati dell'automa come variabili nella grammatica. Per esempio prendiamo il seguente linguaggio

$$L = \{0^n 1^m \mid n \geq 0, m \geq 0\}$$

riconosciuto dal seguente automa



La grammatica che generiamo è $G = (\{q_0, q_1, q_2\}, \{0, 1\}, q_0, P)$ dove in P:

- $q_0 \to 0q_0 \mid 1q_1 \mid \epsilon \mid 1 \mid 0$
- $q_1 \to 0q_2 \mid 1q_2 \mid 1$
- $q_2 \to 0q_2 \mid 1q_2$

Notiamo come nella grammatica è inutile avere la variabile q_2 e si può eliminare in quanto ridondante.

É possibile fare anche il viceversa, ossia data una grammatica lineare a destra $G = (V, T, S, P)$ è possibile definire un NFA con ε transizioni $M = (V \cup \{q_f\}, T, \delta, S, \{q_f\})$ che riconosce il linguaggio generato da G, dove:

- se $A \to a \in P$ allora $q_f \in \delta(A, a)$ ossia se la derivazione porta ad un simbolo terminale, allora spostiamo l'automa in uno stato di accettazione tramite la trnsizione chiamata a;
- se $A \to B \in P$ allora $B \in \delta(A, \varepsilon)$ ossia inseriamo la mossa invisibile per cambiare stato senza simbolo in input;
- se $A \to aB \in P$ allora $B \in \delta(A, a)$ ossia la trasizione a ci porta allo stato B;

Si verifica che $L(G) = L(M)$.
In molti casi potremmo avere delle derivazioni tipo $S \to abcA$. In questo caso scomponiamo le derivazioni (gli automi accettano solo un input e rendiamo più semplice il lavoro) e usiamo delle variabili in più: avremo dunque $S \to aA'$, $A' \to bA''$, $A'' \to cA$.
Per esempio prendiamo il seguente linguaggio

$$L = \{01^n \mid n \geq 0\}$$

generato dalla grammatica $G = (\{S, A, B\}, \{0, 1\}, S, P)$ dove P:

- $S \to 0A$;
- $A \to B$;
- $B \to 1 \mid \epsilon \mid 1A$;

Nota bene: P è inefficiente e ridondante solamente per mostrare ogni casisitica nella definizione dell'automa. La derivazione $A \to B$ è ridondante in quanto $S \to 0A$ può esser riscritto come $S \to B$ e $B \to 1A$ può esser riscritto come $B \to 1B$.
Creiamo l'automa che lo riconosce, codificando ogni variabile ad uno stato. Definiamo perciò $A = (\{q_s, q_A, q_B, q_F\}, \{0, 1\}, S, P)$ dove:

- $\delta(q_S, 0) = \{q_A\}$ codifica la prima derivazione;
- $\delta(q_A, \varepsilon) = \{qB\}$ codifica la seconda derivazione;
- $\delta(q_B, \varepsilon) = \{q_F\}$ codifica la derivazione $B \to \epsilon$;
- $\delta(q_B, 1) = \{q_A, q_F\}$ codifica le derivazioni $B \to 1 \mid 1A$;

3.2 Proprietà dei linguaggi regolari e pumping lemma

Siano L_1 e L_2 linguaggi regolari su Σ_1 e Σ_2 rispettivamente, allora:

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$;
- $\overline{L_1} = \{w \in \Sigma_1^* \mid w \notin L_1\}$;
- $L_1 L_2 = \{w_1 w_1 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$;
- $L_1^* = \{\varepsilon\} \cup L_1 \cup L_1 L_1 \cup \ldots$;
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$;

son linguaggi regolari.
Questo vuol dire che le operazioni descritte sono chiuse rispetto ai linguaggi e possiamo sfruttare questre proprietà nella costruzione degli automi.
Infatti si possono costruire automi che riconoscono i linguaggi ottenuti dalle operazione su di essi.
In questo caso conviene avere un unico stato iniziale ed un unico stato finale.

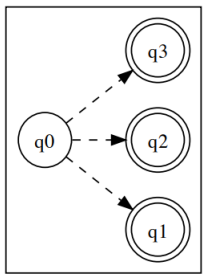
Per ogni ε-NFA $M = (Q, \Sigma, \delta, q_0, F)$ esiste un ε-NFA N tale che $L(M) = L(N)$ e N ha esattamente un solo stato finale.
Per far ciò prendiamo $q_f \notin F$ e definiamo

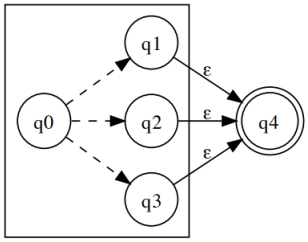
$$N \stackrel{\text{def}}{=} (Q \cup \{q_f\}, \Sigma, \delta', q_0, \{q_f\})$$

dove δ' estende δ con

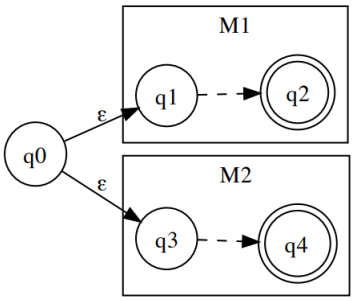
$$\delta'(q, \varepsilon) = \{q_f\}$$

per ogni $q \in F$. Come esempio vediamo il seguente automa a cui se ne trova uno corrispondente con un solo stato finale

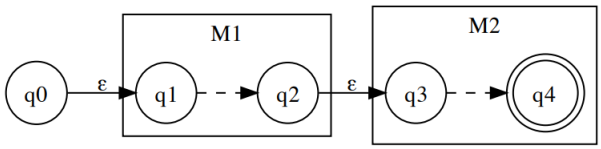




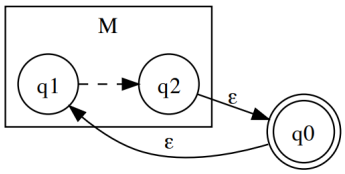
Adesso proviamo come le operazione su linguaggi regolari generano linguaggi regolari.
Partiamo con **l'unione**: se $L_1 = L(M_1)$ e $L_2 = L(M_2)$ allora l'unione dei due linguaggi deve essere accettato da una macchina deterministica.
Per far ciò basta far eseguire M_1 ed M_2 in parallelo e creiamo uno stato



iniziale che va alle due macchine tramite ε-transizioni.
Per la **concatenazione** se $L_1 = L(M_1)$ e $L_2 = L(M_2)$, per accettare $L_1 L_2$ basta mettere M_2 dopo M_1.



Per la **stella di Kleene** basta far reiterare la macchina per ogni nuova parola che viene inserita.



Unione, concatenazione e stella di Kleene sono funzionalmente complementi, ossia tutti gli altri operatori possono esser derivati da essi e perciò è possibile costruire ogni linguaggio regolare a partire da essi.
Per **l'intersezione** la faccenda è più complicata. Se L_1 è riconosciuta da $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ e L_2 è riconosciuta da $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$, l'automa che riconosce $L_1 \cap L_2$ sarà $M = (Q_1 \times Q_2, \Sigma, \delta, (q_1, q_2), F)$ dove

$$\delta((p,q),a) = (\delta_1(p,a), \delta_2(q,a))$$
$$F = \{(p,q) \in Q_1 \times Q_2 \,|\, p \in F_1 \wedge q \in F_2\}$$

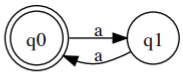
In parole povere l'automa esegue M_1 e M_2 in contemporanea e ogni stato dell'automa M rappresenta una coppia di stati, uno da M_1 e l'altro da M_2. Lo stato iniziale è la coppia degli stati iniziali dei due automi e ogni transizione è determinata eseguendo la mossa corrispettiva in M_1 e in M_2.
Uno stato (p, q) è di accettazione se entrambi p e q sono di accettazione nei

loro rispettivi automi. In alternativa si può fare

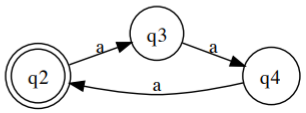
$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Come esempio vediamo i seguenti linguaggi e gli automi che li riconoscono

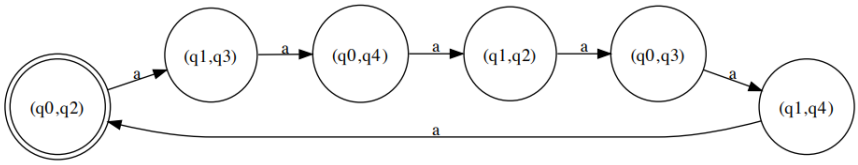
$$L_1 = \{a^{2n} \mid n \geq 0\}$$



$$L_2 = \{a^{3n} \mid n \geq 0\}$$



L'intersezione dei due linguaggi è $L_1 \cap L_2 = \{a^{6n} \mid n \geq 0\}$.
L'automa che lo riconosce è il seguente



Per il **complemento** la situazione è più semplice in quanto basta prendere l'automa che riconosce L e rendere gli stati di non accettazione in stati di accettazione e gli stati di accettazione in stati di non accettazione.
La **differenza** si ottiene tramite intersezione e complemento, i.e. $L_1 \setminus L_2 = L_1 \cap L_2$.

Tutte le proprietà dimostrate fino ad adesso sono *direzionali* : se due linguaggi sono regolari allora *unione/intersezione/differenza* dei due linguaggi sono a loro volta regolari, ma il viceversa in generale non vale.
Per dimostrare ciò basta prendere il seguente linguaggio

$$\{a, b\}^* \cup \{a^n b^n \mid n \geq 0\}$$

Il linguaggio è regolare in quanto $\{a, b\}^*$ è regolare in quanto comprende qualsiasi parola formata dalle lettere a e b e perciò $\{a, b\}^* \cup \{a^n b^n \mid n \geq 0\} = \{a, b\}^*$.
Però $\{a^n b^n \mid n \geq 0\}$ non è un linguaggio regolare!
Inoltre se solo uno tra L_1 o L_2 è regolare allora non è detto che *unione/intersezione/differenza* di essi sia regolare.
Come esempio basta prendere

$$\emptyset \cup \{a^n b^n \mid n \geq 0\}$$

L'insieme vuoto è regolare e l'unione con il linguaggio a destra è il linguaggio stesso che però non è regolare.

Ma come facciamo a dire che un linguaggio non è regolare? Non trovare un automa che lo riconosca non vuol dire niente.
Dobbiamo trovare una proprietà che tutti i linguaggi regolari soddisfano e se un linguaggio non lo soddisfa allora esso non è regolare.
Ci aiuta in questo caso il seguente teorema

Pumping Lemma. *Sia L un linguaggio regolare.*
Allora esiste una costante n tale che per ogni $w \in L$ esiste una decomposizione $w = xyz$ per cui

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. *Per ogni* $k \geq 0$, $xy^k z \in L$

A parole ogni stringa sufficientementa lunga $w \in L$ può essere decomposta in modo tale che un infisso y non troppo distante dall'inizio di w può essere "pompata" un qualsiasi numero di volte, ossia ripetuta più volte, e ottenere una stringa comunque appartenente a L.

Dimostrazione. Se L è un linguaggio regolare, allora è accettato da un'automa deterministico $M = (Q, \Sigma, \delta, q_0, F)$.
Sia $n = |Q|$ ossia il numero di stati.
Sia $w = a_1 a_2 \dots a_n$ una stringa accettata dall'automa tale che $m \geq n$ e $a_i \in \Sigma$.
Poichè l'automa è deterministico, dato un prefisso di w $a_1 a_2 \dots a_i$ esso si troverà in uno stato p_i.
Dunque leggendo la stringa w_0 l'automa seguirà la serie $p_0, p_1, p_2, \dots, p_m$ di stati, dove p_m è uno stato di accettazione in quanto w viene accettata.
La sequenza ha $m + 1$ stati e sappiamo che $m \geq n$: questo vuol dire che alcuni stati p_i sono uguali, ossia vengono ripetuti nel percorso w.
Allora devono esistere per forza due indici i e j, $0 \leq i < j \leq n$ tali che

$$p_i = p_j$$

Prendiamo gli indici i e j più piccoli tali per cui $p_i = p_j$ e scomponiamo la stringa $w = xyz$ così come segue:

- $x = a_1 a_2 \dots a_i$ è il prefisso riconosciuto dall'automa dallo stato iniziale alla prima occorrenza di p_i;
- $y = a_{i+i} a_{i+2} \dots aj$ è la sottostringa di w che segue x riconosciuta dall'automa tra le prime due occorrenze di p_i (ricordandoci che $p_i = p_j$).

- $z = a_{j+1}a_{j+2}\dots a_m$ è il suffisso di w riconosciuta dall'automa dalla prima ricorrenza ripetuta di p_j a p_m.

Ci rimane da verificare se la decomposizione ottenuta rispetta le tre condizioni.
Notiamo che:

1. $y \neq \epsilon$ in quanto $i < j$ e l'automa è deterministico e non ha ε-transizioni: questo vuol dire che per passare da p_i a p_j è stato letto almeno un simbolo.

2. $|xy| \leq n$ poichè i e j sono gli indici più piccoli per cui $p_i = p_j$.
   Quindi gli stati $p_0, p_1, \dots, p_i, \dots, p_{j-1}$ son tutti distinti e il loro numero non supera n in quanto l'automa ha n stati.

3. Poiché p_0 è lo stato iniziale e p_m lo stato finale, è facile verificare che rimuovendo y da xyz, considerando perciò xz, otteniamo un altra stringa accettata dall'automa.
   In aggiunta possiamo ripetere il loop da p_i a p_j due volte e riconoscere $xyyz$. La ripetizione del loop può esser fatta in un numero arbitrario di volte e dunque $xy^kz \in L$ per ogni $k \geq 0$.

Vediamo con un esempio come dimostrare che un linguaggio non è regolare tramite il Pumping Lemma.
Supponiamo che $L = \{0^m1^m \mid m \geq 0\}$ sia regolare e sia n la costante associata a L dal Pumping Lemma.
Consideriamo $w = 0^n1^n \in L$ e assumiamo $w = xyx$ tale che

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. Per ogni $k \geq 0$, $xy^kz \in L$

Poiché

- $|xy| \leq n$, gli x e y son composti solo da 0 poiché $w = 0^n1^n$ comincia con n zero.
- $y \neq \varepsilon$ allora $|x| < n$

Sorge una contraddizione quando $k = 0$, ossia la stringa $xy^0z = xz$ non può appartenere a L. Infatti z è composto solo da 1 e se rimuoviamo la sottostringa y otteniamo una parola che ha meno 0 mentre gli 1 rimangano invariati.
Questo implica che L non è un linguaggio regolare.

Il problema del Pumping Lemma è che è *direzionale*: se un linguaggio è regolare allora soddisfa le condizioni del lemma, ma il viceversa in generale non vale.
Questo vuol dire che esistono linguaggi non regolari che soddisfano le condizioni del Pumping Lemma e dunque un linguaggio che soddisfa le tre condizioni non può esser considerato regolare.
Per dimostrare la non regolarità di un linguaggio si usano le proprietà di chiusura dei linguaggi regolari e si considera il linguaggio come combinazioni di altri linguaggi di cui uno non è regolare. Da li in poi è sufficiente provare per contraddizione che il linguaggio non è regolare.
Vediamo con un esempio.
Il linguaggio $L = \{0^m 1^p \,|\, m, p \geq 0, m \neq 0\}$ soddisfa tutte le condizioni del Pumping Lemma per $y \geq 2$.
Osserviamo che:

- $\overline{L} = L_1 \cup L_2$ dove $L_1 = \{0^n 1^n \,|\, n \geq 0\}$ e $L_2 = \{x1y0z \,|\, x, y, z \in \{0, 1\}\}$
- L_1 abbiamo dimostrato che non è regolare e che L_2 è regolare in quanto si può costruire il DFA che lo accetta.

Adesso dimostriamo per contraddizione: se L è regolare allora lo è pure $\overline{L}$. Se $\overline{L}$ è regolare allora per le proprietà di chiusura dei linguaggi regolari $\overline{L} \setminus L$ è regolare. Ma $\overline{L} \setminus L = L_1$ che non è regolare e si giunge ad una contraddizione. Ciò implica che L non è regolare.

3.3 Espressioni regolari

Oltre alle grammatiche è possibile formalizzare i linguaggi regolari tramite le **espressioni regolari**.
Le espressioni regolari rappresentano linguaggi regolari in modo più capibile. Esse sono un'algebra costruita tramite gli elementi atomici di base $\emptyset$, $\{\varepsilon\}$, $\{0\}, \{1\}$ e sfruttando le operazioni di chiusura dei linguaggi regolari.
Vediamo sintassi e semantica di un espressione regolare:

Sintassi E	Semantica $L(E)$
ε	$\{\varepsilon\}$
$\emptyset$	$\emptyset$
a	$\{a\}\ \ a \in \Sigma$
$E_1 + E_2$	$L(E_1) \cup L(E_2)$
$E_1 E_2$	$L(E_1) \cap L(E_2)$
E^*	$L(E)^*$
(E)	$L(E)$

Per le proprietà di chiusura dei linguaggi regolari, un espressione regolare genererà sempre e soltanto linguaggi regolari. Per convenzione $*$ ha la precedenza su tutti gli altri operatori, seguita dalla concatenazione che è a sua volta seguita dal '+'.
Vediamo degli esempi di espressioni regolari e del loro significato

- ab $\rightarrow L(ab) = L(a)L(b) = \{ab\}$;
- a+b $\rightarrow L(a) \cup L(b) = \{a, b\}$;
- a* $\rightarrow \{\varepsilon\} \cup L(a) \cup L(a)L(a) \cup \cdots = \{a^n \,|\, n \geq 0\}$;
- a*b* $\rightarrow L(a*)L(b*) = \{a^m \,|\, m \geq 0\}\{b^n \,|\, n \geq 0\} = \{a^m b^n \,|\, m, n \geq 0\}$;
- (a+b)* $\rightarrow L((ab)*) = L(ab)^* = \{\varepsilon, ab, abab, ababab, \dots\} =$
  $= \{(ab)^n \,|\, n \geq 0\}$;

Adesso proviamo di partire da un linguaggio a creare l'espressione regolare che lo descrive. Prendiamo L come il linguaggio degli 0 e 1 in posizioni alternate ossia $L = \{\varepsilon, 0, 1, 01, 10, 01010, \dots\}$.
Al primo colpo uno penserebbe di scrivere $(01)^*$ ma ciò non va bene in quando non include 10. Si può rimediare con $(01)^* + (10)^*$. Ma anche ciò non va bene in quanto non vengono generati 101 e 010.
L'espressione regolare che genera il linguaggio è $(01)^*+(10)^*+1(01)^*+0(10)^*$ che ottimizzato e ridotto delle ridondanze diventa $(1 + \varepsilon)(01)^*(0 + \varepsilon)$.

Un problema delle espressioni regolari è che è difficile crearle per i linguaggi la cui descrizione si appoggia delle negazioni.

Questo perché il complemento non è nell'algebra.
In questi casi riscriviamo la descrizione del linguaggio in modo diverso.

Vediamo infine le proprietà algebriche delle espressioni regolari:

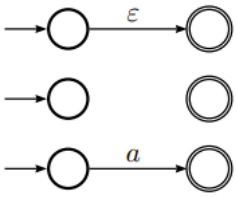
$$
\begin{array}{c}
\hline\hline
E + F = F + E \\
(E + F) + G = E + (F + G) \\
(EF)G = E(FG) \\
\emptyset + E = E + \emptyset = E \\
\varepsilon E = E\varepsilon = E \\
\emptyset E = E\emptyset = \emptyset \\
E + E = E \\
(E^*)^* = E^* \\
\varepsilon * = \varepsilon \\
E(F + G) = EF + EG \\
(F + G)E = FE + GE \\
\hline\hline
\end{array}
$$

3.4 Relazione tra espressioni regolari e automi a stati finiti

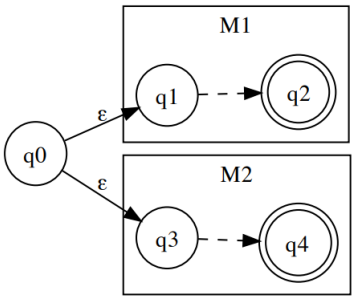
Si dimostra che le espressioni regolari non sono più espressive di un automa a stati finiti.
Data una espressione regolare E è possibile costruire tramite induzione una ε-NFA M tale per cui $L(E) = L(M)$.
I casi base sono

- $E = \varepsilon$;
- $E = \emptyset$;
- $E =$ a;

che possono essere traformati rispettivamente in



Come passo induttivo abbiamo $E = E_1 + E_2$.
Sappiamo che dalle ipotesi che esistono degli automi M_1 e M_2 tali che $L(M_1) = L(E_1)$ e $L(M_2) = L(E_2)$.
Da definizione $L(E_1 + E_2) = L(E_1) \cup L(E_2)$ e dunque per costruire M basta fare l'unione di M_1 e M_2:



Ogni linguaggio generato da un'espressione regolare può anche essere riconosciuto da un automa a stati finiti. Ciò implica che le espressioni regolari non sono più espressive degli automi a stati finiti.

Passare invece da un automa a una espressione regolare è invece difficile: l'automa non ha una struttura e non si può fare induzione sui suoi stati.
Il problema in questo caso è dato un DFA $M = (\{q_1, \ldots, q_n\}, \Sigma, \delta, q_1, F)$, si

deve creare una espressione regolare E tale per cui $L(M) = L(E)$.
Diremo che

$$q_{i_0} \xrightarrow{a_1} q_{i_1} \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_{i_n}$$

è un **cammino** da q_{i_0} a q_{i_n} con **etichetta** $a_1 a_2 \dots a_n$ se

$$\delta(q_{i_k}, a_{k+1}) = q_{i_{k+1}}$$

Scriviamo $q_i \xrightarrow{w}_k q_j$ se esiste un cammino da q_i a q_j con etichetta w che **non** passa mai uno stato con indice superiore a k. Gli indici i e j possono essere superiori a k.
Sia $R_{ij}^{(k)}$ un'espressione regolare tale per cui

$$L\left(R_{ij}^{(k)}\right) = \{w \in \Sigma^* \mid q_i \xrightarrow{w}_k q_j\}$$

Ossia $R_{ij}^{(k)}$ è un'espressione regolare che genera tutte e solo le etichette dei cammini da q_i a q_j che non passano mai in uno stato il cui indice è superiore a k.
Valgono:

1. Per ogni $q_j \in F$, $L\left(R_{1j}^{(n)}\right)$ è l'insieme di tutte le parole accettate da M nello stato finale q_j (notare che lo stato iniziale è q_1 e l'automa non ha stati con indice superiore a n);
2. $L(M) = \{w \mid \hat{\delta}(q_1, w) \in F\} = \bigcup_{q_j \in F} L\left(R_{1j}^{(n)}\right) = L\left(\sum_{q_j \in F} R_{1j}^{(n)}\right)$

Definiamo $R_{ij}^{(k)}$ per induzione su k.
Il caso base è $k = 0$ e poiché gli stati son numerati da 1, tra q_i e q_j non possono esserci stati intermedi. Avremo due alternative:

- Se $i \neq j$ consideriamo tutti i cammini diretti da q_i a q_j;
- Se $i = j$ consideriamo tutti i loop in q_i di lunghezza 1 e consideriamo pure il cammino di lunghezza 0 etichettato con ε;

Ciò che otteniamo è

$$R_{ij}^{(0)} = \begin{cases} \sum_{\delta(q_i, a_l) = q_j} a_l, & \text{se } i \neq j \\ \varepsilon + \sum_{\delta(q_i, a_l) = q_i} a_l & \text{se } i = j \end{cases}$$

Nel caso avessimo $k > 0$ consideriamo il cammino con che va da q_i a q_j che non passa mai per stati con indice superiore a k. Abbiamo due alternative

- Il cammino non passa per q_k e dunque l'etichetta del cammino appartiene a $L\left(R_{ij}^{(k-1)}\right)$;

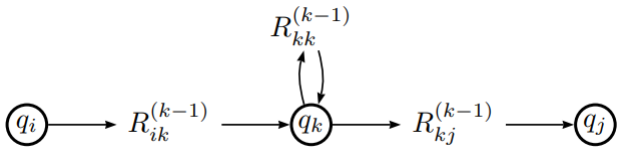
- Il cammino attraversa q_k e perciò lo decomponiamo come segue;

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$$

dove

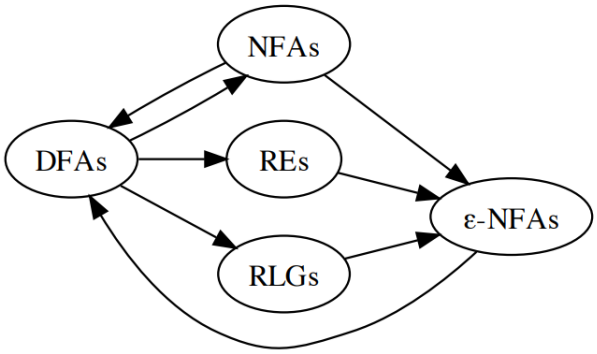
- R_{ij}^{k-1} sono i cammini diretti da q_i a q_j che non passano per q_k e per stati con indice superiore a $k-1$;
- $R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$ è la concatenazione dei cammini da q_i a q_k, i loop in q_k e i cammini che da q_k vanno a q_j;

Visualizzandolo tramite i stati dell'automa abbiamo: Perciò per trasformare



un automa in espressione regolare troviamo i cammini per ogni coppia di stati e per ogni coppia creiamo ogni espressione per k fino a n.

Tramite il seguente diagramma possiamo vedere la relazione tra automi, linguaggi regolare ed espressioni regolari. La freccia indica "si può trasformare in":



Capitolo 4

Linguaggi liberi

4.1 Grammatiche libere e alberi sintattici

Le grammatiche libere sono grammatiche con produzioni del tipo $A \to \alpha$.
Chiamiamo **derivazione a sinistra** $\Rightarrow_l$ (**derivazione a destra** $\Rightarrow_r$) una derivazione

$$\gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n$$

tale per cui in ogni γ_i $(1 \leq i \leq n-1)$ la variabile più a sinistra (a destra) viene espansa.
Diremo che:

- Se $S \Rightarrow^* \alpha$, α è una forma sentenziale;
- Se $S \Rightarrow_l^* \alpha$, α è una forma sentenziale a sinistra;
- Se $S \Rightarrow_r^* \alpha$, α è una forma sentenziale a destra;

Gli **alberi sintattici** di $G = (V, T, S, P)$ sono alberi tali per cui:

- Ogni nodo interno è etichettato con un simbolo in V;
- Ogni foglia è etichettata con un simbolo in $V \cup T \cup \{\varepsilon\}$;
- Se un nodo interno A ha i figli etichettati con

  $$X_1, X_2, \ldots, X_n$$

  (da sinistra a destra), allora

  $$A \to X_1 X_2 \ldots X_n \in P$$

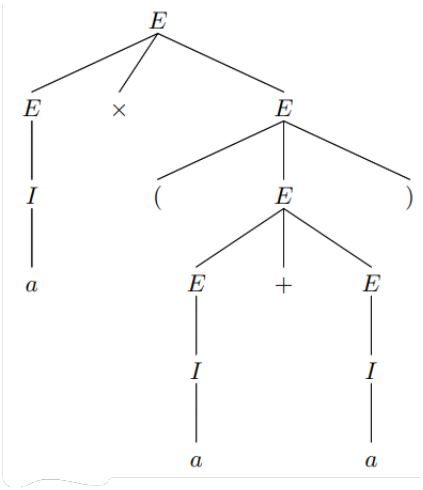
- $X_i = \varepsilon$ solo quando $X_1 X_2 \cdots = \varepsilon$ e $A \to \varepsilon \in P$;

Lo **yield** di un albero sintattico è la stringa ottenuta concatenando le etichette delle sue foglie da sinistra verso destra.
Vale perciò la seguente proposizione:

$$S \Rightarrow^* \alpha \Leftrightarrow \textit{esiste un albero sintattico con radice } S \textit{ e yield } \alpha$$

Vediamo come esempio l'albero sintattico di $a \times (a + a)$ generata tramite le seguenti produzioni

- $E \to E + E \,|\, E \times E \,|\, (E) \,|\, I$;
- $I \to a$;



Può capitare che ci possono essere stringhe associate a più alberi sintattici, i.e. dato $w \in L(G)$ se ci sono due o più alberi sintattici con yield w, allora G è ambigua.
Vale dire anche che dato $w \in L(G)$ se w ha due o più produzioni a sinistra (o a destra) allora G è ambigua.
Questo vuol dire che ci sono diverse interpretazioni per w generando un'ambiguità che deve essere evitata, specialmente nei compilatori.
L'ambiguità è difficile da gestire in quanto il problema dell'ambiguità (decidere se una grammatica è ambigua o meno) è indecidibile.
Inoltre alcuni linguaggi sono *intrinsecamente* ambigui.
Un linguaggio L si dice intrinsecamente ambiguo se ogni grammatica per cui $L(G) = L$ è ambigua, ossia non esista una grammatica non ambigua che la genera.
Un linguaggio L è non ambiguo se esiste una grammatica G per cui $L(G) = L$ non è ambigua.

Vediamo un esempio di linguaggio intrinsecamente ambiguo:

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

Il linguaggio è generato da:

- $S \to AB \mid C$;
- $A \to aAb \mid ab$;
- $B \to cBd \mid cd$;
- $C \to aCd \mid aDd$;
- $D \to bDc \mid bc$;

Se andiamo a vedere, ci saranno sempre due modi per generare $a^n b^n c^n d^n$.

4.2 Grammatiche libere in f. normale di Chomsky

Sia $G = (V, T, S, P)$ una grammatica libera, diremo che:

- $X \in V \cup T$ è **utile** se esiste una derivazione $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ con $w \in T$;
- Se X non è utile, allora è **inutile**

A parole e intuitivamente, se un simbolo è utile esso è presente in una serie di derivazioni (in una forma sentenziale) che permette di creare una parola. Banalmente se un simbolo non permette la creazione di una parola oppure non compare in nessuna forma sentenziale e perciò non contribuisce a creare parole, allora è inutile e può essere eliminato dalla grammatica.
Diremo inoltre che:

- X è un **generatore** se esiste $w \in T^*$ tale per cui $X \Rightarrow^* w$ (ogni terminale è generatore), ossia un simbolo è generatore se esiste una derivazione che partendo dal simbolo arriva ad una parola di terminali;
- X è **raggiungibile** se esiste una derivazione $S \Rightarrow^* \alpha X \beta$ per qualche $\alpha, \beta \in (V \cup T)^*$, ossia esiste una forma sentenziale che lo genera;

Vale la seguente proposizione: se X è utile allora X è generatore e raggiungibile.

Gli algoritmi per trovare i simboli generatori e raggiungibili sfruttano l'induzione.
L'algoritmo per i simboli generatori è definito come segue:

- Base: ogni simbolo in T è generatore;
- Induzione: sia $A \to \alpha$ una produzione in P. Se ogni simbolo in α è generatore allora A è generatore. Se nel caso $\alpha = \varepsilon$, A è generatore;

Invece l'algoritmo per i simboli raggiungibili è definito come segue:

- Base: S è raggiungibile;
- Induzione: se A è raggiungibile e $A \to \alpha \in P$, allora ogni simbolo in α è raggiungibile;

Sfruttando questi due algoritmi troviamo i passi per l'algoritmo che rimuove i simboli inutili da una grammatica G dove $L(G) \neq \emptyset$:

1. Partendo da G creiamo una grammatica G' eliminando i simboli non generatori da G. Poichè $L(G) \neq \emptyset$, S deve essere per forza generatore;
2. Prendendo G', creiamo la grammatica G'' eliminando i simboli non raggiungibili in G';

Avremo che G'' non ha simboli inutili e $L(G'') = L(G)$.
Vediamo un esempio: consideriamo G con le seguenti produzioni:

$$S \to AB \,|\, a$$
$$A \to b$$

I simboli generatori sono $\{a, b, S, A\}$. Consideriamo G':

$$S \to a$$
$$A \to b$$

I simboli raggiungibili sono $\{a, S\}$. Otteniamo infine G''

$$S \to a$$

Continuando con le definizioni, diremo che una variabile A è **annullabile** se $A \Rightarrow^* \varepsilon$.
L'intuizione è che se abbiamo una produzione $B \to CAD$, possiamo scrivere due versioni di essa

$$B \to CD$$
$$B \to CAD$$

per poi omettere tutte le produzioin che rendono A annullabile. Notiamo che se $S \Rightarrow^* \varepsilon$ non possiamo eliminare tutte le ε-produzioni.
Inoltre se $L = L(G)$, possiamo trovare una grammatica G' tale per cui $L(G') = L(G) \setminus \{\varepsilon\}$.
L'algoritmo per trovare i simboli annullabili è definito tramite induzione:

- Base: se $A \to \varepsilon \in P$, allora A è annullabile;
- Induzione: se $B \to C_1 C_2 \dots C_k \in P$ e ogni C_i è annullabile, allora B è annullabile;

Vediamo come eliminare le variabili annullabili con il seguente esempio, prendendo G come segue:

$$S \to AB$$
$$A \to aAA \,|\, \varepsilon$$
$$B \to bBB \,|\, \varepsilon$$

Tutte e tre le variabili sono annullabili, quello che otteniamo è G_1:

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

Data una grammatica $G = (V, T, S, P)$ una produzione del tipo $A \to B$ viene chiamata **unitaria**.
Per esempio:

$$\begin{aligned} I &\to a \\ F &\to I \,|\, (\varepsilon) \\ T &\to F \,|\, T \times F \\ E &\to T \,|\, E + T \end{aligned}$$

Intuitivamente possiamo rimuovere $F \to I$ espandendo I e ottenendo $F \to a$.
Chiamiamo (A, B) una coppia unitaria se $A \Rightarrow^* B$.
L'algoritmo che trova le coppie unitarie è anch'esso induttivo ed è definito come segue:

- Base: per ogni variabile A, (A, A) è una coppia unitaria.
- Induzione: se (A, B) è una coppia unitaria e $B \to C$ è una produzione, allora (A, C) è una coppia unitaria.

Tramite questo algoritmo possiamo eliminare le produzioni unitarie in una grammatica G:

1. Calcolo le coppie unitarie in G;
2. Definisco $G' = (V, T, S, P')$ dove P' è l'insieme delle produzioni $A \to \alpha$ tali per cui
   - $B \to \alpha \in P$;
   - $B \to \alpha$ non è una produzione unitaria;
   - (A, B) è una coppia unitaria;

Si verifica che $L(G') = L(G)$.
Prendendo l'esempio a inizio pagina proviamo ad applicare l'algoritmo appena descritto:

Coppia	Produzione
(E, E)	$E \to E + T$
(E, T)	$E \to T \times F$
(E, F)	$E \to (E)$
(E, I)	$E \to a$
(T, T)	$T \to T \times F$
(T, F)	$T \to (E)$
(T, I)	$T \to a$
(F, F)	$F \to (E)$
(F, I)	$F \to a$
(I, I)	$I \to a$

É possibile semplificare ogni grammatica libera grazie al seguente teorema.

Teorema. *Se G è una grammatica libera che genera una stringa $w \neq \varepsilon$ allora esiste una grammatica libera G' tale per cui $L(G') = L(G) \setminus \{\varepsilon\}$ e G' non ha ε-produzioni, produzioni unitarie e simboli inutili.*

Per semplificare una grammatica libera basta seguire i seguenti passi:

1. Eliminare le ε-produzioni;
2. Eliminare le produzioni unitarie;
3. Eliminare simboli inutili;

Una grammatica si dice in **forma normale di Chomsky** se ogni sua produzione è in una di queste due forme:

1. $A \to BC$ dove A, B, C sono variabili;
2. $A \to a$ dove A è una variabile e a è un terminale e non ci sono simboli inutili.

Teorema. *Se G è una grammatica libera che genera almeno una stringa $w \neq \varepsilon$, allora esiste una grammatica G' in forma normale di Chomsky tale per cui $L(G') = L(G) \setminus \varepsilon$.*

Qualsiasi grammatica può esser trasformata in forma normale di Chomsky seguendo questi passi:

1. Semplifichiamo la grammatica in modo tale da eliminare ϵ-produzioni, produzioni unitarie e simboli inutili;
2. Le produzioni rimanenti saranno soltanto del tipo $A \to a$ o avranno la parte a destra della freccia composta da 2 o più lettere (esempio $A \to BCD$ oppure $A \to aP$). Le prime non vengono toccate;
3. Nelle produzioni in cui son presenti terminali a, creiamo una nuova variabile A e aggiungiamo la produzione $A \to a$ (per esempio se abbiamo $B \to bC$ creiamo $B' \to b$);
4. Ogni occorrenza dei terminali a viene sostituita dalla variabile appena creata (per esempio $B \to bC$ diventa $B \to B'C$);
5. Per ogni produzione $A \to B_1 B_2 \dots B_k$ con $k \geq 3$ si creano le variabili $C_1, C_2, \dots, C_{k-2}$ e rimpizziamo le produzioni in questo modo

$$A \to B_1 C_1 \quad C_1 \to B_2 C_2 \quad \cdots \quad C_{k-2} \to B_{k-1} B_k$$

Vediamo con un esempio come trasformare una grammatica in forma normale di Chomsky, consideriamo G:

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

Rimuoviamo le produzioni unitarie:

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

Creiamo due nuove variabili A' e B' per generare rispettivamente a e b quando non compaiono da sole:

$$\begin{array}{lcl} S & \to & a \mid A'A \mid A'AA \mid b \mid B'B \mid B'BB \mid AB \\ A & \to & a \mid A'A \mid A'AA \\ B & \to & b \mid B'B \mid B'BB \end{array} \qquad \begin{array}{lcl} A' & \to & a \\ B' & \to & b \end{array}$$

Infine sistemiamo le produzioni con un corpo più lungo di 3 caratteri:

$$\begin{array}{lcl} S & \to & a \mid A'A \mid A'A'' \mid b \mid B'B \mid B'B'' \mid AB \\ A & \to & a \mid A'A \mid A'A'' \\ B & \to & b \mid B'B \mid B'B'' \end{array} \qquad \begin{array}{lcl} A' & \to & a \\ B' & \to & b \\ A'' & \to & AA \\ B'' & \to & BB \end{array}$$

4.3 Proprietà dei linguaggi liberi e pumping lemma

Prima di parlare del pumping Lemma per i linguaggi liberi è necessario il seguente teorema.

Teorema (Yield Upperbound). *Sia w lo yield di un albero sintattico di una grammatica in forma normale di Chomsky tale per cui la sua profondità (ossia il numero di archi del cammino radice-foglia più lungo) è h, allora $|w| \leq 2^{h-1}$.*

Dimostrazione. La dimostrazione avviene per induzione su h (profondità dell'albero sintattico).

- Base ($h = 1$): l'albero deve essere della forma $A \to a$ e perciò lo yield è a e perciò $h = 1$.
  Perciò avremo che $|a| = 1 = 2^{h-1} = 2^0$;
- Induzione ($h > 1$): L'albero deve essere della forma $A \to BC$.
  I sottoalberi con radice B e C non possono essere più profondi di $h-1$ da ipotesi (facciamo un salto per andare da A alle variabili).
  Dunque il loro yield massimo non può superare 2^{h-2}.
  Ciò implica che lo yield totale non potrà superare $2^{h-2} + 2^{h-2} = 2 \cdot 2^{h-2} = 2^{h-1}$;

Pumping Lemma (per linguaggi liberi)**.** *Sia L un linguaggio libero.*
Allora esiste una costante n tale per cui per ogni $z \in L$ con $|z| \geq n$ esiste una decomposizione $z = uvwxy$ tale per cui

1. *$|vwx| \leq n$;*
2. *$vx \neq \varepsilon$;*
3. *Per ogni $i \geq 0$, $uv^iwx^iy \in L$;*

Dimostrazione. Sia L generato dalla grammatica $G = (V, T, S, P)$ con G in forma normale di Chomsky e sia $n = 2^{|V|}$ e z una stringa in L tale per cui $|z| \geq n$.
Dal teorema di prima sappiamo che in un albero con profondità $|V|$, lo yield non supera $2^{|V|-1}$.
Poiché $n = 2^{|V|}$ la profondità dell'albero sintattico deve essere almeno $|V|+1$.
Ciò vuol dire che il cammino più lungo dalla radice alla foglia ha etichetta $A_0, A_1, \ldots, A_{|V|}, a$.
Come si nota, le variabili nel cammino sono $V + 1$ e perciò due di esse sono per forza uguali.
Scegliamo $A_i = A_j$ tali per cui $i < j$ e A_i è la prima variabile ripetuta visitata a ritroso dalla foglia:

- Sia w lo yield del sottoalbero sintattico con radice A_j;
- Sia vwx lo yield del sottoalbero sintattico con radice A_i;
- Sia $uvwxy$ lo yield dell'intero albero sintattico con radice S;

Poiché $A_i = A_j$ possiamo sostituire l'albero con radice A_i con quello con radice A_j in quanto la grammatica è libera. Ma ciò vuol dire che

$$uvw \in L(G)$$

Analogamente possiamo sostituire l'albero con radice A_j con un'altra istanza dell'albero con radice A_i e avere che

$$uv^2wx^2y \in L(G)$$

Iterando quest'ultimo procedimento otteniamo che

$$uv^iwx^iy \in L(G)$$

Per quanto riguarda $|vx| \neq 0$, non possiamo avere entrambe le stringhe vuote in quanto $i < j$ e la grammatica non ammette ε-produzioni e produzioni unitarie: per passare da A_i ad A_j deve esser stato generato almeno un simbolo terminale.
Infine, poichè abbiamo preso A_i come prima ripetizione di A_j, la profondità dell'albero con radice A_i non può essere superiore a $|V|+1$: nel caso peggiore visito tutte le variabili, A_i viene visitata due volte e termino in una foglia.
Ricordandoci che A_i produce come yield uvw, dal teorema dello yield upperbound abbiamo che $|uvw| \leq 2^{(|V|+1)-1} = 2^{|V|} = \text{n}$.

Il pumping lemma per i linguaggi liberi, come nel caso dei linguaggi regolari, è direzionale e perciò un linguaggio che verifica tutte e tre le proprietà non è detto che sia libero.
Vediamo delle applicazioni del Pumping Lemma.
Prendiamo come esempio $L = \{0^m1^m2^m \,|\, m \geq 0\}$. Prendiamo qualsiasi decomposizione $0^n1^n2^n = uvwxy$ tale per cui

1. $|vwx| \leq n$;
2. $|vx| \neq \varepsilon$

Poiché $|vwx| \leq n$, vwx non può contenere contemporaneamente sia 0 che 2.
Se vwx non contiene 2, per il pumping lemma uwy deve essere in L: il problema è che uwy ha più 2 che 0 o 1!
Se vwx non contiene 0, per il pumping lemma uwy deve essere in L: il problemaè che uwy ha più 0 che 1 o 2!
Pendiamo anche come esempio $L = \{zz \,|\, z \in \{0,1\}^*\}$.
Sia n la costante del pumping lemma associata ad essa e prendiamo una

qualsiasi decomposizione $uvwxy$ di $0^n1^n0^n1^n$: notiamo come qualsiasi sia la posizione di vwx, la stringa uwy che per il pumping lemma dovrebbe essere in L, non può essere decomposta in modo tale che $uwy = zz$.

Rispetto ai linguaggi lineari, i linguaggi liberi son chiusi in un numero minore di operazioni.
Siano $G_1 = (V_1, T_1, S_1, P_1)$ e $G_2 = (V_2, T_2, S_2, P_2)$ due grammatiche libere per due linguaggi $L_1 = L(G_1)$ e $L_2 = L(G_2)$.
Siano

$$\begin{aligned} G_a &= (\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, \{S \to S_1S_2\} \cup P_1 \cup P_2) \\ G_b &= (\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, \{S \to S_1 \mid S_2\} \cup P_1 \cup P_2) \\ G_c &= (\{S\} \cup V_1, T_1, S, \{S \to \varepsilon \mid S_1S\} \cup P_1) \end{aligned}$$

Allora

$$\begin{aligned} L_1L_2 &= L(G_a) \\ L_1 \cup L_2 &= L(G_b) \\ L_1^* &= L(G_c) \end{aligned}$$

I linguaggi liberi in generale non sono chiusi rispetto all'intersezione. Basta infatti prendere per esempio questi due linguaggi liberi

$$\begin{aligned} L_1 &= \{0^n1^n2^m \mid n, m \geq 1\} \\ L_2 &= \{0^n1^m2^m \mid n, m \geq 1\} \end{aligned}$$

Notiamo che $L_1 \cap L_2 = \{0^n1^n2^n \mid n \geq 1\}$ non è un linguaggio libero.
Ciò implica che neanche il complemento di un linguaggio libero è sempre libero: se ciò fosse si potrebbe dimostrare la chiusura dell'intersezione tramite le leggi di De Morgan ($L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$).
In un linguaggio libero è **dedidibile** verificare se:

- esso è vuoto: basta controllare se la variabile iniziale S è un generatore;
- se una parola w appartiene al linguaggio: dato G in forma normale di Chomsky, l'albero sintattico di w ha esattamente $2^{|w|+1}$ nodi interni e il numero di alberi con questà quantità di nodi è finito.

Invece è **indecidibile** verificare se:

- una grammatica libera è ambigua;
- un linguaggio libero è intrinsecamente ambiguo;
- l'intersezione di due linguaggi liberi è vuoto;
- due linguaggi liberi sono uguali;

Capitolo 5

Automi a pila e parsing

5.1 Automi a pila e relazione con grammatiche libere

Un **automa a pila** consiste in:

- un automa a stati finiti non deterministico con ε-transizioni;
- una pila (o stack) di dimensione illimitata;

Esso cambia stato in base al:

- simbolo attuale nella stringa in input;
- simbolo in cima alla pila;

L'automa può fare push e pop dei simboli dentro e fuori la pila.
Per esempio consideriamo il linguaggio $L = \{ww^R \mid w \in (0+1)^*\}$, ossia il linguaggio delle parole palindrome.
Proviamo a creare, in modo non formale, un automa a pila (da qui in poi chiamato anche PDA da pushdown automata) che riconosca L:

- Lo stato q_0 significa "non ho raggiunto la metà dell'input ancora";
- Ogni simbolo letto da w è inserito nella pila.
- Ogni volta che legge un simbolo in input, l'automa "scommette" che è stata raggiunta la metà di w e cambia stato in q_1;
- In q_1 l'automa confronta l'input col simbolo in cima alla pila: se corrispondono rimuove l'elemento in cima alla pile, altrimenti la scommessa era sbagliata e l'automa fallisce;
- Se quando la stringa di input è finita la pila è vuota, allora l'automa ha riconosciuto ww^R

Formalmente un automa a pila è una tupla $P = (Q, \Sigma, \Gamma, \delta, q_o, Z_0, F)$ dove:

- Q è l'insieme finito degli stati;
- Σ è l'alfabeto dell'input;
- Γ è l'alfabeto della pila;
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to \wp(Q \times \Gamma^*)$ è la funzione transizione;
- q_0 è lo stato iniziale;
- Z_0 è il simbolo iniziale che appare nella pila;
- $F \subseteq Q$ è l'insieme degli stati finali;

Lo stato di un PDA è descritto interamente da:

- stato attuale q;
- stringa w ancora da leggere;
- contenuto γ nella pila;

La tripla (q, w, γ) è chiamata **descrizione istantanea** (o **ID**).
Usando le descrizioni istantanee possiamo descrivere le transizioni dell'automa tramite la relazione $\vdash$.
Se $(p, \alpha) \in \delta(q, a, X)$, allora

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

Ossia se l'automa è nello stato q, deve leggere a dalla stringa aw e nella pila abbiamo $X\beta$ con X in cima, allora l'automa può muoversi a $(p, w, \alpha\beta)$ dove a è stato letto, si cambia stato da q a p e X è stato sostituito da $alpha$.
Definiamo tramite induzione $\vdash^*$ che è la chiusura transitiva e riflessiva di $\vdash$ ed è definita come segue:

- Base: $I \vdash^* I$ per ogni ID I;
- Induzione: $I \vdash J$ se esiste K tale che $I \vdash K$ e $K \vdash^* J$

Chiameremo computazione perciò

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$$

Negli automi a pila il riconoscimento di un linguaggio può esser fatto sia tramite stato finale, i.e.

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha) \wedge q \in F\}$$

oppure il riconoscimento si può fare arrivando allo svuotamento della pila, i.e.

$$N(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

Le due formulazioni sono **equivalenti**, ovvero se un automa riconosce un linguaggio tramite il primo criterio, esiste un automa che tramite il secondo criterio riconosce lo stesso linguaggio.
Per costruire parser che riconoscono i linguaggi liberi, conviene usare il secondo criterio in modo tale da togliere gli stati di accettazione.

Sia $G = (V, T, S, P)$ una grammatica libera, creiamo un'automa a pila

$$M = \{\{q\}, T, V \cup T, \delta, q, S\}$$

che simula le derivazioni a sinistra in G.
Per ogni variabile A, sia

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta \in P\}$$

Per ogni terminale a, sia

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

Avremo che $N(M) = L(G)$.
Con questo tipo di automa creiamo implicitamente l'albero sintattico della grammatica.

I parser efficienti devono essere deterministici. Nella costruzione di PDA è possibile ritrovarci in fonti di non determinismo quali:

1. Per qualche $q \in Q$, qualche $a \in \Sigma \cup \{\varepsilon\}$ e qualche $X \in T$ abbiamo $|\delta(q, a, X)| > 1$.
   Ossia da una descrizione istantanea, l'automa può passare in due o più descrizioni diverse;

2. Per qualche $q \in Q$, qualche $a \in \Sigma$ e qualche $X \in \Gamma$ abbiamo che $\delta(q, a, X) \neq \emptyset$ e $\delta(q, \varepsilon, X) \neq \emptyset$.
   Ossia l'automa può scegliere se consumare o meno il simbolo di input a quando in cima alla pila abbiamo X;

Se in un automa a pila non si verificano queste due situazioni, diremo che esso è **deterministico**.

Riprendendo il linguaggio di esempio visto a inizio capitolo, notiamo come non sia possibile far capire all'automa quando si è raggiunta la metà della stringa: infatti non esiste nessun PDA deterministico che riconosce il linguaggio.
Però se aggiungiamo dello "zucchero sintattico" possiamo riconoscere un linguaggio simile che usa un carattere che denota la metà della stringa, i.e. il linguaggio sarà

$$L_2 = \{wcw^R \mid w \in \{0, 1\}^*\}$$

Questo linguaggio viene riconosciuto da un automa a pila deterministico.
Vale la seguente proposizione: se un linguaggio L è riconosciuto da un automa a pila, allora L ha una grammatica non ambigua.
Il viceversa non vale, basta considerare

$$S \to 0S0 \,|\, 1S1 \,|\, \varepsilon$$

che non è ambigua, ma genera $L = \{ww^R \,|\, w \in (0+1)^*\}$.

5.2 Parsing top-down

Il parsing top-down cerca di trovare una derivazione a sinistra della stringa in input. Prendiamo per esempio questa grammatica

$$S \to aAB \,|\, b$$
$$A \to c \,|\, d$$
$$B \to e$$

Prendendo in input ace, sappiamo che a è creata da $S \to aAB$, c è creata da A e e da B.
Ad ogni passo sappiamo la prossima variabile da espandere e il prossimo simbolo in input determina la produzione $X \to \alpha$ da applicare.
Dato un simbolo in input a e una variabile A da espandere, viene decisa quale delle produzioni tra le alternative

$$A \to \alpha_1 \,|\, \alpha_2 \,|\, \alpha_2 \,|\, \ldots \,|\, \alpha_n$$

è quella unica che deriva una stringa che inizia per a. In un parser predittivo è necessario calcolare l'insieme dei simboli terminali che sono in prima posizione in una qualsiasi stringa di simboli terminali che viene generata da una sequenza di variabili e terminali α.
Chiameremo $FIRST$ questa funzione

$$FIRST(\alpha) = \{a \,|\, \alpha \Rightarrow^* a\beta\} \cup \{\varepsilon \,|\, \alpha \Rightarrow^* \varepsilon\}$$

Il calcolo di $FIRST$ segue un procedimento induttivo e si divide nel caso α è composto da una sola variabile o multiple.
Per $FIRST(X)$ abbiamo:

- $a \in FIRST(a)$;
- Se $X \to \alpha$ è una produzione, allora $FIRST(\alpha) \subseteq FIRST(X)$;

Per $FIRST(Y_1 \ldots Y_k)$ abbiamo:

- Se $k = 0$ allora $\varepsilon \in FIRST(Y_1 \ldots Y_k)$;
- $FIRST(Y_1) \setminus \{\varepsilon\} \subseteq FIRST(Y_1 \ldots Y_k)$;
- Se $\varepsilon \in FIRST(Y_1)$ allora $FIRST(Y_2 \ldots Y_k) \subseteq FIRST(Y_1 \ldots Y_k)$;

Vediamo un esempio di utilizzo del FIRST:

$$\begin{array}{ll} S \to Ac \,|\, Ba & FIRST(S) = \{a, b, c\} \\ A \to \varepsilon \,|\, a & FIRST(A) = \{\varepsilon, a\} \\ B \to b & FIRST(B) = \{b\} \end{array}$$

Il $FIRST$ però molte volte non basta.
Prendiamo la grammatica appena usata per l'esempio e supponiamo di capire se la stringa c può essere generata. In partenza sappiamo che la produzione iniziale è $S \to Ac$. Ma qui sorge un problema: quale produzione dobbiamo applicare ad A per produrre c? Se andiamo a vedere $FIRST(A)$ non contiene c.
Alla fine $A \to \epsilon$ è la produzione giusta, poiché c può seguire A.
Per gestire questi casi in cui ci sono ε-produzioni definiamo la funzione $FOLLOW$

$$a \in FOLLOW(A) \Leftrightarrow S \Rightarrow^* \alpha A a \beta$$

che definisce l'insieme dei simboli terminali che seguono A in una forma sentenziale.
Per il $FOLLOW(S)$ usiamo il carattere sentinella \$ che indica che la stringa in input è finita.
$FOLLOW$ è definita seguendo un'algoritmo:

1. $\$ \in FOLLOW(S)$;
2. Se $A \to \alpha B \beta$ è una produzione, allora $FIRST(\beta) \backslash \{\varepsilon\} \subseteq FOLLOW(B)$;
3. Se $A \to \alpha B$ è una produzione o se $A \to \alpha B \beta$ è una produzione dove $\varepsilon \in FIRST(\beta)$, allora $FOLLOW(A) \subseteq FOLLOW(B)$;

Vediamo il calcolo del FOLLOW in un esempio

$$\begin{array}{ll} S \to Ac \,|\, Ba & \$ \in FOLLOW(S) \\ & c \in FOLLOW(A) \\ & a \in FOLLOW(B) \\ A \to \varepsilon \,|\, a & \\ B \to b & \\ C \to a \,|\, Cb & b \in FOLLOW(C) \\ D \to \varepsilon \,|\, d \,|\, Db & b \in FOLLOW(D) \end{array}$$

Adesso possiamo creare il parser predittivo.
Abbiamo bisogno di:

- Buffer per l'input che contiene la stringa da analizzare;
- Uno stack;
- Una tabella di parsing M che contiene le produzioni di G tale per cui $M : V \times (T \cup \{\$\}) \to \wp(P)$.
  Nella tabella nelle righe abbiamo le variabili e nelle colonne i terminali incluso \$.
  Ogni cella $M[X, a]$ o è vuota e indica che non ci sono produzioni per il simbolo a o contiene una produzione $X \to \alpha$ che ci dice che se il prossimo input è a e X è la variabile da espandere, allora bisogna seguire la produzione $X \to \alpha$;

Per riempire la tabella di parsing seguiamo questi due passi:

1. Se $A \to \alpha \in P$: per ogni $a \in FIRST(\alpha) - \{\varepsilon\}$ aggiungi $A \to \alpha$ a $M[A, a]$;

2. Se $A \to \alpha \in P$ e $\varepsilon \in FIRST(\alpha)$: per ogni $b \in FOLLOW(A)$ aggiungi $A \to \alpha$ a $M[A, b]$;

Se per ogni cella c'è al più una produzione, allora la grammatica è $LL(1)$: la prima L indica una lettura da sinistra a destra dalla stringa, la seconda L indica che si usa una derivazione a sinistra e l'uno indica che che il parser usa un solo input dalla stringa per decidere quale produzione applicare.
Il parser per riconoscere una stringa procede in questo modo:

1. La pila viene inizializzata a $S\$$ (S in cima);

2. Controlla il simbolo X in cima alla pila e il simbolo a in input corrente;

3. Se $X = a = \$$, allora **accetta** la stringa;

4. Se $X = a \neq \$$, allora X viene rimosso dalla pilae si avanza al prossimo simbolo in input;

5. se X è una variabile e $M[X, a]$ contiene $X \to Y_1 Y_2 \dots Y_k$, allora rimpiazza X in cima alla pila con $Y_1 Y_2 \dots Y_k$ (con Y_1 in cima) e vai al punto 2;

6. Termina con un **errore**;

5.3 Grammatiche $LL(1)$

Una fonte di problemi per i parser predittivi sono le produzioni che hanno un prefisso in comune.
Prendiamo in considerazione:

$$A \to \alpha\beta_1 \,|\, \alpha\beta_2$$

Quando bisogna espandere A, il parser non sa quale delle due produzioni applicare, introducendo così non determinismo e rendendo la grammatica non di tipo $LL(1)$.
La soluzione in questo caso è di differire la scelta fino a che uno tra β_1 o β_2 emerge, introducendo una nuova variabile A':

$$\begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 \,|\, \beta_2 \end{aligned}$$

Una grammatica è detta **immediatamente ricorsiva a sinistra** se presenta produzioni del tipo

$$A \to A\alpha \,|\, \beta$$

La presenza di queste produzioni è problematica. Basta osservare la seguente derivazione:

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow \cdots \Rightarrow A\alpha\cdots\alpha\alpha \Rightarrow \beta\alpha\cdots\alpha\alpha$$

Un parser vedrebbe β all'inizio ma non saprebbe quante volte la produzione $A \to A\alpha$ è stata applicata.
Per risolvere il problema ricompongo le produzioni in modo da generare β per prima introducendo anche qui una nuova variabile:

$$\begin{aligned} A &\to \beta A' \\ A' &\to A'a \,|\, \varepsilon \end{aligned}$$

In questo modo sappiamo che β viene prodotta per prima.
In generale le produzioni della forma

$$A \to A\alpha_1 \,|\, A\alpha_2 \,|\, \cdots \,|\, A\alpha_m \,|\, \beta_1 \,|\, \beta_2 \,|\, \cdots \,|\, \beta_n$$

sono sostituite da

$$\begin{aligned} A &\to \beta_1 A' \,|\, \beta_2 A' \,|\, \cdots \,|\, \beta_n A' \\ A' &\to \alpha_1 A' \,|\, \alpha_2 A' \,|\, \cdots \,|\, \alpha_m A' \,|\, \varepsilon \end{aligned}$$

Anche le ricorsioni indirette causano problemi.
Prendiamo per esempio:

$$\begin{aligned} S &\to Aa \,|\, b \\ A &\to Ac \,|\, Sd \,|\, \varepsilon \end{aligned}$$

La derivazione $S \Rightarrow Aa \Rightarrow Sda \Rightarrow bda$ è possibile e anche qui quando il parser si trova b, non sa quante volte è stata applicata la produzione $S \to Aa$.
La soluzione consiste nell'espandere le produzione ed eliminare le ricorsioni a sinistra:

$$\begin{array}{lcl} S \to Aa \,|\, b & \Rightarrow & S \to Aa \,|\, b \\ A \to Ac \,|\, Aad \,|\, bd \,|\, \varepsilon & \Rightarrow & A \to bdA' \,|\, A' \\ & & A' \to cA' \,|\, adA' \,|\, \varepsilon \end{array}$$

Proviamo adesso a creare un parser per la seguente grammatica:

$$\begin{array}{lcl} E & \to & E\overbrace{+T}^{\alpha} \,|\, \overbrace{T}^{\beta} \\ T & \to & T \times F \,|\, F \\ F & \to & a \,|\, (E) \end{array}$$

Eliminiamo la ricorsioni:

$$\begin{array}{lcl} E & \to & TE' \\ E' & \to & +TE' \,|\, \varepsilon \\ T & \to & FT' \\ T' & \to & \times FT' \,|\, \varepsilon \\ F & \to & a \,|\, (E) \end{array}$$

Calcoliamo il $FIRST$:

X			FIRST(X)
E	$\to$	TE'	$\{a, (\}$
E'	$\to$	$+TE' \,\|\, \varepsilon$	$\{+, \varepsilon\}$
T	$\to$	FT'	$\{a, (\}$
T'	$\to$	$\times FT' \,\|\, \varepsilon$	$\{\times, \varepsilon\}$
F	$\to$	$a \,\|\, (E)$	$\{a, (\}$

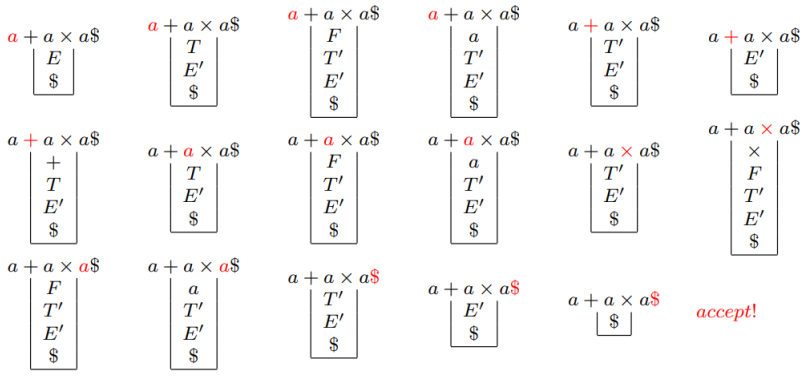
E successivamente il $FOLLOW$:

X			FIRST(X)	
E	$\to$	TE'	$\{a, (\}$	$+ \in \text{FOLLOW}(T)$
				$\text{FOLLOW}(E) \subseteq \text{FOLLOW}(T)$
				$\text{FOLLOW}(E) \subseteq \text{FOLLOW}(E')$
E'	$\to$	$+TE' \,\|\, \varepsilon$	$\{+, \varepsilon\}$	$\text{FOLLOW}(E') \subseteq \text{FOLLOW}(T)$
T	$\to$	FT'	$\{a, (\}$	$\times \in \text{FOLLOW}(F)$
				$\text{FOLLOW}(T) \subseteq \text{FOLLOW}(F)$
				$\text{FOLLOW}(T) \subseteq \text{FOLLOW}(T')$
T'	$\to$	$\times FT' \,\|\, \varepsilon$	$\{\times, \varepsilon\}$	$\text{FOLLOW}(T') \subseteq \text{FOLLOW}(F)$
F	$\to$	$a \,\|\, (E)$	$\{a, (\}$	$) \in \text{FOLLOW}(E)$

X			FIRST(X)	FOLLOW(X)
E	$\to$	TE'	$\{a, (\}$	$\{\$,)\}$
E'	$\to$	$+TE' \mid \varepsilon$	$\{+, \varepsilon\}$	$\{\$,)\}$
T	$\to$	FT'	$\{a, (\}$	$\{+, \$,)\}$
T'	$\to$	$\times FT' \mid \varepsilon$	$\{\times, \varepsilon\}$	$\{+, \$,)\}$
F	$\to$	$a \mid (E)$	$\{a, (\}$	$\{\times, +, \$,)\}$

	$+$	$\times$	a	$($	$)$	$\$$
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \varepsilon$	$E' \to \varepsilon$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to \varepsilon$	$T' \to \times FT'$			$T' \to \varepsilon$	$T' \to \varepsilon$
F			$F \to a$	$F \to (E)$		

Da queste possiamo creare la tabella di parsing:
Poichè in ogni cella c'è al massimo una produzione, la grammaticha è $LL(1)$.
Adesso vediamo lo stack quando la stringa da riconoscere è $a + a \times a$:



5.4 Parsing bottom-up

Il parsing bottom-up consiste nella creazione dell'albero di parse partendo dalle foglie (bottom) e aivando alla radice (top).
Prendiamo per esempio la seguente grammatica:

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

Vediamo come viene fatto il parsing per la stringa *abbcd*: viene letto a e messo nello stack, successivamente viene letto b e messo anche lui nello stack.
Il parser vede che b è stato prodotto da $A \to b$ e sostituisce b con A nello stack.
Successivamente vengono letti b e c nello stack. Il parser vede che nello stack è presente Abc (dal fondo alla cima) e sa che è prodotto da $A \to Abc$ e li sostituisce con A.
Legge poi d e analogamente a prima lo sostituisce con B nella pila. Infine legge e e nella pila abbiamo $aABe$ che è prodotto da $S \to aABe$ e sostituisce tutto con S.
Poiché non rimane niente da leggere e nella pila è rimasto solo S, la stringa viene accettata.
Se ripercorriamo i passi all'indietro notiamo che abbiamo trovato la derivazione a destra della parola in input:

$$S \Rightarrow_r aABe \Rightarrow_r aAde \Rightarrow_r aAbcde \Rightarrow_r abbcde$$

Infatti il parsing bottom-up si basa sul trovare la derivazione a destra per la stringa in input da sinistra a destra.

Quando si forme un'espansione nello stack, diremo che si è formato un **handle** che deve essere subito ridotto.
Formalmente un **handle** di una forma sentenziale a destra γ è una produzione $A \to \beta$ e una posizione di γ dove la stringa β può esser trovata e rimpiazzata da A per produrre la forma sentenziale a destra precedente in una derivazione a destra di γ.
Se un handle compare nella pila, esso è sempre in cima.

Il parser durante il riconoscimento di una parola compie le seguenti azioni nella pila:

1. **shift**: sposta il prossimo simbolo in input nella pila;
2. **reduce**: riduce l'handle in cima alla pila;
3. **accept**: parsing avvenuto con successo;
4. **error**: errore nel parsing;

I parser che compiono queste azioni vengono chiamati shift-reduce.

Un parser LR (L per indicare la lettura da sinistra a destra, R che usa derivazioni a destra) è composto da:

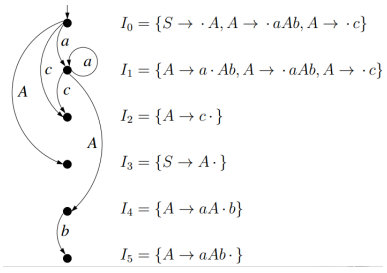
1. Una **stringa in input** con carattere di terminazione $a_1 a_2 \dots \$$;
2. Una **pila** $s_0 X_1 s_1 X_2 \dots X_m s_m$ con s_m in cima dove:
   - ogni X_i è un simbolo della grammatica;
   - ogni s_i è un simbolo, chiamato **stato** che riassume le informazioni contenute nella pila sotto di esso;
3. Una **tabella di parsing** dove nelle righe abbiamo gli stati e nelle colonne i terminali e le variabili;
4. Un **programma driver o guida** che legge s_m (il simbolo in cima alla pila e a_i (simbolo attuale in input) e consulta la cella $[s_m, a_i]$ della tabella di parsing che può essere

$$\text{shift } s, \text{ reduce } A \to \beta, \text{ accept, error}$$

L'algoritmo di parsing, dalla configurazione $(s_0 X_1 s_1 X_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$, segue i seguenti passi:

1. Se $[s_m, a_i] = \text{shift } s$, allora muoviti a
$$(s_0 X_1 s_1 X_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$$
2. Se $[s_m, a_i] = \text{reduce } A \to \beta$, allora muoviti a
$$(s_0 X_1 s_1 X_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$$
3. Se $[s_m, a_i] = \text{accept}$, il parsing è stato completato.
4. Se $[s_m, a_i] = \text{error}$, il parser ha scoperto un errore sintattico;

Gli stati nello stack possono essere gestiti tramite un automa a stati finiti. Per esempio, sia $G : S \to A$ $A \to aAb \,|\, c$



Lo stato iniziale codifica l'inizio con quello che espande S.
Il puntino è un placeholder che indica in che punto sono nella generazione e a destra di esso troviamo ciò che deve ancora finire nello stack.
Le produzioni di G con un puntino in una qualsiasi posizione nella parte di destra vengono chiamati **item** $LR(0)$.
Per esempio la produzione $A \to XYZ$ fornisce quattro item $LR(0)$:

- $A \to \cdot XYZ$;
- $A \to X \cdot YZ$;
- $A \to XY \cdot Z$;
- $A \to XYZ\cdot$;

$A \to \varepsilon$ fornisce solo l'item $A \to \cdot$.
Se I è un insieme di item per una grammatica G, allora $chiusura(I)$ è l'insieme degli item costruiti da I induttivamente come segue:

- Base: ogni item I è in $chiusura(I)$;
- Induzione: se $A \to \alpha \cdot B\beta \in chiusura(I)$ e $B \to \gamma$ è una produzione di G, $B \to \cdot\gamma$ è in $chiusura(I)$;

L'intuizione sta nel fatto che $A \to \alpha \cdot B\beta$ indica che abbiamo già visto α e prevediamo di vedere $B\beta$ successivamente: se $B \to \gamma$ allora prevediamo con sicurezza di vedere $\gamma\beta$ nel nostro input.
Vediamo un esempio e consideriamo la seguente grammatica:

$$\begin{array}{lcl} E & \to & E \\ E' & \to & E + T \,|\, T \\ T & \to & T \times F \,|\, F \\ F & \to & (E) \,|\, a \end{array}$$

Calcoliamo la chiusura di un item della produzione iniziale:

$$\begin{aligned} chiusura(\{E' \to \cdot E\}) = \{&E' \to \cdot E, E \to \cdot E + T, E \to \cdot T, \\ &T \to \cdot T \times F, T \to \cdot F, F \to \cdot(E), \\ &F \to \cdot a\} \end{aligned}$$

Sia I un insieme di item e X un simbolo della grammatica, allora

$$goto(I, X) = chiusura(\{A \to \alpha X \cdot \beta \,|\, A \to \alpha \cdot X\beta \in I\})$$

L'operazione di *goto* è semplicemente lo shift a destra del puntino. Dopo lo shift è necesserio calcolare la chiusura dell'item creato.

Calcoliamo il goto di un insiseme di item derivante dalla grammatica usata in precedenza:

$$goto(\{E' \to E\cdot, E \to E \cdot +T\}, +) = \{E \to E + \cdot T, T \to \cdot T \times F, \\ T \to \cdot F, F \to \cdot(E), F \to \cdot a\}$$

Data una grammatica $G = (V, T, S, P)$, aumentiamo G a G' in modo tale che $G' = (V \cup \{S'\}, T, S', P \cup \{S' \to S\})$.
La produzione $S' \to S$ garantisce che il parsing abbia esattamente uno stato iniziale e che verrà usato per identificare univocamente la condizione di terminazione del parsing.
Infine ci creiamo l'insieme degli stati che è una collezione $\mathcal{I} = \{I_0, I_1, \cdots, I_n\}$ di insiemi di item costruita ricorsivamente:

- Base: $chiusura(\{S' \to \cdot S\})$ è I_0;
- Induzione: Se $I \in \mathcal{I}$ e $X \in V \cup T$ e $goto(I, X) \neq \emptyset$, allora $goto(I, X) \in \mathcal{I}$;

L'idea è quella di creare gli stati dell'automa partendo dalla chiusura di $S' \to \cdot S$ per trovare incrementalmente tutti gli stati raggiungibili da esso.

5.5 Grammatiche SLR, LR(1), LALR(1)

In questa sezione vedremo tre varianti che implementano il parsing pushdown.
La prima è il parser SLR (simple left right). La tabella di parsing viene costruita tramite gli stati derivanti dagli item (Lo stato i corrisponde a I_i):

1. Se $A \to \alpha \cdot a\beta \in I_i$ e $goto(I_i, a) = I_j$, allora imposta $[i, a]$ a shift j;
2. Se $A \to \alpha\cdot \in I_i$, allora imposta $[i, a]$ a rid $A \to \alpha \, \forall a \in FOLLOW(A)$;
3. Se $S' \to S\cdot \in I_i$, allora imposta $[i, \$]$ ad accept;

Ogni cella rimasta vuota è impostata a error.
Se una parsing table SLR per una grammatica G non ha celle con azioni multiple, allora G è una grammatica SLR.
Nella shift se:

- Siamo nello stato i;
- $A \to \alpha \cdot a\beta \in I_i$;
- Il simbolo lookahead è a;
- $a \in FOLLOW(A)$;

Allora

- La pila non contiene ancora l'handle.
- Leggi a e inseriscilo nella pila;
- Muoviti allo stato $goto(I_j, a)$;

Nella reduce se:

- Siamo nello stato i;
- $A \to \alpha\cdot \in I_i$;
- Il simbolo lookahead è a;
- $a \in FOLLOW(A)$;

Allora

- Riduci $A \to \alpha$, rimpiazzando α con A nella pila;
- Muoviti nello stato $goto(I_j, A)$ dove j è il simbolo di stato che sta sotto A;

Nella accept se:

- Siamo nello stato i;
- $S' \to S\cdot \in I_i$;
- \$ è il simbolo lookahed;

Allora il parsing è andato a buon fine.
Ogni grammatica SLR è non ambigua, ma il viceversa non vale, basta considerare

$$\begin{array}{lcl} S & \to & L = R \,|\, R \\ L & \to & *R \,|\, a \\ R & \to & L \end{array}$$

L'itemset I_2 contiene $\{S \to L\cdot = R, R \to L\cdot\}$ e $= \in FOLLOW(R)$, creando un conflitto shift/reduce. Me non esiste nessuna forma sentenziale a destra che inizia con $R = \cdots !$
Le regole usate fino ad adesso mettono i reduce in più celle possibili, creando però problemi come nel caso appena menzionato.
L'idea per risolvere questo tipo di problemi è quella di avere un estensione degli item $[A \to \alpha\cdot, \text{informazione}]$ dove l'informazione riguarda un sottoinsieme del follow di A.
Questi item saranno chiamati item $LR(1)$ e son composti come segue

$$A \to \alpha \cdot \beta,\, a$$

dove $A \to \alpha \cdot \beta$ è una produzione e a è un simbolo terminale o \$.

- $FOLLOW(A)$ contiene tutti i possibili terminali che possono seguire ogni possibile occorrenza di A;
- In $A \to \alpha \cdot \beta, a$, quella occorrenza di A puo essere seguita solo da a;
- $a \in FOLLOW(A)$;
- L'item $A \to \alpha\cdot, a$ permette di ridurre $A \to \alpha$ solo se il prossimo simbolo in input è a;

Cambia anche la funzione *chiusura*. Se I è un insieme di item della grammatica G, allora $chiusura(I)$ è l'insieme degli item costruiti da I come segue:

- Base: ogni item in I è in $chiusura(I)$;
- Induzione: se $A \to \alpha \cdot B\beta$, $a \in chiusura(I)$ e $B \to \gamma$ è una produzione di G, per ogni $b \in FIRST(\beta a)$ aggiungi $B \to \cdot\gamma$, b a $chiusura(I)$;

L'intuizione sta nel fatto che sappiamo che B è seguito da β, ma può essere che $\epsilon \in FIRST(\beta)$ e dunque mostrando cosa può seguire la variabile A. Inseriamo perciò pure il simbolo a e perciò i simboli che possono seguire B in quella determinata produzione sono quelli in $FIRST(\beta a)$.
Il primo stato del parser è determinato da $chiusura(\{S' \to \cdot S, \$\})$.
Il *goto* non cambia di molto rispetto all versione SLR:

$$goto(I, X) = chiusura(\{A \to \alpha X \cdot \beta,\, a \mid A \to \alpha X \cdot \beta,\, a \in I\})$$

Calcoliamo gli stati del parser della seguente grammatica come esempio:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & CC \\ C & \to & cC \mid d \end{array}$$

Partiamo da $S' \to \cdot S$, \$ e calcoliamo l'itemset iniziale I_0: seguendo la definizione sappiamo che $\beta \to \gamma$ è $S \to CC$.
Confrontando la produzione generica $A \to \alpha \cdot B\beta$ abbiamo che S è A, ε è α, B è S e ε è β.
Sappiamo che $FIRST(\varepsilon\$) = \$$ dunque aggiungiamo $S \to \cdot CC$, \$ nell'itemset ottenendo:

$$\begin{aligned} I_0 = \{&(S' \to \cdot S,\, \$), \\ &(S \to \cdot CC,\, \$)\} \end{aligned}$$

Continuiamo applicando la chiusura sul nuovo item ottenuto. Sapendo che $FIRST(C\$) = c/d$ finiamo di costruire l'itemset:

$$\begin{aligned} I_0 = \{&(S' \to \cdot S,\, \$), \\ &(S \to \cdot CC,\, \$) \\ &(C \to \cdot cC,\, c/d \\ &(C \to \cdot d, c/d\} \end{aligned}$$

Adesso applicando il *goto* per ogni item trovato, ossia spostando il puntino a destra per ogni item, otteniamo gli stati del parser, sapendo che per ogni item creiamo un corrispettivo stato:

$$\begin{aligned} I_1 &= \{(S \to S\cdot, \$)\} \\ I_2 &= \{(S \to C \cdot C,\, \$), \\ &\quad (C \to \cdot cC,\, \$), \\ &\quad (C \to \cdot d,\, \$)\} \\ I_3 &= \{(C \to c \cdot C, c/d), \\ &\quad (C \to \cdot cC, c/d), \\ &\quad (C \to \cdot d, c/d)\} \end{aligned}$$

D'ora in poi la costruzione della parsing table è uguale a quella SLR.
Se in ogni cella, abbiamo al massimo un'azione allora la grammatica è $LR(1)$.

Molte volte ci può capitare di avere stati uguali che differiscono solo per i simboli di lookahead. L'idea è quella di fondere questi stati e ridurne il numero: otteniamo i parser $LALR(1)$.
Un parser $LALR(1)$ ha lo stesso numero di stati del corrispettivo parser SLR.
Il problema di questo approccio è che in alcuni casi si introducono conflitti *reduce/reduce*.
Nessuna grammatica ambigua è $LR(1)$, ma in generale il viceversa non vale e basta considerare:

$$E \to E + E \,|\, E * E \,|\, (E) \,|\, a$$

Capitolo 6

Semantica denotazionale

6.1 Sem. denotazionale di programmi sequenziali aciclici

Lo scopo delle **semantiche formali** è la descrizione **rigorosa** del **significato** dei programmi.
La **semantica denotazionale** descrive *cosa fa* un programma. In tutto il capitolo per far vedere come funziona la semantica denotazionale useremo la seguente grammatica di un semplice linguaggio di programmazione imperativo:

$$
\begin{array}{lcl}
S & \to & \texttt{skip} \\
 & | & x := A \\
 & | & S; S \\
 & | & \texttt{if}(B)\, S\, \texttt{else}\, S \\
 & | & \texttt{while}(B)\, S \\
A & \to & n \\
 & | & x \\
 & | & A + A \mid A * A \mid \cdots \\
B & \to & true \\
 & | & false \\
 & | & A = A \mid \cdots \\
 & | & B\texttt{\&\&}B \mid \neg B
\end{array}
$$

Abbiamo che S produce gli statement, A produce le espressioni aritmetiche e B produce le espressioni booleane.
n fa da alias per i valori numerali interi e x rappresenta le variabili, intere anch'esse.

Il significato di un'espressione è dipendente dai valori nelle variabili presenti in esso. Per esempio $x + 1$ vale 4 se x è 3 o vale 2 se x è 1.
Introduciamo il concetto di **stato**: lo stato è una funzione che mappa ogni

variabile ad un valore:

$$\mathbf{State} = \mathbf{Var} \to \mathbb{Z}$$

Uno stato s può essere rappresentato come lista in cui vengono associati ad ogni variabile i loro rispettivi valori, per esempio

$$[x \mapsto 3, y \mapsto 9, z \mapsto 5]$$

Quando scriviamo $s[z \mapsto 2]$ indichiamo di prendere lo stato s così com'è tranne per z che assume valore 2.
Il valore delle espressioni aritmetiche/booleane è denotato dai **domini semantici**:

$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$
$$\mathbb{B} = \{T, F\}$$

Dato uno stato s e un espressione aritmetica a siamo in grado di determinare il valore dell'espressione.
Ci rimane da definire il significato delle espressioni aritmetiche tramite **funzioni semantiche**:

$$\mathcal{A}[[\cdot]]\cdot = \mathbf{Aexp} \to (\mathbf{State} \to \mathbb{Z})$$

La funzione prende come input l'oggetto sintattico rappresentante l'espressione e lo stato e restituisce il valore dell'espressione.
$\mathcal{A}[[\cdot]]\cdot$ prende gli argomenti uno alla volta e perciò passiamo prima l'espressione e studiare $\mathcal{A}[[expr]]\cdot$ per poi fornire lo stato s e determinare il valore dell'espressione $expr$ in base ai valori forniti dallo stato.
Il ragionamento è analogo per le espressioni booleane:

$$\mathcal{B}[[\cdot]]\cdot = \mathbf{Bexp} \to (\mathbf{State} \to \mathbb{B})$$

Adesso vediamo la semantica delle espressioni

$$\mathcal{A}[[n]]s = n \text{ (Il valore delle costanti è uguale per ogni stato.)}$$
$$\mathcal{A}[[x]]s = s\,x \text{ (Il valore di } x \text{ in base allo stato } s.)$$
$$\mathcal{A}[[A_1 + A_2]]s = \mathcal{A}[[A_1]]a + \mathcal{A}[[A_2]]s$$

$$\mathcal{B}[[true]]s = T$$
$$\mathcal{B}[[false]]s = F$$
$$\mathcal{B}[[A_1 = A_2]]s = \begin{cases} T & \text{se } \mathcal{A}[[A_1]]s = \mathcal{A}[[A_2]]s \\ F & \text{se } \mathcal{A}[[A_1]]s \neq \mathcal{A}[[A_2]]s \end{cases}$$
$$\mathcal{B}[[B_1 \texttt{\&\&} B_2]]s = \begin{cases} T & \text{se } \mathcal{B}[[B_1]]s = T \text{ e } \mathcal{B}[[B_2]]s = T \\ F & \text{se } \mathcal{B}[[B_1]]s = F \text{ o } \mathcal{B}[[B_2]]s = F \end{cases}$$

Notiamo come ci sia una notevole differenza tra i simboli a sinistra e a destra dell'equazione: nel primo caso infatti non hanno un significato intrinseco e sono solo degli oggetti sintattici.
Il loro significato è dato dalle definizioni delle equazioni.

Ci manca la defizione delle funzioni semantiche per gli statement: l'idea è di avere una funzione che dato uno statement restituisce una funzione parziale ceh dato uno stato ce ne restituisce un altro:

$$\mathcal{S}[[\cdot]]\cdot = \mathbf{Stmt} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

Definiamo $\mathcal{S}[[S]]$ trmite induzione sulla struttura di S:

$$\begin{aligned}
\mathcal{S}[[x := A]]s &= s[x \mapsto \mathcal{A}[[A]]s] \\
\mathcal{S}[[\texttt{skip}]]s &= id \\
\mathcal{S}[[S_1; S_2]]s &= \mathcal{S}[[S_2]] \circ \mathcal{S}[[S_1]] \\
\mathcal{S}[[\texttt{if}(B)\ S_1\ \texttt{else}\ S_2]]s &= cond(\mathcal{B}[[B]], \mathcal{S}[[S_1]], \mathcal{S}[[S_2]])s
\end{aligned}$$

dove

$$\mathcal{S}[[S_2]] \circ \mathcal{S}[[S_1]] = \begin{cases} s'' & \text{se esiste } s' \text{ tale che } [[S_1]]s = s' \text{ e } \mathcal{S}[[S_2]]s' = s'' \\ \textbf{undef} & altrimenti \end{cases}$$

e

$$cond(p, f, g)s = \begin{cases} f\, s & \text{se } p\, s = T \\ g\, s & \text{se } p\, s = F \end{cases}$$

6.2 Sem. denotazionale dell'operatore WHILE

Per definire gli effetti dell'operatore `while` cerchiamo di "srotolarlo" e ottenere

$$\texttt{while}(B)\, S = \texttt{if}(B)\, S; \texttt{while}(B)\, S\, \texttt{else skip}$$

Per come abbiamo definito $\mathcal{S}[[\cdot]]$ otteniamo perciò

$$\mathcal{S}[[\texttt{while}(B)\, s]] = cond(\mathcal{B}[[B]], \mathcal{S}[[\texttt{while}(B)\, S]] \circ \mathcal{S}[[S]], id)$$

che però **non** è una definizione.
Se poniamo $g = \mathcal{S}[[\texttt{while}(B)\, S]]$ e denotiamo con F il funzionale

$$cond(\mathcal{B}[[B]], g \circ \mathcal{S}[[S]], id)$$

osserviamo che g diventa il **punto fisso** di F in quanto l'equazione diventa

$$g = cond(\mathcal{B}[[B]], g \circ \mathcal{S}[[S]], id) = F\, g$$

Chiamiamo punti fissi di una funzione f i valori x per cui $x = f(x)$; un esempio è 0 per la funzione $\sin x$.
Per le funzioni semantiche che definiamo, è necessario che il punto fisso esista: ci viene in aiuto il seguente teorema

Teorema. *Sia $f : D \to D$ una funzione continua nel ccpo $(D, \sqsubseteq)$, allora*

$$FIX f = \sqcup\{f^n \perp \mid n \geq 0\}$$

dove

- $\sqsubseteq$ è la relazione d'ordine;
- $\perp$ (bottom) è l'elemento più piccolo in D in base alla relazione d'ordine;
- $\sqcup$ indica il più piccolo dei maggioranti (l'estremo superiore);

Nel nostro caso definiamo per poter usare il teorema, definiamo $\sqsubseteq$ su funzioni $State \hookrightarrow State$ e avremo che

$$g_1 \sqsubseteq g_2$$

se $g_1\, s = s'$ e $g_2\, s = s'$ per ogni scelta di s e s'.
Ciò significa che g_2 si comporta esattamente come g_1 dove sono definite insieme: g_2 è un'estensione di g_1.
In base alla relazione che abbiamo definito, $\perp$ è la funzione indefinita per qualsiasi s.
Andando a vedere $f^n \perp$ vedremo che è

$$f^n \perp = \perp, f(\perp), f(f(\perp)), \ldots$$

Il teorema ci dice che per un certo valore di n, $f^n \perp$ smette di divergere e il più piccolo di essi (ossia la prima occorrenza dei valori uguali) è il nostro punto fisso.
Dato l'operatore `while`, la funzione semantica è definita come

$$\mathcal{S}[[\texttt{while}(B)\, S]] = FIX\ F$$

dove

$$F\ g = cond(\mathcal{B}[[B]], g \circ S[[S]], id)$$

Perciò basta trovare il punto fisso di $cond(\mathcal{B}[[B]], g \circ S[[S]], id)$.
Prendiamo come esempio:

$$\mathcal{S}[[\texttt{while}\ \neg(x = 0)\ \texttt{skip}]]$$

Dbbiamo trovare il punto fisso di $F\ g = cond(\mathcal{B}[[\neg(x = 0)]], g \circ id, id)$ dove

$$(F\ g)\ s = \begin{cases} g\ s & se\ s\ x \neq 0 \\ s & se\ s\ x = 0 \end{cases}$$

Partiamo dalla funzione $\perp$, abbiamo che $(F\ \perp)\ s$ è indefinita se $s\ x \neq 0$ in quanto $\perp\ s$ è indefinita per ogni s.
Se $s\ x = 0$, invece $(F\ g)$ vale s in quanto in questo caso la funzione non tiene conto del valore di g, i.e.

$$(F\ \perp)\ s = \begin{cases} \texttt{undef} & se\ s\ x \neq 0 \\ s & se\ s\ x = 0 \end{cases}$$

Con ragionamento analogo, facciamo le stesse osservazioni pure per $(F\ F\ perp)\ s$ che è uguale a $(F\ \perp)\ s$, i.e.

$$(F\ F\ \perp)\ s = \begin{cases} \texttt{undef} & se\ s\ x \neq 0 \\ s & se\ s\ x = 0 \end{cases} \quad = (F\ \perp)\ s$$

Questo vuol dire che $(F\ \perp)$ è il punto fisso per F.
Infatti se andiamo a vedere cosa succede nel while, se $x = 0$ esso non viene eseguito, mentre se $x \neq 0$ esso entra in un loop da cui non esce, dando il comportamento non definito.

6.3 Sem. denotazionale con blocchi e procedure

In questa situazione vedremo come la semantica denotazionale gestisce i blocchi e le procedure.
Prima di far ciò estendiamo la nostra grammatica usata fin'ora: Vediamo

$$\begin{array}{rcl} S & ::= & \ldots \mid \texttt{begin}\ D_V\ D_P\ S\ \texttt{end} \mid \texttt{call}\ p \mid \ldots \\ D_V & ::= & \texttt{var}\ x := A;\ D_V \mid \varepsilon \\ D_P & ::= & \texttt{proc}\ p\ \texttt{is}\ S;\ D_P \mid \varepsilon \end{array}$$

un brano di codice prodotto dalla grammatica estesa

```
begin var x:= 7; proc p is x:=0;
  begin var x:=5; call p end
end
```

Consideremo regole di scoping *statico*.
Se vediamo il brano di codice, la procedura si riferirà alla `x` più esterna, in quanto per determinare la referenza corretta si guarda alla struttura sintattica del programma, ovvero la variabile in interesse è quella dichiarata nello stesso blocco dove è dichiarata la procedure.
Nel caso lo scoping fosse *dinamico*, verrebbe modificata la `x` nel blocco interno: questo perché in questo caso viene usato il flusso di esecuzione per determinare l'istanza da usare.

Per implementare lo scoping statico, lo stato non ci basta e dobbiamo introdurre le *locazioni* (o location): ad ogni variabile associamo una locazione unica e ad ogni locazione associamo un valore.
Quando dichiariamo una nuova variabile, associamo una nuova locazione ad essa, mentre il valore nella locazione è quello presente nell'assegnamento.
Per far ciò dividiremo lo stato e introduciamo gli ambienti delle variabili che associano per ogni variabile una località e gli store che associano le località ai valori in memoria:

$$\begin{aligned} \mathbf{Env}_V &= \mathbf{Var} \to \mathbf{Loc} \\ \mathbf{Store} &= \mathbf{Loc} \to \mathbb{Z} \end{aligned}$$

Dati $e_V : \mathbf{Var} \to Loc$ e $sto : \mathbf{Loc} \to \mathbb{Z}$, recuperiamo lo stato come $sto \circ e_v$.
Ovviamente le funzioni semantiche cambiano poiché dobbiamo tenere traccia del fatto che solo lo store viene aggiornato durante l'esecuzione degli statement:

$$\begin{aligned} \mathcal{S} &: \mathbf{Stmt} \to \mathbf{Env}_V \to (\mathbf{Store} \hookrightarrow \mathbf{Store}) \\ \mathcal{A} &: \mathbf{Aexp} \to \mathbf{Env}_V \to (\mathbf{Store} \to \mathbb{Z}) \\ \mathcal{B} &: \mathbf{Bexp} \to \mathbf{Env}_V \to (\mathbf{Store} \to \mathbb{B}) \end{aligned}$$

Vediamo come cambia l'assegnazione delle variabili

$$\mathcal{S}[[x := A]]\, e_V\, sto = sto[l \mapsto \mathcal{A}[[A]](sto \circ e_v)]$$

dove $l = e_v\, x$. La lettura che diamo è che la semantica dell'assegnamento è prendere la locazione di x nell'ambiente delle variabili e aggiornare il valore ad esso associato nella store con il significato dell'espressione A in base a sto ed ev.
La semantica degli altri statemente non differisce di molto da prima se non per l'aggiunta della specifica dell'ambiente delle variabili:

$$\begin{aligned}
\mathcal{S}[[\texttt{skip}]]\, e_V &= id \\
\mathcal{S}[[S_1; S_2]]\, e_V &= (\mathcal{S}[[S_2]]\, e_V) \circ (\mathcal{S}[[S_1]]\, e_V) \\
\mathcal{S}[[\texttt{if}(B)\ S_1\ \texttt{else}\ S_2]]\, e_V &= cond(\mathcal{B}[[B]]\, e_V, \mathcal{S}[[S_1]]\, e_V, \mathcal{S}[[S_2]]\, e_V) \\
\mathcal{S}[[\texttt{while}(B)\, S]]\, e_V &= FIX\, F \\
&\quad \text{dove } F\, g = cond(\mathcal{B}[[B]]\, e_V, g \circ (\mathcal{S}[[S]]\, e_V), id)
\end{aligned}$$

L'ambiente delle variabili deve essere aggiornata ogni volta che entriamo in un blocco che contiene dichiarazioni di variabili locali.
Per fare ciò usiamo una funzione, chiamata $update_V[[D_V]]$ che prende in ingresso l'ambiente delle variabili e lo store attuale e restituisce l'ambiente aggiornato e lo store in virtù della dichiarazione delle variabili D_V.
Formalmente:

$$\begin{aligned}
&update_V[[\varepsilon]] = id \\
&update_V[[\texttt{var}\ \ x := A; D_V]](e_V, sto) = update_V[[D_V]](e_V[x \mapsto l], sto[l \mapsto v])
\end{aligned}$$

dove $l = newloc()$ è la località di memoria allocata per ospitare la variabile e $v = \mathcal{A}[[A]](sto \circ e_V)$ è il significato della espressione da assegnare alla variabile (il valore da memorizzare nella store).
Per accomodare la dichiarazione di blocchi è necessaria una nuova regola semantica:

$$\mathcal{S}[[\texttt{begin}\ \ D_V\ S\ \ \texttt{end}]]\, e_V\, sto = \mathcal{S}[[S]]\, e'_V\, sto'$$

dove e'_v e sto' sono l'ambiente e la store aggiornate dopo $update_V[[D_V]]\, e_V\, sto$.

Per gestire le procedure abbiamo bisogno dell'*ambiente delle procedure*: una funzione totale che associa ogni procedura all'effetto dell'esecuzione del loro corpo. Questo vuol dire che un ambiente di procedura è un elemento del tipo

$$\mathbf{Env}_P = \mathbf{Name} \hookrightarrow (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

Per aggiornare l'ambiente delle procedure useremo la funzione $update_P$:

$$\begin{aligned}
&update_V[[\varepsilon]]\, e_V = id \\
&update_V[[\texttt{proc}\ \ p\ \ \texttt{is}\ \ S; D_P]]\, e_V\, e_P = update_P[[D_P]]\, e_V\, (e_P[p \mapsto g])
\end{aligned}$$

dove $g = \mathcal{S}[[S]]\, e_V\, e_P$.
In pratica quando in un blocco incontriamo la dichiarazione di procedura, andiamo nell'ambiente delle procedure e associamo il nome della procedura a S, ossia il suo corpo. Inoltre notiamo come venga passato l'ambiente delle variabili e l'ambiente delle procedure del blocco in cui è presente la dichiarazione: questo è per avere lo scoping statico.
La funzione semantica dovrà ospitare anche l'ambiente delle procedure:

$$\mathcal{S} : \mathbf{Stmt} \to \mathbf{Env}_V \to \mathbf{Env}_P \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

La semantica di un blocco sarà

$$\mathcal{S}[[\texttt{begin}\, D_V\, D_P\, S\ \ \texttt{end}]]\, e_V\, e_P\, sto = \mathcal{S}[[S]]\, e'_V\, e'_P\, sto'$$

dove $update_V[[D_V]](e_V, sto) = (e'_V, sto')$ ci aggiorna l'ambiente delle variabili e $update_P[[D_P]]\, e'_V\, e_P = e'_P$ aggiorna l'ambiente delle procedure in base all'ambiente delle variabile appena aggiornato.
La semantica di `call` p è quella di andare nell'ambiente delle procedure ed eseguire la procedura con il nome p

$$\mathcal{S}[[\texttt{call}\ \ p]]\, e_V\, e_P = e_P p$$

Le altre funzioni semantiche non cambiano rispetto a prima, se non per ricevere come parametro anche l'ambiente delle procedure

$$\begin{aligned}
\mathcal{S}[[x := A]]\, e_V\, e_P\, sto &= sto[l \mapsto \mathcal{A}[[A]](sto \circ e_v)] \\
&\quad \text{dove } l = e_V x \\
\mathcal{S}[[\texttt{skip}]]\, e_V\, e_P &= id \\
\mathcal{S}[[S_1; S_2]]\, e_V\, e_P &= (\mathcal{S}[[S_2]]\, e_V\, e_P) \circ (\mathcal{S}[[S_1]]\, e_V\, e_P) \\
\mathcal{S}[[\texttt{if}(B)\ S_1\ \texttt{else}\ S_2]]\, e_V\, e_P &= cond(\mathcal{B}[[B]]\, e_V\, e_P, \mathcal{S}[[S_1]]\, e_V, \mathcal{S}[[S_2]]\, e_V\, e_P) \\
\mathcal{S}[[\texttt{while}(B)\, S]]\, e_V\, e_P &= FIX\, F \\
&\quad \text{dove } F\, g = cond(\mathcal{B}[[B]]\, e_V, g \circ (\mathcal{S}[[S]]\, e_V\, e_P), id)
\end{aligned}$$

Per vedere il funzionamento della semantica andiamo a considerare il brano di codice a pag 75 a inizio 6.3. La prima riga di codice ci indica che la semantica da valutare è $\mathcal{S}[[\texttt{begin}\, D_V\, D_P\, S\ \ \texttt{end}]]$ dove $D_V = (\texttt{var}\ \ x := 7;)$ e $D_P = (\texttt{proc}\ \ p\ \ \texttt{is}\ \ x := 0;)$.
Dovremo dunque aggiornare l'ambiente delle variabili e associare x alla locazione l_1 e associare la locazione l_1 a 7 nella store:

$$\begin{aligned}
update_V[[\texttt{var}\ \ x := 7;]] &= update[[\varepsilon]](e_V[x \mapsto l_1], sto[l_1 \mapsto x]) \\
&= (e_V[x \mapsto l_1], sto[l_1 \mapsto x])
\end{aligned}$$

Successivamente aggiorniamo l'ambiente delle procedure:

$$\begin{aligned}
&update_P[[\texttt{proc}\ \ p\ \ \texttt{is}\ \ x := 0;]](e_V[x \mapsto l_1])\, e_P = \\
&update[[\varepsilon]](e_V[x \mapsto l_1])(e_P[p \mapsto g]) = e_P[p \mapsto g]
\end{aligned}$$

dove g è l'effetto dell'esecuzione di p, i.e.

$$g\, sto = \mathcal{S}[[x := 0]](e_V[x \mapsto l_1])\, e_P\, sto = sto[l_1 \mapsto 0]$$

ossia presa la variabile d'ambiente appena aggiornata, quando p viene chiamata il valore associato a l_1 passa a 0.
Notiamo che per effetto dello scoping statico andiamo a modificare la x nel blocco più esterno che nella variabile d'ambiente è associata a l_1.
Ci ritroviamo un altro blocco (chiamiamolo P') e dunque dovremo trovare la semantica di

$$\mathcal{S}[[P']]\,(e_V[x \mapsto l_1])(e_P[p \mapsto g])(sto[l_1 \mapsto 7])$$

Incontriamo una dichiarazione di variabile. Dobbiamo ricordarci che la variabile pur avendo lo stesso nome del blocco esterno, riceve una nuova locazione in quanto la funzione $update_V$ crea una nuova locazione.
Poiché non abbiamo dichiarazioni di procedure, l'ambiente delle procedure rimane lo stesso:

$$\begin{aligned} update_V[[\texttt{var } x := 5;]]\,(e_V[x \mapsto l_1], sto[l_1 \mapsto 7]) = \\ update_V[[\varepsilon]]\,(e_V[x \mapsto l_2], sto[l_1 \mapsto 7][l_2 \mapsto 5]) = \\ (e_V[x \mapsto l_2], sto[l_1 \mapsto 7][l_2 \mapsto 5]) \end{aligned}$$

Infine abbiamo la chiamata alla procedura e dovremo perciò chiamare la funzione g:

$$\begin{aligned} \mathcal{S}[[\texttt{call } p]]\,(e_V[x \mapsto l_2])\,(e_P[p \mapsto g])\,(sto[l_1 \mapsto 7][l_2 \mapsto 5]) = \\ g\,(sto[l_1 \mapsto 7][l_2 \mapsto 5]) = sto[l_1 \mapsto 0][l_2 \mapsto 5] \end{aligned}$$

Nel caso volessimo permettere la definizione di funzioni ricorsive abbiamo bisogno di una funzione g : **Store** $\hookrightarrow$ **Store** che soddisfa

$$g = \mathcal{S}[[S]]\, e_V\, (e_P[p \mapsto g])$$

in modo tale da assicurarci che il significato di ogni chiamata ricorsiva è lo stesso della procedura che viene definita.
Dobbiamo cambiare la funzione $update_P$:

$$\begin{aligned} &update_P[[\varepsilon]] = id \\ &update_P[[\texttt{proc } p \texttt{ is } S; D_P]]\, e_V\, e_P = update_P[[D_P]]\, e_V\, (e_P[p \mapsto FIXF]) \end{aligned}$$

dove

$$F\, g = \mathcal{S}[[S]]\, e_V\, (e_P[p \mapsto g])$$

Usiamo il punto fisso come nel while e in modo tale definiamo la semantica del caso base e il passo induttivo della procedura.

Estendiamo il nostro linguaggio imperativo per permettere il passaggio di un parametro alle procedure.
Usiamo due semantiche diverse:

- Passaggio per valore: la procedura riceve una copia del parametro e la sua modifica non ha effetti sul chiamante;
- Passaggio per indirizzo: la procedura riceve la referenza al parametro e la sua modifica ha effetti sul chiamante;

Per il passaggio per valore l'ambiente delle procedure è un oggetto del tipo

$$\mathbf{Env}_P = \mathbf{Name} \hookrightarrow (\mathbf{Z} \to (\mathbf{Store} \hookrightarrow \mathbf{Store}))$$

Mentre una procedura $p(x)$ è una funzione g che prende il valore corrispondente al parametro x e trasforma una store in un altra store ed è di tipo

$$\mathbf{Z} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

La funzione $update_P$ diventa:

$$update_P[[\varepsilon]] = id$$
$$update_P[[\texttt{proc}\ \ p(x)\ \ \texttt{is}\ \ S; D_P]]\, e_V\, e_P = update_P[[D_P]]\, e_V\, (e_P[p \mapsto g])$$
$$\text{dove } g\, v\, sto = \mathcal{S}[[S]]\, e_V[x \mapsto l]\, e_P\, sto[l \mapsto v] \text{ e } l = newloc()$$

L'idea è quella di creaare una locazione nuova nell'ambiente delle variabili per il parametro in modo tale che ogni modifica rimanga locale alla funzione. Ovviamente associamo alla locazione il valore nella store.
Tra le regole semantica, l'unica che cambia è la chiamata:

$$\mathcal{S}[[\texttt{call}\ \ P(A)]]\, e_V\, e_P\, sto = (e_P\, p)\, (\mathcal{A}[[A]](sto \circ e_V))\, sto$$

Nel caso della chiamata per riferimento, l'ambiente delle procedure è un oggetto del tipo

$$\mathbf{Env}_P = \mathbf{Name} \hookrightarrow (\mathbf{Loc} \to (\mathbf{Store} \hookrightarrow \mathbf{Store}))$$

Mentre una procedura $p(x)$ è una funzione g che prende la locazione del parametro x e trasforma una store in un'altra store ed è di tipo

$$\mathbf{Loc} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

La funzione $update_P$ diventa:

$$update_P[[\varepsilon]] = id$$
$$update_P[[\texttt{proc}\ \ p(x)\ \ \texttt{is}\ \ S; D_P]]\, e_V\, e_P = update_P[[D_P]]\, e_V\, (e_P[p \mapsto g])$$
$$\text{dove } g\, l\, sto = \mathcal{S}[[S]]\, e_V[x \mapsto l]\, e_P\, sto$$

Anche qui l'unica funzione semantica che cambia è la chiamata

$$\mathcal{S}[[\texttt{call}\ \ P(y)]]\, e_V\, e_P\, sto = (e_P\, p)\, (e_V y)\, sto$$

Capitolo 7

Semantica operazionale

7.1 Sem. operazionale naturale di programmi sequenziali

Nella semantica operazionale naturale, la semantica di un programma è data da un **sistema di transizioni** e l'effetto dell'esecuzione di un programma è il cambiamento dello **stato**.
Il sistema di transizione avrà due configurazioni:

- s è una **configurazione terminale** che rappresenta un programma che è terminato;
- $\langle S, s\rangle$ rappresenta un programma S da eseguire quando lo stato è s;

Le relazioni di transizioni descrivono come l'esecuzione va avanti e una transizione

$$\langle S, s\rangle \to s'$$

indica che l'esecuzione di S nello stato s ci porta nello stato s'. La semantica naturale è una logica che comprende assiomi e sistema deduttivo.
Gli assiomi sono la semantica delle istruzioni di salto e di assegnamento

$$\text{(SKIP)} \quad \langle \texttt{skip}, s\rangle \to s$$
$$\text{(ASSIGN)} \quad \langle x := A, s\rangle \to s[x \to \mathcal{A}[[A]]\, s]$$

La semantica delle altre istruzioni è del tipo $\frac{A}{B}$ dove A è composto dalle premesse e B dalla transizione dell'istruzione che è la conclusione: la conclusione è valida solo se la premessa è vera.

$$\text{(COMP)} \quad \frac{\langle S_1, s\rangle \to s' \quad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

Nella composizione se dall'esecuzione dello statement S_1 nello stato s si passa allo stato s' e se l'esecuzione dello statement S_2 si passa allo stato s''

allora l'esecuzione della composizione dei due statement ci porta allo stato s''. La semantica di `if` è divisa in due parti in mutua esclusione tra di loro (ossia vale solo una di esse) in cui nella premessa è presente la valutazione della condizione Nel caso del `while`, se la condizione è vera l'idea è eseguire

$$(\text{IF}T)\quad \frac{\mathcal{B}[\![B]\!]s = T \qquad \langle S_1, s\rangle \to s'}{\langle \texttt{if } (B) \ S_1 \ \texttt{else } S_2, s\rangle \to s'}$$

$$(\text{IF}F)\quad \frac{\mathcal{B}[\![B]\!]s = F \qquad \langle S_2, s\rangle \to s'}{\langle \texttt{if } (B) \ S_1 \ \texttt{else } S_2, s\rangle \to s'}$$

il primo ciclo di esso e poi rivalutare la successiva iterazione con lo stato aggiornato. Se la condizione è falsa si salta e lo stato non viene aggiornato Prendiamo come esempio lo statement $(z := x; x := y); y := z$, la costruzione

$$(\text{WHILE}T)\quad \frac{\mathcal{B}[\![B]\!]s = T \qquad \langle S, s\rangle \to s' \quad \langle \texttt{while } (B) \ S, s'\rangle \to s''}{\langle \texttt{while } (B) \ S, s\rangle \to s''}$$

$$(\text{WHILE}F)\quad \frac{\mathcal{B}[\![B]\!]s = F}{\langle \texttt{while } (B) \ S, s\rangle \to s}$$

dell'albero di iterazione è iterativa e si fa passo per passo.
Per prima cosa scriviamo la parte sotto della "frazione", ossia la conslusione però senza specificare lo stato finale in quanto non lo sappiamo ancora

$$\frac{?}{\langle (z := x; x := y); y := z, s\rangle \to ?}$$

Abbiamo una composizione, dunque possiamo scrivere le premessa

$$\frac{\langle z := x; x := y, s\rangle \to ? \quad \langle y := z, ?\rangle \to ?}{\langle (z := x; x := y); y := z, s\rangle \to ?}$$

Abbiamo un altra composizione, scriviamo la composizione di essa

$$\frac{\frac{\langle z:=x,s\rangle \to s' \langle x:=y,s'\rangle \to s''}{\langle z:=x;x:=y,s\rangle \to s''} \quad \langle y := z, ?\rangle \to ?}{\langle (z := x; x := y); y := z, s\rangle \to ?}$$

Da qui possiamo complentare la semantica dell'intero statement

$$\frac{\frac{\langle z:=x,s\rangle \to s' \langle x:=y,s'\rangle \to s''}{\langle z:=x;x:=y,s\rangle \to s''} \quad \langle y := z, s''\rangle \to \ s'''}{\langle (z := x; x := y); y := z, s\rangle \to s'''}$$

Duqnue se inizialmente $x = 5$ e $y = 7$ allora $s''' = s[z \mapsto 5][y \mapsto 5][x \mapsto 7]$.

Una volta che abbiamo formalizzato la semantica di un programma possiamo dimostrare proprietà sui programmi.
Notiamo che la relazione di transizione è **deterministica** ossia

$$\langle S, s\rangle \to s' \text{ e } \langle S, s\rangle \to s'' \implies s' = s''$$

Diremo che due programmi S_1 e S_2 sono equivalenti se per ogni stato s, s' abbiamo che

$$\langle S_1, s\rangle \to s' \iff \langle S_2, s\rangle \to s'$$

Possiamo dunque dimostrare l'associatività della composizione in quanto $(S_1; S_2); S_3$ e $S_1; (S_2; S_3)$ sono equivalenti.
Si dimostra anche che `while`$(B)\, S$ e `if`$(B)\, (S;$ `while`$(B)\, S)$ `else skip` sono equivalenti.

Il significato degli statement è adesso quello di una funzione parziale che dato uno stato, ne restituisce un altro

$$\mathcal{S}[[S]] : \mathbf{Stmt} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

ed è definita come

$$\mathcal{S}[[S]]\, s = \begin{cases} s' & \text{se } \langle S, s\rangle \to s' \\ \mathbf{undef} & \text{altrimenti} \end{cases}$$

Per esempio $\mathcal{S}[[$`while (true) skip`$]]\, s$ è **undef** per ogni stato s.

7.2 Sem. operazionale naturale con blocchi e procedure

Per poter definire la semantica naturale dei blocchi dobbiamo arricchire il sistema di transizioni con configurazioni della forma $\langle D_V, s\rangle$ e transizioni della forma $\langle D_V, s\rangle \to_D s'$.
La semantica naturale per la dichiarazioni di variabili sarà

$$(\textsc{novar}) \quad \langle \varepsilon, s\rangle \to_D s$$

$$(\textsc{var}) \quad \frac{\langle D_V, s[x \mapsto \mathcal{A}[\![A]\!]s]\rangle \to_D s'}{\langle \texttt{var } x := A;\ D_V, s\rangle \to_D s'}$$

Mentre per la dichiarazione dei blocchi avremo

$$(\textsc{block}) \quad \frac{\langle D_V, s\rangle \to_D s' \quad \langle S, s'\rangle \to s''}{\langle \texttt{begin } D_V\ S\ \texttt{end}, s\rangle \to s''[\mathrm{DV}(D_V) \mapsto s]}$$

Se la dichiarazioni delle variabili ci porta allo stato s' e l'esecuzione del corpo del blocco nello stato s' ci porta allo stato s'', allora l'esecuzione del blocco ci porta allo stato s'', ma in cui le variabili dichiarate locali ritornano allo stato di partenza s e perciò esse cessano di esistere fuori dal blocco.
Questo è infatti il significato di $s''[\mathrm{DV}(D_V) \mapsto s]$.

Per le procedure abbiamo tre tipi di semantiche:

1. Scoping dinamico sia per le procedure che per le variabili:
2. Scoping dinamico per le variabili, ma statico per le procedure;
3. Scoping statico sia per le procedure che per le variabili.

Per lo scoping dinamico completo dobbiamo tenere traccia dell'associazione tra corpo della procedura e il suo nome: per fare ciò usiamo l'**ambiente**, una mappa dai nomi delle procedure ai loro corpi

$$\mathbf{Env}_P = \mathbf{Name} \hookrightarrow \mathbf{Stmt}$$

Dobbiamo cambiare forma delle transizioni per gli statement:

$$e \vdash \langle S, s\rangle \to s'$$

indica che l'esecuzione di S nello stato s e nell'ambiente e ci porta allo stato s'.
Nell'operazione di update associamo il nome della procedura al suo corpo

$$update_P(\varepsilon, e_P) = e_P$$
$$update_P(\texttt{proc } p \texttt{ is } S; D_P, e_P) = update_P(D_P, e_P[p \mapsto S])$$

Ovviamente dovranno essere aggiunte delle nuove regole.
La regola per la dichiarazione del blocco sarà:

$$(\textsc{block}) \quad \frac{\langle D_V, s\rangle \to_D s' \quad update_P(D_P, e_P) \vdash \langle S, s'\rangle \to s''}{e_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end}, s\rangle \to s''[\mathrm{DV}(D_V) \mapsto s]}$$

Lo scoping dinamico viene attuato nella chiamata a procedura in quanto il corpo di una procedura p viene eseguito nell'ambiente **attuale** e_P (ossia quello del chiamante) e non nell'ambiente dove è stato definito p:

$$(\textsc{call}) \quad \frac{e_P \vdash \langle (e_P\ p), s\rangle \to s'}{e_P \vdash \langle \texttt{call}\ p, s\rangle \to s'}$$

Se vogliamo lo scoping statico per le procedure (non per le variabili) dobbiamo modificare l'ambiente:

$$\mathbf{Env}_P = \mathbf{Name} \hookrightarrow (\mathbf{Stmt} \times \mathbf{Env}_P)$$

In questo modo oltre ad associare il nome di una procedura al suo corpo, lo associamo anche all'ambiente in cui è stato dichiarato. La funzione $update_P$ diventa dunque

$$update_P(\varepsilon, e_P) = e_P$$
$$update_P(\texttt{proc}\ p\ \texttt{is}\ S; D_P, e_P) = update_P(D_P, e_P[p \mapsto (S, e_P)])$$

L'unica regola che cambia è quella della chiamata:

$$(\textsc{call}) \quad \frac{e'_P \vdash \langle S, s\rangle \to s'}{e_P \vdash \langle \texttt{call}\ p, s\rangle \to s'}$$

dove $(e_p\, p) = (S, e'_p)$. Lo scoping statico è garantito dal fatto che e'_P è l'ambiente delle procedure quando la procedura p è stata dichiarata e S è il suo corpo sempre nello stesso ambiente.
Se vogliamo anche gestire la ricorsione dobbiamo assicurarci che le occorrenze di `call` p all'interno del corpo di p si riferiscono alla procedura stessa. Per far ciò nella premessa quando denotiamo l'ambiente delle variabili, andiamo a specificare la procedura stessa.

$$(\textsc{call}) \quad \frac{e'_P[p \mapsto (S, e'_P)] \vdash \langle S, s\rangle \to s'}{e_P \vdash \langle \texttt{call}\ p, s\rangle \to s'}$$

Per permettere lo scoping statico per le variabili dobbiamo introdurre l'**ambiente delle variabili** che associa il nome delle variabili alla loro locazione e di una **store** che associa ogni locazione al suo valore:

$$\mathbf{Env}_V = \mathbf{Var} \mapsto \mathbf{Loc}$$
$$\mathbf{Store} = \mathbf{Loc} \to \mathbb{Z}$$

Il sistema delle transizioni per la dichiarazioni di variabili cambia come segue:

$$\langle D_V, e_V, sto\rangle \to_D (e'_V, sto')$$

Cambia la semantica naturale della dichiarazione delle variabili che si deve occupare adesso di memorizzare gli assegnamenti a variabile

$$(\textsc{novar}) \quad \langle \varepsilon, e_V, sto\rangle \to_D (e_V, sto)$$

$$(\textsc{var}) \quad \frac{\langle D_V, e_V[x \mapsto l], sto[l \mapsto v]\rangle \to_D (e'_V, sto')}{\langle \texttt{var } x := A;\ D_V, e_V, sto\rangle \to_D (e'_V, sto')}$$

dove $v = \mathcal{A}[[A]](sto \circ ev)$ e $l = newloc()$.
Per le procedure dovremo cambiare l'ambiente delle procedure in modo tale da associare il nome della procedura non solo al corpo e all'ambiente delle procedure in cui è stato definito, ma anche all'ambiente delle variabili al momento della definizione:

$$\mathbf{Env}_P = \mathbf{Name} \hookrightarrow \mathbf{Stmt} \times \mathbf{Env}_V \times \mathbf{Env}_P$$

Cambia anche ovviamente $update_P$ che deve memorizzare per ogni procedura dichiarata, anche l'ambiente delle variabili al momento della dichiarazione:

$$update_P(\varepsilon, e_V, e_P) = e_P$$
$$update_P(\texttt{proc } p \texttt{ is } S, e_V, e_P) = update_P(D_p, e_V, e_P[p \mapsto (S, e_V, e_P)])$$

Il sistema delle transizioni dovrà tener conto anche a che ambiente di variabili ci riferiamo:

$$e_V, e_p \vdash \langle S, sto\rangle \to sto'$$

Cambia la semantica dello skip e dell'assegnamento che dovrà associare alle variabili il loro valore nella store

$$(\textsc{skip}) \quad e_V, e_P \vdash \langle \texttt{skip}, sto\rangle \to sto$$

$$(\textsc{assign}) \quad e_V, e_P \vdash \langle x := A, sto\rangle \to sto[l \mapsto v]$$

dove $l = ev\, x$ e $v = \mathcal{A}[[A]](sto \circ e_V)$.
Nella dichiarazione dei blocchi ci occupiamo di fornire le locazioni alle variabili

$$(\textsc{block}) \quad \frac{\langle D_V, e_V, sto\rangle \to_D (e'_V, sto') \quad e'_V, e'_P \vdash \langle S, sto'\rangle \to sto''}{e_V, e_P \vdash \langle \texttt{begin } D_V\ D_P\ S \texttt{ end}, sto\rangle \to sto''}$$

Nella semantica della chiamata a procedura assicuriamo lo scoping statico di variabili e procedure nella premesse specificando gli ambienti al momento della dichiarazione

$$(\textsc{call}) \quad \frac{e'_V, e'_P[p \mapsto (S, e'_V, e'_P)] \vdash \langle S, sto\rangle \to sto'}{e_V, e_P \vdash \langle \texttt{call } p, sto\rangle \to sto'}$$

dove $(e_P\, p) = (S, e'_V, e'_P)$

7.3 Sem. operazionale strutturata

Nella semantica operazionale strutturata continuiamo ad avere un sistema di transizione, ma l'enfasi è sui **passi individuali** dell'esecuzione.
La relazione di transizione ha forma

$$\langle S, s\rangle \Rightarrow \gamma$$

γ può essere di due forme:

1. Se è nella forma $\langle S', s'\rangle$, l'esecuzione di S da s non è completata e ci porta alla **configurazione intermedia** $\langle S', s'\rangle$;
2. Se è nella forma s', l'esecuzione di S da s è **terminata** e lo stato finale è s';

Se non esiste γ tale per cui $\langle S, s\rangle \Rightarrow \gamma$ allora diremo che $\langle S, s\rangle$ è **bloccato**.
Vediamo la semantica del linguaggio imperativo usato fin'ora:

$$(\textsc{skip}) \quad \langle \texttt{skip}, s\rangle \Rightarrow s$$

$$(\textsc{assign}) \quad \langle x := A, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![A]\!]s]$$

Dividiamo la composizione di istruzioni nel caso che la prima termini subito la sua esecuzione o meno:

$$(\textsc{comp1}) \quad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

$$(\textsc{comp2}) \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

Per l'`if` differenziamo in base alla condizione booleana:

$$(\textsc{ifT}) \quad \langle \texttt{if } (B) \ S_1 \ \texttt{else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle$$

se $\mathcal{B}[[B]]\, s = T$

$$(\textsc{ifF}) \quad \langle \texttt{if } (B) \ S_1 \ \texttt{else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$$

se $\mathcal{B}[[B]]\, s = F$.
L'idea nel `while` è di trattarlo come un `if`, eseguire il primo ciclo e reiterare con lo stato aggiornato:

$$(\textsc{while}) \ \langle \texttt{while } (B) \ S, s\rangle \Rightarrow \langle \texttt{if } (B) \ (S; \texttt{while } (B) \ S) \ \texttt{else skip}, s\rangle$$

La relazione $\Rightarrow$ è **deterministica**, i.e. $\langle S, s\rangle \Rightarrow \gamma$ e $\langle S, s\rangle \Rightarrow \gamma'$ implica che $\gamma = \gamma'$ per ogni scelta di S, s, γ, γ'.

Adesso confrontiamo le tre semantiche viste fin'ora e denotiamo con:

- $\mathcal{S}_{ds}[[\cdot]]$ la semantica denotazionale;
- $\mathcal{S}_{ns}[[\cdot]]$ la semantica naturale;
- $\mathcal{S}_{sos}[[\cdot]]$ la semantica operazionale;

Per ogni statement S visto nel linguaggio imperativo fino ad adesso abbiamo che:

$$\mathcal{S}_{ns}[[S]] = \mathcal{S}_{sos}[[S]]$$
$$\mathcal{S}_{sos}[[S]] = \mathcal{S}_{ds}[[S]]$$

Cerchiamo adesso di illustrare l'espressività e le debolezze dei due tipi di semantica operazionali estendo il nostro linguaggio con lo statement `abort` che ha come effetto la terminazione dell'esecuzione del programma.
Per ogni stato s le configurazioni del tipo $(\texttt{abort}, s)$ sono **bloccate**.

$$\langle \texttt{abort}, s\rangle \not\rightarrow$$
$$\langle \texttt{abort}, s\rangle \not\Rightarrow$$

Dal punto di vista della semantica operazionale `abort` e `skip` non sono equivalenti semanticamente in quanto $\langle \texttt{skip}, s\rangle \Rightarrow s$, mentre $\langle \texttt{abort}, s\rangle \not\Rightarrow$.
Analogamente `abort` non è semanticamente equivalente a `while (true) skip` in quanto

$$\begin{aligned}
&\langle \texttt{while (true) skip}, s\rangle \\
&\quad\Rightarrow\ \langle \texttt{if (true) (skip; while (true) skip) else skip}, s\rangle \\
&\quad\Rightarrow\ \langle \texttt{skip; while (true) skip}, s\rangle \\
&\quad\Rightarrow\ \langle \texttt{while (true) skip}, s\rangle \\
&\quad\Rightarrow\ \ldots
\end{aligned}$$

esprime una sequenza di derivazione infinita, mentre `abort` nessuna.
Dal punto di vista della semantica naturale `abort` e `skip` non sono semanticamente equivalenti, ma `abort` e `while (true) skip` lo sono: questo perchè con la semantica naturale ci concentriamo solo sulle esecuzioni che terminano propriamente e dunque se non abbiamo un'albero di derivazione per una transizione non siamo in grado di di stabilire se è per colpa di una configurazione bloccante o per un loop infinito.

Adesso estendiamo il nostro linguaggio in modo da permettere il nondeterminismo.

$$S ::= \ldots \mid S \ \texttt{or}\ S$$

L'idea è che possiamo scegliere in modo nondeterministico di eseguire una delle due istruzioni.

Le regole per la semantica operazionale saranno:

$$(\text{OR1}) \quad \langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_1, s\rangle \quad (\text{OR2}) \quad \langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$$

Mentre per la semantica naturale avremo

$$(\text{OR1}) \quad \frac{\langle S_1, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2, s\rangle \to s'} \quad (\text{OR2}) \quad \frac{\langle S_2, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2, s\rangle \to s'}$$

Consideriamo (`while (true) skip`) `or` $(x := 2; x := x + 2)$: in SOS abbiamo due sequenze di derivazione, la prima infinita e l'altra che ci porta a $s[x \mapsto 4]$, mentre in NS abbiamo solo l'ultima.
In $\mathcal{S}_{ns}[[\cdot]]$ il **nondeterminismo sopprime il looping** quando possibile, mentre in $\mathcal{S}_{sos}[[\cdot]]$ ciò non succede.
Consideriamo infine l'ultima estensione, il parallelismo nella composizione di istruzioni, in modo da eseguirle in qualsiasi ordine (interleving)

$$S ::= \ldots \mid S \texttt{ par } S$$

Vediamo la semantica operazionale

$$(\text{PAR1}) \quad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1' \texttt{ par } S_2, s'\rangle}$$

$$(\text{PAR2}) \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$(\text{PAR3}) \quad \frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1 \texttt{ par } S_2', s'\rangle}$$

$$(\text{PAR4}) \quad \frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

Se prendiamo per esempio $(x := 1)$ `par` $(x := 2; x := x + 2)$ possiamo avere tre risultati diversi:

1. Se eseguiamo prima $x := 1$ e poi $x := 2; x := x+2$ otteniamo $s[x \mapsto 4]$;
2. Se eseguiamo prima $x := 2; x := x+2$ e poi $x := 1$ otteniamo $s[x \mapsto 1]$;
3. Se eseguiamo prima $x := 2$ e poi $x := 1$ e infine $x := x + 2$ otteniamo $s[x \mapsto 3]$;

Vediamo la semantica naturale invece

$$(\textsc{par1}) \quad \frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1 \texttt{ par } S_2, s\rangle \to s''}$$

$$(\textsc{par2}) \quad \frac{\langle S_2, s\rangle \to s' \qquad \langle S_1, s'\rangle \to s''}{\langle S_1 \texttt{ par } S_2, s\rangle \to s''}$$

Le regole esprimono solo il fatto che S_1 può essere eseguito prima di S_2 o viceversa. Questo implica che non è possibile esprimere l'interleaving con la semantica naturale.