Indice

1 Introduzione

1.1 Motivazione

Un linguaggio è uno strumento per descrivere come risolvere i problemi in maniera rigorosa, in modo tale che sia eseguibile da un calcolatore Perché è utile studiare come creare un linguaggio di programmazione?

- non rimanere degli utilizzatori passivi
- capire il funzionamento dietro le quinte di un linguaggio
- domain-specific language (DSL): è un linguaggio pensato per uno specifico problema
- model drivern software development: modo complesso per dire UML e simili
- model checking

1.2 Definizioni base

Un linguaggio è composto da:

- lessico e sintassi
- compilatore: parser + generatore di codice oggetto

La generazione automatica di codice può essere dichiarativa lessico (espressioni regolari o automa a stati finite) o sintassi(grammatiche o automa a pile). Un automa a stati finiti consuma informazioni una alla volta, ne salva una quantità finita. Alcuni esempi di applicazione di automa a stati finiti: software di progettazione di circuiti, analizzatore lessicale, ricerca di parole sul web e protocolli di comunicazione.

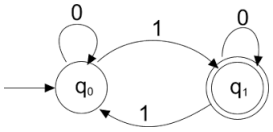


Figura 1: Semplice automa

1.3 Contenuti del corso

- Linguaggi formali e Automi:
  - Automi a stati finiti, espressioni regolari, grammatiche libere, automi a pila, Macchine di Turing, calcolabilità
- Compilatori:
  - Analisi lessicale, analisi sintattica, analisi semantica, generazione di codice
- Logica di base:
  - Logica delle proposizioni e dei predicati
- Modelli computazionali:
  - Specifica di sistemi tramite sistemi di transizione, logiche temporali per la specifica e verifica di proprietà dei sistemi (model checking), sistemi concorrenti (algebre di processi e reti di Petri)

1.4 Informazioni utili

Parte integrante del corso:

- Supporto alla parte teorica usando tool specifici.
  - JFLAP 7.1: http://www.jflap.org (automi/grammatiche)
  - Tina 3.7.5: http://projects.laas.fr/tina (model checking di sistemi di transizione e reti di Petri)
  - LTSA 3.0: http://www.doc.ic.ac.uk/ltsa (sistemi di transizione definiti tramite algebre di processi)
- Nel resto del corso utilizzeremo un ambiente di sviluppo per generare parser/compilatori
  - IntelliJ esteso con plug-in ANTLRv4, ultima versione 1.20 (generatore ANTLR: http://www.antlr.org/)

Libri di testo suggeriti:

- J. E. Hopcroft, R. Motwani e J. D. Ullman: Automi, linguaggi e calcolabilita', Addison-Wesley, Terza Edizione, 2009. Cap. 1–9
- A. V. Aho, M. S. Lam, R. Sethi e J. D. Ullman: Compilatori: principi tecniche e strumenti, Addison Wesley, Seconda Edizione, 2009. Cap. 1–5
- M. Huth e M. Ryan: Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, Second Edition, 2004. Cap. 1–3

2 Linguaggi regolari

2.1 Alfabeti

Un *alfabeto* è un insieme finito e non vuoto di simboli, comunemente indicato con Σ. Seguono alcuni esempi di alfabeti:

- Σ = {0,1} alfabeto binario
- Σ = {a,b,...,z} alfabeto di tutte lettere minuscole
- L'insieme ASCII

2.1.1 Stringhe

Una stringa/parola è un insieme di simboli di un alfabeto, 0010 è una stringa che appartiene Σ = {0,1}.
La *stringa vuota* è una stringa composta da 0 simboli.
La lunghezza della stringa sono il numeri di caratteri che la compongono (non devono essere unici). La sintassi per la lunghezza di una stringa w è $|w|$, quindi $|001| = 3$ oppure $|\epsilon| = 0$ (nota bene, $\epsilon \neq 0$ ma è di lunghezza 0).

Potenze di un alfabeto

Se Σ è un alfabeto si può esprimere l'insieme di tutte le stringhe di una certa lunghezza con una notazione esponenziale: Σ^k denota tutte le stringhe di lunghezza k con simboli che appartengono a Σ.
Per esempio:
Σ^1 = {0,1}
Σ^2 = {00, 01, 10, 11}
Σ^2 = {000, 001, 010, 011, 100, 101, 110, 111}
L'insieme delle stringhe meno quella vuota è segnato come Σ^+, mentre l'insieme che include la stringa vuota è Σ^*,

2.1.2 Concatenazione di stringhe

Siano x e y stringhe, dove i è la lunghezza di x e j è la lunghezza di y, la stringa xy è la stringa risultata dalla concatenazione delle stringhe xy di lunghezza i+j.

2.2 Definizione di linguaggio

Un insieme di stringhe a scelta L $\subseteq \Sigma^*$ si definisce linguaggio su Σ.
Un modo formale per definire un alfabeto è il seguente {w | enunciato su w}, che si traduce in "w tale che enunciato su w".
$\{0^n1^n|n \geq 1\}$ si traduce in "l'insieme di 0 elevato alla n, 1 alla n tale che n è maggiore o uguale a 1"

3 Automa a stati finiti deterministico

Un automa a stati finiti deterministico consiste in:

1. Un insieme di stati finiti Q
2. Un insieme di simboli di input, Σ
3. Una funzione di transizione, che prende in input uno stato e un simbolo e restituisce uno stato. Tale funzione è spesso indicato con δ ed è usata per rappresentare i archi nella rappresentazione grafica. Ovvero sia q uno stato, a un input allora δ(q,a) è lo stato p tale che esista un arco da q a p.
4. Uno stato iniziale (naturalmente che appartiene a Q)
5. Un insieme di stati accettati finali F. Questo è un sottoinsieme di Q.

Un automa a stati finiti deterministico è spesso chiamato con l'acronimo DFA e viene può essere rappresentato nella seguente maniera concisa:

$$A = (Q, \Sigma, \delta, q_0, F)$$

Dove A rappresenta il DFA.

3.1 Elaborazione di stringhe

Per elaborare una stringa è si definisce lo stato iniziale, quello finale e una serie di regole di transizione per poterci arrivare. Se dovessi decodificare la stringa 01 il DFA risulterebbe:

$$A = (Q = \{q1, q2, q3\}, \{0, 1\}, \delta, q_0, \{q1\})$$

I stati sono i sequenti:
$\delta(q_0, 1) = q_0$: leggo come primo stato 1, nessun progresso fatto
$\delta(q_0, 0) = q_2$: leggo come primo stato 0, posso andare avanti e cercare un 1
$\delta(q_2, 1) = q_1$: leggo 1 dopo lo 0, ho trovato la stringa
$\delta(q_2, 0) = q_2$: leggo 0 dopo lo 0, non ho fatto progresso
Nota bene: questa è una notazione arbitraria del libro, q1 e q2 si possono invertire.

3.1.1 Notazioni semplici per DFA

Diagramma di transizione

Dato un DFA $A = (Q, \Sigma, \delta, q_0, F)$ un suo diagramma di transizione è composto da:

- Ogni stato Q è un nodo
- Ogni funzione δ è una freccia
- La freccia Start che denota il primo input
- Gli stati accettati F hanno un doppio cerchio

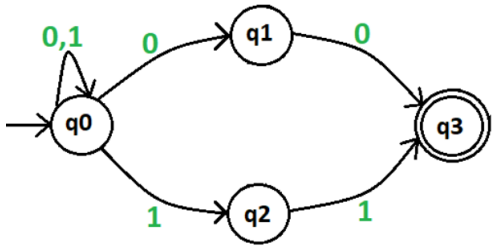


Figura 2: Diagramma di transizione

Tabelle di transizione

Una tabella di transizione è costituita nelle riga dalle funzioni δ e nelle colonne dagli input. Ogni incrocio equivale a uno stato della funzione δ con un input generico *a*.

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

Tabella 1: Esempio di tabella

La freccia è lo start e l'asterisco è lo stato finale.

3.1.2 Estensione della funzione di transizione di stringhe

Allo scopo di poter seguire una sequenza di input ci serve definire una funzione di transizione estesa. Se δ è una funzione di transizione, chiameremo $\hat{\delta}$ la sua funzione estesa. La funzione estesa prende in input q e una stringa w e ritorna uno stato p.
Ogni stato viene calcolato grazie allo stato esteso precedente:

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

Esempio

L = { w | w ha un numero pari di 0 e di 1 }
Nota bene: 0 (numero di simboli) è pari quindi conta come stato accettato, ed è l'unico stato accettato.

q_0: 0 e 1 sono pari
q_1: 0 pari 1 dispari
q_2: 1 pari 0 dispari
q_3: 0 dispari 1 dispari

$$A = (\{q0, q1, q2, q3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

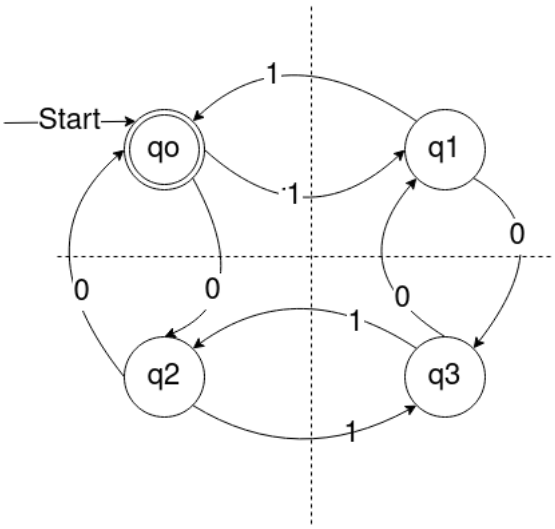


Figura 3: Diagramma

	0	1
$\rightarrow * \mathrm{q}_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Tabella 2: Esempio funzioni

Ora applichiamo le funzione di transizione estesa per verificare che 110101 abbia 0 e 1 pari:

- $\hat{\delta}(q_0, \epsilon) = \mathrm{q}_0$
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = \mathrm{q}_1$
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = \mathrm{q}_0$
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_1, 11), 0) = \delta(q_0, 1) = \mathrm{q}_2$
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = \mathrm{q}_3$
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = \mathrm{q}_1$
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = \mathrm{q}_0$

A ogni simbolo aggiunto posso usare la funzione estesa precedente per calcolare il prossimo stato, in questo caso la sequenza ha un numero pari di 0 e 1.

4 Automa a stati finiti non deterministici

Un NFA (nondeterministic finite automaton) può trovarsi contemporaneamente in diversi stati. L'automa "scommette" sul input su certe proprietà dell'input.
I NFA sono spesso più succinti e facili da definire rispetto ai DFA, un DFA può avere un numero di stati addirittura esponenziale rispetto a un NFA. Ogni NFA può essere convertito in un DFA.

4.1 Descrizione informale

A differenza di un DFA, una funzione di stato in un NFA può restituire 0 o più stati. Immaginiamo di dover identificare se una stringa finisce con 01. Di seguito il diagramma di transizione sarà il seguente. Come è possibile

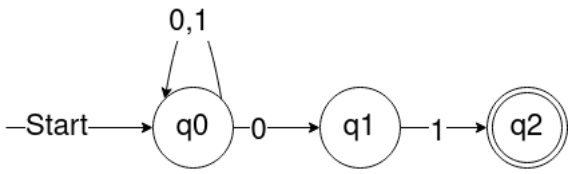


Figura 4: NFA che accetta stringa che finisce con 01

notare q_0 può può restituire due stati se riceve uno 0. Il NFA esegue molteplici stadi alla ricerca del pattern (simile a un processo che si moltiplica).

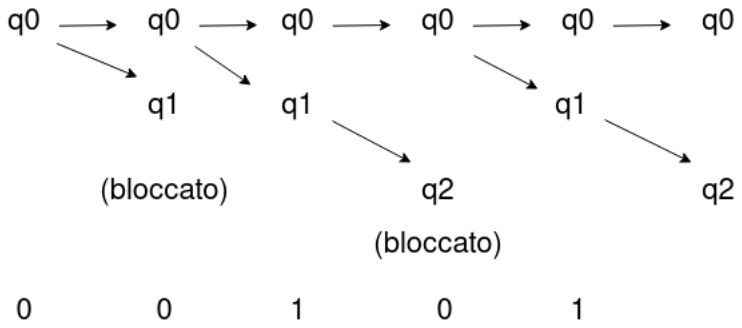


Figura 5: Gli stati del NFA

Ogni volta che il NFA accetta uno stato 0 crea due processi, un q_1 e q_0 A ogni successivo input tutti i processi vanno avanti, nel nostro caso il q_1 "muore". Al secondo giro viene creato q_1 che muore alla quarta iterazione perché non è l'ultimo simbolo. Durante la quarta iterazione nasce q_1 che alla quinta ci porta uno stato accettato.

4.2 Definizione formale

Formalmente un NFA si definisce come un DFA.

$$A = (Q, \Sigma, \delta, q_0, F)$$

1. Un insieme di stati finiti Q
2. Un insieme di simboli di input, Σ
3. Una funzione di transizione, che prende in input uno stato e un simbolo e restituisce ***un insieme di stati***. Questa è l'unica differenza rispetto al DFA, dove ci viene restituito un singolo stato.
4. Uno stato iniziale (naturalmente che appartiene a Q)
5. Un insieme di stati accettati finali F. Questo è un sottoinsieme di Q.

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Tabella 3: Tabella di transizione di una NFA che accetta una stringa che finisce con 01

L'unica differenza con una tabella DFA è che negli incroci ci sono dei insiemi di stati di output (singoletto quanto è uno solo), mentre se la transizione non esiste viene segnata con $\emptyset$.

4.3 Funzione di transizione estesa

Come per i DFA bisogna prendere la funzione di transizione e renderla estesa. In questo caso lo stato precedente può ritorna un insieme di stati, quindi bisognare fare l'unione di questi. La funzione estesa di δ si chiamerà $\hat{\delta}$.

$$\bigcup_{x=2}^{k} \delta(p_i, a) = \{r_1, r_2, ..., r_m\}$$

Usiamo $\hat{\delta}$ per calcolare se la stringa 00101 finisce con 01.

1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Abbiamo un risultato positivo, q_2 mentre q_0 viene scartato

4.4 Linguaggio NFA

Come abbiamo visto sopra, il fatto di avere uno stato non accettabile al termine dell'operazione non significa che non abbia avuto successo.
Formalmente se $A = (Q, \Sigma, \delta, q_0, F)$ è un NFA allora:

$$L(A) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

In parole povere L(A) è l'insieme delle stringhe w in Σ^* tale che $\hat{\delta}(q_0, w)$ contenga almeno uno stato accettante.

4.5 Equivalenza tra DFA e NFA

Di solito è più facile ottenere un NFA piuttosto che un DFA per un linguaggio. Nel migliori dei casi un DFA ha circa tanti stati quanti un NFA, ma più transizioni. Nel caso peggiore un DFA ha 2^n stati, mentre un NFA n.
Come detto in precedenza ogni NFA può essere ricondotto a un DFA, questo andrà dimostrato costruendo un DFA per insiemi a partire da un NFA.
Dato un NFA $A = (Q_N, \Sigma, \delta_N, q_0, F_N)$ possiamo costruire un DFA
$A = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tale che L(D)=L(N) (che i linguaggio sono uguali).
Si noti che i due linguaggi condividono lo stesso alfabeto.
Gli altri D componenti sono fatti nel seguente modo:

- Q_D è formato da un insieme di insiemi di Q_N, in termini formali Q_D è l'insieme potenza di Q_N. Quindi se Q_N ha n stati allora Q_D ha 2^n stati, questo è vero nella teoria, nella pratica gli stati non raggiungibili non contano quindi tendono a essere meno di 2^n.

- F_D è l'insieme dei sottoinsiemi di S di Q_N tale che $S \cap F_N \neq \emptyset$. F_D è quindi formato dagli sottoinsiemi di stati N che includono almeno uno stato accettante.

- Per ogni insieme $S \subseteq Q_N$ e per ogni simbolo a in Σ,

$$\delta_D(S, a) = \bigcup_{p\ in\ S} \delta_N(p, a)$$

Ovvero l'insieme $\delta_D(S, a)$ è calcolato tramite l'unione di tutti gli insiemi p in S.

La tabella precedente era deterministica nonostante fosse formata da insiemi, *ogni insieme è uno stato*, e non sono insieme di stati. Per rendere più chiara l'idea possiamo cambiare notazione.
Tra gli 8 stati presenti in tabella possiamo raggiungere: B, E e F. Gli atri stati sono irraggiungibili o non esistenti. È possibile evitare di costruire questi stati compiendo una "valuta differita".
Trattando i l'insieme di stati come un unico stato composto da un insieme è possibile riscrivere la DFA in questo modo:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Tabella 4: Stringa che termina con 01, NFA $\rightarrow$ DFA

	0	1
A	A	A
$\rightarrow$B	E	B
C	A	D
$*$D	A	A
E	E	F
$*$F	E	B
$*$G	A	D
$*$H	E	F

Tabella 5: Stringa che termina con 01, notazione nuova

Teorema

Se $D = (Q_N, \Sigma, \delta_N, q_0, F_N)$ è il DFA trovato per costruzione a partire dal NFA $N = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ allora L(D)=L(N).

Teorema

Un linguaggio L è accettato da un DFA se e solo se L è accettato da un NFA.

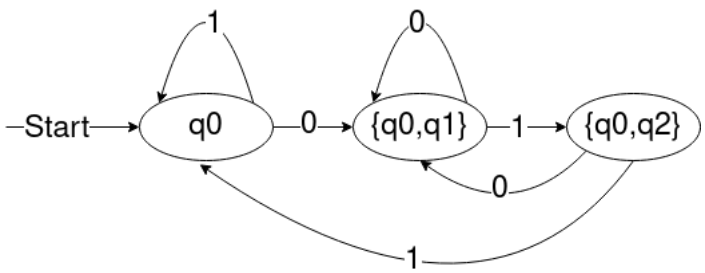


Figura 6: Grafico DFA convertito da NFA

5 Automa con epsilon-transazioni

Un estensione degli automa è la capacità di poter ammettere come input la stringa vuota ϵ. È come se l'NFA compisse una transizioni spontaneamente. Tale NFA si chiamerà ϵ-NFA

5.1 Uso delle epsilon-transizioni

L'esempio di seguito tratta le ϵ come invisibili, possono mutare lo stato ma non sono contante nella catena.

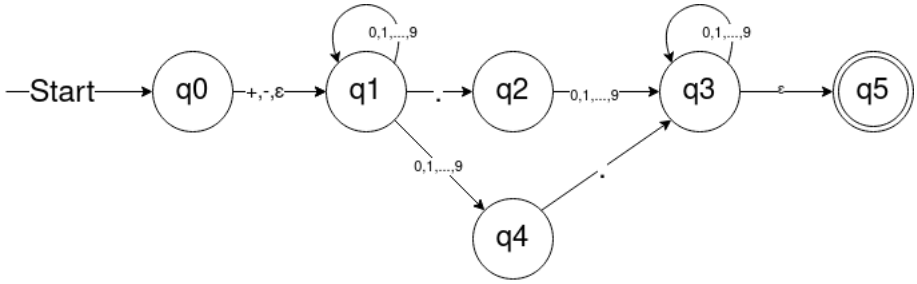


Figura 7: epsilon-NFA che accetta numeri decimali

L'ϵ-NFA in figura accetta numeri decimali formati da:

1. un segno +,- facoltativo
2. una sequenza di cifre
3. un punto decimale
4. una seconda sequenza di cifre

È possibile avere input vuoti prima della virgola $\delta(q_1, .) = q_2$ e dopo la virgola $\delta(q_4, .) = q_3$ ma non entrambi. Il segno è facoltativo $\delta(q_0, \epsilon) = q_1$.
In q_3 l'automa può "scommettere" che la sequenza sia finita oppure può andare avanti a leggere.

5.2 Notazione formale di epsilon-NFA

La definizione forma di un ϵ-NFA è uguale a quella di un NFA, va solo specificate le informazioni relative alla transizione ϵ.
Una ϵ-NFA è definita con $A = (Q, \Sigma, \delta, q_0, F)$, dove δ è una funzione di transizione che richiede come input:

1. uno stato Q
2. un elemento $\Sigma \cup \{\epsilon\}$, ovvero un simbolo di input oppure il simbolo ϵ. Questa distinzione viene fatta per evitare confusione.

ϵ-NFA per riconoscere un numero decimale

$$E = (\{q_0, q_1, ..., q_5\}, \{., +, -, 1, ..., 9\}, \delta, q_0, \{q_5\})$$

	ϵ	+,-	.	0,1,...,9
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tabella 6: Tabella di transizione per un numero decimale

5.3 Epsilon chiusure

Un ϵ-chiusura è un cammino fatto solo di transizioni ϵ. Formalmente tale stato si scrive ENCLOSE(q) = insieme di stati.

5.4 Transizioni estese di epsilon-NFA

Grazie alle ϵ-chiusure possiamo definire cosa significa accettare un input.
Supponiamo $E = (Q, \Sigma, \delta, q_0, F)$ un σ-NFA, $\hat{\delta}$(q,w) è la funzione di transizione estesa le cui etichette concatenate descrivono la stringa w.
BASE $\hat{\delta}$(q,w) = ENCLOSE(q), se l'etichetta è ϵ posso seguire solo cammini ϵ, definizione di ENCLOSE.
INDUZIONE Supponiamo w abbia forma xa, dove a è l'ultimo simbolo, che non può essere ϵ perché non appartiene a Σ:

1. Poniamo $\hat{\delta}$(q,x) = $\{p_1, p_2, ..., p_k\}$ in questo modo tutti i cammini p_i sono tutti gli stati raggiungibili da q a x. Questi stati possono terminare con ϵ oppure contenere altre ϵ transizioni

2. Sia $\bigcup_{i=1}^{k} \delta(p_i, a)$ l'insieme $\{r_1, r_2, ..., r_m\}$, ovvero tutte le transizioni da a a x.

3. Infine $\hat{\delta}$(q,w) = $\bigcup_{j=1}^{m} ENCLOSE(r_j)$, questo chiude gli archi rimasti dopo a

Forma contratta

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

Il linguaggio accettato è L(E) = $\{w | \hat{\delta}(q_0, w) \cap F \neq 0\}$

5.5 Da epsilon-NFA a DFA

Dato un ϵ-NFA possiamo costruire un equivalente DFA per sottoinsiemi. Sia $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ un ϵ-NFA il suo equivalente DFA è

$$D = (Q_D, \Sigma, \delta_D, q_0, F_D)$$

ovvero:

1. Q_D è l'insieme di sottoinsiemi Q_E. Ogni stato accessibile in D è un sottoinsieme ϵ-chiuso di Q_E, in termini formali S$\subseteq$ Q_E tale che S = ENCLOSE(S).

2. q_D=ENCLOSE(q_0)

3. F_D contiene almeno uno stato accettante in E.
   $F_D = \{S | S$ è in Q_D e $S \cap F_E \neq 0\}$

4. $\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} (\bigcup_{t \in \delta(q,a)} ENCLOSE(t))$

Teorema Un linguaggio è linguaggio L è accetto da un ϵ-NFA se è solo se è accettato da un DFA.

6 Espressioni regolari

Le espressioni regolari definiscono gli stessi linguaggi definiti dai vari automi: *linguaggi regolari*. A differenza degli automi, le espressioni regolari descrivono linguaggi in maniera dichiarativa. Per questo motivo le espressioni regolari sono molto diffuse, per esempio nel commando unix *grep* oppure negli analizzatori lessicali.

6.1 Operatori lessicali

L'espressione lessicale 01*+10* denota il linguaggio 0 seguito da qualsiasi numero di 1 oppure 1 seguito da qualsiasi numero di 0.
Per poter definire le operazioni sulle regex (sinonimo di espressione regolare) dobbiamo definire tali operazioni sui linguaggi che esse rappresentano:

1. *Unione* di due linguaggi L ed M, L∪M, indica tutte le stringhe che appartengono ad L e ad M oppure a entrambi.

2. *Concatenazione* di due linguaggi L ed M è l'insieme di stringhe formate dalla concatenazione di una qualsiasi stringa L con una qualsiasi stringa M. Tale operazione è indicata così: L·M oppure semplicemente LM. Per L={001,10,111} e M={ϵ,001} LM={001,10,111,001001, 10001, 111001}

3. *Chiusura*(o *star* o chiusura di Kleene) di un linguaggio L, indicata come L*, rappresenta l'insieme delle stringhe che si possono formare tramite concatenazione e ripetizione di qualsiasi stringa in L. Nel caso L={0,1} L* rappresenta l'alfabeto binario, qualsiasi combo di 0 e 1. Nel caso L={0, 11} L* rappresenta qualsiasi stringa che abbia una o più coppie di 1, NB 011 è valido ma come 01111, mentre 101 non è valido, non abbiamo né la stringa 10 né la stringa 01. Formalmente L* è l'unione infinita $\bigcup_{i>0} L_i$ dove $L^0 = \{\epsilon\}$, $L^1 = L$, $L^i = LL...L$.

6.2 Proprietà regex

- $L \cup M = M \cup L$ L'unione è commutativa
- $(L \cup M) \cup M = L \cup (M \cup L)$ L'unione è associativa
- $(LM)M = L(ML)$ La concatenazione è associativa (LM $\neq$ ML)
- $\emptyset \cup L = L \cup \emptyset = L$
- $\{\epsilon\} \cup L = L \cup \{\epsilon\} = L$
- $\emptyset L = L\emptyset = \emptyset$
- $L(M \cup N) = LM \cup LN$
- $(M \cup N)L = ML \cup NL$
- $L \cup L = L$
- $\emptyset^* = \{\epsilon\}, \{\epsilon\}^* = \{\epsilon\}$
- $L^+ = LL^* = L^*L,\ L^* = L^* \cup \{\epsilon\}$

6.3 Costruzione di regex

Servono modi per raggruppare le espressioni regolari, in questo caso vengono usati operatori algebrici comuni. Di seguito verranno definite regex lecite E con il loro corrispondente linguaggio L(E).

BASE

1. le costanti ϵ e $\emptyset$ sono regex, rispettivamente del linguaggio $\{\epsilon\}$ e $\{\emptyset\}$, in altri termini L(ϵ) =$\{\ \epsilon\}$ e L($\emptyset$) = $\{\emptyset\}$.
2. Se a è un simbolo allora **a** è una regex che denota il linguaggio {a}, ovvero L(**a**) = {a}. (si usa il grassetto per distinguere simboli da regex)
3. Una lettera maiuscola qualsiasi, di solito L, viene usata per indicare un linguaggio arbitrario

INDUZIONE

1. Data E ed L regex, allora $E + L$ è una regex che indica l'unione dei due linguaggi L(E) e L(L), in altre parole L(E+F) = L(E) $\cup$ L(F)
2. Date E e F due regex, EF indica la concatenazione tra i due linguaggio L(E) e L(F), in altri termini L(EF) = L(E)L(F).
3. Data E una regex, E* indica la chiusura del linguaggio L(E), in altri termini L(E*) = (L(E))*
4. Data E una regex, allora anche (E) è una regex valida che appartiene sempre al linguaggio E, in termini formali L((E)) = L(E)

Esempio di regex

Si crei una regex che descriva una linguaggio che è fatto di 0 e 1 alternati.
Intuitivamente si potrebbe provare **01***, che è errato, questo indica tutte le stringhe che hanno uno 0 e un numero arbitrario di 1. (**01**)* è corretto, però indica per forza un linguaggio di 01 alternati, quindi 101010 non sarebbe valido
Uniamo regex per descrivere il caso: (**10***) 10 alternato, **0(10)*** 10 con 0 all'inizio, **1(01)*** 01 con 1 all'inizio, in conclusione

$$\mathbf{(01)^*+(10)^*+0(10)^*+1(01)^*}$$

Un modo più contratto sarebbe quello di aggiungere un 1 facoltativo all'inizio e uno 0 facoltativo alla fine

$$\mathbf{(\epsilon+1)(01)^*)(\epsilon+0)}$$

6.4 Precedenza degli operatori

1. Star ha la precedenza massima
2. concatenazione
3. unione

Naturalmente si possono usare parentesi per decidere il proprio ordine e inoltre è consigliato farlo anche se non fosse necessario per rendere più chiara l'espressione.

7 Automa a stati finite e regex

Abbiamo visto che le regex e gli automi a stati finiti possono descrive gli stessi linguaggi, va solo dimostrato che formalmente.
Dobbiamo dimostrare che:

1. Ogni linguaggio definito da un automa è definito anche da una regex, useremo un DFA per comodità

2. Ogni linguaggio definito da una regex è definita da un automa, useremo un ϵ-NFA per comodità

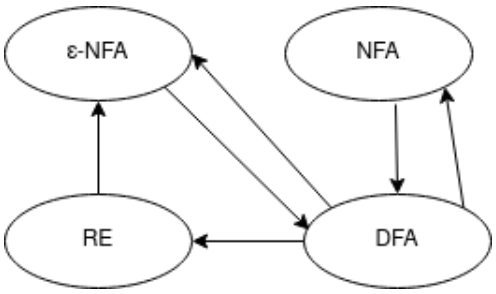


Figura 8: Conversioni

7.1 Da DFA a regex

Teorema
Se L = L(A) per un DFA A, allora esiste una regex R tale che L = L(R).
Il procedimento formale e matematico è formato dal espansione di ogni singolo stato tramite la formula:

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

In parole povere sto calcolando l'espressione regolare da uno stato j a uno stato i k volte, una per ogni stato.
Questo procedimento è molto lungo, perché l'espressione va effettuata per ogni transizione, unita e poi ridotta.
Tenendo però a mente questa formalità è possibile usare un metodo più gestibile, ovvero *l'eliminazione per stati*.

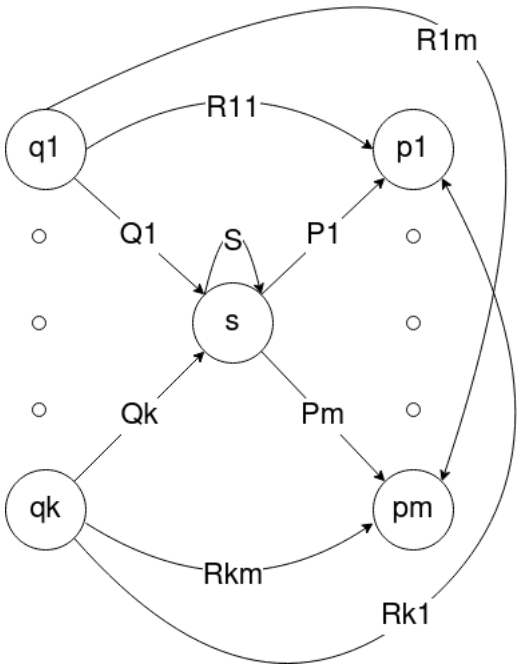


Figura 9: Eliminazione per stati

- s è lo stato generico che sta per essere eliminato
- $q_1, q_2, ..., q_k$ sono i k stati precedenti a s
- Q_i sono tutte le transizioni precedenti
- $p_1, p_2, ..., p_k$ sono i k stati successi a s
- P_i sono tutte le transizioni successive
- R_{ij} sono tutte le transizioni tramite regex, bisogna definirne una per ogni direzione ij ma se non dovesse esistere basterà scrivere $\emptyset$

A questo punto possiamo iniziare a costruire l'espressione regolare a partire dall'automa.

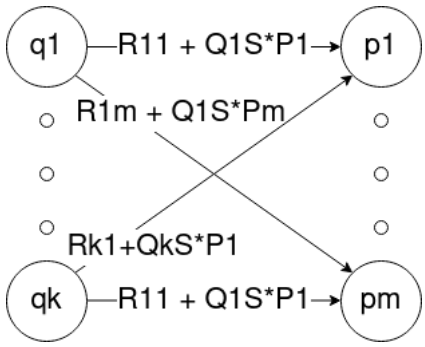


Figura 10: Eliminazione di s

1. Bisogna eliminare tutti gli stati intermedi ad eccezione di q_0.
2. Se $q_0 \neq q_1$ allora questo stato può essere espresso come E_q = (R + SU*T)*SU*, un cammino generico illustrato in figura.

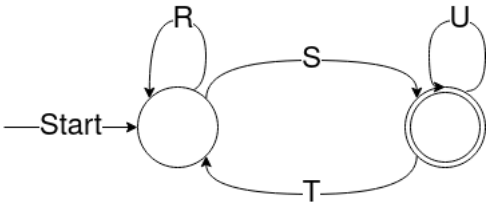


Figura 11: Automa generico 2 stati

3. Se q_0 è accettante allora la regex è data da R*

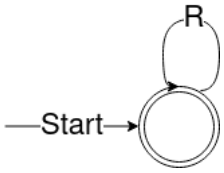


Figura 12: Automa generico 1 stato

Dato un NFA come segue.

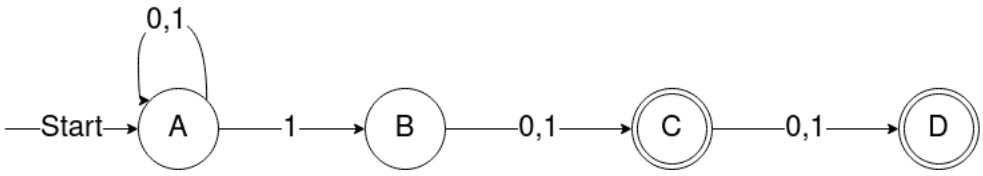


Figura 13: NFA esempio

Esprimiamo le sue funzioni di transizioni come regex

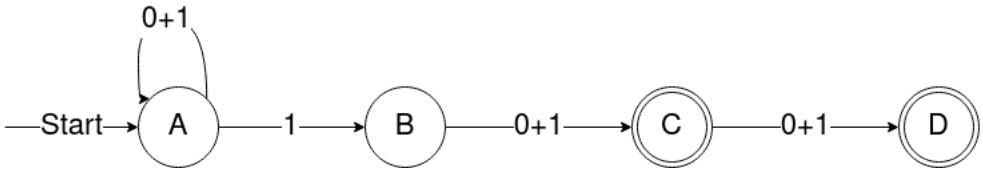


Figura 14: NFA con archi regex

Il primo stato che rimuoviamo è B, applicando la fromula. $R_{11} + Q_1 S^* P_1$, in questo caso risulta $\emptyset + 1\emptyset^*(0+1)$, che si può ridurre in $1(0+1)$.
NB $\emptyset^*$ equivale a ϵ, non annulla le regex, mentre $\emptyset$ sì

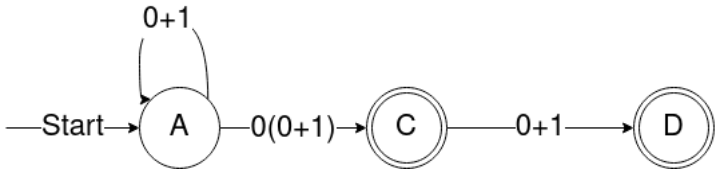


Figura 15: B rimosso

Eliminiamo C

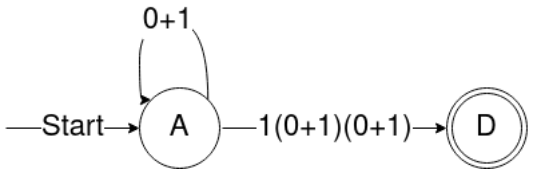


Figura 16: C rimosso

Ora possiamo applicare (R + SU*T)*SU*, quindi

- R=(0+1)
- S=1(0+1)(0+1)
- T=$\emptyset$
- U=$\emptyset$.

$$((0+1)+1(0+1)(0+1)\emptyset^*\emptyset)^*1(0+1)(0+1)\emptyset$$

Possiamo semplificare U* perché è equivalente a ϵ e possiamo eliminare SU*T perché è T è $\emptyset$.

$$(0+1)^*1(0+1)(0+1)$$

Questo è lo stato accettante D, è necessario calcolare lo stato accettante C. Ripartendo dalla fig.15 applichiamo di nuovo E_Q ottenendo (0+1)*1(0+1). L'espressione finale è data dalla **somma** delle 2 espressioni.

$$(0+1)*1(0+1)+(0+1)^*1(0+1)(0+1)$$

7.2 Da regex a automi

Teorema Per ogni rex R possiamo costruire un ϵ-NFA A tale che L(R)=L(A). Questo si dimostra per induzione strutturale, prendendo come base gli automi ϵ, $\emptyset$ e a.

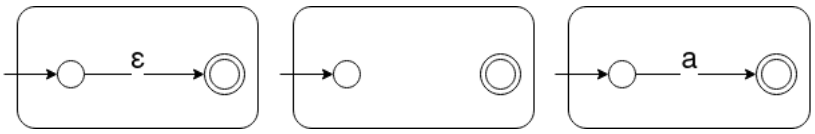


Figura 17: Stati base

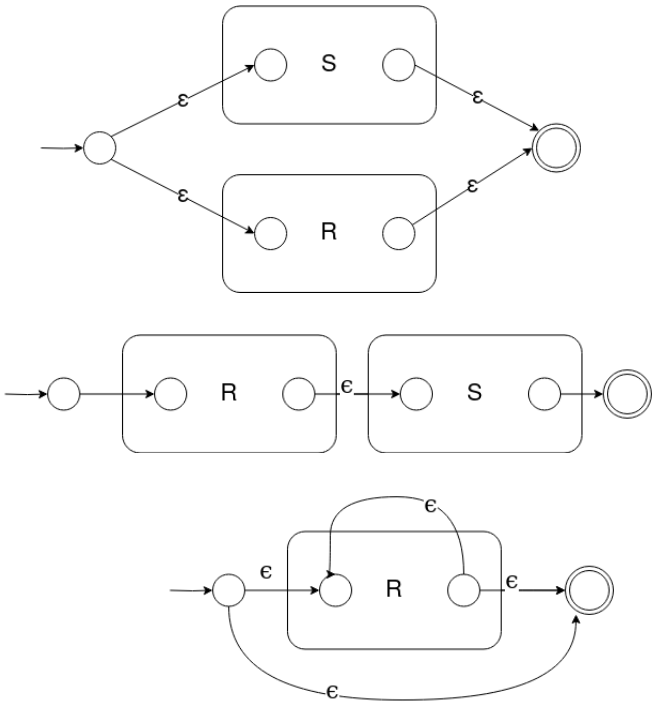


Figura 18: R+S, RS e R*

R+S significa che viene percorso 1 dei 2 espressioni. RS significa che una volta percorso R, quello diventa lo stato iniziale di S. R* va in loop su se stesso.

Usando i blocchi precedenti convertiamo (0+1)*1(0+1)

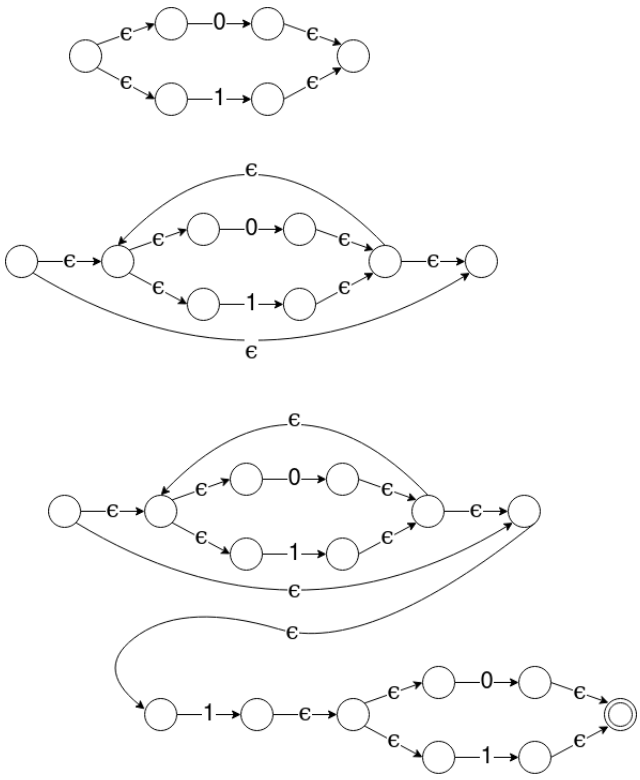


Figura 19: R+S, RS e R*

8 Proprietà dei linguaggi regolari

- *Pumping Lemma*: Ogni linguaggio regolare soddisfa il pumping di lemma
- *Proprietà di chiusura*: Possibilità di costruire un nuovo automa a partire da altri automi, seguendo specifiche operazioni
- *Proprietà di decisione*: Analisi di automi, come l'equivalenza
- *Tecniche di minimizzazione*: Possiamo ridurre un automa

8.1 Pumping Lemma

Un linguaggio non è detto che sia regolare.
Immaginiamo di avere un linguaggio $L_{01} = \{0^n\ 1^n | n \geq 1\}$. Questo è un linguaggio che accetta una stringa con tanti 1 quanti 0. Perché questo linguaggio possa essere un DFA deve avere un numero finito di stati, diciamo k. Quindi dopo $k+1$ simboli, $\epsilon, 0, 00, ..., 0^k$ ci troviamo in un qualche stato. Poiché gli stati sono limitati esistono 2 strade diverse per cui ci troviamo nello stesso stato, chiamiamoli 0^j e 0^i.
Ora immaginiamo dallo stato j di iniziare a leggere 1, l'automa deve fermarsi quando ha letto j quantità di 1, ma non può farlo perché non ricorda lo stato, potrebbe finire dopo i quantità di 1, L_{01} non è regolare.
Teorema Sia L un linguaggio regolare, allora esiste una costante n tale che, per ogni stringa w in L dove $|w| \geq n$ possiamo scomporre w in 3 stringhe $w = xyz$ tale che:

1. $y \neq \epsilon$
2. $|x, y| \leq n$
3. per ogni $k \geq 0$ anche xy^kz è in L

Ovvero c'è una stringa non vuota replicabile da qualche parte, senza uscire dal linguaggio.
Dimostrazione Supponiamo che L sia regolare. Allora L=L(A) e supponiamo che A abbia n stati. Ora consideriamo una stringa w dove $w = a_1, a_2, ..., a_m$ $m \geq n$ e ogni a_i è un simbolo di input. Definiamo la sua funzione $\delta(a_1, a_2, ..., a_n)$ che descrivere tutte le p_i transizioni, e $q_0 = p_0$.

Per il principio della piccionata tutti gli stati non possono essere distinti, quindi esistono due stati p_i e p_j dove $0 \leq i \leq j \leq n$ tale che $p_i = p_j$. Possiamo scomporre w in w=xyz:

1. $x = a_1, a_2, ..., a_i$
2. $y = a_{i+1}, a_{i+2}, ..., a_j$
3. $x = a_{j+1}, a_{j+2}, ..., a_m$

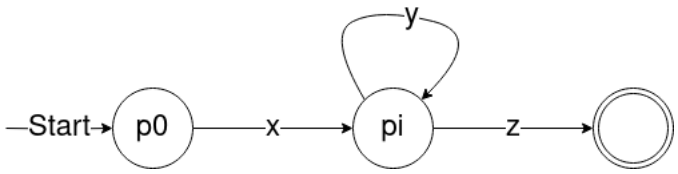


Figura 20: A un certo punto i stati si ripetono

Se k=0 siamo allo stato accettante, se $k \geq 0$ allora dobbiamo necessariamente fare dei loop, perché l'input è xy^kz.

8.2 Chiusura dei linguaggi regolari

Sia L e M due linguaggi regolari allora i seguenti sono a loro volta linguaggi regolari.

- *Unione*: $L \cup M$
- *Intersezione*: $L \cap M$
- *Complemento*: N
- *Differenza*: $L \backslash M$
- *Inversione*: $LR = \{wR : w \in L\}$
- *Chiusura*: L^*
- *Concatenazione*: $L \cdot M$

Teorema Sia L e M linguaggi regolari allora anche L ∪ M è un linguaggio regolare.
Dimostrazione L ed M sono linguaggi descritti dalle espressioni regolari S ed R, quindi L=L(S) e M=L(R) quindi L∪M=L(R+S).

Teorema Se L è un linguaggio regolare sull'alfabeto Σ allora anche $\overline{L} = \Sigma^* - L$.
Dimostrazione Sia L=L(A) per un DFA A=$(Q, \Sigma, \delta, q_0, F)$, allora $\overline{L}$=L(B) dove B è il DFA $(Q, \Sigma, \delta, q_0, Q - F)$, quindi B ha gli stati accentanti opposti a quelli di A. In questo caso l'unico modo per cui w è in L(B) se e solo se $\delta(q_0, w)$ è in $Q - F$, ovvero **non** è in L(A).

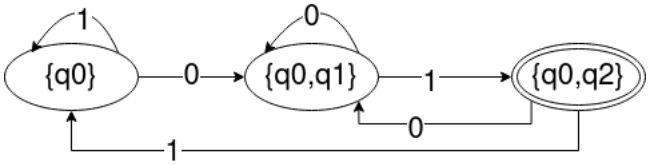


Figura 21: Diagramma di A

Il diagramma di B risulta opposto

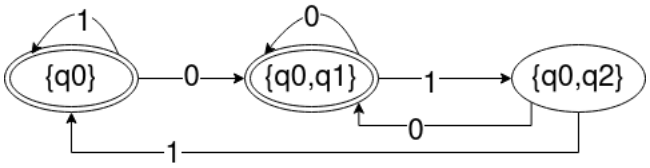


Figura 22: Diagramma di B

Teorema 4.8. Se L e M sono regolari, allora anche $L \cap M$ è regolare.

Dimostrazione Le 3 operazioni booleane sono interdipendenti, quindi possiamo usare le due precedenti per dimostrare l'Intersezione

$$L \cap M = \overline{\overline{L} \cup \overline{M}}$$

Dimostrazione alternativa Sia L il linguaggio $L = (Q_L, \Sigma, \delta_L, q_0, F_L)$ e M il linguaggio $M = (Q_M, \Sigma, \delta_M, q_0, F_M)$ assumiamo per semplicità che siano dei DFA Se A_L passa da p a s e A_M passa da q a t (sono tutti stati) allora $A_{L \cap M}$ passera da $(p, s)a(q, t)$ quando legge una stringa a.
Formalmente il linguaggio risultante dall'intersezione diventa

$$A = (Q_M \times Q_L, \Sigma, \delta_{M \cap L}, (q_M, q_L), F_L \times F_M)$$

Si può dimostrare che $\hat{\delta}((q_M, q_L), w) = (\hat{\delta}(q_M, w), \hat{\delta}(q_L, w))$, questo perché A accetta solo quando entrambi gli stati sono accettanti, quindi accetta per forza anche l'intersezione.

8.2.1 Chiusura rispetto alla differenza

Teorema Sia L e M dei linguaggi regolari allora anche $L - M$ è un linguaggio regolare

Dimostrazione $L - M = L \cap \overline{M}$, ma sappiamo che $\overline{M}$ è regolare e l'intersezione di 2 linguaggi è regolae, quindi $L - M$ è anche esso regolare.

8.2.2 Inversione

L'inversione di una strina $a_1, a_2, ..., a_n$ è la stringa $a_n, ..., a_2, a_1$ questa stringa la denotiamo come w^R e notiamo che $\epsilon^R = \epsilon$.
Un linguaggio L^R inverso di L presenta tutte le stringe al suo interno inverse. Se L è un linguaggio regolare allora lo è anche L^R.

8.3 Proprietà di decisione

Questi non scontanti che vanno affrontate:

- Un linguaggio descritto è vuoto?
- Una stringa appartiene al linguaggio?
- Due linguaggi equivalenti?

8.3.1 Verificare se un linguaggio è vuoto

Se il linguaggio A è rappresentato da un automa finito posso attraversare tutti i nodi, se trovo uno stato accettante allora il linguaggio **non** è vuoto. Quest'operazione richiede O(n^2) perché è un semplice attraversamento di grafo.
Se iniziamo da un espressione regolare, possiamo trasformarla in un ϵ-NFA e poi effettuare i cammini a costo O(n).
È possibile anche determinare se il linguaggio è vuoto in base alla regex direttamente. Se il linguaggio non ha $\emptyset$ sicuramente non può essere vuoto, altrimenti posso determinarlo ricorsivamente seguendo le regole algebriche delle regex.
Di seguito elenco i casi:

- $R = \emptyset$. $L(\emptyset)$ è vuoto
- $R = \epsilon$. $L(\epsilon)$ **non** è vuoto
- $R = a$ (qualsiasi stringa a) L(a) non è vuoto
- $R = R_1 + R_2$. $L(R)$ è vuoto se sia $L(R_1)$ che $L(R_2)$ siano vuoti
- $R = R_1R_2$. $L(R)$ è vuoto se $L(R_1)$ o $L(R_2)$ è vuoto
- $R = R_1*$. $L(R)$ non è mai vuoto, al massimo è ϵ
- $R = R_1$. $L(R)$ è vuoto solo se $L(R_1)$ è vuoto, sono lo stesso linguaggio

8.3.2 Appartenenza a un linguaggio

Per controllare se una qualsiasi stringa $w \in L(A)$ per un DFA è sufficiente simulare w su A, se $|w| = n$ il tempo risulta O(n).

Se A è un NFA e ha s stati, allora O(ns^2), vale lo stesso epr ϵ-NFA.

Se L=L(R) è una regex, la converto in ϵ-NFA ovvero O(ns^2)

8.3.3 Equivalenza e minimizzazione di automi

Dobbiamo esplorare la possibilità di dire che 2 DFA sono equivalenti, un modo per farlo è minimizzarli, se sono equivalenti basterà cambiare etichette finché non coincidono.
Iniziamo definendo cosa rende equivalenti 2 stati p e q. Data una stringa w, p e q sono equivalenti se $\hat{\delta}(q, w)$ e $\hat{\delta}(p, w)$ sono entrambi accentanti oppure non accentanti.
Nel caso in cui uno sia accentante e l'altro no, allora si dicono distinti. NB. 2 stati equivalenti non ci dicono niente sulla stringa w e non ci dice se i due stati sono lo stesso.
Possiamo raggruppare le nostre distinzioni degli stati in una tabella, tramite l'algoritmo $riempit - tabella$.

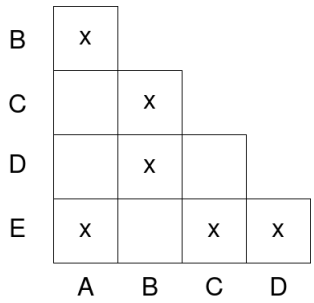


Figura 23: Algoritmo riempi tabella

Ogni x è uno stato distinguibile, un quadrato vuoto significo stati equivalenti.
Per testare se 2 linguaggi L e M sono equivalenti dobbiamo:

- Convertire L e M in DFA

- Costruire il DFA unione dei 2 linguaggi
- Se l'algoritmo dice che i 2 stati iniziali sono equivalenti allora L=M, altrimenti L≠M

Partendo da 2 DFA costruisco un DFA B che ha lo stato iniziale che contiene quello di A e lo stato accentante che contiene quello di A. Ogni altra funzione di un blocco deve essere equivalente.
L'algoritmo non può essere applicato a un NFA.

9 Grammatiche libere da contesto

Esempio informale

Definiamo un linguaggio delle palindrome. Una stringa è palindroma se si legge allo stesso modo in entrambi i versi, come *otto* oppure *madamimadam* (madame I'm Adam). Perciò w è palindroma se $w = w^R$.
Si può facilmente dimostrare che questo linguaggio non è regolare usando il pumping lemma. Scegliamo $w = 0^n10^n$, scomponiamo in $w = xyz$ tale che y sia fatto di vari 0 e scegliamo k=0, xz dovrebbe adesso appartenere a L_{pal} ma non è così, perché ho meno 0 a sinistra rispetto che a destra.
Posso definire le stringhe che appartengono a L_{pal} in maniera ricorsiva.

Base ϵ, 0 e 1 sono palindrome
Induzione Se w è palindroma allora $0w0$ e $1w1$ sono palindrome

Una **grammatica libera** è una notazione formale per esprimere linguaggi ricorsivamente. Una grammatica consiste in una o più serie di variabili che rappresentano classi di stringhe, ovvero linguaggi.
Ogni classe definisce come costruire le stringhe in ogni classe. Definiamo le classi del linguaggio palindroma:

1. $P \to \epsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

0 e 1 sono terminali, P è una variabile, P è anche la categoria iniziale, 1-5 sono produzioni.

9.1 Definzione formale di CFG

Un CFG è formato da 4 elementi:

1. Un insieme di simboli detti *terminali*
2. Un insieme di variabili, detti *non termali* oppure *categorie sintattiche*
3. Una variabile detto *simbolo iniziale*
4. Un insieme finito di *produzioni* o *regole* che definiscono il linguaggio ricorsivamente. Ogni produzione consiste in 3 parti
   (a) Una variabile che è definita parzialmente dalla produzione, *testa*
   (b) Il simbolo di produzione $\rightarrow$
   (c) Il *corpo* della produzione, ovvero la stringa o il terminale che la forma. Le stringhe vengono formate sostituendo le variabili.

In maniera contratta, CFG=(V,T,P,S) rispettivamente variabile, terminale, produzioni e simbolo iniziale.
Il linguaggio palindromo descritto come CFG è $G_{pal} = (\{P\}, \{0,1\}, A, P)$ A è l'insieme delle produzioni del linguaggio.

9.2 Derivazione in CFG

Possiamo definire le stringhe tramite concatenazione delle produzione, *inferenza ricorsiva*
Il secondo modo per *derivazione*, ovvero uso le produzioni fino a quanto ho solo simboli terminali. Noi studieremo la derivazione.
Sia $G = (V, T, P, S)$ una grammatica libera e sia αAB una stringa mista di terminali e variabili, dove A è variabile e $\alpha B \in (V \cup T)$ allora se G risulta chiara nel contesto, posso scrivere:

$$\alpha AB \underset{G}{\Longrightarrow} \alpha\gamma B$$

A è una derivazione di G, $A \Rightarrow \gamma$. Posso usare il simbolo * per denotare "zero o più passi"(chiusura transitiva).
Base $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \alpha$, vale per ogni stringa terminale o variabile, ovvero ogni stringa deriva se stessa
Induzione Se $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ e $\beta \underset{G}{\Longrightarrow} \gamma$ significa che $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \gamma$
Se la grammatica è chiara posso scrivere $\overset{*}{\Rightarrow}$

9.3 Derivazione a sinistra e a destra

È possibile arrivare a diverse conclusioni a seconda della scelta di quali produzioni fare, quindi per evitare di avere incertezze si può scegliere un verso di come derivare ogni volta.
Derivazione a destra: $\underset{rm}{\Longrightarrow}$
Derivazione a sinistra: $\underset{lm}{\Longrightarrow}$

9.4 Linguaggio di una grammatica

Se $G(V, T, P, S)$ è una CFG allora il suo linguaggio è:

$$L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Longrightarrow}} w\}$$

Ovvero l'insieme delle stringhe T^* derivabili da w

9.5 Alberi sintattici

Data una grammatica $G = (V, T, S, P)$, gli alberi sintattici di G soddisfano i seguenti requisiti:

1. Ogni nodo interno è una variabile V
2. Ogni foglia è un variabili, terminale o ϵ. Quando è ϵ deve essere l'unico figlio
3. Se un nodo A con i figli $X_1, X_2, ..., X_k$ allora $A \to X_1, X_2, ..., X_k$ è una produzione in P. X può essere ϵ solo nel caso in qui $A \to \epsilon$

Nella grammatica seguente:

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$

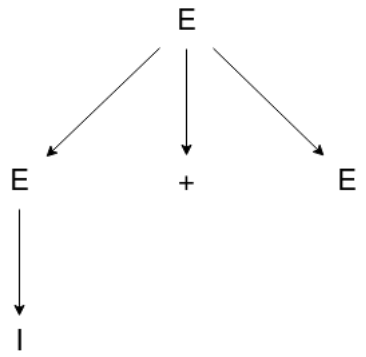


Figura 24: Parsing tree

Questo albero sintattico rappresenta la produzione $E \overset{*}{\Rightarrow} I + E$

9.5.1 Prodotto di un albero sintattico

Se concateniamo le foglie di un albero otteniamo una stringa, ovvero il *prodotto*. Inoltre la radice deve essere il simbolo iniziale e ogni foglia è $\emptyset$ oppure un terminale.

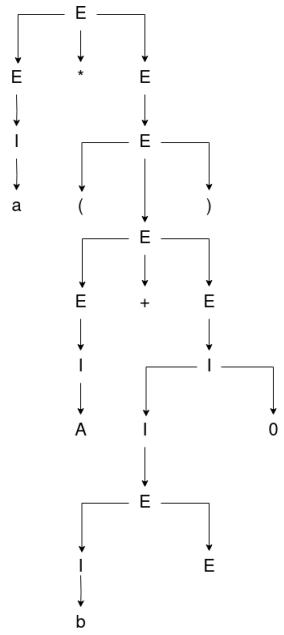


Figura 25: Produzione

Il risultato della produzione è $a * (a + b00)$

9.5.2 Inferenza, derivazione e alberi sintattici

Sia $G = (V, T, P, S)$ una CFG e $A \in V$ allora i seguenti sono equivalenti:

1. $A \overset{*}{\Rightarrow} w$
2. $A \underset{lm}{\overset{*}{\Longrightarrow}} w$
3. $A \underset{rm}{\overset{*}{\Longrightarrow}} w$
4. C'è un albero sintattico G con radice A e prodotto w

Costruiamo il prodotto della fig. 25 per derivazione sinistra:

- $E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow}$
- $I * E \underset{lm}{\Longrightarrow}$

- $a * E \underset{lm}{\Longrightarrow}$
- $a * (E) \underset{lm}{\Longrightarrow}$
- $a * (E + E) \underset{lm}{\Longrightarrow}$
- $a * (I + E) \underset{lm}{\Longrightarrow}$
- $a * (a + E) \underset{lm}{\Longrightarrow}$
- $a * (a + I) \underset{lm}{\Longrightarrow}$
- $a * (a + I0) \underset{lm}{\Longrightarrow}$
- $a * (a + I00) \underset{lm}{\Longrightarrow}$
- $a * (a + b00) \underset{lm}{\Longrightarrow}$

9.6 Ambiguità in grammatiche e linguaggi

Nella grammatica:

- $E \to I$
- $E \to E + E$
- $E \to E * E$
- $E \to (E)$

L'espressione $E+E*E$ ha due possibili derivazioni: $E \Rightarrow E+E \Rightarrow E+E*E$ oppure $E \Rightarrow E * E \Rightarrow E + E * E$, da qui i due alberi sintattici

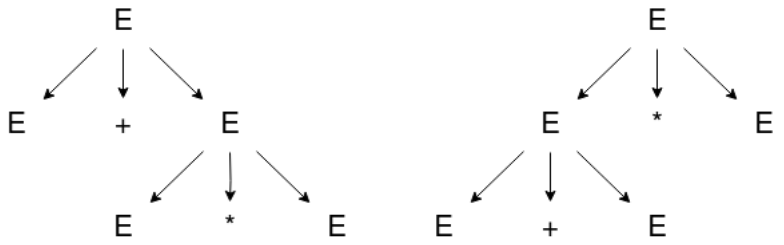


Figura 26: Semplice automa

Queste sono 2 operazioni diverse, usiamo i valori $1+2*3$, dal primo albero troviamo $1 + (2 * 3) = 7$ mentre dal secondo albero troviamo $(1 + 2) * 3 = 9$.
Definizione Sia $G = (V, T, P, S)$ una CFG. Diciamo che G è ambigua se esiste almeno una stringa T* che ha più di un albero sintattico.

9.6.1 Rimuovere ambiguità

È possibile, in alcuni casi, rimuovere l'ambiguità. Non esiste però un modo sistematico e alcuni CFL hanno solo CFG ambigue.
Studiamo la grammatica

$$E \to I \mid E + E \mid E * E \mid (E)$$

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

Dobbiamo decidere:

1. Chi ha precedenza tra + e *
2. Come raggruppare un operatore

Una soluzione è l'introduzione di una gerarchia di variabili:

1. espressioni E: composizione di uno o più termini T tramite +
2. termini T: composizione di uno o più fattori F tramite *
3. fattori F:
   (a) identificatori I
   (b) espressioni E racchiuse tra parentesi

Ogni più è costretto a essere una T, ogni T può generare E solo chiuse nelle parentesi, questo significa che * ha precedenza rispetto al +.
La seguente grammatica risulta quindi non ambigua:

1. $E \to T|E+T$
2. $E \to F|T*F$
3. $F \to I|(E)$
4. $E \to a|b|Ia|Ib|I0|I1$

9.6.2 Ambiguità inerente

Un CFL è inerentemente ambiguo se tutte le grammatiche per L sono ambigue.

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

Il linguaggio è inerentemente ambiguo.

10 Automi a pila

Un automa a pila (PDA - pushdown automaton) è un ϵ-NFA con una pila che è la sua memoria.
Durante le transizione:

1. Consuma un simbolo di input o esegue una transizione ϵ
2. Va in un nuovo stato o rimane dov'è
3. Rimpiazza la cima della pila con una stringa (lo stack è cambiato), con ϵ (c'è stato un pop) oppure con lo stesso simbolo (nessun cambiamento)

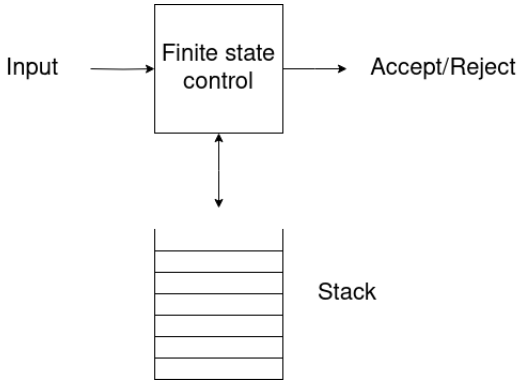


Figura 27: PDA, automa con pila

Esempio: Consideriamo il linguaggio palindromo:

$$L_{wwr} = \{ww^R : w \in \{0,1\}^*\}$$

Una PDA per L_{wwr} ha 3 stati:

1. In q_0 si presume l'input non sia esaurito, quindi leggo 1 alla volta tutti i simboli di input e li accumulo nello stack.
2. Scommettiamo di aver trovato la sequenza corretta, quindi passiamo in q_1 ma continuando a leggere input
3. Confronta la cima della pila con il simbolo in q_1, se sono uguali consumiamo l'elemento, altrimenti il ramo muore
4. Se lo stack è vuoto passiamo in q_2

10.1 Definizione formale PDA

Un PDA è una tupla di 7:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

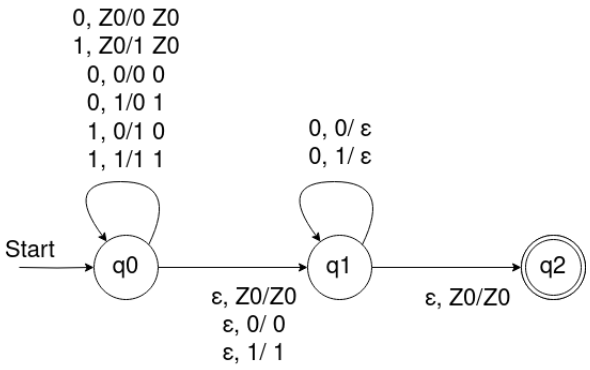
dove

- Q è un insiemi di stati finiti
- Σ è un alfabeto finito di input
- Γ è un alfabeto finito di pila
- (δ è una funzione di transizione da $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ a sottoinsieme di $Q \times \Gamma^*$ - definizione del prof, poco chiara)
  δ è un funzione di transizione che prende 3 input (q, a, X) dove:
  - q è uno stato in Q
  - a è un simbolo in Σ oppure la ϵ, NB ϵ non fa parte dell'input
  - X è un simbolo in Γ, ovvero è un simbolo **sullo stack**

  L'output di σ è una coppia (p, γ), dove p è uno stato e γ è una nuova stringa che rimpiazza X sullo stack. Se $\gamma = \epsilon$ X viene eliminato, se $\gamma = X$ non cambia nulla e se $\gamma = YZ$ allora Y sostituisce X e Z viene aggiunto allo stack
- q_0 stato iniziale
- $Z_0 \in \Gamma$ simbolo iniziale per la pila
- $F \subseteq Q$ è l'insieme degli stati di accettazione

Prendiamo come esempio il PDA di L_{wwr}:

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{Z_0, 0, 1\}, \delta, q_0, Z_0, \{q_2\}$$



dove δ è definita dalle seguenti regole:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$ una di queste regole è applicata all'inizio, l'input viene inserito lasciando Z_0 come indice del fondo.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$,
   $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, $\delta(q_0, 1, 1) = \{(q_0, 11)\}$
   leggo un input, lo salvo nello stack e continuo a leggere

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$,
   $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$
   $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$
   passiamo da q_0 a q_1 lasciando intantto lo stack

4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$,
   $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$
   rimaniamo su q_1 ma eliminiamo un membro dallo stack, se sono uguali

5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$ se non ho niente nello stack, passo alla soluzione accettante

10.2 Descrizioni istantanee

Un PDA passa da una configurazione all'altra consumando un simbolo in input oppure la cima della stack.
Possiamo rappresentare una configurazione tramite *descrizioni istantanee* (ID) che sono un tupla (q, w, γ) dove: q è lo stato, w è l'input rimanente e γ è il contenuto della pila.
Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA, allora $\forall w \in \Sigma^*, \beta \in \Gamma^* : (p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha, \beta)$

Il simbolo $\vdash$ significa "deduzione logica", mentre definiamo $\overset{*}{\vdash}$ come la chiusura transitiva di $\vdash$

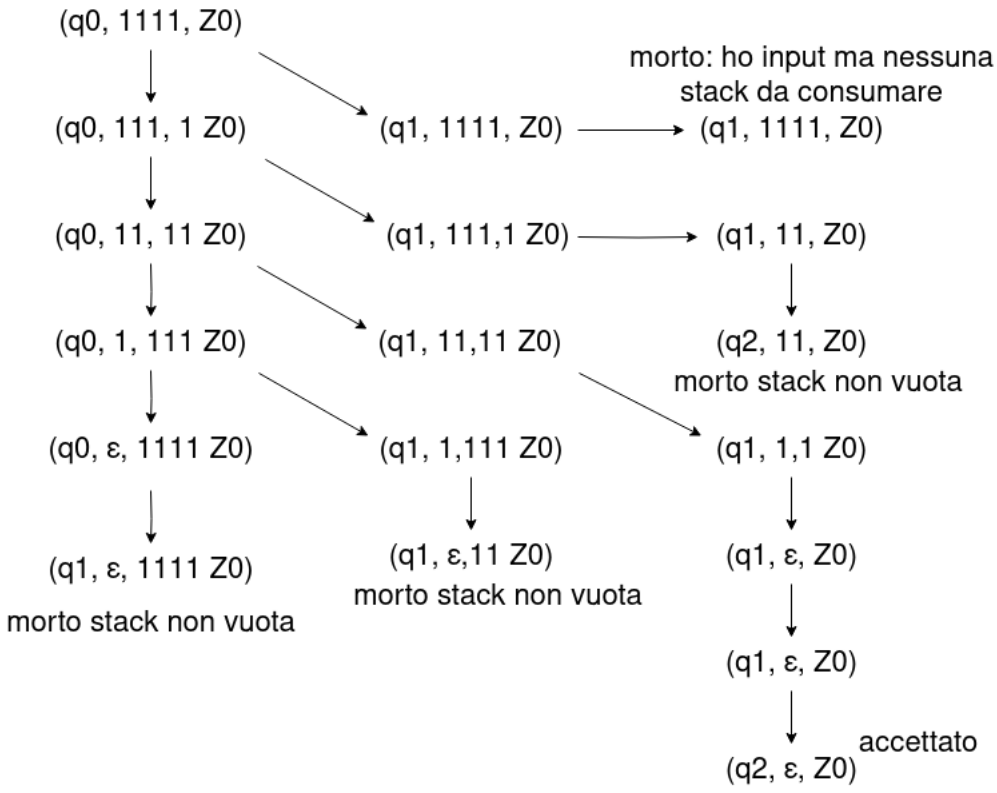


Figura 28: Diagramma PDA

10.3 Accettazione per stato finale

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA, il *linguaggio accettato* da P per stato finale è:

$$L(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha, q \in F)\}$$

In altre parole una volta che w è in uno stato accettante il contenuto della pila è irrilevante

10.4 Accettazione per pila vuota

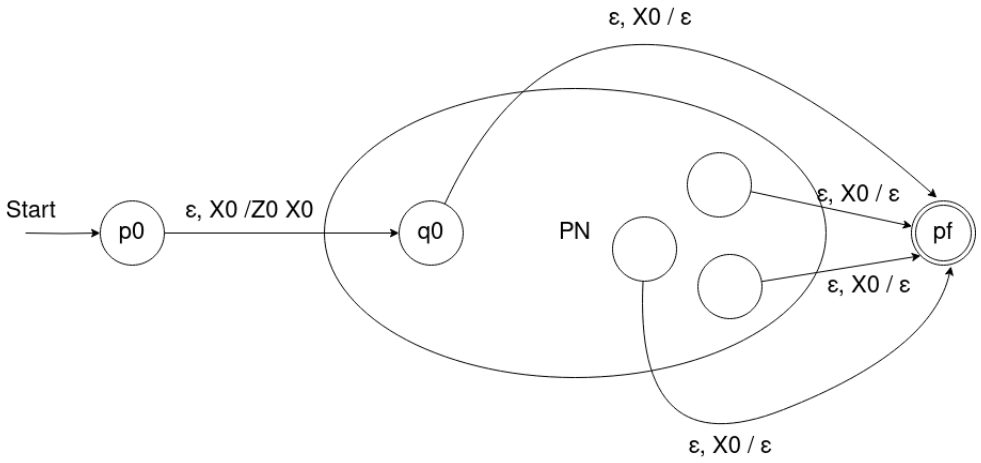
Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA, il *linguaggio accettato* da P per pila vuota è:

$$N(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

NB q è un generico stato, dunque $N(P)$ è degli input w che svuotano la stack

10.5 Da stack vuota a stato finale

Dimostriamo ora che per pila vuota o per stato finale sono linguaggi equivalenti.
Teorema: Se $L = N(P_N)$ per un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ allora $\exists$ PDA P_F tale che $L = N(P_F)$.
A parole: L'idea è avere un simbolo nuovo X_0 che specifica quando arrivano allo stato finale sia P_F che P_N. p_0 ci server per inserire il primo simbolo nello stack ed andare in q_0. Andiamo in p_F quando P_N svuota la stack.



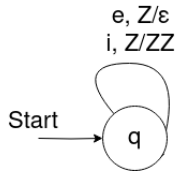
Dimostrazione Sia:

$$P_F = \{Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\}\}$$

dove δ è definita come:

1. $\delta(p_0, \epsilon, X_0) = \{(q_0, Z_0, X_0)\}$ transazione spontanea da P_F a P_N
2. $\forall q \in Q, a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma : \delta_F(q, a, Y) = \delta_N(q, a, Y)$
3. $\delta_F(q, a, Y)$ contiene (p_j, ϵ) per ogni q in Q

Consideriamo un automa *if else* in C, dobbiamo leggere tanti if quanti else quindi usiamo Z per contare la differenza tra i 2 simboli.



Leggendo i inseriamo una Z, leggendo e rimuoviamo una Z.Formalmente:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

dove δ:

1. $\delta(q, i, Z) = \{(q, ZZ)\}$ leggo i aggiungo Z
2. $\delta(q, e, Z) = \{(q, \epsilon)\}$ leggo e rimuovo Z

A partire da P_N costruiamo P_F:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

dove δ:

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$ simulo lettura primo simbolo, X_0 va in fondo allo stack
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$ leggo i aggiungo Z
3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$ leggo e rimuovo z
4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$ P_F accetta quando P_N si svuota

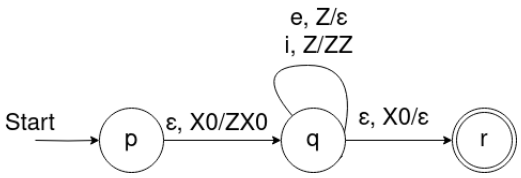


Figura 29: Diagramma P_F

10.6 Da stato finale a pila vuota

Facciamo il percorso inverso $P_F \to P_N$. Si aggiunge un ϵ-transizione da ogni stato accettante di P_F a un nuovo stato p. Quando siamo in p, consumo lo stack senza leggere input.
Per evitare di svuotare lo stack per stringhe non valide uso X_0 come indicatore di fondo. Il nuovo stato p_0 serve solo ad arrivare allo stato iniziale q_0 e mettere X_0 in fondo allo stack.
Teorema: Se $L = N(P_F)$ per un PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ allora $\exists$ PDA P_N tale che $L = N(P_N)$.
Dimostrazione:

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

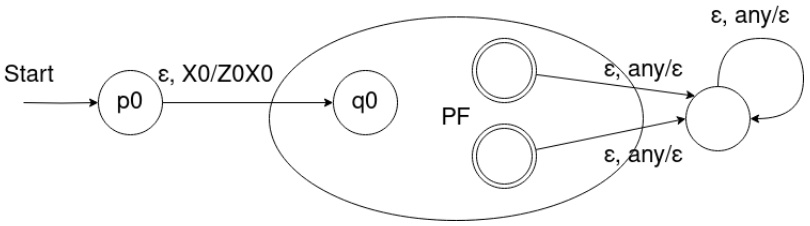
dove δ_N è:

1. $\delta_N(p, \epsilon, X_0) = \{(q, ZX_0)\}$ simulo lettura primo simbolo, X_0 va in fondo allo stack

2. $\forall$ q in Q, a in Σ e ogni Y in Γ: $\delta_N(q, a, Y)$ contiene tutte le coppie in $\delta_F(q, a, Y)$. Ovvero P_N simula P_F.

3. $\forall$ q in F e ogni Y in Γ: $\delta(q, a, Y)$ contiene (p, ϵ). Ogni volta che P_F accetta P_N può svuotare lo stack

4. $\forall$ Y in Γ: $\delta(q, a, Y) = \{p, \epsilon\}$ quando arrivo a p, quindi P_F ha accettato, P_N svuota lo stack senza leggere input



10.7 Equivalenza tra PDA e CFG

Un linguaggio è generato da una CFG se e solo se è accettato da un PDA per pila vuota se e solo se è accettato da un PDA per stato finale

10.8 Da CFG a PDA

Data una CFG costruiamo una PDA che simula la derivazione a sinistra. Ogni espressione non terminale si può scrivere come $xA\alpha$, dove A è la variabile più a sinistra, x sono gli simboli terminali a sinistra di A e α sono le variabili alla destra di A.
Chiamiamo $A\alpha$ la coda della forma/espressione. Una coda di terminali è da considerarsi ϵ. Dobbiamo simulare un PDA, dove accettiamo una stringa terminale w. La coda di $xA\alpha$ compare sullo stack con A in cima, x è quello che consumiamo per aggiungere una stack, $w = xy$.
Quindi data la transazione $(q, y, A\alpha)$, che rappresenta $xA\alpha$, l'automa sceglie di espandere $A \to \beta$. β va in cima allo stack entrando nella ID $(q, y, \beta\alpha)$, il PDA ha un unico stato q.
Non è detto però che β sia terminale, e potrebbe avere dei terminali che lo precedono, quindi dobbiamo eliminare ogni terminale all'inizio di $\beta\alpha$. Confronto i terminali con i successi input per verificare correttezza, altrimenti il processo muore.
Nel caso in cui tutto vada bene, lo stack è vuoto e abbiamo la w corretta.
Formalmente: Sia $G = (V, T, Q, S)$ una CFG, costruiamo il PDA P che accetta $L(G)$ per stack vuoto:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

dove δ è definita come segue

$$\delta(q, \epsilon, A) = \{(q, \beta) : A \to \beta \in Q\}$$

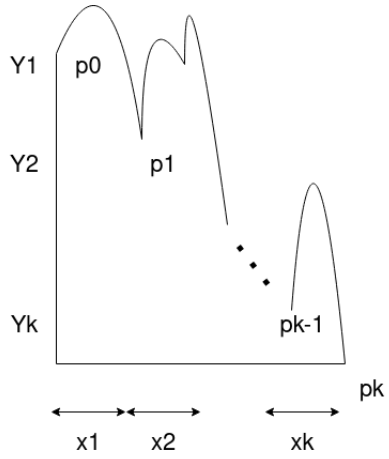
per ogni terminale a, $\delta(q, a, a) = \{(q, \epsilon)\}$
Esempio:
Considero la grammatica $S \to \epsilon|SS|iS|iSe$. Il PDA corrispondente è:

$$P = (\{q\}, \{i, e\}, \{S, i, e\}, \delta, q, S)$$

dove $\delta(q, \epsilon, S) = \{(q, \epsilon), (q, SS), (q, iS), (q, iSe)\}, \delta(q, i, i) = \{(q, \epsilon)\}$
$\delta(q, e, e) = \{(q, \epsilon)\}$.

10.9 Da PDA a CFG

Per ogni PDA P possiamo costruire un CFG G il cui linguaggio coincide con quello accettato dallo stack vuoto di P. Dobbiamo prendere atto del momento più importante di una PDA, ovvero quando viene fatto il pop di un elemento dello stack, perché è cambiato lo stato. Nel nostro caso server tenere traccia dello stato passato. Ogni stato che cambia "scendo" nello stack.



Nel grafico si può notare che ogni eliminazione $Y_1, Y_2, ..., Y_k$ coincide con la lettura di un input x. Non importa quanti passaggi siano stati fatti, qua viene visto lo step finale, quando Y esce.
Inoltre viene visto anche il passaggio di stato, che anche esso coincide con l'eliminazione.
Per costruire una CFG a partire dal PDA usiamo variabile che rappresentano un "evento" con due parti:

1. l'eliminazione definitiva dallo stack di X
2. il passaggio da p a q, dopo che scambio X con ϵ sullo stack

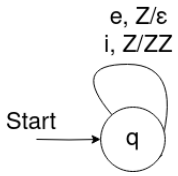
Questa variabile la rappresentiamo come $[pXq]$.
Formalmente: Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ un PDA.
Definiamo $G = (V, \Sigma, R, S)$ con:

- $V = \{[pXq] : \{p, q\} \subseteq Q, X \in \Gamma\} \cup \{S\}$ ovvero V contiene il simbolo iniziale S e tutti i simboli in Q e in Γ

- $R = \{S \to [q_0 Z_0 p] : p \in Q\} \cup$
  $\{qXr_k] \to a[rY_1r_1]...[r_{k-1}Y_kr_k] :$
  $a \in \Sigma \cup \{\epsilon\},$
  $\{r_1, ..., r_k\} \subseteq Q,$
  $(r, Y_1Y_2...Y_k) \in \delta(q, a, X)\}$

  - $[q_0 Z_0 p]$ genera tutte le stringhe w e passa dallo stato q_0 a p. Al termine dell'operazione lo stack P sarà svuotato
  - $\{qXr_k] \to a[rY_1r_1]...[r_{k-1}Y_kr_k]$: è una produzione che indica il passaggio q a r_k e il susseguirsi di passaggi per arrivarci. Al primo passaggio leggo l'input a e itero k volte finché non ho eliminato Y. a può essere ϵ
  - se k=0 allora $Y_1Y_2...Y_k = \epsilon$ e $r_k = r$

Esempio: Convertiamo



$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

dove

- $\delta(q, i, Z) = \{(q, ZZ)\}$
- $\delta(q, e, Z) = \{(q, \epsilon)\}$

in una grammatica

$$G = (V, \{i, e\}, R, S)$$

dove

- $V = \{[qZq], S\}$
- $R = \{S \to [qZq], [qZq] \to i[qZq[qZq], [qZq] \to e\}$

Per semplicità $[qZq] = A$ quindi $S \to A$ e $A = iAA|e$

Esempio: Convertiamo

$$P = \{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$$

dove δ è data da:

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$
2. $\delta(q, 1, X) = \{(q, XX)\}$
3. $\delta(q, 0, X) = \{(p, X)\}$
4. $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
5. $\delta(p, 1, X) = \{(p, \epsilon)\}$
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$

in una CFG.
Otteniamo $G = (V, \{0, 1\}, R, S$, dove

$$V = \{[qZ_0q], [pZ_0q], [qZ_0p], [pZ_0p], [qXq], [pXq], [qXp], [pXp], \}$$

e le produzioni in R

$$S \to [qZ_0q][qZ_0p]$$

Dalla transizione (1) $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$ si ha:
$[qZ_0q] \to 1[qXq][qZ_0q]$
$[qZ_0q] \to 1[qXp][pZ_0q]$
$[qZ_0p] \to 1[qXq][qZ_0p]$
$[qZ_0p] \to 1[qXp][pZ_0p]$

Dalla transizione (2) $\delta(q, 1, X) = \{(q, XX)\}$
$[qXq] \to 1[qXq][qXq]$
$[qXq] \to 1[qXp][pXq]$
$[qXp] \to 1[qXq][qXp]$
$[qXp] \to 1[qXp][pXp]$

Dalla transizione (3) $\delta(q, 0, X) = \{(p, X)\}$
$[qXq] \to 0[pXq]$
$[qXp] \to 0[pXp]$

Dalla transizione (4) $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
$[qXq] \to \epsilon$

Dalla transizione (5) $\delta(p, 1, X) = \{(p, \epsilon)\}$
$[pXp] \to 1$

Dalla transizione (6) $\delta(p, 0, Z_0) = \{(q, Z_0)\}$
$[pZ_0q] \to 0[qZ_0q]$
$[pZ_0p] \to 0[qZ_0p]$

11 PDA deterministici

I PDA deterministici sono quelli usati per i parser, sono un sotto caso utile per noi.
Un PDA deterministico non deve avere mosse alternative, se $\delta(q, a, X)$ ha più di una coppia sicuramente non è deterministico. Questo però non è sufficiente però, perché potresti avere due prodotti diversi e bisogna scegliere.
Definiamo quindi un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ deterministico se e solo se:

1. $\delta(q, a, X)$ ha al massimo un elemento $\forall q \in Q, a \in \Sigma, X \in \Gamma$
2. Se $\delta(q, a, X)$ non è vuoto per un $a \in \Sigma$ allora $\delta(q, \epsilon, X)$ deve essere vuoto

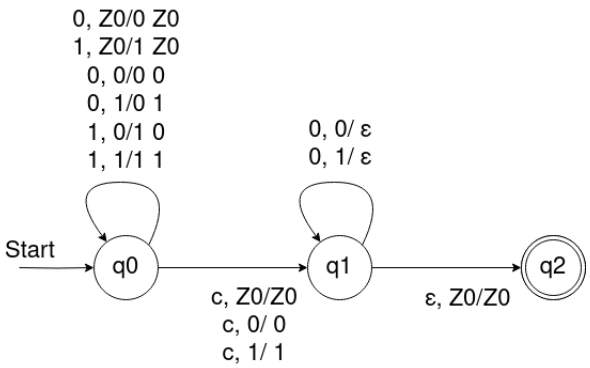


Figura 30: Esempio di PDA deterministico (DPDA)

11.1 DPA che accettano per stato finale

Regex $\subset$ L(DPDA) $\subset$ CFL.
Teorema Se L è regolare allora L=L(P) per un qualsiasi DPDA P.
Prova: Dato che L è regolare $\exists$ DFA A tale che L=L(A)

$$A = (Q, \Sigma, \delta_A, q_0, F)$$

definiamo il DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

dove

$$\sigma_P(q, a, Z_0) = \{(\delta_A(q, a), Z_0\}$$

$\forall\ p, q \in Q$ e $a \in \Sigma$
applichiamo un induzione su $|w|$

$$(q_0, w, Z_0) \overset{*}{\vdash} (p, \epsilon, Z_0) \iff \hat{\delta}(q_0, w) = p$$

11.2 DPDA che accettano per la pila vuota

Si usa la **proprietà del prefisso**: un linguaggio ha la proprietà del prefisso se NON esistono due stringhe uguali una che è il prefisso dell'altra.
L_{pal} ha la proprietà del prefisso
$\{0\}^*$ non ha la proprietà del prefisso
Teorema L è N(P) per un qualche DPDA P se e solo se L ha la proprietà del prefisso e L è $L(P')$ per qualche DPDA P'

11.3 DPDA e non ambiguità

L(DPDA) non per forza coincide con CFL non ambigue, viceversa invece è vero.

Theorem 1 *Se $L = N(P)$ per qualche DPDA P, allora L ha una grammatica non ambigua*

Dimostrazione stessa dimostrazione che fai da PDA a CFG, solo che parti con un DPDA

L'enunciato può essere rafforzato

Theorem 2 *Se $L = L(P)$ per qualche DPDA P, allora L ha una grammatica non ambigua*

Dimostrazione
Sia \$ un simbolo non presente in L, e sia $L' = L\$$. In altri termini L' sono le stringhe L seguite da \$. Dal Teorema. 1 L' ha la proprietà di prefisso. Dal Teorema. 2 esiste una grammatica G' che genera un linguaggio $N(P')$ che è L'.
Costruiamo quindi una grammatica G tale che $L(G) = L$, dobbiamo solo togliere \$ dalla fine delle stringhe, così che G' coincida con G. Introduciamo \$ in G

$$\$ \rightarrow \epsilon$$

Visto che $L(G') = L'$ consegue che $L(G) = L$. G non è ambigua.

12 Proprietà di CFG

Elenco delle proprietà:

- *Semplificazione* di una CFG. Una CFL ha una grammatica in forma speciale
- *Pumping Lemma* per CFG, simile alle regex
- *Proprietà di chiusura.*
- *Proprietà di decisione.* Verifichiamo l'appartenenza e l'essere vuoto

12.1 Forma normale di Chomsky

Ogni CFL (senza ϵ) è generato da una CFG dove tutte le forme sono

$$A \to BC, A \to a$$

dove A, B, C sono variabili, mentre a è un terminale. Questa è detta forma di Chomsky e per ottenerla dobbiamo pulire la grammatica:

- Eliminare i *simboli inutili*, ovvero ogni simbolo che non appartiene a $S \overset{*}{\Rightarrow} w$
- Eliminare le *produzioni* ϵ, dalla forma $A \to \epsilon$
- Eliminare le *produzioni unità* ovvero le produzioni $A \to B$

12.2 Eliminazione di simboli inutili

Un simbolo X è *utile* per una grammatica $G = (V, T, P, S)$ se esiste una derivazione

$$S \underset{G}{\overset{*}{\Rightarrow}} \alpha X \beta \underset{G}{\overset{*}{\Rightarrow}} w$$

per una stringa $w \in T^*$. Possiamo eliminare i simboli inutili senza cambiare il significato della grammatica. Un simbolo utile deve avere 2 caratteristiche:

1. X è un *generatore* se esiste una stringa w tale che $X \overset{*}{\Rightarrow} w$. Ogni terminale è un generatore di se stesso in 0 passi

2. X è *raggiungibile* se esiste una derivazione $S \stackrel{*}{\Rightarrow} \alpha X \beta$ per qualche $\{\alpha, \beta\} \subseteq (V \cup T)^*$

Un simbolo deve essere sia raggiungibile che generatore, eliminando prima i non generatori e poi i non raggiungibili abbiamo una grammatica solo di simboli utili.
Esempio: Consideriamo la grammatica

$$S \to AB|a$$

$$A \to b$$

A genera b, S genera A, B non è un simbolo generatore. Eliminando B:

$$S \to a$$

$$A \to b$$

Adesso solo a è raggiungibile quindi

$$S \to a$$

che è la nostra grammatica iniziale
NB se elimino prima i simboli non raggiungibili non cambia nulla, tutto è raggiungibile

$$S \to AB|a$$

$$A \to b$$

ora elimino B

$$S \to a$$

$$A \to b$$

e ho una grammatica con simboli inutili

12.3 Eliminazione delle produzioni epsilon

Se L è un CFL, allora L $\{\epsilon\}$ ha una grammatica priva di produzioni ϵ.
La variabile A è annullabile se $A \stackrel{*}{\Rightarrow} \epsilon$

Sia a annullabile, rimpiazzo

$$B \to \alpha A \beta$$

con

$$B \to \alpha\beta$$

indico con n(G) l'insieme dei simboli annullabili della grammatica $G = (V, T, P, S)$.
Esempio Sia la grammatica

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

Abbiamo $n(G) = \{A, B, S\}$, la prima regola diventa

$$S \to AB|A|B$$

la seconda regola diventa

$$A \to aAA|aA|aA|a$$

la terza regola diventa

$$B \to bBB|bB|bB|b$$

La nuova grammatica sarà:

$$S \to AB|A|BA \to aAA|aA|aB \to bBB|bB|b$$

12.4 Eliminazione produzioni unità

$$A \to B$$

è una produzione unità nel caso in cui A e B siano variabili. Le produzioni variabili sono eliminabili.
Data la grammatica
$E \to T|E + T$
$T \to F|T * F$
$F \to I|(E)$
$I \to a|b|Ia|Ib|I0|I1$
le produzioni unità sono $E \to T, T \to F, F \to I$
Considero la produzione $E \to T$ e la espando

$$E \to F, E \to T * F$$

poi espando $E \to F$

$$E \to I, (E), T * F$$

Infine espando $E \to I$

$$E \to a|b|Ia|Ib|I0|I1|(E)|T * F$$

lo stesso applico lo stesso procedimento con $T \to F$ e $F \to I$:

$$T \to a|b|Ia|Ib|I0|I1|(E)$$

$$F \to a|b|Ia|Ib|I0|I1$$

la nuova grammatica sarà
$E \to a|b|Ia|Ib|I0|I1|(E)|T * F|E + T$
$T \to a|b|Ia|Ib|I0|I1|(E)|T * F$
$F \to a|b|Ia|Ib|I0|I1|(E)$
$I \to a|b|Ia|Ib|I0|I1$
Questo procedimento non funziona nel caso in cui ci siano dei cicli. Se trovo un unità già espansa la rimuovo.

Esempio Si consideri la grammatica
$A \to B|a$
$B \to C|b$
$C \to A|c$
Espando $A \to B$
Da $A \to B$ ottengo

$$A \to C|b$$

Da $A \to C$ ottengo

$$A \to A|c|b$$

Da $A \to A$ ottengo

$$A \to B\ a|c|b$$

Qui mi fermo perché tornei a fare le stesse produzioni, mi rimane:

$$A \to a|c|b$$

eseguo per le altre 3 e la nuova grammatica risulta

$$A \to a|b|c$$

$$B \to a|b|c$$

$$C \to a|b|c$$

12.5 Sommario

Per pulire una grammatica bisogna:

1. Eliminare le produzioni ϵ
2. Eliminare le produzioni unità
3. Eliminare simboli inutili

in quest'ordine

12.6 Forma normale di Chomsky, CNF

Ogni CNF non vuoto, senza ϵ, ha una grammatica G di simboli inutili nella seguente forma $A \to BC$ dove $\{A, B, C\} \subseteq V$ oppure $A \to a$ dove $a \in T$ e $A \in V$.
Per arrivare a questa forma bisogna:

- Pulire la grammatica
- Modificare le produzioni con 2 o più simboli in modo che siano tutte variabili
- Ridurre le produzioni con più di 2 simboli in catene da 2 simboli

Quindi per ogni terminale a più lungo di 1 creo una nuova variabile $A \to a$.
Mentre per ogni regola

$$A \to B_1, B_2, ..., B_k$$

$k \geq 3$ creo nuove variabili $C_1, C_2, ..., C_{k-2}$

$$A \to B_1, C_1$$

$$C_1 \to B_2, C_2$$

$$...$$

$$C_{k-3} \to B_{k-2}, C_{k-2}$$

$$C_{k-2} \to B_{k-1}, B_k$$

Esempio Usiamo la grammatica
$E \to |E + F|T * F|(E)|b|Ia|Ib|I0|I1$

$T \to (E)|T * F|a|b|Ia|Ib|I0|I1$
$F \to (E)|a|b|Ia|Ib|I0|I1$
$I \to a|b|Ia|Ib|I0|I1$

applichiamo il passo 2
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

otteniamo
$E \to EPF|TMF|LER|b|IA|IB|IZ|IO$
$T \to LER|TMF|a|b|IA|IB|IB|IO$
$F \to LER|a|b|IA|IB|IZ|IO$
$I \to a|b|IA|IB|IZ|IO$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

Applichiamo il passo 3
$E \to EPT$ diventa $E \to EC_1, C_1 \to PT$

$E \to TMF, T \to TMF$ diventa $E \to TC_2, T \to TC_2, C_2 \to MF$

$E \to LER, T \to LER, F \to LER$ diventa $E \to LC_3, T \to TC_3, F \to LC_3, C_3 \to ER$

La grammatica finale diventa
$E \to EC_1|TC_2|LC_3|b|IA|IB|IZ|IO$
$T \to LC_3|TC_2|a|b|IA|IB|IB|IO$
$F \to LC_3|a|b|IA|IB|IZ|IO$
$I \to a|b|IA|IB|IZ|IO$
$C_1 \to PT$, $C_2 \to MF$, $C_3 \to ER$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

12.7 Pumping lemma per CFL

Pumping lemma per i linguaggi regolari: per una stringa sufficiente lunga è possibile causare un ciclo e creare un infinità di stringhe che appartengono al linguaggio
Pumping lemma per CFL: per una stringa sufficiente lunga è possibile

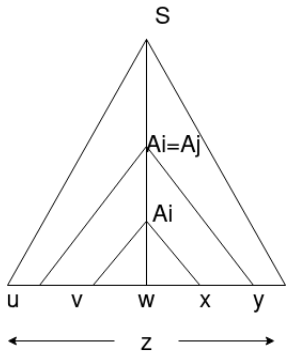
trovare due sottostringhe vicine che si possono iterare "in tandem". Posso iterare i volte per trovare nuove stringhe appartenenti al linguaggio.
Formalmente:
Sia L un CFL. Allora $\exists n \geq 1$ che soddisfa: ogni $z \in L : |z| \geq n$ è composto da 5 stringhe $z = uvwxz$ tali che:

1. $|vwx| \leq n$
2. $|vx| > 0$
3. $\forall i \geq 0, uv^iwx^iy \in L$

Dimostrazione:

- Si consideri una grammatica per $L\backslash\{\epsilon\}$ in CNF
- Assumiamo che la grammatica abbia m variabili, con $n = 2^m$
- Sia $z \in L$ una qualsiasi stringa tale che $|z| \geq 2^m$ allora ogni albero sintattico di z ha un cammino $\geq m + 1$
- Se tutti i cammini dell'albero hanno lunghezza $\leq m$ allora la stringa generata ha lunghezza $\leq 2^{m-1}$

Prendendo un cammino sufficientemente lungo, a partire da A fino a A_0 fino a A_k a un certo punto troverò due sotto alberi A_i, A_j con $i \neq j$ che sono uguali. Questo succede perché $k \geq m$, quindi avendo solo m variabili distinte è normale che si ripetano. Ora posso radicare l'albero A_j sotto A_i e la stringa finale non cambia, perché rispetta $z = uv^iwx^iy$ Notiamo che:



- l'albero in A_i ha altezza $\leq m+1$ quindi la stringa corrispondente ha lunghezza $\leq 2^m = n$ e quindi ($|vwx| \leq n$)
- v e x non possono essere vuote, perché la grammatica genera variabili non terminali perciò $|vx| > 0$
- posso ripetere l'albero A_i n volte e creo sempre una stringa valida ($uv^iwx^iy \in L$)

12.8 Applicazione Pumping lemma

Possiamo usarlo per dimostrare che un linguaggio non è libero.

Esempio: Si consideri $L = 0^m10^m10^m : m \geq 1$, dimostra che L non è CFL.
Dimostrazione: Assumiamo, per assurdo, che L sia CFL. Sia n la costante del Pumping lemma. Si consideri la strina $z = 0^n10^n10^n$. Si ha che $z \in L$ e $|z| \geq n$. Allora per il Pumping lemma, $z = uvwxy$ con $|uwx| \leq n$, $|vx| > 0$ e $uv^iwx^iy \in L$ per ogni $i \geq 0$. Consideriamo i seguenti casi:

- vx contiene almeno un 1, la catena può essere replicata all'infinito quindi sicuramente per $i >= 2$ $|uwx| \notin L$
- vx contiene almeno un 0, in questo caso non viene rispettata 0^n perché sia se vx forma un gruppo di 0, sia se ne forma 2, andrà in ogni caso a ridurre il numero di 0 totali

In entrambi i casi $uwy \notin L$

Esempio 2: Si consideri $L = 0^{k^2} : k > 1$, dimostra che L non è CFL.
Dimostrazione: Assumiamo, per assurdo, che L sia CFL. Sia n la costante del Pumping lemma. Si consideri la strina $z = 0^{n^2}$. Si ha che $z \in L$ e $|z| \geq n$. Allora per il Pumping lemma, $z = uvwxy$ con $|uwx| \leq n$, $|vx| > 0$ e $uv^iwx^iy \in L$ per ogni $i \geq 0$. Consideriamo il seguente caso:
$uv^2wx^2y \in L$ per Pumping Lemma
(Presumo che qua stiamo contando il numero di 0) $n^2 < |uv^wx^2y| \leq n^2+n < n^2+2n+1 = (n+1)^2$. Non esiste un quadrato perfetto tra n^2 e $(n+1)^2$, il numero di 0 non torna, quindi $uv^2wx^2y \notin L$.

12.9 Proprietà di chiusura dei CFL

Theorem 3 *I CFL sono chiusi rispetto ai seguenti operatori: unione, concatenazione e chiusura di Kleene (*) e chiusura positiva +*

Theorem: se L è CFL allora lo è anche L^R
Dimostrazione: Supponiamo che L sia generato da $G = (V, T, P, S)$. Costruiamo $G^R = (V, T, P^R, S)$ dove

$$P^R = \{A \to \alpha^R : A \to \alpha \in P\}$$

Si può dimostrare per induzione che $(L(G))^R = L(G^R)$

12.10 I CFL NON sono chiusi rispetto all'intersezione

Sia $L_1 = \{0^n 1^n 2^i : n \geq 1, i \geq i\}$. Allora L_1 è CFL con grammatica

$$S \to AB$$

$$A \to 0A1|01$$

$$B \to 2B|2$$

Inoltre $L_2 = \{0^i 1^n 2^n : n \geq 1, i \geq i\}$. Allora L_1 è CFL con grammatica

$$S \to AB$$

$$A \to 0A0|0$$

$$B \to 1B2|12$$

Invece, $L_1 \cap L_2 = \{0^n 1^n 2^n : n \geq 1\}$ non è CFL per pumping lemma.

12.11 Operazioni su linguaggi liberi e regoli

Theorem 4 *Se L è CFL e R è regolare, allora $L \cup R$ è CFL*

Prova: Sia L che accetta PDA

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

per stato finale, e sia R accettato dal DFA

$$A = (Q_A, \Sigma, \delta_A, q_A, F_a)$$

Costruiamo un PDA per $L \cup R$:

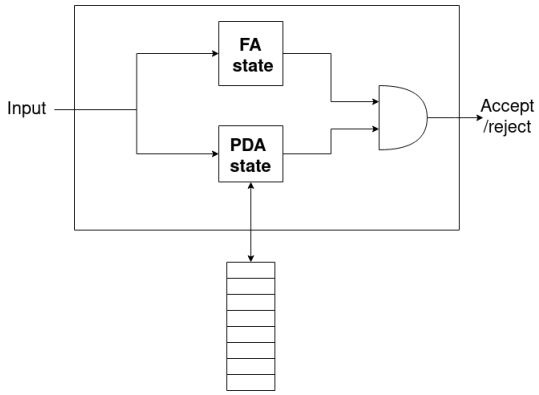


Figura 31: Semplice automa

Formalmente definiamo

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A$$

dove

$$\delta((q,p),a,X) = \{((r, \hat{\delta}, (p,a)), \gamma) : (r,\gamma) \in \delta_P(q,a,X)\}$$

Per induzione si può provare per induzione su $\overset{*}{\vdash}$ che

$$(q_P, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \gamma) \; in \; P$$

se e solo se

$$((q_P, q_A), w, Z_0) \overset{*}{\vdash} ((q, \hat{\delta}(q_A, w)), \epsilon, \gamma) \; in \; P'$$

Theorem 5 *Siano L, L_1, L_2 CFL e R regolare, Allora*

- *$L \backslash R$ è CFL*
- *$\overline{L}$ è per forza CFL*
- *$L_1 \backslash L_2$ non è per forza CFL*

Prova:

1. $\overline{R}$ è regolare, $R \cap \overline{R}$ è CFL e $L \cap \overline{R}$ è CFL e $L \cap \overline{R} = L \backslash R$

2. Se $\overline{L}$ fosse sempre CFL, seguirebbe che

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

è sempre una CFL

3. Notare che Σ^* è CFL; quindi se $L_1 \backslash L_2$ è CFL allora lo è anche $\Sigma^* \backslash K = \overline{L}$

12.12 Proprietà di decisione per CFL

Analizziamo i seguenti problemi decidibili:

- Verificare che $L(G) \neq \emptyset$ per una CFG G
- Verificare che $w \in L(G)$, per una stringa w ed una CFG G

E faremo un elenco di **problemi indecidibili**

12.12.1 Verificare se un CFL è vuoto

$L(G)$ è non vuoto se il simbolo iniziale S è generante.
Con un implementazione naive il tempo è $O(n^2)$, ottimizzando si passa $O(n)$

12.12.2 Appartenenza a un CFL

$$w \in L(G)?$$

Tecnica inefficiente
Supponiamo che G sia in CNF, e che la stringa w abbia lunghezza $|w| = n$, allora il suo albero binario ha al più $2n - 1$ nodi.
Genero tutti gli alberi sintattico di G e controllo se w è in uno di quelli, tempo esponenziale in n

12.12.3 Problemi indecidibili per CFL

Elenco di problemi indecidibili:

1. Una data CFG G è ambigua?
2. Un dato CFL L è inerentemente ambigua?

3. L'intersezione di 2 CFL è vuota?

4. 2 CFL sono uguali?

5. Un CFL è universale?

13 Analisi Lessicale e sintattica

There are 3 kinds of execution environment (language processor):

- Compilers: translate code in a forma that is executable by computers. More generally a compiler can *translate* code into an equivalent code in another language.

  Compilers also check for errors. If the compiled language is executable the users can add input and get output with the program.

  C and Java are compiled.

- Interpreters: Unlike a compiler the interpreter directly executes a program without producing an executable.

  Interpreters are slower than compilers, but can have better errors recognition since they execute the source statement by statement.

  Scheme, lisp and perl are interpreted

- Virtual machines: You can also combine the interpreter and the compiler. This is the case in Java where the code is begin compiled into *bytecode* and executed by the JVM.

  The advantage is you can 2 machines, one for compiling and the other for interpreting thus speeding the interpretation process.

  Some java compilers translate the code *just-in-time* (JIT) for the execution.

The whole process is the following: source -¿ preprocessor -¿ compiler -¿ assembler -¿ linker/loader.

13.1 The compiler

The operation of the compiler can be divided in 2 parts: *analysis* and *synthesis*.

The analysis breaks the program into pieces and enforces grammatical structure. If there are syntactic errors the user will get an error message. If everything goes well then the program is translated it into a intermediate representation. It also creates a *symbol table*, which will be used later.

The synthesis translates into the desired program, using the intermediate representation and the symbol table.

Here's the steps:

- Scanning (Lexical analysis)
- Parsing (Syntactic analysis)
- Type checking (Semantic analysis)
- Optimisation
- Code generation

13.1.1 Lexical analysis

Translates words into units.
Ex. if x==y ⇒ if, x, ==, y

13.1.2 Parsing

Once the lexical analysis is over, you can translate it into some kind of representation like *diagram sentences*(it's just a tree).

13.2 Building a lexer

Our input is this code snippet.

```
1    if(i==j):
2        z=0
3    else
4        z=1
```

The compiler actually sees it as such:

$$\backslash tif(i == j)\backslash n\backslash t\backslash tz = 0;\backslash n\backslash telse\backslash n\backslash t\backslash tz = 1;$$

We need to partition this string into substrings (lexemes) and classify them into role (tokens).
Each lexeme has a name and value $< tokenname, value >$
Therefore our input will be split. White spaces don't count and some special tokens will be grouped, line \t means newline.

It's not a good idea to write a lexer by hand it's: tedious, error prone and non-maintainable. There are *lexer generators* where you can define the lexemes and tokens and it will do the rest.
This way we can focus on what the lexer should do rather than how. The what is done with declarative programming, while the how with imperative programming.
First we build the lexer by hand in java, to get the feel of the code that will be hidden.

ID	sequence of alphanumeric that start with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

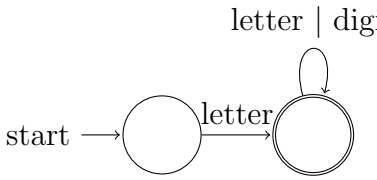
Tabella 7: Esempio di tabella

```
    c=nextChar();
    if (c == '=') { c=nextChar(); if (c == '=')
       {return EQUALS;}}
    if (c == '+') { return PLUS; }
    if (c == '*') { return TIMES; }
    if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) { c=NextChar(); }
    undoNextChar(c);
    return ID;
}
```

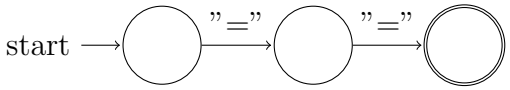
Imperative lexer

undoNextChar() determines if the next character is part of the lexeme or not. Each lexeme is usually partitioned at the maximum match, in order to avoid errors like "iffy" partition into "if" "fy" instead of the whole ID.
Each comparison is the *how*, but anything else is the *what*. We could abstract the how using automata. Each token is an automa or a regex, and the token reading is an automa responsible for the scanning.
Let's define the automata:

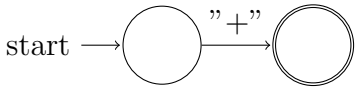
- ID: [a-zA-Z][a-zA-Z0-9]*, ovvero legge una lettera e poi legge lettere o numeri



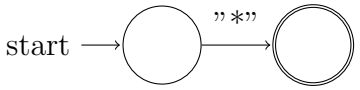
- EQUALS: ==



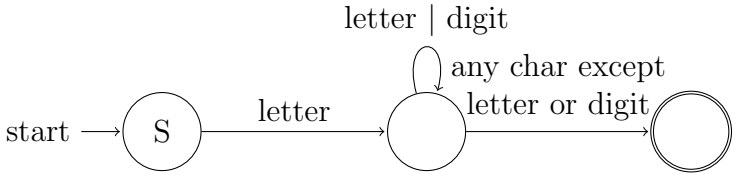
- PLUS: +



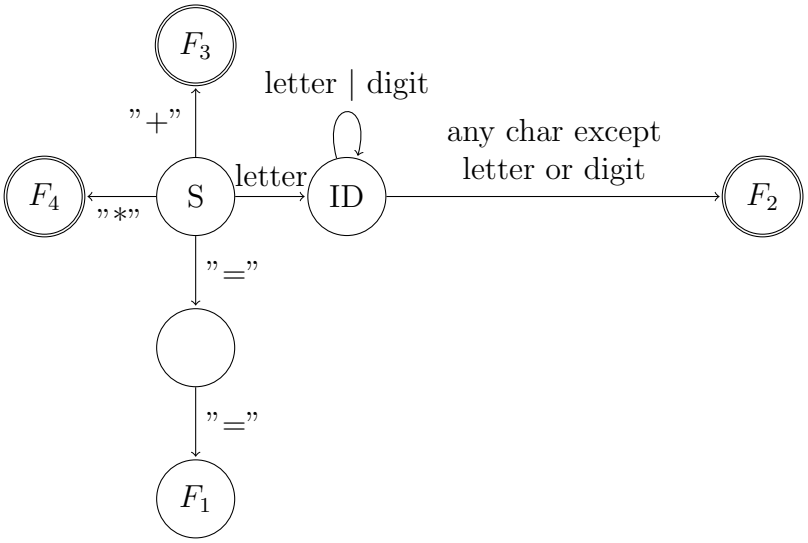
- TIMES: *



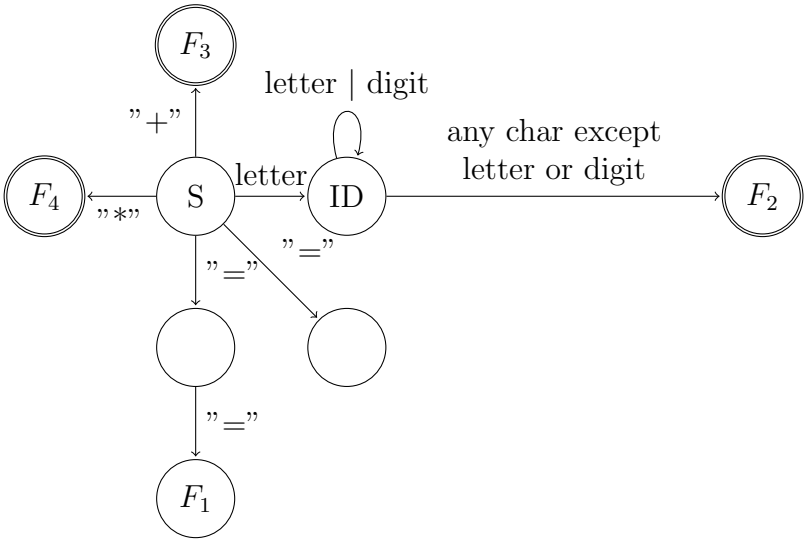
Now let's extend the FA to accept the whole input.



By the end of the operation we return a token and reset the automa to S.
If we put everything together into a single automa:



Let's add the ASSIGN



At this point we need to know if the automa is not deterministic, then we convert it to a deterministic one. The only missing piece is remembering the position of the input, if the lexer errors we can return information about where.

Some practical concerns:

- Ambiguity: if the input is "if" is it an ID or a keyword? We can solve

this by giving priority to the keywords, if has priority over ID.

- Discard white space: final state white space jumps to start
- Error inputs: discard illegal lexemes and return an error token, this token has the lowest priority

Automata are not the only way to build a lexer, we can also use regex. The advantage is that regex are way easier to represent textually, thus being a better specification. Regex also are very compact.
Let's define the regex for ID: [a-zA-Z]. ([a-zA-Z] | [0-9])*

- . is any character (prof says followed by???), middle priority
- * is 0 or more repetitions, highest priority
- | is the or operator, lowest priority
- () is the grouping operator

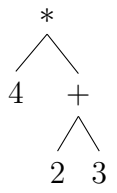
In ANTLR operands corresponds to FA edge labels, which are single char. [a-z] is written as 'a'..'z' in ANTLR. [0-9] is written as '0'..'9' in ANTLR.
Let's define the regex for Integer:

1. in english: ϵ, + or - followed by a digit followed by 0 or more digits
2. regex: (+ | - | ϵ).[0-9].[0-9]*
3. we can omit the .: (+ | - | ϵ)[0-9][0-9]*
4. we can omit the * using +: (+ | - | ϵ)[0-9]+

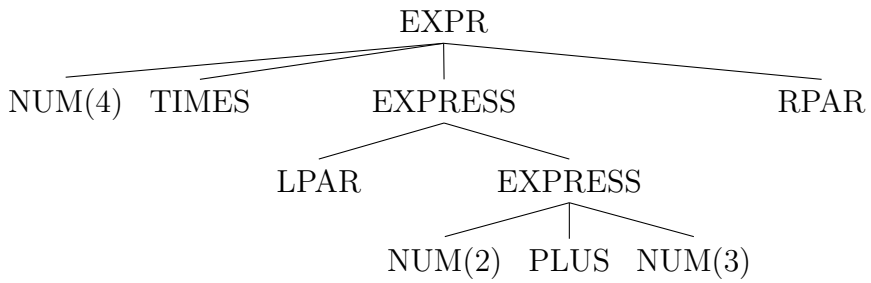
A DFA in practice is implemented as a 2D table T. One dimension is the state, the other is the input symbol. Every transition becomes T[i,a]=k. During execution I check the input the skip to the next state, very efficient. We could use flex to generate the lexer, the input is a set of regex and some C instruction. The output is C program that has a function yylex() which reads inputs and executes the C functions.

13.3 Building a parser

A parser checks the syntax of the program and builds a representation of the program.
After reading the tokens provided by the lexer, the parser generates a parse tree and then the AST. The intermediate step between tokens and AST is rarely explicit, but for out purpose it will be.
Let's parse this expression 4*(2+3).
Parser input: NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR
Parse output AST



Parse tree



Leaves are tokens.
The AST is a simplified version of the parse tree, it's more compact and easier to process. You can think of the parse tree as concrete syntax.
The parser therefore has 2 tasks: syntax checking and AST generation (from parse tree).
We can a parser by hand but, same as the lexer, it's hard, errore prone and there are parser generators. We will write one manually just to see why it's no fun.
The first problem is syntax checks: are parentheses balanced? We can't do this using the automa because (this is my guess) the language is non deterministic.
Define the implementation:

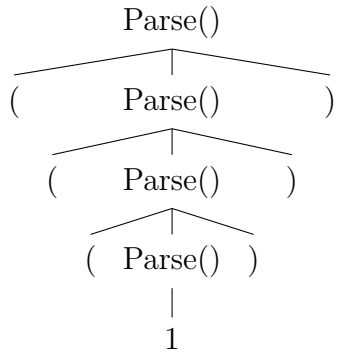
- let TOKEN be enums: NUM, LPAR, RPAR, PLUS, MINUS,TIMES, DIV

- in[] is a global with the inputs
- next is a global with the index of next input

```
1    void Parse() {
2        nextToken = in[next++];
3        if (nextToken == NUM) return;
4    }
5    if (nextToken != LPAR) print("syntax␣error");
6    Parse();
7    if (in[next++] != RPAR) print("syntax␣error");
```

Balanced parentheses parser

The output for (((1))) is:



Let's add subtraction:

```
1    void Parse() {
2        if ((nextToken = in[next++]) == NUM) {
3            if (in[next] == MINUS) { next++;
                 Parse(); }
4        } else if (nextToken == LPAR) {
5            Parse();
6            if (in[next++] != RPAR) print("syntax␣
                 error");
7            if (in[next] == MINUS) { next++;
                 Parse(); }
8        } else print("syntax␣error");
9    }
```

Balanced parentheses parser

Obviously this is tedious, we will therefore use a parser generator. To determine how our language will be generate we will use a CFG. Here's fore example a simple CFG for arithmetic expressions: E $\rightarrow$ n | id | (E) | E + E | E * E.
CFG will guarantee that the there is no syntax error, how? One way is the naive solution, derive any possibile string and check if you program is among them. Actually used in real word (damn).

14 Top-down parsing

Let's parse a string. Recursive descent parsers try each production in turns, always left most, until there is a mismatch or a match.

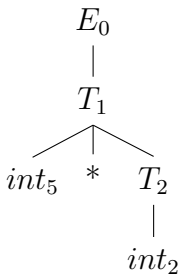
14.1 Recursive descent parser

Consider the following grammar:

- E → T + E | T
- T → (E) * T | (E) | int * T | int

Let's parse the string int_5 * int_2, starting with E.

1. $E_0 \rightarrow T_1 + E_1$
2. $T_1 \rightarrow (E_2) * T_2$, no token int_5
3. $T_1 \rightarrow (E_2)$, no token int_5
4. $T_1 \rightarrow int * T_2$, now we do all T_1 rules until we match. But after



Define function that checks the token for match of the following:

- a given token terminal
  bool term(TOKEN tok) { return in[next++] == tok; }
- a given production of S (the n^{th})
  bool Sn() { ... }
- any production of S:
  bool S() { ... }

Each function advance next.
Now let's get specific:

- E → T + E
  bool E_1() { return T() && term(PLUS) && E(); }
- E → T
  bool E_2() { return T(); }
- any production from E:
  bool E() { int save = next; return E_1() || (next = save,E_2()); }

Let's define non-terminal T functions:

- bool T_1() { return term(OPEN) && E() && term(CLOSE)
  && term(TIMES) && T(); }
- bool T_2() { return term(OPEN) && E() && term(CLOSE) }
- bool T_3() { return term(INT) && term(TIMES) && T() }
- bool T_4() { return term(INT) }
- bool T() {
  int save = next;
  return T_1()
  || (next = save,T_2()) || (next = save,T_3()) || (next = save,T_4()) }

To start the parser next have to point to the first token and then you can invoke E(). If we have a special char $ at the end of our token array, E() shall return true and the pointer to $.
This approach is easy to implement, but does not always work. Consider $S \to Sa$ this is a infinite loop, also called left-recursive grammar. We can fix the recursion.
Consider $S \to aS|b$, we can rewrite it as $S \to bS'$ and $S' \to aS'|\epsilon$.
In general

$$S \to S\alpha_1|...|S\alpha_n|\beta_1|...|\beta_m$$

should be rewritten as

$$S \to \beta_1 S'|...|\beta_m S'$$
$$S' \to \alpha_1 S'|...|\alpha_n S'|\epsilon$$

ANTLR4 does this automatically.

Another left recursive grammar is

$$S \to S\alpha|\delta$$

$$S \to S\beta$$

can be eliminated using this algorithm:
List all non terminals symbols $A_1, A_2, ..., A_n$, for i:=1 to n:

- replace all $A_i \to A_j\beta$ such as j¡i with $A_i \to \delta_1\beta|\delta_2\beta|...|\delta_k\beta)$ where $A_j \to \delta|\delta_2|...|\delta_k$
- eliminate direct left recursion in A_i
- after i-th steps all productions are $A_i \to A_k\beta$ where k¿1

This algorithm it's good enough for small languages but inefficient.

14.2 Predictive parsing

It would be nice if the parse "knew" which production to expand next.
Let's try and replace:

```
return E() || (next = save,E2())
```

with

```
switch( something ) {
    case L1: return E1();
    case L2: return E2();
    otherwise: print ("syntax␣error");
}
```

"something" is the lookahead, the next token.
This parser:

- uses the recursive descent but can predict which production to use
  the prediction are made by looking the next otkesn
  no backtracking

- accept LL(k) grammas
  L: left to right scan of inputs
  L: leftmost derivation
  k: k tokens of lookahead

- we will use LL(1) grammars
  ANTLR can do LL(*) grammars

LL(1) means that for each non-terminal and token there is at most 1 production that leads to success. This method can be specified as a 2D table:

- one dimensions are non-terminals

- one dimensions are tokens

- each cell contains a production

For example the grammar:
E $\rightarrow$ T + E | T
T $\rightarrow$ (E) * T | (E) | int * T | int
is impossibile to predict, because for T 2 productions start with int and (. E is unclear how to predict.
In general a language must be left-factore(ANTLR does this on it's own). The new fixed grammar is:
E $\rightarrow$ T + X
X $\rightarrow$ +E | ϵ
T $\rightarrow$ (E) * Y | int * Y
Y $\rightarrow$ *T | ϵ

14.2.1 LL(1) parser

For simplicity sake we will use a LL(1) *table* and *parse stack*.
Here's our left-factore grammar
E $\rightarrow$ TX
X $\rightarrow$ +E |ϵ
T $\rightarrow$ (E)Y | int Y
Y $\rightarrow$ *T |ϵ

The corresponding LL(1) parsing table: The empty spots are errors. Like

	int	*	+	(	)	$
T	int Y			(E)Y		
E	T X			TX		
X			+E		ϵ	ϵ
Y		*T	ϵ		ϵ	ϵ

Tabella 8: Esempio di tabella

[E,*], we can't know that it's a string if it starts with *.
We use a similar method to recursive descent. The difference is: we choose the production based on the next token.
There is a stack used to keep track on pending non terminals . We reject when there is an error and accept when we encounter end of input.
Here's the pseudo code of the algorithm:

```
1     // We start the stack with <S,\$>
2     repeat
3         case stack of
4         <X rest> : if T[X,*next] = Y1...Yn
5             then stack := <Yn...Y1 rest>
6             else error()
7         <t rest> : if t == *(next++)
8             then stack := <Yn...Y1 rest>
9             else error()
10    until stack == <>
```

LL(1) languages are defined by a parsing table. No table entry can be multiply defined. We want to generate parsing tables from CFG.
Consider the state $S\$ \stackrel{*}{\Rightarrow} \beta A \gamma$, b is the next token and we want to match the string $\beta b \delta$.
FIRST(α) is the set of terminals that are at the beginning of a string derived from α, α is a generic string. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$ then $\epsilon \in$ First(α).

FOLLOW(A), for non terminal A, is the set of terminals that can follow A in a derivation. There is such a derivation $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, a is in FOLLOW(A), \$ (the symbol that show the end of the stack) is also in FOLLOW(A).

Let's define formally first sets. First(X)=$\{b|X \stackrel{*}{\Rightarrow} b\beta \cup \{\epsilon|X \stackrel{*}{\Rightarrow} \epsilon\}$
Therefore:

- First(b) = b
- For all productions $X \rightarrow A_1...A_n$ with n≥0.
  Add First(A_1) - {ϵ} to First(X). Stop if $\epsilon \notin$ First(A_1).
  Add First(A_2) - {ϵ} to First(X). Stop if $\epsilon \notin$ First(A_2).
  ...
  Add First(A_n) - {ϵ} to First(X). Stop if $\epsilon \notin$ First(A_2).
- repeat step 2 until there is no more First

Here's the definition with n≥0:
First($X_1X_2...X_n$) = $\{b|X_1X_2...X_n \stackrel{*}{\Rightarrow} b\alpha\} \cup \{\epsilon|X_1X_2...X_n \stackrel{*}{\Rightarrow} \epsilon\}$. The steps are the same as above.

Let's define formally follow sets: Follow(X) = $\{b|S\$ \stackrel{*}{\Rightarrow} \beta X b\gamma\}$.

1. Add $ to Follow(S), if S is the start symbols
2. For all productions $Y \rightarrow \alpha X A_1...A_n$ with n≥1
   Add First($A_1...A_n$) -{ ϵ } to Follow(X)
   and for all productions $Y \rightarrow \beta X$ or $Y \rightarrow \alpha X \beta$ with $\epsilon \in First(\beta)$
   Add Follow(Y) to Follow(X)
3. repeat step 2 until there is no more Follow

Here's an example of the algorithm. Using the grammar:
E → TX
X → +E |ϵ
T → (E)Y | int Y
Y → *T |ϵ
First sets:

- Terminals:
  - First(+) = { + }
  - First(*) = { * }
  - First(() = { (}

  - First()) = {) }
  - First(int) = { int }

- Non-terminal:
  - First(T) = { (, int }
  - First(X) = { +, ϵ }
  - First(Y) = { *, ϵ }
  - First(E) = { (, int }

Follow sets:

- Follow(E) = { $,) }
- Follow(X) = { $,) }
- Follow(T) = { +, $,) }
- Follow(Y) = { +, $,) }

14.2.2 Constructing LL(1) parsing table

Construct a parsing table T for CFG G.
For each production $A \rightarrow \alpha$ in G, do the following:

- For each terminal b in First(α), T[A,a] = α
- If $\epsilon \in$ First(α), for each b *in* Follow(A) do, T[A,a] = α
- If $\epsilon \in$ First(α) and \$ $\in$ Follow(A), T[A,\$] = α

Notes on the algorithm:

- If any entry is multiply defined, then G is not LL(1):
  if G is ambiguous
  if G is left-recursive
  if G is not left-factored
  and in other cases
- Most programming grammas are NOT LL(1)
- There are tools that build LL(1) tables that are fully declarative

14.2.3 Ambiguity

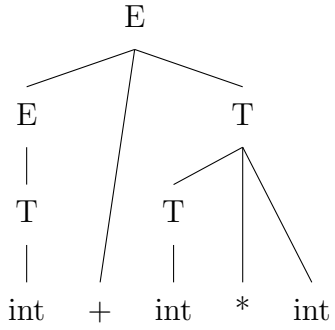
A grammar is ambiguous if it has more than one parse tree for any string. This is bad and is common with arithmetic expressions and if then else.
There are ways to deal with ambiguity, the most direct one is to rewrite the grammar, so that you can enforce symbol precedences.
There is no way to automatically correct ambiguous grammar.

15 Bottom-up parsing

Let's build a parser for this grammar:
E → E + T | E - T | T
T → T * int | int
This gramma is:

- non ambiguous
- left recursive
- not left factored

And our parser will not mind, because we will build it in revers.
E
E + T
T + T
T + T * int
T + int * int
int + int * int



Chaos bottom-up:

1. stare the input string s
2. find in s a right-hand side or r of a production N → r
3. reduce the found string r into non terminal N
4. if all string reduced to start non terminal we'r done
5. otherwise repeat from step 1

This wont work, we will get stuck eventually:
E + E * E (we are stuck here)
E + E * T
E + E * int
T + E * int
int + E * int
int + T * int
int + int * int If you are lucky you might get by, but let's define ourt luck by using non-determinism.
The new algorithm:

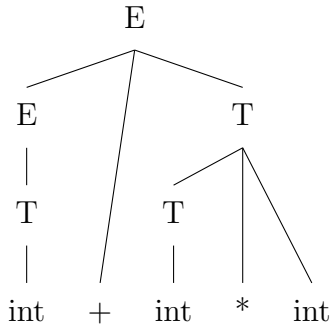
1. find all strings that can be reduced, let's say there are k of them
2. create k copies of the input, k instances of the parser
3. each instance reduces one of the k strings
4. if any instance reaches the start symbol we are done

This will work, but it's not efficient. We could do exponential copies. We could reduce the instances using LR parser. Basically you will try any combination, but keep only 1 instance.
Why won't this get me stuck? Because each tree follow a different derivation.
Also keep in mind, the LR parser builds the right most derivation but in revers. Reads from left to right, and deriving the right most symbol: hence LR parser.
Here's the action of an LR parser:

- The left part of the string will be on the stack
  the ▷ symbol is the top of the stack
- We have 2 simple actions:
  reduce: replace a string on the stack
- shift: move the ▷ symbol to the right

This actions are chose non-deterministically.

E ▷
E + T ▷
E + T * int ▷
E + T * ▷ int
E + T ▷ * int
E + int ▷ * int
E + ▷ int * int
E ▷ + int * int
T ▷ + int * int
int ▷ + int * int
▷ int + int * int



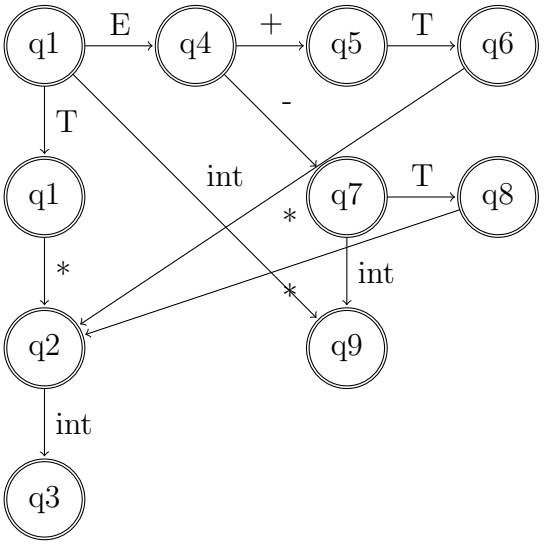
The algorithm will be:

- find all reductions allowed on top of the stack
- create k new instances of the parser
- each instance reduces one of the k strings, in the original instance do no reduction but a shift
- stop when one instance is at the start

But we can do better, kill any instance that you know will get stuck ASAP. For example T + ▷ int * int, we know that this will get stuck, so we can kill it. This is because there is no way that T can be on top of the stack.
How can we tell if a stack is doomed? We list all legal stacks and kill the illegal ones. The listed legal stacks are described as a DFA, if one configuration fails the DFA then it's doomed. In this DFA every state is accepting.

15.1 Simple LR parser

In order to produce a DFA we add a new initial variable E', and a new derivation E' → E.
Our simple grammar becomes:
E' → E
E → E + T | E - T | T
T → T * int | int



Each state is:

- q0:
  E'→ ∘ E
  E → ∘ E + T
  E → ∘ E - T
  E → ∘ T
  T → ∘ T * int
  T → ∘ int

- q1:
  E → T ∘
  T → T ∘ * int

- q2:
  T → T * ∘ int

- q3:
  T → T * int ∘
- q4:
  E' → E ∘
  E → E ∘ + T
  E → E ∘ - T
- q5:
  E → E + ∘ T
  T → ∘ T * int
  T → ∘ int
- q6:
  T → E + T ∘
  T → T ∘ * int
- q7:
  E → E - ∘ T
  T → ∘ T * int
  T → ∘ int
- q8:
  E → E - T ∘
  T → T ∘ * int
- q9:
  T → int ∘

This is a SLR item: $X \to \alpha \circ \beta$, with $X \to \alpha\beta$ being a production.
$X \to \alpha \circ \beta$ describes the context:

- we are trying to find X
- we have α on top of the stack
- we need to see the next a string derived from β

$\circ$ and $\triangleright$ are the same thing, but $\circ$ is used for SLR items and $\triangleright$ for LR items. A DFA state is a closet set of SLR(1) items, which means we perform Closure. The stating state is Closure({E' → ∘ E })

The operation of extending the context with items is called closure operation:
Closure(Items) =
    repeat
        for each X $\rightarrow \alpha \circ Y\beta$ in Items
            for each production $Y \rightarrow \gamma$
                add $Y \rightarrow \circ\gamma$ to items
until Items is unchanged

A State that contains at least an item, X $\rightarrow \alpha \circ y\beta$ has a transition label y to a state Transitions(State,y), y can be terminal or not.
Transitions(State,y) =
    Items $\leftarrow \emptyset$
    for each X $\rightarrow \alpha \circ y\beta$ in State
        add X $\rightarrow \alpha y \circ \beta$ to Items
    return Closure(Items)
The SLR contains the correct stack configuration but also indicates when to shift/reduce. When the $\circ$ is at the end of the rule reduce, otherwise shift. If there are conflicts use the lookahead. If the look ahead finds a discrepancy, it will change the next transition.
The two rules are:

- lookahead for shit: only on the expected terminals
- lookahead for reduce: only on terminals in the Follow set of the variabile on the l.h.s of the rules used to reduce

Consider our example grammar:
E' $\rightarrow$ E
E $\rightarrow$ E + T | E - T | T
T $\rightarrow$ T * int | int
First(E') = int (no need to calculate)
First(E) = int
First(T) = int
First(E') = $ (no need to calculate)
First(E) = $, +, -
First(T) = $, +, -
Here's the SLR table representation This table describes the SLR(1) parsing algorithm. This works only if each cell have one action, meaning there is no conflict.
In this case the grammar is called SLR(1) grammar.

	int	+	-	*	$	E	T
1	shift,9					goto,2	goto,5
2		shift,3	shift,6		accept		
3	shift,9						goto,4
4		red,E→E+T	red,E→E+T	shift,8	red,E→E+T		
5		red, E→T	red, E→T	shift,8	red, E → T		
6	shift,9						goto,7
7		red,E→E-T	red,E→E- T	shift,8	red, E → E -T		
8	shift,10						
9		red,T→ int	red, T → int	red, T → int	red, T → int		
10		red,T→T*int	red,T→T*int	red,T→T*int	red,T→T*int		

Tabella 9: SLR(1) table

15.2 Bottom-up parsing algorithm

This algorithm is stronger that LL(1) because the decision are made after seeing all the symbols not just one ahead.
The parser represents the DFA as a 2D table. Each lines is a state, each column is a terminal or non-terminal. Columns are split into actions and goto.
After each shift or reduce we rerun the DFA on the entire stack, this is a waste. A smarter approach is to save the state of the DFA after that element.
The stack is saved as such: $< sym_1, state_1 > ... < sym_n, state_n >$.
The algorithm is:
Let I = $w_1 w_2 ... w_n$\$ initial input
Let j=i
Let DFA state 0 be the starting state
Let stack = $< dummy, 0 >$
repeat
case $action[topState(stack), I[j])]$ of
shift k: push <I[j],j>; j+=1
reduce X → α:
pop $|\alpha|$ pairs,
push $< X, goto[topState(stack), X] >$
accept: halt normally
accept: halt and report error

Bison it's a tool that takes a grammar and generates a parser. It's a LALR(1) parser generator, which is slightly more complex that a SLR(1).
Bison takes in input grammar rules and C instructions and generates the table and a C program as output.

16 Semantic analysis

16.1 Symbol table

We will learn how to build symbol tables and how to use them to find multiply-declared and undeclared variables.
So far our compiler does:

- lexical analysis: detects illegal tokens
- parsing: detects ill formed parse tree
- semantic analysis: catches all remaining errors

Here's some typical semantic errors:

- multiple declarations
- undeclared variables
- type mismatch
- wrong argument number/types

The semantic analysis is done in 2 steps (we visit the AST):

1. Generation of Enriched AST (performed top-down)
   For each scope of the programma:
   process declarations: add new entries to the symbol table and report any variable/methods method is multiply declared
   process statements: check that variables are declared and in "ID" nodes add a pointer to the appropriate symbol table entry
2. type checking (performed bottom-up)
   Process all of the statement in the program again, use the symbol-table entry information to type check

The symbol table has to map names of anything declared in the program (variables, classes, fields, methods).
The symbol table entry is a set of attributes associated with a name:

- kind of name (variable, class, field, method)

- type
- nesting level
- memory location

The symbol table is designed based on the scoping of the language. Usually a variable can be re-declared once per scope.
For example java uses these names:

- class
- field of class
- method of class
- local variable

Java and C++ support method overloading, method can have same name but have to be unique (by having different params or return).
Java and C++ use static scoping:

- mapping are made at compile time
- C++ uses the "most closely nested rule"
- in Java inner scopes cannot define variables defined in outer scopes

Each function has one or more scopes: one for params, one for body and any nested block inside.
Not all languages use static scoping. For example python uses dynamic scoping, which means that the mapping are made at run time. This is not a great idea, a variable can have different declaration and different types.
In Java it's possible to use a field or a method before the definition, but not a variable.
From now on we will assume that our language:

- uses static scoping
- requires declaration before usage
- does not allow multiply declarations in the same scope, no method overloading even with different kinds of names

- does allow the same method to be defined multiply but once per scope

In addition our language will answer 2 questions: is this name already declared?, to which declaration does our name correspond?. Once we answer this, the table is no longer needed, names are forgotten.
Here's the needed operations:

- look the name in the current scope, if not found insert a new name in the table
- look the name in the current and enclosing scope, to check and link undeclared names
- operations need when a new scope is entered
- operations need when a new scope is exited

The implementation can be a list of table or a table of lists. For simplicity we will assume each table only includes type and nesting level.

16.1.1 List of hashtables

The idea is a list of hashtables, one per scope.
The process:

- On scope entry increment the current level number and add a new empty hashtable at the start of the list O(how hashtable size)
- To process a declaration of x, if it's in the first table the we error otherwise add x to the first hashtable with it's name, type and nesting level. (O(1)) lookup in hashtable is constant
- To process a use of x, search for x in the list, if found link it otherwise error. O(depth of nesting)
- On scope exit remove the first hashtable from the list. O(1)

16.1.2 Hashtable of lists

We have one big hashtable with all the names, and each name has a list of symbol-table entries. The first item is the closest scope. We also store the level number attribute, which tells us in which enclosing the declaration was made.
The operations:

- On scope entry increment the current level number. O(1)
- To process a declaration of x, look up x in the symbol table, if x has the same nesting number error otherwise add a new item with appropriate current level number (O(1))
- To process a use of x, if there is an entry in the symbol-table link the use otherwise error (Undeclared variable) (O(1))
- On scope exit, scan all names at first level, if the level is the same as current then remove it from the list. If a name has empty list, then remove it. Finally decrement the current number O(number or names)

16.2 Type checking

Types are a set of values and operations on those values, like class is a type. Type are needed because we can make operations with different types.
There are 3 kinds of languages:

- statically typed: types are known at compile time (C)
- dynamically typed: types are known at run time (Scheme, lisp dialect...figures)
- untyped: no type checking (machine code)

The are lots of type wars, but the truth is that both are good.
Static typing avoids errors at compile time and overhead at run time.
Dynamic typing allows more flexibility and is easier to prototype.
Type checking is the process of checking that the program obeys the type system. Type inference is when the compiler automatically deduces the type.
We have so far 2 formal notations: regex for lexer and CFG for parser. The way to enforce type checking is having logical rules of inference. Inference

rules have this form: *If Hypothesis is true, then Conclusion is true* therefore *If E_1 and E_2 have certain types, then E_3 has a certain type.*
The notation is simple to read, here's a few building blocks:

- $\wedge$ is "and"
- $\Rightarrow$ is "if then"
- x:T is "x has type T"

Here's an example:
If e_1 has type Int and e_2 has type Int then $e_1 + e_2$ has type Int

(e_1 has type Int $\wedge$ e_2 has type Int) $\Rightarrow$ $e_1 + e_2$ has type Int

(e_1: Int $\wedge$ e_2: Int) $\Rightarrow$ $e_1 + e_2$: Int

$$\frac{\vdash e_1 : Int \qquad \vdash e_2 : Int}{\vdash e_1 + e_2 : Int}$$

$\vdash$ means "we can prove that..."
An inference system for proving x<y:

$$\vdash x < (x + 1)$$

[A] means that A is an axiom, and in the case above it means that any number is smaller than it's successor.

$$\frac{\vdash x < y \qquad \vdash y < z}{\vdash x < z}$$

[T] means that the rule is transitive. Therefore $<$ is transitive.
Let's prove that 6<10.

$$\dfrac{\dfrac{}{\vdash 6 < 7}[A] \qquad \dfrac{\dfrac{\dfrac{}{\vdash 7 < 8}[A] \qquad \vdash 8 < 9[A]}{\vdash 7 < 9}[T] \qquad \vdash 9 < 10[A]}{\vdash 7 < 10}[T]}{\vdash 6 < 10}[T]$$

In our case

$$\frac{\overline{\vdash 1 : Int} \qquad \overline{\vdash 2 : Int}}{\vdash 1 + 2 : Int}$$

A type system is sound if we can infer ⊢e:T.
Here's rules for constants:

$$\frac{}{\vdash false : Bool}[Bool]$$

$$\frac{}{\vdash s : String}[String](\text{s is constant})$$

Object creation:

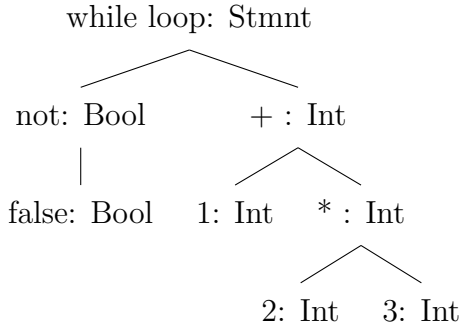
$$\frac{}{\vdash new\ C() : C}[New](\text{C is a class})$$

more rules

$$\frac{\vdash e : Bool}{\vdash not\ e : Bool}\ [Not]$$

$$\frac{\vdash e_1 : Bool \qquad \vdash e_2 : T}{\vdash while\ e_1\ loop\ e_2\ pool : Stmnt}\ [Loop]$$

Let's type: while not false loop 1 + 2 * 3 pool



$$\frac{\dfrac{\vdash false : Bool}{\vdash not\ false : Bool} \qquad \dfrac{\overline{\vdash 1 : Int} \qquad \dfrac{\vdash 2 : Int \qquad \vdash 3 : Int}{\vdash 2 * 3 : Int}}{\vdash 1 + 2 * 3 : Int}}{\vdash while\ not\ false\ loop\ 1 + 2 * 3\ pool : Stmnt}$$

- The root of the tree is the whole expression
- Each node is an instance of typing rule
- Leaves are the rules with no hypotheses

But how can we know the type of a variable?

$$\frac{}{\vdash x :?}[Var] \text{ (x is an identifier)}$$

We don't know enough, it's need to have knowledge of the scope. So let's put more information in the scope.
Let O be a function from Identifiers to Types:
The sentence O $\vdash$ e:T reads:
Under the assumption that the variables in the current scope have the types given by O, it is provable that expression e has type T (this are info store in the symbol table btw).
A type environment O gives types for **free** variables. A free variable is a variable out of current scope.
Here's the previous rules but modified:

$$\frac{}{O \vdash i : Int}[Bool]$$

$$\frac{O \vdash e_1 : Int \qquad O \vdash e_2 : Int}{O \vdash e_1 + e_2 : Int}\,[Add]$$

Now we can write new rules:

$$\frac{}{O \vdash x : T}\ [Id]\ \text{(if O(x)=T)}$$

The types of a function are usually written as:

- $T_1, ..., T_n \to T$
- with $T_1, ..., T_n \to T$ being the params
- and T the output return

$$\frac{O \vdash f : (T_1, ..., T_n \to T) \qquad O \vdash e_1 : T_1 \; ... \; O \vdash e_n : T_n}{O \vdash f(e_1, ..., e_n) : T} \; [FuncCall]$$

The let statement declares a variable x with given type T_0 that is then defined throughout e_1:

- $let\ x : T_0\ in\ e_1$ without initialization
- $letx : T_0 \leftarrow e_0 ine_1$ with initialization

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash let\ x : T_0\ in\ e_1 : T_1} \; [Let - No - Init]$$

For example:
let x: T_0 in ((let y: T_1 in $E_{x,y}$) + (let x: T_2 in $F_{x,y}$)
$E_{x,y}$ and $F_{x,y}$ are expressions that use x and y
The type of x in $E_{x,y}$ is T_0 and the type of x in $F_{x,y}$ is T_2.
Notes:

- The type environment gives types to free identifier in current scope
- The type environment is passed down the from root to leaves
- Types are computed from leaves to root

The downward phase of compilers enrich the AST linking identifies with the their symbol table entry.

Consider let with initialization:

$$\frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash let\ x : T_0 \leftarrow e_0\ in\ e_1 : T_1} \; [Let - Init]$$

This rule is too restrictive. Because it does not allow inheritance.
class C inherits P ...
let x: P $\leftarrow$ new C() in ...

Let's add a relation X $\leq$ Y on classes that means:

- X can be used when Y is acceptable or equivalent
- X conforms with Y

Also know as X is a subclass of Y. Let's define de relations $\leq$:

- X $\leq$ Y
- X $\leq$ Y if X inherits from Y
- X $\leq$ Z if X $\leq$ Y and Y $\leq$ Z

Let's re-write our Let rule:

$$\frac{O \vdash e_0 : T_0 \qquad T \leq T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash let\ x : T_0 \leftarrow e_0\ in\ e_1 : T_1}\ [Let - Init]$$

It's still sound but more flexible. There is tension between flexible and sounds rules.
A static type system enables the compiler to detect many common errors but at the cost of disallowing correct programs. Some argue that the solution is more expressive type system, but this leads to more complex language. Others argue that you should use dynamic type instead.
The static type of an object variable is the class C used to declare the variable, a compile-time notion.
The dynamic type of an object variable is the class C that is used to create its object **value** ("new C"), a run-time notion.
In early type systems those two corresponded (dynamicType(E) = staticType(E)), but in modern languages they don't. You could for instance change class of an object a with class A, if class B inherits from A. Hence dynamicType(E) $\leq$ staticType(E). Soundness Theorem
$\forall$E. dynamicType(E) $\leq$ staticType(E)
This works if the subclass has all public methods and fields of the parent class.

We also need a rule if x is already declared:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad O \vdash e_1 : T_1}{O \vdash x\ \ e_0\ ;\ \ e_1 : T_1}\quad \text{[Assign] (if O(x) = } T_0$$

Let's add subtyping to functions, our input parameters need to account for subtyping $T_{0,1}, ..., T_{0,n} \rightarrow T$

$$\frac{O \vdash f : (T_{0,1}, ..., T_{0,n} \rightarrow T) \qquad O \vdash e_1 : T_1\ ...\ O \vdash e_n : T_n \qquad \forall i T_i \leq T_{0,i}}{O \vdash f(e_1, ..., e_n) : T}\ [Fun]$$

You can subtype arrays but only the content of the cells.

Classes usually can override methods and fields. This is the case in our language FOOL.
Let's start from fields.

- Class A { ..., T:f, ... }
- Class B inherits from { ..., T':f, ... }

Fields can be seen as arrays cells, they can be modified but can't change type. Fields subtyping is not sound, that's why in java we cannot override fields. But if fields are immutable then subtyping is admitted, if T'≤T for every fields then B ≤ A. These are calle *covariant fields*.

Consider methods overriding by changing the return type and the params:

- Class A { ..., T : m(T_1 p_1,...,T_n p_n){e} ... }
- Class B inherits from { ..., T' m(T'_1 p_1,...,T'_n p_n) {e' } ... }

Consider "let x:T ← y.m($e_1, ..., e_n$) in e" assuming "y" with static type A and dynamic type B.

- the B return type T' must be usavble in place of the A return type T, T'≤T.
- the params types must be usable in place of B params types, $T_i \leq T'_i$

The general subtyping rule for functions is:

$$\frac{T_1 \leq T'_1 ... T_n \leq T'_n \text{ and } T' \leq T}{(T'_1, ..., T'_n \to T') \leq (T_1, ..., T_n \to T)}$$

"covariant" output return type
"contravariant" input params types.

You have to make a tradeoff, the less sound the language then more bad programs will be accepted. The more sound the language, the less good program will be accepted.

17 Code generation

We covered the front-end phases: lexical analysis, parsing, semantic analysis. Now we will cover the back-end phases: code generation and optimization.

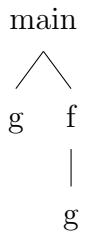
17.1 Memory management

The initial execution of the program is in the hands of the computer. The OS allocates memory for the program, the code in loaded and the OS jumps to the entry point (the main).
The memory get's split into code and other space (not necessarily contiguous memory). The other space is usually data.
The compiler is responsible for generating the code and organizing the use of the data area. The goals for memory management is to be fast and correct, which is hard.
Let's make some assumptions about executions:

1. Execution is sequential
2. When a procedure (functions) is called, we save the pointer to the current position, execute and return to current position

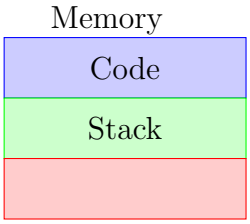
An invocation of procedure P is an activation of P. The *lifetime* of P is: all the steps to execute P, including any procedures P calls.
The *lifetime* of a variable x is the portion of the execution in which x is defined. The difference between variable scope and lifetime is: lifetime dynamic run-time concept while scope is a static concept.
Our next assumption (2) is that when P calls Q, then Q returns before P does (like in recursions). Lifetimes of procedure activations are nested and can be depicted as trees.

```
1    class Main {
2        int g() { return 1; }
3        int f() { return g(); }
4        void main() { g(); f(); }
5    }
```



Note:

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Currently active procedures are tracked in a stack



The information needed to manage one procedure activation is calle **activation record** or **frame**. If procedure f calls g, the g's activation record contains a mix of info about f and g.
When f calls g, f is "suspended" until g completes. f will be resumed by g, g knows how thanks to it's AR.
g's AR could contain:

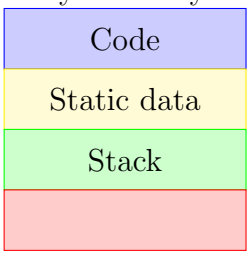
- the return address of f
- g params
- space for g's local variables
- other temporary values

g's AR should have:

- pointer to the previous AR, called control link and points to the caller of g

- machine status prior to calling g, contents of the registers and program counter (pointer to current position in stack).

The compiler therefore must determine at compile time the **layout** of the activation record and generate code that access correctly the locations in the activation record: AR layout and code generator must be designed together. You could have a simple stack that has a fixed offset which works, but if organized it could help speed and simplicity.
All reference to global variables have a fixed address, they can be statically or dynamically allocated.



Reference to a variable declared in an outer scope should point to a variable stored in another record. Any block has it's own activation record. In particular it should point to the most recent AR within the immediately enclosing scope.
The access link is used to find outer scope variables and always point to the enclosing syntax block, like say a functions.
The value of the access link is defined as follows:

- when we enter a new scope or create a function: access link = current AR (we save the current access link address)
- a function calls herself or another function: access link = caller's access link
- call of a function outside the current scope: access link = follow the chain of access link until you find the correct one

Any data that outlives the procedure can't be saved in the AR, such data must be saved in the *heap*.
Unlike di AR data the heap data should be remove when it becomes garbage. The garbage collection is done by the system when needed, it is execute at the same time as the program.
In some languages like C the deallocation is under the responsibility of the

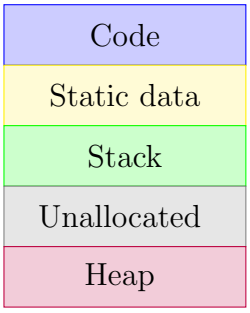
programmer which can lead to big problems when pointers share the same memory.

Mark and sweep algorithm tags all resources with 0, then follows all links and set those tags to 1. Finally removes any tag that is 0.

Reference counting algorithm counts the number of references to a resource, when it reaches 0 it removes the resource. When the pointer is set to a resource it increments the counter, when the pointer is set to null it decrements the counter.
Datum = data element, which is basically a resource.

Notes:

- the code portion contains object code, usually fixed size and read only
- the static area contains data with fixed addresses, fixed size and could be readable and writable
- the stack contains activation records, grows and shrinks dynamically. A single procedure is usually fixed size and contains locals
- heap contains all the other data and it can grow
- heap and stack could grow into each other! One way to avoid it's to have them at opposite memory locations.



17.2 Code generation for stack machine

It's a machine that computes using a stack. Both the operations and the results are pushed on the stack.

Say you want to add to numbers: push 5, push 7, add. The add operation will pop 5 and 7, add them and push the result.
The big advantage of the stack machine is it's simplicity:

- input and output in the same place
- uniform compilation which means simple compiler
- the stack is invariant, results stack on top22
- items are always on top of the stack: location is implicit
- no need to specify operands and results
- compact programs and small instructions
- java bytecode is a stack machine

Any operation need the stack will perform 2 reads and 1 write (2 pops and 1 push). We could optimize this by keeping the top of the stack in a register (accumulator) therefore removing one read. We will not do it, for simplicity sake.

17.2.1 From stack machine to assembly language

We will consider a stack machine without an accumulator.We want to run this code on a processor, in our case it will be MIPS (Microprocessor without Interlocked Pipeline Stage). There are several emulators of this processor. To be more precise we implement the stack machine using MIPS instructions and registers.
MIPS architecture:

- Prototype RISC (Reduced Instruction Set Computer) architecture
- arithmetic operations use register for operands and results
- use load and store instructions to access memory
- 32 general use registers, each 32 bits. We will use \$sp and $\$t_0$,$\$t_1$,$\$t_2$ (temporary registers)

Therefore the stack will be kept in memory and grow towards lower addresses. The \$sp register keeps track of the top of the stack. Let's define some macros:

- push $t becomes addiu2 $sp, $sp, -4; sw $t, 0($sp)
- pop $t becomes addiu $sp, $sp, 4
- $t → pop becomes lw $t, 0($sp); addiut $sp, $sp, 4
- pop*n becomes addiu $sp, $sp, z where z = 4*n

Let's define a small language with integers, integer operation and global functions with at least one parameter.

P → D ; P | S

D → fun id(ARGS) = E

ARGS → id, ARGS | id

E → int | id | if E_1 = E_2 then E_3 else E_4 | E_1 + E_2 | E_1 - E_2 | id(E_1, ..., E_n)

The first function definition f is the main routine.

Here's the program for computing Fibonacci:
fun fib(x) = if x=1 then 0 else:
if x=2 then 1 else
fib(x-1) + fib(x-2)

For each expression e we will generate MIPS code that:

- computes the result of e and pushes it to the stack
- preserves the content of the stack that came before e, we simply add on the top the result of e

cgen(e) is generation function which generates code for e. Eg: Say you have a constant i, the code cgen(i) = li $\$t_0$ i ; push $ t_0.
Let's see cgen(e_1, e_2):

1. cgen(e_1)
2. cgen(e_2)
3. $\$t_2 \leftarrow$ pop
4. $\$t_1 \leftarrow$ pop
5. add $\$t_0$, $\$t_1$, $\$t_2$

6. push $\$t_0$

What if the put the result in t_1 to optimize operations? We would have wrong code! You can't preserve the value of a register between operations!
Code generation notes:

- stack machine code generation is recursive
- code generations is performed bottom-up by a recursive traversal of the AST.

Let's add sub to our instructions: sub reg_0 reg_1 reg_2 or $reg_0 \leftarrow reg_1$ reg_2.
cgen(e_1 - e_2):

- cgen(e_1)
- cgen(e_2)
- $\$t_2 \leftarrow$ pop
- $\$t_1 \leftarrow$ pop
- sub $\$t_0$, $\$t_1$, $\$t_2$
- push $\$t_0$

We also need to add flow control instructions: beq reg_1 reg_2 label or if reg_1 = reg_2 then goto label.
Another new instruction is b label, unconditional jump to label.
cgen(if $e_1 = e_2$ then e_3 else e_4):

```
    cgen(e1)
    cgen(e2)
    $t2 ← pop
    $t1 ← pop
    beq $t1, $t2, true_branch
    cgen(e4)
    b end_if
true_branch:
    cgen(e3)
end_if:
```

You are supposed to also generate your labels with cgen().
Code for functions and functions definition depend on the activation record layout. In our case a simple AR will suffice:

- the result of the function is on top of the stack, no need to reserve space for the result in the AR
- the AR holds parameters values, these are the only variables in our current language

We still need to store the return address and the control link. The access link is still not needed, we don't have nested declarations, we have local params and global functions.
It's handy to have a pointer to the current position (reference pointer) in the AR. This pointer is saved in $fp (frame pointer). The reference position is part of AR layout design. $fp points to the position of the first argument and it's used by the code generator to locate elements (params, based on offset). We need to store the caller $fp (control link) because it has to be restored after the call.
Now let's add a new instructions: jal label (jump and link). Jump to label and save the return address in $ra register.
Code generation for a function call:
cgen(f(e_1, ..., e_n)):
    push $fp
    cgen(e_n)
    ...
    cgen(e_1)
    jal f_entry
We save the frame pointer, push each value in reverse order, save the return address in $ra and so far our memory used is 4*n+4 bytes (4 bytes for each param and 4 byte for the $fp).
Let's add another istruction: jr reg (jump register). Jump to the address contained in reg.
cgen(fun f(x_1, ..., x_n) = e):
    f_entry:
    move $fp, $sp
    push $ra
    cgen(e)
    $$t_0 \leftarrow$$ pop
    $ra $\leftarrow$ pop
    pop*n
    $fp \leftarrow$ pop
    push $$t_0$$

jr $ra

Save the current stack in $fp into $sp, save the return address in the temporary field $t_0, we do all the operations save the return and jump to the $ra. Activation record are **not** saved adjacently in the stack, because the stack grows during execution.

Be mindful, the $sp always changes! We can't use to perform operations with our params, however this is not true for the $fp. Also remember that the $fp keeps the location of the first param.

Therefore in fun (f(x_1, x_2) = e) x_1 is at $fp and x_2 is at $fp + 4.

Thus based on the offset of the params x_i: cgen(x_i) = lw t_0 z($fp); push t_0. The offset is z=4*(i-1) and should be saved inside the symbol table.

17.2.2 Code generation for OOP languages

We have 2 issues:

- How to represent objects in memory.
- How to dynamically dispatch methods.

Consider the following code:

```
1   class A {
2       int a = 0;
3       int d = 1;
4       int f() { return a = a + d; }
5   }
6
7   class B extends A {
8       int b = 2;
9       int f() { return a; } // override
10      int g() { return a = a - b; }
11  }
12
13  class C extends B {
14      int c = 3;
15      int h() { return a = a * c; }
16  }
```

Fields a and d are inherited by B and C. All methods in all classes refer to a. For a to work in all classes a must be in the same place.
An object is like a struct in C, foo.field is an index into foo struct at an offset corresponding to field.
Objects are laid out in a contiguous memory section. Each field is stored at a fixed index. An object is created from the corresponding layout (I assume statically saved) in the heap.
Given a layout for class A, we can define a subclass B which extends the layout with new name. This way A is unchanged.
Each class has a fixed set of methods, saved in the **dispatch table**.
The dispatch table is a table of function pointers, one for each method. There is also a method f at a fixed offset for a class and all of its subclasses. We can override f but it won't change its index. All other methods get a on offset, inside the dispatch table, from f.

18 Basic logic

Logic is a foundational theory. In computer science logic provides a language for expressing property of interest of a system and a way of enacting these properties into the same system.
The basic syntax of logic is: a set of admissible formulas F = { $\phi_0, \phi_1, ...$ }.
The basic semantics of logic is by two elements: $< \mathcal{M}, \models >$:

- $\mathcal{M}$ is a set of models (M $\in \mathcal{M}$)
- $\vdash \subset \mathcal{M} \times \mathcal{F}$ the satisfaction relation

Terminology and notation

- $M \models \phi$ means that model M satisfies formula ϕ
- (or: formal ϕ is satisfied in interpretation/valuation M)
- Also is valid to satisfy a set of formulas $\mathcal{M} \models \phi, \phi', ...$

Given a finite set of propositions symbols (P = {p,q,r,...}) a formula can be:

- T (true constant)
- $\perp$ (false constant)
- propositional symbol p $\in$ P
- of the kind: $\neg\phi$ (negation)
- of the kind: $\phi \vee \psi$ (or)
- of the kind: $\phi \wedge \psi$ (and)
- of the kind: $\phi \rightarrow \psi$ (implication)

nothing else can be a formula. This can also be written as a grammar: $\phi ::=$ p | T $|\perp|\neg\phi, \phi \vee \psi, \phi \wedge \psi, \phi \rightarrow \psi$
A model M is a function M: P $\rightarrow$ {F,T}.
Let's define the satisfaction relation M $\models \phi$:

- M $\models$ p $\iff$ M(p) = T

- M $\models$ T
- M $\not\models \bot$
- M $\models \neg\phi \iff$ M $\not\models \phi$
- M $\models \phi \vee \psi \iff$ M $\models \phi$ or M $\models \psi$
- M $\models \phi \wedge \psi \iff$ M $\models \phi$ and M $\models \psi$
- M $\models \phi \rightarrow \psi \iff$ M $\not\models \phi$ implies M $\models \psi$

Let's define some logical concepts.
Logical consequence:

- Formal ϕ is a logical consequence of $\phi_1, ..., \phi_n$ if for any M, when M $\models \phi_1, ..., \phi_n$ then M $\models \phi$
- This form is equivalent to: $\phi_1, ..., \phi_n \vdash \phi$

Tautology/Validity definition: A formula ϕ that is logical consequence of no formulas (always true in any model) is called tautology. The notation is: $\models \phi$ or we can simply say ϕ is valid.
We can prove by contradiction: If $\phi_1, ..., \phi_n \neg\phi \models \bot$ then $\phi_1, ..., \phi_n \models \phi$.
Two formulas logically equivalent if $\phi \models \psi$ therefore $\psi \models \phi$.
A formula ϕ is satisfiable if there is a model M such that M $\models \phi$. ϕ is valid if and only $\neg\phi$ is not satisfiable.
Logical consequence: $\phi_1, ..., \phi_n$ if and only if $\models (\phi_1 \wedge ... \wedge \phi_n \rightarrow \phi)$.

Logical equivalence: $\phi \equiv \psi$ if and only if $\models (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. This could also be written using $\leftrightarrow$: LE (... $\models \phi \leftrightarrow \psi$. Note: not all logics has the symbol $\rightarrow$.

All operators can be defined in terms of $\neg$ and $\rightarrow$:

- T $\equiv p \rightarrow q$
- $\bot \equiv \neg(p \rightarrow q)$
- $\phi \vee \psi \equiv \neg\phi \rightarrow \psi$
- $\phi \wedge \psi \equiv \neg(\phi \rightarrow \neg\psi)$

We can now simplify the grammar from before like so: $\phi ::= p|\neg\phi|\phi \to \phi$ or $\phi ::= p|\bot|\phi \to \phi$
How can we know if $\phi \models \psi$, after all this should hold for any M, which are infinite. Fortunately M using variables in ϕ and ψ are limited, basically it's sufficient to build the truth table.
In logic there is no obvious way to compute validity, we need some mechanism to do that.
In propositional logic we can use truth tables. There is a more efficient technique called **CNF** (Conjunctive Normal Form), there are tool which are optimize to compute this form.
CNF is a conjunction ($\wedge$) of clauses, where a clause is a disjunction ($\vee$) of literals. A literal is a propositional symbol or it's negation.

18.1 Deduction system

Validity can be deduced:

- $(a \vee b \vee c \vee d \vee e) \to c$ since on the left we have a conjunction that includes the implied formula (we have c on the left)
- $\neg((a \vee b \vee c \vee d \vee e) \to c)$ is false
- a false formula can be any formula

We use axiom to represent this cases: $\phi \vee \psi \to \phi$, $\neg\neg T$, $\bot \to \phi$.
Let's induce a deduction symbol $\vdash$. $\phi_1, ..., \phi_n \vdash \phi$ means that ϕ can be deduced from $\phi_1, ..., \phi_n$. $\vdash \phi$ means ϕ can't be deduced by any formula. ($\models$ is called semantic entailment, while $\vdash$ is syntactic entailment).
We can use $\models$ in any mathematical form while $\vdash$ should be constructing use rules and axioms

- Form axioms: $\vdash \phi$
- Form for rules: $\dfrac{\vdash \phi_1 ... \vdash \phi_n}{\vdash \phi}$

where $\phi, \phi_1, ...$ are formulas or formula schemas.
Hilbert deduction system minimal syntax: $\phi ::= p|\neg\phi|\phi \to \phi$. Axioms:

- $\vdash \alpha \to (\beta \to \alpha)$

- $\vdash (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$
- $\vdash (\neg\alpha \to \neg\beta) \to (\alpha \to \beta)$

Rule: $\dfrac{\vdash \alpha \vdash \alpha \to \beta}{\vdash \beta}$

To prove $\phi_1 ... \phi_n \vdash \phi$ we need to prove $\dfrac{\vdash \phi_1 ... \vdash \phi_n}{\vdash \phi}$

How sound is this system? Well if $\phi_1 ... \phi_n \vdash \phi$ then $\phi_1 ... \phi_n \models \phi$ which works on reverse by the way $\phi_1 ... \phi_n \models \phi$ then $\phi_1 ... \phi_n \vdash \phi$.

18.2 Predicate logic

Predicate logic should be more structure and discuss individual and global properties. For example:
"All men are mortal, Socrates is a man, therefore Socrates is mortal"
translates to
$\models (\forall(man(X) \to mort(X)) \wedge man(socrat)) \to mort(socrat)$.
Another example:
Every student is younger than some professor
translates to
M $\models (\forall X(stud(X) \to (\exists Y(instr(Y) \wedge younger(X, Y)))))$
A term t over (C,F) has syntax: t ::= x | c | f(t_1, ..., t_n) where x is a variable, c $\in$ C is a constant and f $\in$ F is a with n terms as arguments.
A formula ϕ over (P,C,F) has syntax: $\phi ::= p(t_1, ..., t_n)|T|\bot|\neg\phi|\phi \vee \phi|\phi \wedge \phi|\phi \to \phi|\forall x\phi|\exists x\phi$

- an atomic formula: p $\in$ P with n terms over (C,F)
- a formula $T, \bot, \neg\phi, \phi \vee \phi, \phi \wedge \phi, \phi \to \phi$
- a universal quantification of x in ϕ
- an existential quantification of x in ϕ

The structure of a model M over (C,P,F) is of the kind $< \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathcal{P} >$:

- A is a non empty universe of concrete values
- $\mathcal{C}$: C $\to$ A associate to each constat a concerete value

- $\mathcal{F}$: F $\rightarrow$ (A^n $\rightarrow$A) associate to each function name a concrete funcion
- $\mathcal{P}$: P $\rightarrow$ (A^n $\rightarrow$ {F, P}) associate to each predicate name a concrete predicate name

In a specific model M a term is associated with a concrete value while an atomic formula p is associated with a boolean.
Let's make an example using the previous student example:

$$M \models (\forall X(stud(X) \rightarrow (\exists Y(instr(Y) \wedge younger(X, Y)))))$$

this translates to:
C = {}, F = {}, P = { stud/1, instr/1, younger/2}
Therefore our an example model M = $< \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathcal{P} >$ could be:

- A = {bravetti,rossi,omicini,bianchi,natali}
- C = {}, F = {}
- P = {
  - stud(bravetti) = F, stud(rossi) = T, stud(omicini) = F, stud(bianchi) = T, stud(natali) = F
  - instr(bravetti) = T, instr(rossi) = F, instr(omicini) = T, instr(bianchi) = F, instr(natali) = T
  - younger(rossi, natali) = T, younger(natali, omicini) = F, younger(bianchi, bravetti) = T, ...

  }

A variable is bound if it's inside a *binder* $\forall$x and $\exists$x otherwise is called free. ϕ is closed if it does not include free variables. We will talk only about closed formulas, here's the semantics:

- M $\models p(t_1, ..., t_n) \iff$ M(p$(t_1, ..., t_n)$) = T
- M $\models$T, M $\not\models \bot$
- M $\models \neg\phi \iff$ M $\not\models \phi$
- M $\models \phi \wedge \psi \iff$ M $\models \phi$ and M $\models \psi$

- M $\models \phi \vee \psi \iff$ M $\models \phi$ or M $\models \psi$
- M $\models \phi \rightarrow \psi \iff$ M $\models \phi$ implies M $\models \psi$
- M $\models \forall\phi \iff$ for all v $\in$ A we have M $\models \phi[v/x]$
- M $\models \exists\phi \iff$ exists v $\in$ A such as M $\models \phi[v/x]$

where $\phi[v/x]$ is the formula obtained by replacing all free occurrences of x in ϕ with v.
Let's use some more formal definitions of M(p(t_1, ... , t_n)), remember the model $< \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathcal{P} >$.
M() over atomic fomrulas looks like this: M(p(t_1, ... , t_n)) = P(p)M(t_1,...,M(t_n))
M() over terms looks like this:

- M(v) = v
- M(c) = C(c)
- M(f(t_1,...,t_n)) = $\mathcal{F}$(f)(M(t_1,...,$M(t_n)$)

Here's an example of equivalent logic: $\exists x\phi \equiv \neg\forall x\neg\phi$.
And here's a simplified version of predicate logic: $\phi ::= p(t_1, ..., t)n)| \neq \phi|\phi \rightarrow \neg|\forall x\phi$.

Turing and Church proved that $\models \phi$ is undecidable. In a procedural implementation if $\models \phi$ has an answer in a finite time then is valid, while $\not\models \phi$ happens where the procedure does not terminate.

18.3 Hilber dedection system

Minimal syntax: $\phi ::= p(t_1, ..., t_n)|\neg\phi|\phi \rightarrow \phi|\forall x\phi$.
Axioms:

- those of Propositional Logic
- $\vdash (\forall x\alpha) \rightarrow \alpha[t/x]$ with no free variable in t becoming bound
- $\vdash \forall x(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \forall x\beta)$ with x not free in α.

Rules:

- $\dfrac{\vdash \alpha \vdash \alpha \to \beta}{\vdash \beta}$ (Modus Ponenes)

- $\dfrac{\vdash \phi}{\vdash \forall x \phi}$ (Universal Generalisation)

where $\phi[t/x]$ is the formula obtained by replacing all free occurrences of x in ϕ with t.
It can be proved that $\vdash$ is correct and complete, but is undecidable (from $\models$).

19 Model checking

19.1 Logiche temporali

Nella logica classica c'è un solo "mondo" su cui si posso esprimere proprietà sempre vere o false: "oggi è lunedì" o è sempre vera o è sempre falsa.
I sistemi software sono dinamici: "x vale 10" può essere a volte vera a volte falsa. Le logiche temporali esprimo proprietà temporali: vere in alcuni mondi false in altri.
I "mondi— corrispondo a diversi momenti temporali. Queste relazioni possono essere: linear time oppure branching time.
Una sequenza lineare di mondi indicano fatti veri in quel instante. Estendiamo la nostra logica per specificare proprietà sui mondi:

- $\bigcirc\ \phi$: ϕ è vera nel prossimo momento
- $\Box\ \phi$: ϕ è vera in tutti i momenti successivi
- $\Diamond\ \phi$: ϕ è vera in alcuni momenti successivi
- $\phi\,\mathcal{U}\psi$: ϕ è vera fino a quando ψ vera

Sintassi LTL:
$\phi, \psi \rightarrow$ p | (atomic proposition)
T | (true)
$\perp$| (true)
$\neg\phi$| (complement)
$\phi \wedge \psi$| (conjunction)
$\phi \vee \psi$| (disjunction)
$\bigcirc$ (next time)
$\Box$ (always)
$\Diamond$ (sometime)
$\phi\,\mathcal{U}\psi$: ϕ (until)

Si considerino formule di LTL che utilizzano proposizioni p $\in$ P. Un modello π è una sequenza infinita di $\pi_1, \pi_2, ...$ di mondi dove pi_i è una funzione: $\pi_i : P \rightarrow \{T, F\}$.
Definiamo la relazione $\pi \models \phi$:

- $\pi \models p \Longleftrightarrow \pi_1$(p) = T (unica differenza con logica proposizionale)

- $\pi \models$T, $\pi \not\models \perp$
- $\pi \models \neg\phi \iff \pi \not\models \phi$
- $\pi \models \phi \wedge \psi \iff \pi \models \phi$ and $\pi \models \psi$
- $\pi \models \phi \vee \psi \iff \pi \models \phi$ or $\pi \models \psi$
- $\pi \models \phi \rightarrow \psi \iff \pi \models \phi$ implies $\pi \models \psi$

L'operatore next indica la proposizione corretto nel prossimo momento temporale. Esempio: (sad $\wedge\neg$ rich) $\bigcirc$0.15 sad. La semantica corretta sarebbe: $\pi \models \bigcirc 0.15 \iff \pi^2 \models \phi$. ($\pi_i$ è la sequenza di mondi, quindi al quadrato significa solo che è il secondo mondo)

L'operatore sometime è vero ora o in qualche momento futuro. Esempio: sad $\Rightarrow \Diamond$happy. Semantica: $\pi \models \Diamond\phi \iff$ there is some i ≥ 1 such that $\pi_1 \models \phi$.

L'operatore always è vero ora e in ogni momento futuro. Esempio: lotter-win $\Rightarrow \Box$0.3 rich. Semantica: $\pi \models \Box 0.3\ \phi \iff$ for all i ≥ 1 we have $\pi^i \models \phi$.

L'operatore until mettere in relazione due proposizioni, una vera ora e per tutti i momenti successivi finché non è vera un altra. Esempio born $\Rightarrow$ alive $\mathcal{U}$ dead. Semantica: $\pi \models \phi\mathcal{U}\psi \iff$ there is some i ≥ 1 such that $\pi^i \models \psi$ and for all j = 1...i-1 we have $\pi^j \models \psi$.

Possiamo minimizzare gli operatori:

- always e sometime sono duali: $\neg\Box 0.30\phi \equiv \Diamond\neg\psi$
- sometime può essere espresso usando until: $\Box\phi \equiv T\mathcal{U}\psi$
- quindi tutti gli operatori si possono esprimere tramite next e until

Relazione con operatori classici:

- sometime distribuisce su or e always su and
  $\Diamond\ (\phi \vee \psi) \equiv \Diamond\phi \vee \Diamond\psi$
  $\Box\ (\phi \wedge \psi) \equiv \Box\phi \wedge \Box\psi$
- relazione con il not:
  $\neg \bigcirc \phi \equiv \bigcirc \neg\phi$
  $\neg(\psi\mathcal{U}\phi) \equiv (\neg\psi\mathcal{U}(\neg\phi \wedge \neg\psi)) \vee \Box\ \neg\psi$

Possiamo usare notazione alternative:

- $\Diamond \rightarrow$ F oppure <>
- $\Box \rightarrow$ G oppure []
- $\bigcirc \rightarrow$ X oppure ()

Ecco alcuni concetti importanti per i sistemi informatici:

- safety ($\Box\neg$ ....): non raggiungo mai stati con errori
- liveness ($\Diamond$....): primo o poi eseguo un azione
- fairness ($\Box \Diamond$ ready $\Rightarrow \Box \Diamond$ run): se chiedo una cosa infinite volte, verrà eseguita infinite volte

Le proposizioni π_i(p) per i ≥ 1 sono codificati tramite predicati unari p(i) su numeri naturali. Una formula LTL ϕ è codificata dalla formula $[\![\phi]\!]_{s(z)}$, dove:
$[\![p]\!]_t = \text{p(t)}$
$[\![\bigcirc\phi]\!]_t = [\![\phi]\!]_{s(t)}$
$[\![\Diamond\phi]\!]_t = \exists t'\ t' \geq t \wedge [\![\phi]\!]_{t'}$
$[\![\Box\phi]\!]_t = \forall t'\ t' \geq t \rightarrow [\![\phi]\!]_{t'}$
$[\![\phi \mathcal{U} \psi]\!]_t = \forall t'\ t' \geq t \rightarrow [\![\psi]\!]_{t'} \wedge \forall t''(t \leq t'' \wedge s(t'') \leq t') \rightarrow [\![\phi]\!]_{t''}$

19.2 Labeled transition system e logiche temporali

Un LTS è un NFA senza stati di accettazione F. Una traccia di un LTS è una sequenza di etichette I_n. Le tracce ricordano le stringe degli NFA, a noi interessa le massimali: infinito oppure il programma si blocca.
Formalmente una traccia σ di un LTS (Q, Σ, δ, q_0) è una sequenza di etichette I_1 I_2 ... tale che esistono stati q_1, q_2 ... con $\text{q}_i \in \delta(\text{q}_{i-1}, \text{I}_I)$ per ogni i =1 ... len(σ).
Una traccia δ è un massimale se len(σ) = ∞ oppure se $\text{q}_{len(\delta)}$ è di blocco, ovvero $\delta(\text{q}_{len(\delta)}, \text{O}) = \emptyset$ per ogni I $\in \Sigma$.

Lungo una traccia vale solo una proposizione che coincide con l'etichetta della transizione successiva. Un LTS soddisfa una formula se tutte le sue tracci massimali soddisfano la formula.

Formalmente una traccia $\sigma = I_1\ I_2\ ...$ di un LTS (Q,Σ, δ, q_0) individua il modello π definito da:

$$\pi_i(p) = T \Longleftrightarrow i\ len(\sigma) \wedge p = I_i$$

dove i ≥ 1 e p $\in \Sigma$.
Un LTS soddisfa una formula ϕ di LTL se i modelli π individuati da tutte le sue tracce massimali sono tali che: $\pi \models \phi$.
Non sono super sicuro sulla mutua esclusione... Riguardalo post esercizi.
I sistemi modellati da automi LTS sono detti "sistemi concorrenti". La "process algebra" sono un modo naturale per rappresentare sistemi di questo tipo.

19.3 Process algebra

Esistono molti tipi di process algebra, noi trattiamo FSP(finite state process) usato dal LTSA (labeled transition system analyzer). Un processo è rappresentato come LTS, ovvero diamo nomi agli stati e descriviamo le transizione che può fare.

Esempio interruttore:
SWITCH = OFF,
off = (on $\rightarrow$ ON),
on = (off $\rightarrow$ OFF).
oppure se vogliamo essere concisi
SWITCH = (on $\rightarrow$ off $\rightarrow$ SWITCH).
Anche gli LTS hanno il concetto di concorrenza (interleaving), inoltre c'è il concetto di sincronizzazione che è l'esecuzione fisica contemporanea.
Interleaving è l'esecuzione di azioni che mancano in altri processi in ordine arbitraria, mentre la sincronizzazione è l'esecuzione simultanea di etichette comuni in più LTS.

Esempio interleaving:
ITCH = (scratch $\rightarrow$ STOP).
CONVERSE = (think $\rightarrow$ talk $\rightarrow$ STOP).
CONVERSE_ITCH = (ITCH || CONVERSE)
Siccome i due alfabeti sono disgiunti si possono fare le 2 azioni in contemporanea.

Esempio sincronizzazione:

MAKER = (make → ready → used → MAKER →).
USER = (ready → use → used → USER →).
|| MAKER_USER = (MAKER || USER)
I due alfabeti si sincronizzano su used.
Formalmente:
Dati due LTS $A_i = (Q_i, \Sigma_i, \delta_i, q_0^i)$, la loro composizione parallela A_1|| A2 è l'LTS $(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_0^1, q_0^1))$, dove δ è definita:

- per ogni a $\in \Sigma_1 \cap \Sigma_2$ (sincronizzazione)

  $$\delta((p,q),a) = \delta_1(p,a) \times \delta_2(q,a)$$

  cioè se p $\xrightarrow{a}$ p' e q $\xrightarrow{a}$ q' allora (p,q) $\xrightarrow{a}$ (p', q')

- per ogni a $\notin \Sigma_1 \cap \Sigma_2$ (interleaving)

  $$\delta((p,q),a) = \delta_1(p,a) \times \{q\} se\ a \in \Sigma_1$$

  cioè se p $\xrightarrow{a}$ p' e allora (p,q) $\xrightarrow{a}$ (p', q)

  $$\delta((p,q),a) = \{p\} \times \delta(q,a) se\ a \in \Sigma_2$$

  cioè se q $\xrightarrow{a}$ q' e allora (p,q) $\xrightarrow{a}$ (p, q')

La composizione parallela è commutativa e associativa:
(A_1|| A_2) = (A_2|| A_1)
(A_1|| A_2) || A_3 = A_1||(A_2 || A_3)
Quindi la composizione può essere denotata in paralleli di multipli: A_1|| A_2|| ... || A_n.
Possono limitare un alfabeto P usando "P@{a_1,...,ax}" alle azioni {a_1,...,ax}. Altre azioni vengono trasformate in **tau** in modo che si eviti la sincronizzazione con altri processi.
User = (acquista → use → release → USER) @{acquire, release}.
Formalmente:
Data un LTS A = (Q, Σ, δ, q_0), l'LTS ottenuto limitando il suo alfabeto a $\Sigma' \in \Sigma$ è A@Σ' = (Q, Σ', δ, q_0) dove δ' è definita:

- per l'azione speciale tau (nascondere azioni)

  $$\delta'(p,\tau) = \bigcup_{a \notin \Sigma'} \delta(p,a)$$

  cioè se p $\xrightarrow{a}$ p' $\wedge \in \Sigma'$ allora in A@Σ' si ha p $\xrightarrow{tau}$ p'

- per ogni a $\in \Sigma'$ (azioni non nascoste)

$$\delta'(p, a) = \delta(p, a)$$

cioè se p $\xrightarrow{a}$ p' $\wedge \in \Sigma'$ allora in A@Σ' si ha p $\xrightarrow{a}$ p'

19.4 Rete di Petri (Petri nets)

È un approccio grafico alla rappresentazione dei sistemi concorrente. Può essere vista com un estensione degli automi per sistemi concorrenti dove: gli stati sono "piazze" (place), più piazze sono contemporaneamente attive(indicate da token) e le transizioni modificano gli stati attivi (generano e consumano multi insieme di stati attivi/token)
Formalmente una Petri Net(PN) è una quintupla N = (P, T, F, W, M_0) dove:

- P = { p_1, ... , p_m } è un insieme finito di piazze
- T = { t_1, ... , t_n } è un insieme finito di transizioni
- F $\subseteq$ (P $\times$ T) $\cup$ (T $\times$ P) è una relazione di flusso (insieme di archi)
- W: F {1,2,3,..} è una funzione di peso
- M_0: P {0,1,2,3,..} è una funzione di marcatura iniziale
- P $\cap$ T = $\emptyset$ e P $\cup$ T $\neq \emptyset$

Il marking graph di una Petri Net è un LTS che ne definisce il comportamento. C'è una configurazione iniziale (marking) che segna la quantità di token in ogni piazza. Da una configurazione all'altra si modificano il marking corrente. Il marking iniziale è M_0.
Formalmente:

- Un marking M è un multi insieme di piazzeç
  M: P {0,1,2,3,..}
- Una transizione t $\in$ T di una PN è abilitata in un marking M se:
  Per ogni p $\in$ P (M(p) $\geq$ W(p,t)
- L'esecuzione di una t $\in$ T abilitata nel marking M fa passare la PN al marking M' dato da:
  Per ogni p $\in$ P (M'(p) = M(p) - W(p,t) + W(t,p)

È possibile che il marking sia infinito, succede quando la rete di Petri è unbounded. Questo non si verifica se la formula di LTL è soddisfatta.

20 Macchina di Turing

Turing arrivò alla conclusione seguente conclusione: "Se un problema è intuitivamente calcolabile (risolvibile con seguendo una procedura) allora esiste una touring machine che lo risolve".
Definizione formale di touring machine: una TM è una 7-pla (Q,Σ, Γ, δ,q_0, B, F) con:

- Un insieme finito di stati Q
- Un alfabeto di input Σ
- Un alfabeto di nastro Γ con $\Sigma \subseteq \Gamma$
- Una funzione di transizione δ
- Uno stato iniziale $q_0 \in Q$
- Un simbolo di blank B $\in \Gamma \setminus \Sigma$ (tutto il nastro, meno l'input, è blank
- Un insieme di stati finali F $\subseteq$ Q

C'è una testina che come azione cambia lo stato cambiando il simbolo che indica e si muove alla cella destra o sinistra. Il nastro è un numero infinito di celle che contengono simboli di un alfabeto Γ.
Per convenzione:

- a, b... sono simboli di input
- ..., X, Y, Z sono simboli del nastro
- ..., w, x, y, z sono stringhe in input
- α, β,... sono stringhe sul nastro

Dati due argomenti q in Q e Z in Γ la funzione di transizione δ(q,Z) è indefinita oppure è una tripla dalla forma (p, Y, D):

- p è uno stato
- Y nuovo simbolo del nastro
- D è una direzione, L o R

Quindi se ho $\delta(q, Z) = (p, Y, D)$ allora la TM cambia stato da q a p, cambia il simbolo Z con Y sul nastro e si muove nella direzione D(destra o sinistra, R o L).
Inizialmente una TM ha una string in input e un infinità di blank in entrambe le direzioni. La testina è sul simbolo di input più a sinistra.
Questo stato si chiama ID (descrizione istantanea) e si descrive tramite una stringa $\alpha\mathbf{q}\beta$ dove:

- $\alpha\beta$ è il contenuto del nastro. Tutto quello che è a destra si assume sia blank
- **q** è lo stato corrente della TM
- La testina è sul primo simbolo di β, se non c'è il carattere la testina punta a un blank.

Come per i PDA usiamo $\vdash$ e $\vdash^*$v per indicare i passaggi di ID (mossa singola e mossa multipla).
Formalmente una mossa è definita in questo modo:

- Se δ(q,Z) = (p,Y,R) allora:
  - $\alpha qZ\beta \vdash \alpha Yp\beta$
  - se Z è B, allora anche $\alpha q \vdash \delta Yp$
- Se δ(q,Z) = (p,Y,L) allora:
  - $\alpha qZ\beta \vdash \gamma pXY\beta$ dove $\alpha = \gamma X$ oppure $\alpha = \gamma = \epsilon$ e X=B
  - se Z è B, allora anche $\alpha q \vdash \gamma pXYp$

Una TM M definisce un linguaggio per stato finale:
L(M) = { w — $q_0 w \vdash *\alpha\mathbf{q}\beta$ con q finale}
Se w non è in L(M) allora M o si blocca o non finisce mai. Tali linguaggi vengono detti: **linguaggi ricorsivamente enumerabili**. Se L = L(M) per una qualche Tm M che è un algoritmo, diciamo che L è un linguaggio ricorsivo.
Ogni CFL è un linguaggio ricorsivo, inoltre ogni linguaggio che data una stringa si può vedere se essa appartiene al linguaggio è anche lui ricorsivo.
Ci sono alcuni trucchi per rendere le TM più potenti:

- tracce multiple: ho una serie di tracce parallele, l'input ha una fila di blank e un solo simbolo, i simbolo del nastro sono un vettore pieno di simboli e ogni blank è un vettore blank
- marcatura: posso usare un simbolo speciale per marcare delle celle
- stato vettore: anche lo stato può essere un vettore, il primo elemento è lo stato il resto è la nostra memoria

Esempio di applicazione: una TM che copia l'input infinite volte.
Stati di controllo:

- q: marca la posizione e memorizza il simbolo letto
- p: va a destra, quando trova un blank scrive il simbolo in memoria
- r: va a sinistra fino alla marcatura
- ogni stato ha una forma [x,Y] dove:
  - x è q, p oppure r
  - Y è 0, 1 oppure B (0 e 1 solo quando x è p)

I simboli del nastro hanno forma [U,V] dove:

- U è X (marcatura) oppure B
- U è 0, 1 (input) oppure B

  B,B è il blank; [B,0] e [B,1] sono gli input

La funzione di transizione è definita come segue:

- a e b sono simboli di input (0 e 1)
- δ([q,B], [B,a]) = ([p,a], [X,a],R)
  - nello stato q copia a, il simbolo sotto la testina
  - marca la posizione letta
  - va allo stato successivo
- δ([p,a], [B,b]) = ([p,a], [B,b],R)

  - nello stato p va a destra in attesa di un blank (deve essere blank sia la marcatura che il simbolo sopra)

- δ([p,a], [B,B]) = ([r,B], [B,a],L)
  - quando trovo B, lo rimpiazzo con il simbolo in memoria, ovvero a
  - entra nello stato r e va a sinistra
- δ([r,a], [B,a]) = ([r,B], [B,a],L)
  - in r cerco la marcatura verso sinistra
- δ([r,B], [X,a]) = ([q,B], [B,a],R)
  - quando trovo la marcatura entro in Q e vado a destra
  - rimuovo la marcatura
  - q inserirà la marcatura e si ripete il ciclo

Invece che avere nastro infinito in entrambe le direzione, possiamo avere due tracce. Uno conteggia le posizioni a destra 1,2, ... e l'altro le posizioni a sinistra -1,-2,... questo è il nastro semi infinito.
Inoltre invece che avere un nastro possiamo usare due stack, al posto delle due tracce. Quindi una TM è un PDA ma con due stack.
Possiamo inoltre avere k nastri e k testine, le mosse dipendono dallo stato e dal simbolo di ogni testina di ogni nastro.
Ogni nastro è una traccia. C'è una traccia per la posizione della testina. La TM accetta l'input se esiste una possible sequenza di scelte che porta a uno stato accettante.
Infine posso fare una DTM che tiene conto delle ID di una NTM, scrivendo solo le ID possibile. Se troviamo un ID accettante allora termina l'esecuzione.

I linguaggi ricorsi sono chiusi rispetto a: unione, intersezione, concatenazione, stella di Kleene e reverse.Sono ricorsivi chiusi rispetto a complemento e differenza. Sia $L_1 = L(M_1)$ e $L_2 = L(M_2)$, assumiamo che M_1 e M_2 siano TM

con un solo nastro semi infinito. Costruiamo una TM con 2 nastri che copia l'input su entrambi i nastri e accetta se entrambe le TM accettano: unione.
Nel caso del complemento possiamo determinare se un linguaggio è ricorsivo,

ovvero quando le due macchine, una che riconosce un L RE e l'altra il suo complemento. Quindi esce fuori uno stato accettante e uno che rifiuta.
La differenza tra L_1 e L_2 è l'intersezione di L_1 con il complemento di L_2. I linguaggi ricorsivi sono chiusi rispetto a entrambe le operazioni.
Nella concatenazione ho due nastri non deterministici. Suddivido l'input w=xy e sposto y sul secondo nastro. Se entrambi i nastri accettano allora accetto. Nel caso ricorsivo devo ripetere l'operazione per ogni x e y e accettare se almeno una suddivisione accetta.
La stella di Kleene è un caso particolare della concatenazione. RE scommette su una suddivisione mentre i ricorsivi provano ogni suddivisione.
Per il reverse inverto l'input e simulo L sul input invertito, con entrambe i linuaggi.

20.1 Decidibilità

È possibile ordinare tutte le stringe finite su un alfabeto. Siccome sono infinite esiste una funzione biunivoca da numeri naturali a stringhe. Possiamo fare lo stesso con le TM. È una tabella stringa-TM che ha 1 accetta e 0 non accetta.
Questa tabella è diagonalizable, ovvero costruiamo un vettore D = a_1, a_2... con i termini invertiti. D è un linguaggio che nessuna TM conosce.
Il linguaggio di diagonalizzazione è L_d = {w | w è la string i-esima, e la i-esima TM non riconosce w }. Questo linguaggio non ha impatto sul mondo reale.
UTM è il linguaggio della Macchina di Turing Universale. (se servisse si approfondisce in un secondo momento)
Non esiste un linguaggio senza computazioni non terminanti.
Teorema di rice: L_p è ricorsivo se P è sempre oppure mai soddisfatta.

21 Compilatore di Function and Object Oriented Language

Sviluppo di un compilatore per il Function and Object Oriented Language (FOOL). La sintassi c'è nel file FOOL.g4. I metodi di visita ci sono in BaseASTVisitor.java.
È il linguaggio che abbiamo fatto in laboratorio, ma con operatori aggiuntivi e OOP. Il compilatore produce codice per la SVM sviluppata a lezione (senza necessità di modifica).
C'è da implementare lo heap, ma niente garbage collection.
Sintassi astratta:

```
class ID1 [extends ID2] ( campi dichiarati come
   parametri) {
    metodi dichiarati come funzioni
}

// Utilizzo
ID1.ID2()
new ID()
null
```

La classe è di tipo ID.
Sintassi:

```
let
    class A (a:int, b:bool) {
        fun n:int(...) ... ;
        fun m:bool(...) ... ;
    }
in
```

a e b sono parametri. Gli oggetti una volta creati sono immutabili. I campi sono accessibli solo da dentro la classe o suo figlio. I metodi del padre sono invocabili dal figlio. O anche dal esterno.
Sintassi ereditarietà:

```
    let
        class A (a:int, b:bool) {
            fun n:int(...) ... ;
            fun m:bool(...) ... ;
        }
        class B extends A (c:int) {
            fun l:int(...) ... ;
        }
    in

    // nuovo oggetto con a=5, b=true e c=7
    new B(5, true, 7)
```

Overriding di campi.

```
    let
        class A (a:int, b:bool) {
            fun n:int(...) ... ;
            fun m:bool(...) ... ;
        }
        //override di a
        class B extends A (c:int, a:bool) {
            fun l:int(...) ... ;
        }
    in

    // nuovo oggetto con a=false, b=true e c=7
    new B(false, true, 7)
```

Layout oggetti nel heap:

PRIMA POSIZIONE LIBERA HEAP
dispatch pointer
valore primo campo dichiarato
.
.
valore ultimo (n-esimo) campo Layout oggetti nel heap:

PRIMA POSIZIONE LIBERA HEAP

addre ultimo metodo
.
.
addr primo metodo dichiarato

L'AR funziona uguale, mentre il suo access link è relativo a "this" ovvero la classe.

21.1 AST

Dichiarazioni:

- ClassNode: salvare i campi figli in fields e metodi in "methods", mentre in "superID" ci va l'ID della classe padre oppure niente
- FieldNode: come ParNode
- MethodNode: come FunNode

I nodi ora eridato da DecNode per poter scoprire il tipo con getType()
Espressioni:

- IdNode ID
- CallNode ID()
- ClassCallNode ID.ID()
- NewNode new ID()
- EmptyNode null

Il nostro type checking ti ritorna RefTypeNode (ID) oppure EmptyTypeNode (null).
Il nesting level globale è 0, da fare -2 ogni volta che dichiaro una classe. ClassTypeNode ha come campi, ArrayList⟨TypeNode⟩ allFields e ArrayList⟨ArrowTypeNode⟩ allMethods. Entrambi in ordine di apparizione e inclusi ereditati.
Abbiamo anche una virtual symbol table. Ogni volta che definisco una classe devo creare una symbol table per salvare i simboli ereditati che non sono stati overloadedati.
Abbiamo anche una Class Table che mappa ogni nome di classe nella propria

Virtual Table: Map⟨ String, Map ⟨ String, STentry ⟩⟩ classTable. Serve per rendere accessibile le dichiarazione anche dopo aver dichiarato la classe (ID1.ID2()).
Quando si dichiara una classe salvo il nome nella Symbol Table livello 0, se non eredita niente ha fields e calls vuoti altrimenti copia tutto il contenuto del ClassTypeNode. Nella class table aggiungo il nome della classe la sua virtual table, come prima vuota se non eredita altrimenti la copio tutta dal padre.
Quando entro nella classe ogni volta che incontro un campo o un metodo aggiorno Vitual Table e oggetto ClassTypeNode.

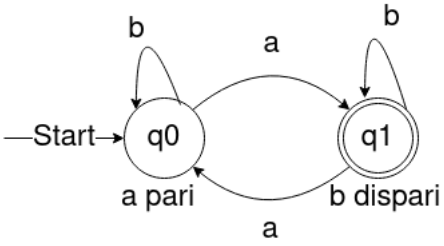
22 Esercitazioni

22.1 Esercitazione 21/09/23

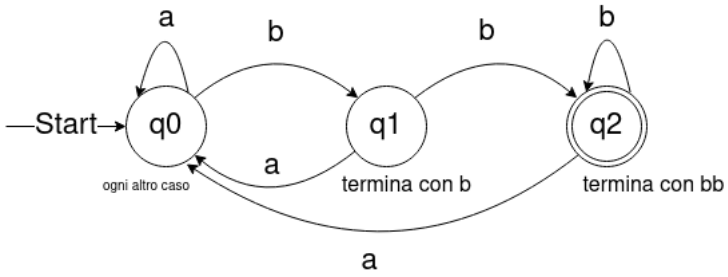
22.1.1 Costruzione DFA

Consideriamo l'alfabeto {a,b}. Realizzare dei DFA che riconoscono i seguenti linguaggi:

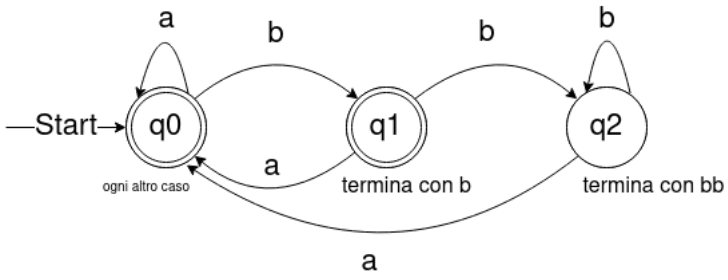
1. stringhe con un numero dispari di a
   SI ab,aaa,bba,aaba
   NO ϵ,aa



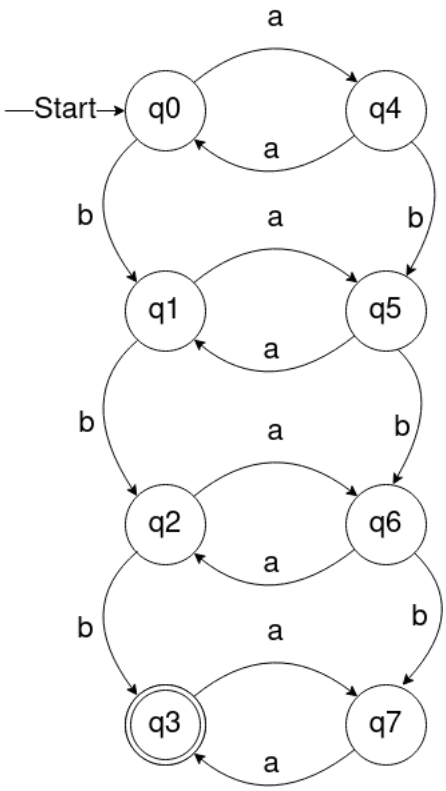
2. stringhe che terminano con bb
   SI bb,babb
   NO ϵ,ba,a,aba



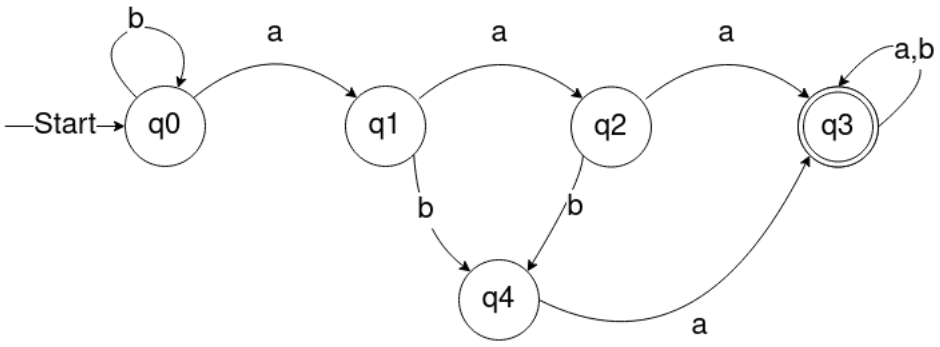
3. stringhe che non terminano con bb
   SI ϵ,ba,a,aba
   NO bb,babb



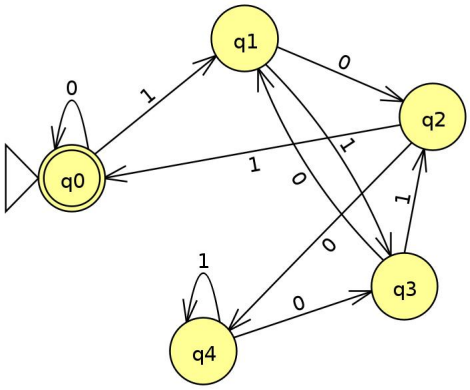
4. stringhe con un numero pari di a ed almeno 3 b
   SI bbb,bababb
   NO ϵ,bbaa,bababa



5. stringhe che contengono la sottostringa aaa o la sottostringa aba (contengono almeno una delle due)
   SI babab,aaaa,aaaba
   NO ϵ,abba,a,b,ab

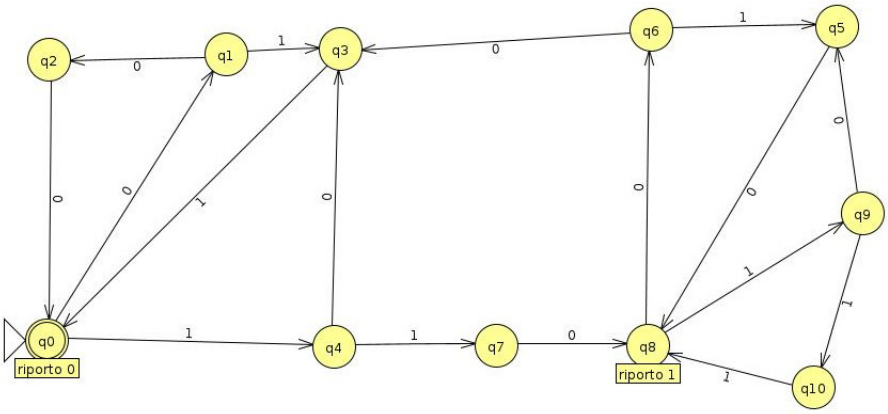


6. Realizzare un DFA che riconosca il seguente linguaggio su alfabeto {0,1}: stringhe che interpretate come numero binario risultano un multiplo di 5
   SI 101,1010,1111,0
   NO 111,1,10



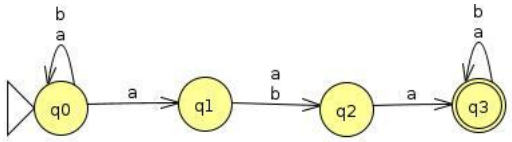
NB: Devi considerare tutti i numeri fino a 5!

7. Sempre considerando alfabeto {0,1}, realizzare un DFA che controlla la correttezza delle somme binarie: data la stringa: $a_0b_0c_0a_1b_1c_1...a_nb_nc_n$ controlla se $a_n...a_1a_0+b_n...b_1b_0 = c_n...c_1c_0$ (cioè a+b=c con a,b,c numeri binari con stessa lunghezza)

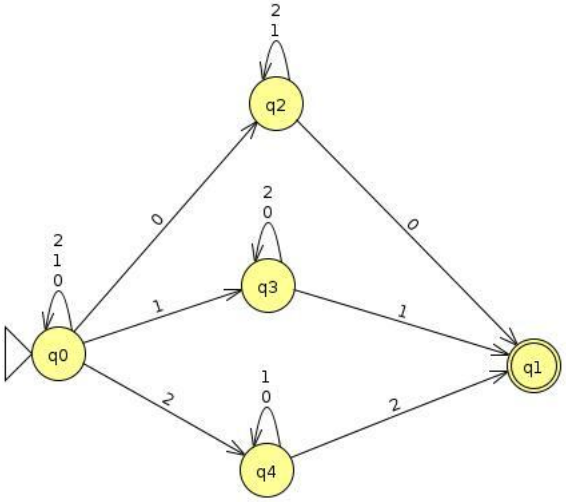


22.1.2 NFA

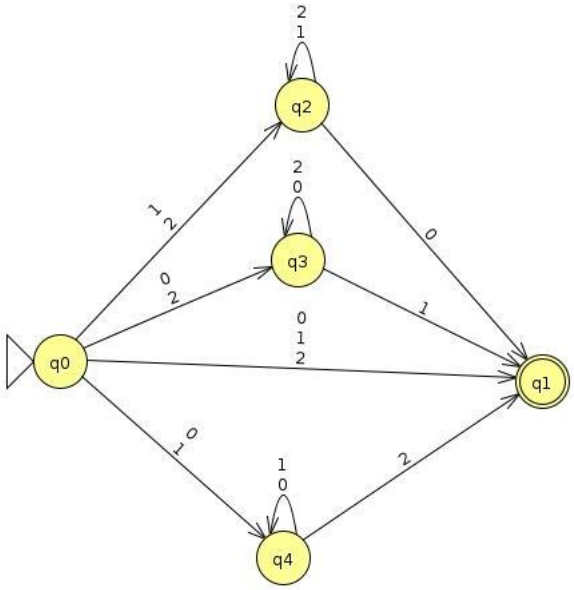
1. Dato l'alfabeto {a,b} realizzare un NFA che riconosce le stringhe che contengono aaa oppure aba
   SI babab,aaaa,aaaba
   NO ϵ,abba,a,b,ab



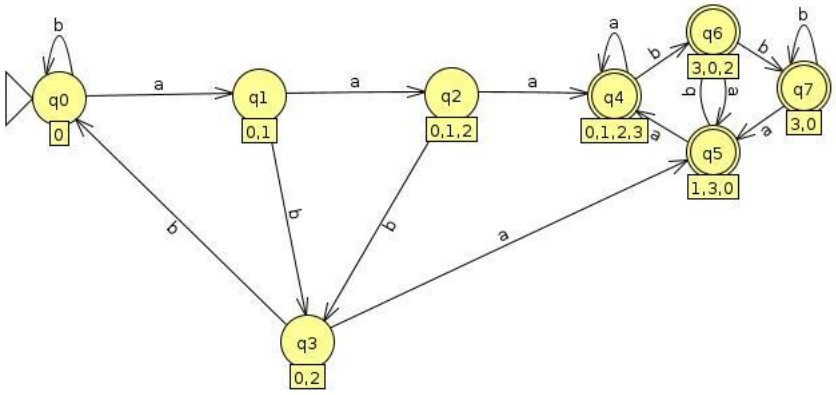
2. Realizzare un NFA che riconosce le stringhe non vuote sull'alfabeto {0,1,2} in cui l'ultima cifra appare almeno una volta in precedenza
   SI 011,121,22,0120
   NO ϵ,012,20,1



3. Realizzare un NFA che riconosce le stringhe non vuote sull'alfabeto 0,1,2 in cui l'ultima cifra NON appare in precedenza
   SI 012,20,1
   NO ϵ,011,121,22,0120



4. Dato l'alfabeto a,b si consideri l'NFA fatto all'esercizio 1, che riconosce le stringhe che contengono aaa oppure aba. Trasformarlo in DFA.



22.2 Esercitazione 28/09/23

22.2.1 Pumping lemma

Mostrare che i seguenti linguaggi non sono regolari:

1. Linguaggio sull'alfabeto {a,b} delle stringhe $a^h b^m$ con $m \geq 2h$ **Soluzione**
   Si supponga per assurdo che L sia regolare.
   Deve quindi esistere un DFA A tale che $L = L(A)$.
   Quindi, chiamato n il numero degli stati di A, si ha che:

   $\exists n >= 1 : \forall w \in L, |w| \geq n$, w può essere scomposta in $w = xyz$ con $|y| > 0, |xy| \leq n$, $xy^k z$ appartiene ad L per ogni $k \geq 0$
   (PUMPING LEMMA)

   Considero la stringa $w = a^n b^{2n}$, e decido che la stringa y è composta da sole a e almeno una. Prendiamo $k > 2$ allora la stringa risultate è $z = xyyz$, li numero delle a è aumentato ma il numero delle b no, paradosso. Si può concludere che $w \notin L$ quindi L non è regolare.

2. Linguaggio delle parentesi tonde bilanciate; cioè, delle stringhe, sull'alfabeto composto dai due simboli "(" e ")", che sono espressioni Exp fatte come segue.
   Una Exp e' nella forma: (Exp1) oppure Exp1 Exp2 oppure ϵ
   Esempi di stringhe del linguaggio:
   ()()((()))
   (()())
   (())((()))
   Esempi di stringhe non del linguaggio:
)(
   (()

   Formalmente dimostro che $L = \{"(", ")"\}$ non è regolare:
   Si supponga per assurdo che L sia regolare.
   Deve quindi esistere un DFA A tale che $L = L(A)$.
   Quindi, chiamato n il numero degli stati di A, si ha che:

   $\exists n >= 1 : \forall w \in L, |w| \geq n$, w può essere scomposta in $w = xyz$ con $|y| > 0, |xy| \leq n$, $xy^k z$ appartiene ad L per ogni $k \geq 0$
   (PUMPING LEMMA)

Prendiamo la stringa $w = ({}^{n})^{n}$, e decido che y è composta da "(" e che $k > 2$. Allora la stringa $z = xyyz$ ma così ho più "(" che ")", assurdo.

Si può concludere che $w \notin L$ quindi L non è regolare.

3. Linguaggio sull'alfabeto binario delle stringhe del tipo xy dove y coincide con il complemento di x (0 diventano 1, 1 diventano 0).
   Sinceramente il testo mi confonde e alla fine dei conti è uguale alle 2 dimostrazioni precedenti, quindi non la metto.

4. Linguaggio sull'alfabeto binario delle stringhe del tipo ww

   Formalmente dimostro che $L = \{w^2 n\}$ non è regolare:
   Si supponga per assurdo che L sia regolare.
   Deve quindi esistere un DFA A tale che $L = L(A)$.
   Quindi, chiamato n il numero degli stati di A, si ha che:

   $\exists n >= 1 : \forall w \in L, |w| \geq n$, w può essere scomposta in $w = xyz$ con $|y| > 0, |xy| \leq n$, $xy^k z$ appartiene ad L per ogni $k \geq 0$
   (PUMPING LEMMA)

   Prendiamo la stringa $s = w^2 n$, e decido che y è composta da w e che $k > 0$. Allora la stringa $z = xyz$ ma così ho più w dispari.

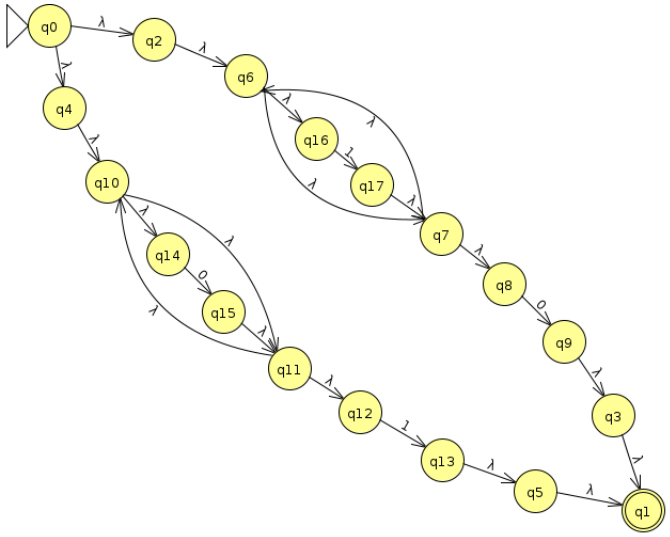
   Si può concludere che $s \notin L$ quindi L non è regolare.

22.2.2 Espressioni regolari

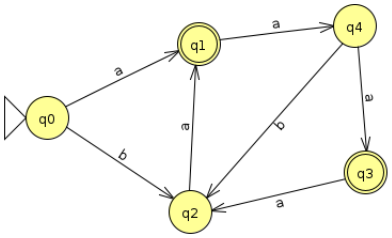
1. Considerare l'espressione regolare 0*(0+11)1*. Dire quale linguaggio rappresenta.

$$L = \{0^n \times 1^m | n \geq 0, m \geq 0, x \in \{0, 11\}\}$$

2. Considerare l'espressione regolare 1*0+0*1. Trasformalo in un ϵ-NFA.



3. Trasformare in RE il seguente automa



(a+ba)(aba+aaaa)*+(a+ba)(aba)*aa(aa(aba)*aa)*

È una banale applicazione della formula

4. 5 Si consideri l'alfabeto {a,b}.

Definisci un RE che accetta un numero pari di a seguito da un numero di b che diviso per 3 da' resto 2

(aa)*+bb(bbb)*

22.3 Esercitazione 05/10/23

22.3.1 CFG

1. Definire una grammatica libera dal contesto (CFG) per i seguenti linguaggi sull'alfabeto {a,b,c}:

   (a) linguaggio delle stringhe $a^n\ b^{2n}$ (mostrare che "aabbbb" è derivabile dalla CFG)
   $S \to aSbb$
   $S \to \lambda$

   (b) linguaggio delle stringhe $(abc)^n\ (cba)^n$
   $S \to abcScba$
   $S \to \lambda$

   (c) linguaggio delle stringhe $a^n b^m$ in cui $n \leq m$
   $S \to aSb$
   $S \to Sb$
   $S \to \lambda$

   (d) linguaggio delle stringhe $a^n b^m c^n$ con n pari e m dispari
   $S \to aaScc$
   $S \to bA$
   $A \to bbA$
   $S \to \lambda$

   (e) linguaggio delle stringhe $a^n b^m c^p$ in cui $n \leq m + p$
   $S \to aSc$
   $S \to Sc$
   $B \to \lambda$
   $S \to B$
   $B \to aBb$
   $B \to Bb$

2. Dire se la CFG del punto "e" dell'esercizio 3 è ambigua o meno. In caso affermativo mostrare una stringa generata dalla CFG che abbia (almeno) due alberi sitattici/derivazioni canoniche sinistre. Mostrare tali alberi sintattici e le corrispondenti derivazioni canoniche sinistre.

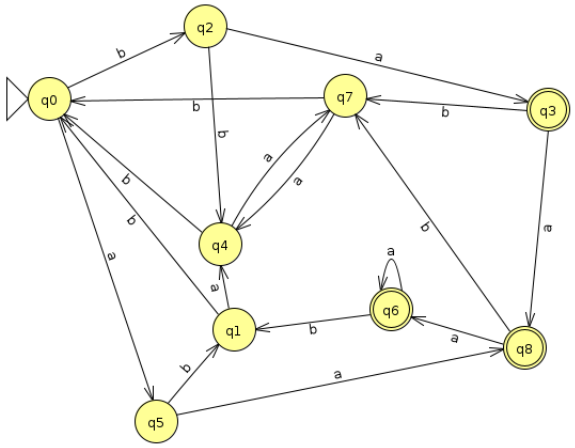
È ambigua, ho due derivazioni di abc

$S \to aSc$ $\qquad$ $S \to Sc$

$S \to aBc$ $\qquad$ $S \to aBbc$

$S \to abc$ $\qquad$ $S \to abc$

3. Esercizio su minimizzazione DFA. Effettuare (a mano con l'algoritmo riempi-tabella visto a lezione) la minimizzazione dell'automa in exMinimization.jff



	q_0	q_1	q_2	$*q_3$	q_4	q_5	$*q_6$	q_7	$*q_8$
q_1	+								
q_2	o	o							
$*q_3$	x	x	x						
q_4	o		o	x					
q_5	o	o		x	o				
$*q_6$	x	x	x		x	x			
q_7	o		o	x		o	x		
$*q_8$	x	x	x		x	x			

Tabella 10: Tabella riepita

(a) x: primo giro, distinguo gli stati

(b) o: secondo giro, trovo lo stato di ogni copia con entrambi i simboli. Se lo stato trovato è una x allora metto una o

(c) +: terzo giro, ripeto il 2