UPPSALA UNIVERSITET

Föreläsningsanteckningar

Automatateori

Rami Abou Zahra

Inlämningsdatum
October 6, 2022

Contents

1. Mål med kursen

Målet med kursen är att studera matematiska modeller för beräkningar/beräkningsbarhet
På detta vis vill vi kunna förstå vilka problem som överhuvudtaget är beräkningsbara samt nå en förståelse om hur svåra olika beräkningsbara problem är att lösa. Vi vill även bli bekanta med matematiska modeller för beräkningar som tillämpas inom datavetenskap och andra ämnen. Vi kommer även kika lite på användbara algoritmer inom programkonstruktion.

1.1. **Några tillämpningar av teorin.**

De bästa metoderna för översättning av hög-nivå språk såsom Python till maskinkod bygger på teorin om *sammanhangsfria grammatiker*

Det visar sig även att data som följer ett visst mönster kan metoder som är beroende på *ändliga automater* och *reguljära uttryck* användas för att exempelvis söka i datan/databaser.

Det går även att tillämpa teorin för att bestämma om ett konkret problem ens är lösbart algoritmiskt (dvs beräkningsbar)

Fler tillämpningsområden:

- Lingvistik
- Bio-informatik

2. Alfabet och Strängar

2.1. **Terminologi.**

Definition/Sats 2.1: Alfabet

En ändlig (icke-tom) mängd av tecken/symboler. Brukar betecknas stora sigma (Σ)

Exempel:

Mängden $\{0, 1\}$ är ändlig, icke-tom, och består av tecken/symboler. Alltså är det ett korrekt alfabet enligt vår definition

Mängden $\{\bot, \vdash, \wedge\}$ är ändlig, icke-tom, och består av symboler. Alltså är även detta ett korrekt alfabet

Mängden $\mathbb{N}$ däremot, är ej ändlig, den är icke-tom och består av tecken/symbler, men ej ändlig. Alltså är de naturliga talen ej ett korrekt alfabet.

Definition/Sats 2.2: Sträng

En sträng definieras som en kombination av karaktärerna i alfabetet. En intuitiv fråga som kanske dyker upp är om det finns något supremum (max-längd) på strängarna och svaret är nej. Vi ska kika mer på det snart

Låt $\Sigma = \{0, 1\}$ vara ett alfabet. Då är 0 en valid sträng, men även 1 och 01 och 00 och 11 och $\cdots$

I definitionsrutan sade vi att strängarna kan bli hur stora som helst, och eftersom vi bara kan lägga till fler symboler från vårt alfabet in i strängen så kan mängden av strängar bli oändligt stor.
Men, hur oändligt stor?

Definition/Sats 2.3: Uppräknelig mängd

En mängd A kallas för en **uppräknelig mängd** om det finns en *surjektiv* funktion $f : \mathbb{N} \to A$

Mycket ord att ta in, vi påminner oss om vad surjektion betyder:

Definition/Sats 2.4: Surjektiv

En **surjektiv** funktion är en funktion vars målmängd träffas helt.
Alltså, givet en avbildning (funktion) $f : A \to B$ kommer funktionen att evalueras till alla tal i mängden B. Detta betyder inte nödvändigtvis att hela A används

Ett exempel på en surjektiv funktion är en funktion $f(x)$ så att givet 2 olika x-värden ges samma y-värde. $f(x) = x^2$ är en sådan funktion

Anmärkning:
Om en mängd A är uppräknelig så kan A skrivas som $A = \{a_n : n \in \mathbb{N}\}$

Anmärkning:
Om en mängd A är ändlig så är A uppräknelig. Exempelvis är vår mängd Σ alltid ändlig och därmed alltid uppräknelig.

Kuriosa/Anmärkning:
Varje program kan betraktas som en ändlig sträng över alfabetet $\{0, 1\}$, alltså är mängden av program uppräknelig!

Påstående:
Mängden av funktioner $g : \mathbb{N} \to \{0, 1\}$ är *inte* uppräknelig

Proof 2.1: Bevis av påstående

Antag för motsägelse att mägnden av funktioner $g : \mathbb{N} \to \{0, 1\}$ är uppräknelig.
Då kan vi göra en lista $g_0, g_1, g_2, \cdots$ av alla sådana funktioner.
Defniera nu en funktion h enligt följande:

$$h(n) = \left.\begin{array}{l} 0 \text{ om } g_n(n) = 1 \\ 1 \text{ om } g_n(n) = 0 \end{array}\right\}$$

Då gäller att $h : \mathbb{N} \to \{0, 1\}$, alltså finns h med i mängden av funktionerna.
Men! Enligt definition av h så finns det inte en funktion g_n så att $g_n = h$ (vi tar ju motsatt värde till g_n) för varje $n \in \mathbb{N}$, men detta betyder att h *inte* finns med i lista, vilket är en motsägelse □

Definition/Sats 2.5

Det finns någon funktion $g : \mathbb{N} \to \{0, 1\}$ som *inte* kan beräknas av något program

Proof 2.2

Låt A vara mängden av funktioner $g : \mathbb{N} \to \{0, 1\}$ och låt P vara mängden av program.
Antag att för varje $g \in A$ så finns $p_g \in P$ som beräknar g.
Låt $g' \in A$ vara godtycklig funktion ur mängden A.

Eftersom P är uppräknelig så finns en surjektiv avbildning $f : \mathbb{N} \to P$.
Men då kan vi definiera en surjektiv funktion $h : \mathbb{N} \to A$ genom att låta $h(n)$ vara g om $f(n) = p_g$ för något $g \in A$ och g' annars

Detta motsäger att A inte var uppräknelig. □

2.2. **Operationer på strängar.**

Låt $v = a_1 \cdots a_n$ och $w = b_1 \cdots b_n$ vara strängar. Vi vill, likt hur man definierar plus och gånger med tal, definiera operationer för strängar:

Definition/Sats 2.6: Sammanfogning

Sammanfogning (sträng1 + sträng2) definieras enligt följande: $vw = a_1 \cdots a_n b_1 \cdots b_n$
Notera! Ordning spelar roll ($a_1 b_1 \neq b_1 a_1$)

Definition/Sats 2.7: Repetition n-gånger

Detta fungerar ungefär som att ta (sträng)^n:
$v^n = \underbrace{vvv \cdots v}_{n \text{ gånger}}$

Definition/Sats 2.8: Reversering

Här vill vi härma ”ta ordet baklänges” fast med strängar, alltså:
$v^{rev} = a_n \cdots a_1$ istället för $v = a_1 \cdots a_n$

Givetvis finns det strängar där $v = v^{rev}$, låt $v =$ racecar och se vad som händer om vi reverserar den!

En ganska central grej med både addition och multiplikation inom matematik är ”enheten”, det vill säga 1-1 = 0 (där 0 är enheten för addition) och $2 \cdot \frac{1}{2} = 1$ (där 1 är enheten för multiplikation).
Vi vill försöka härma enheten, det vill säga, finna en sträng sådant att den inte påverkas av operationerna vi tidigare har definierat.
En egenskap hos dessa enheter som vi vill bevara är deras ”längd”, tänk exempelvis enhetsvektorn som har längd 1 för att åstadkomma att vi ej förflyttar oss. Med addition och multiplikation får vi tänka oss tallinjen, 0 i addition betyder att vi inte rör oss i tallinjen på samma sätt som 1an gör med multiplikation.

Definition/Sats 2.9: Längd av sträng

Längden av en sträng $v = a_1 \cdots a_n$ betecknas $|v| = n$ och betyder ”hur många tecken från alfabetet Σ har jag använt?”

Definition/Sats 2.10: Tomma strängen

Vi definierar den **tomma strängen** (ε)
Under operationer beter sig ε på följande sätt:

$$\varepsilon^{\text{rev}} = \varepsilon$$
$$\varepsilon\varepsilon = \varepsilon$$
$$v\varepsilon = v = \varepsilon v$$

Varför kallas den för tomma strängen om det är en enhet? Jo! Vi testar att evaluera längden av ε:

$$|\varepsilon| = 0$$

Det som kan vara ointuitivt är att den tomma strängen *inte* finns i något alfabet omm (om och endast om) det inte specifieras i definitionen, men ε den är en valid sträng i alla alfabet.
Den beter sig lite som tomma mängden, i den att följande gäller för en godtycklig mängd A:

$$\emptyset \notin A \qquad \varepsilon \notin \Sigma$$
$$\emptyset \subseteq A \qquad \varepsilon \subseteq \Sigma$$

Låt x, y vara strängar, då gäller följande:

Definition/Sats 2.11: Prefix

Vi säger att x är en **prefix** till y om det finns en sträng z så att $xz = y$

Exempel:
Låt $y =$ sportbil, $x =$ sport, då är $z =$ bil och ”sport” (x) är prefixet.

Definition/Sats 2.12: Suffix

Vi kallar x för **suffix** till y om det finns en sträng z så att $zx = y$

Exempel:
I föregående exempel med strängen ”sportbil”, så är ”bil” suffixet till ”sportbil”.

3. Språk

Vi har definierat vårt alfabet (exvis siffror), bildat strängar m.h.a operationer (operationer med tal). Kombinerar vi dessa bör vi rimligtvis få ut något som påminner om vektorrum, det vill säga en mängd med tal med några operationer (i vårat fall är mängden tal = mängden strängar, och operationerna de operationer vi definierat överst).

Definition/Sats 3.1: Språk

Ett **språk** över ett alfabet Σ är en mängd strängar över Σ

Några speciella språk:

- $\emptyset$ (tomma mängden)
- $\{\varepsilon\}$
- Σ
- Σ^* (mängden av alla strängar över Σ)

Med liknande resonemang som tidigare kan man visa att:

- Σ^* är uppräknelig
- Mängden av alla språk över Σ är inte uppräknelig

3.1. Operationer på språk.

Låt Σ vara ett alfabet och låt L, L_1, L_2 vara språk över Σ.
Nya språk (över Σ) kan bildas genom:

- Union $L_1 \cup L_2$
- Snitt $L_1 \cap L_2$
- Differens $L_1 - L_2$
- Komplement i Σ^* är $\Sigma^* - L$
- Sammanfogning $L_1L_2 = \{wv : w \in L_1 \wedge v \in L_2\}$
- Repetition:
  - $L^0 = \{\varepsilon\}$
  - $L^{n+1} = L^nL$

3.2. Kleenestjärnatillslutning.

Vi har använt notationen "upphöjt till en stjärna" för att på sätt och vis indikera mängden av alla kombinationer av element. Exempelvis har vi använt Σ^* för att beteckna alla möjliga konstruerbara strängar i ett alfabet.
Av dessa strängar kunde vi skapa språk, detta betecknades med L, och med dessa språk hade vi operationer, precis som vi hade operationer med våra strängar där vi kunde konstruera nya språk givet andra språk, då är frågan om vi kan konstruera mängden av alla språk!

Kleenestjärnatillslutning går ut på följande princip:

$$\sum_{i=0}^{\infty} x^i = x^0 + x^1 + x^2 + x^3 \cdots$$

Där vi på något sätt vill sammanfoga alla språk L^i. Detta gör vi på följande sätt:

$$L^* = \{w : w \in L^n \text{ för något } n \in \mathbb{N}\} = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$$

L^* uttalas "L-stjärna". Notera att vi har med mängden ε eftersom vi har L^0, vi kan välja att omittera denna genom att använda L-plus istället (L^+).

Exempel:

Låt $\Sigma = \{a, b\}$, $L_1 = \{aa\}$ och $L_2 = \{aa, bb\}$. Då gäller följande:

$$(L_1)^* = \{(aa)^n : n \in \mathbb{N}\} = \{\varepsilon, aa, aaaa, aaaaaa, \cdots\}$$
$$(L_1)^+ = \{aa, aaaa, aaaaaa, \cdots\}$$
$$(L_2)^* = \{\varepsilon, aa, bb, aaaa, aabb, bbaa, bbbb, \cdots\}$$

4. Reguljära språk och uttryck

Låt Σ vara ett alfabet. Vi definierar följande:

Definition/Sats 4.1: Reguljära uttryck

- $\emptyset$ är ett reguljärt uttryck för språket $\emptyset$
- ε är ett reguljärt uttryck för språket $\{\varepsilon\}$
- För varje $\sigma \in \Sigma$ så är σ ett reguljärt uttryck för språket $\{\sigma\}$
- Om α och β är reguljära uttryck för språken L_1 och L_2 så är följande även reguljära uttryck för språken $L_1 \cup L_2$, $L_1 L_2$, L_1^* och L_1^+:
  - $(\alpha \cup \beta)$
  - $(\alpha\beta)$
  - (α^*)
  - (α^+)

Definition/Sats 4.2: Reguljära språk

Om L är ett språk och det finns ett reguljärt uttryck för L så säger vi att L är reguljärt

Exempel:
Låt $\Sigma = \{0, 1\}$
Då är $(0 \cup (1(0^*)))$ ett reguljärt uttryck för språket $L = \{0\} \cup (\{1\}(\{0\}^*)) = \{0\} \cup \{10^n : n \in \mathbb{N}\}$
Eftersom det finns ett reguljärt uttryck för L, så är även språket L reguljärt i detta fall.

Anmärkning:
För notationens enkelhet så reducerar vi paranteser enligt följande paranteskonvention:

- Först * eller $^+$
- Sedan Sammanfogning
- Sist union

Jämför med paranteskonvention för aritmetiska uttryck som ser ut på följande vis:

- Först exponent
- Sedan multiplikation
- Sedan addition

Exempel:
Vi skriver $0 \cup 10^*$ istället för $(0 \cup (1(0^*)))$
(Notera! Det är inte siffran tio, utan siffran ett sammanfogat med noll).

5. Deterministiska finita (ändliga) automater [DFA]

Förr i tiden programmerades datorer genom att man hade ”punch-cards”, det vill säga kort av styvt papper som hade ”hål” som agerade som dagens transistorer gör.
DFA:er påminner lite om hur dessa fungerar, det vill säga att vi ska tänka oss en slags textremsa som inmatning:

a	b	a	b	b	$\cdots$

Där ändligt många inledande rutor innehåller tecken från en sådan textremsa med alfabet Σ.

Vi har också en ”kontrollmekanism”, det vill säga själva DFA:n som vi kan tänka oss har ett ”läshuvud” som avläser en ruta i taget (det är viktigt att precisera, DFA:er läser EN ruta i taget).
DFA:n befinner sig alltid i ett tillstånd, av ändliga antal möjliga. När den avläser en ruta så övergår den till ett nytttillstånd samt flyttar läshuvudet ett steg till höger.
Det nya tillståndet beror (endast) på det tidigare tillståndet samt det just avlästna tecknet (i föregående ruta).

De två ”viktigaste” tillstånd kallas för **starttillståndet** och **accepterande tillståndet**.
Vi antar att en DFA alltid befinner sig i starttillståndet då den sätts igång.
Om en DFA befinner sig i ett accepterande tillstånd då en sträng har avlästs (det vill säga, läshuvudet befinner sig på den första blanka rutan till höger om strängen) så säger vi att DFA:n **accepterar** strängen.
Om strängen inte accepteras, så avvisas strängen.

5.1. **Grafisk beskrivning av en DFA.**

Vi väljer att representera tillstånden med hjälp av noder, och tillståndsövergångar med pilar:

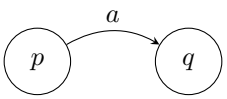


Figure 1.

Figur 1 betyder att om DFA:n befinner sig i tillstånd p och avläser a på textremsan så övergår DFA:n till tillstånd q (och flyttar därmed läshuvudet ett steg till höger).

Anmärkning:
För varje tillstånd och varje tecken från input-alfabetet skall det finnas *preci* en utgående pil som bär detta tecken.

Starttillstånd markeras med en pil mot sig:

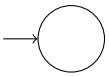


Figure 2.

Accepterande tillstånd markeras med en pil från sig *eller* dubbelrand:

Figure 3.

Exempel:

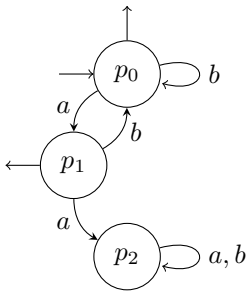


FIGURE 4.

Kom ihåg att en sträng accepteras om vi befinner oss i ett accepterande tillstånd då hela strängen är avläst.
Vilka strängar accepterar av ovanstående DFA? $((b \cup ab)^*(\varepsilon \cup a))$

Definition/Sats 5.1

Antag att M är en DFA.
Mängden av strängar som M accepterar kallas för M**:s språk** och betecknas $L(M)$

Definition/Sats 5.2: Formell definition av DFA

En **DFA** är en 5-tupel $M = (Q, \Sigma, \delta, s, F)$ där:

- Q är en ändlig mängd (tillståndsmängden)
- Σ är en ändlig mängd (inputalfabet)
- δ är en funktion (övergångsfunktion) från $Q\text{x}\Sigma \to Q$
- $s \in Q$ (s är Starttillstånd)
- $F \subseteq Q$ (accepterande tillstånden)

För att formellt kunna definiera *acceptans* av en sträng behöver vi en **utvidgad övergångsfunktion**. Denna kan vi löst definiera på följande vis:

Definition/Sats 5.3: Utvidgad övergångsfunktion

En funktion $T(\text{nuvarande tillstånd}, \text{nuvarande symbol på läshuvu}) \to \text{nästa tillstånd}$

6. Icke-Determiniska Finita (åndliga) automater [NFA]

En NFA är en generalisering av en DFA på följande sätt:

- En NFA kan ha flera Starttillstånd
- En NFA tillåts kunna läsa flera tecken på en gång, dvs vi kan tillståndsövergångar på formen:

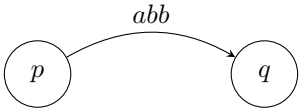


Figure 5.

- En NFA kan göra en tillståndsövergång utan att läsa något (läser tecknet ε)
- En NFA kan ha flera valmöjligheter i en given situation (icke-determinism), dvs vi kan ha följande situation:

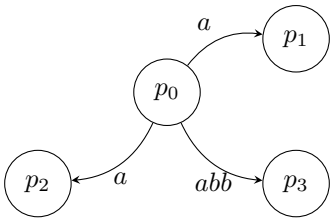


Figure 6.

- Det följer att en NFA (i allmänhet) kan ”hänga sig”

Exempel: (På en NFA)

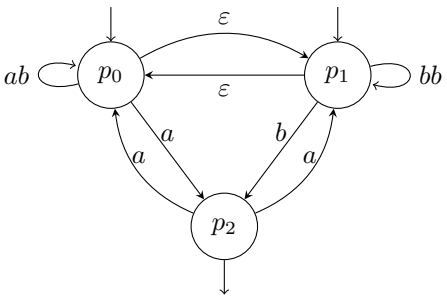


Figure 7.

Definition/Sats 6.1: NFA Acceptans

En NFA accepterar en sträng w om vi kan börja i något starttillstånd och gå från tillstånd till tillstånd genom att avläsa en del av strängen w och slutligen hamna i ett accepterande tillstånd. Det vill säga, om w kan delas upp i delsträngar $w = v_1 v_2 \cdots v_n$ där vi tillåter $v_k = \varepsilon$ *och* det finns tillstånd $p_0, p_1, \cdots, p_n$ där p_0 är ett starttillstånd och p_n är ett accepterande tillstånd samt följande övergångar:

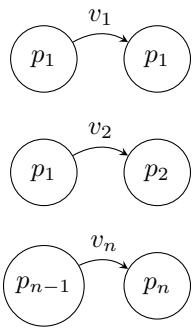


FIGURE 8.

Exempel:
Betrakta föregående exempel och strängarna:

$$aaa$$
$$aaaa$$
$$aabbaa$$
$$abbba$$

Vilka strängar accepteras av NFA:n?

Definition/Sats 6.2

Mängden av strängar som accepteras av en NFA M kallas för M:s språk, och betecknas precis som i DFA fallet med $L(M)$

6.1. **Omvandling av en NFA till en DFA som accepterar samma språk.**

Betrakta följande NFA:

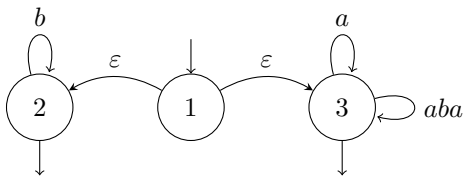


FIGURE 9.

Vi konstruerar först en "icke-glupsk" NFA M' som avlöser högst ett tecken i taget (försöker gå från generaliseringen tillbaks till en DFA) och accepterar samma strängar som M.
$M' =$

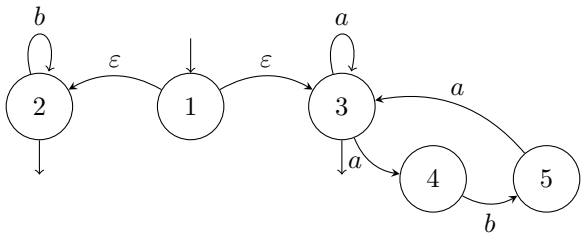


FIGURE 10.

Sedan använder vi den så kallade "delmängdsalgoritmen" för att konstruera en DFA M'' som accepterar samma strängar som M'.
M'':s starttillstånd är mängden av alla tillstånd i M' som kan nås uitan att avläsa några tecken alls.

I vårat fall når vi naturligtvis noden 1 när vi startar, men genom att använda den tomma strängen kan vi nå nod 2 och 3, alltså blir denna mängd $\{1, 2, 3\}$.
Notera att de element från vårat alfabet som vi använder är a och b, därför kommer vi nu följa var vi hamnar med just dessa två bokstäver.

Vi ställer oss frågan, vilka noder som nås från $\{1, 2, 3\}$ genom att *bara* avläsa a. Börjar vi i nod 1 kan vi ta ε till 3, loopa runt till 3 igen, och sedan nå nod 4 genom att avläsa a. Vi kan inte nå nod 5 och avläsa a för att gå tillbaks till nod 3, ty det kräver att vi avläser b. Mängden här blir alltså $\{3, 4\}$.

Vårat diagram ser just nu ut på följande:

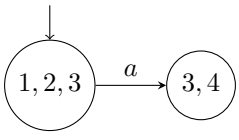


FIGURE 11.

Vi undersöker nu villka tillstånd vi kan nå genom att avläsa b. Vi kan avläsa ε och komma till nod 2, varpå vi kan avläsa b för att återigen komma till nod 2. Mängden blir därför $\{2\}$ och vårat diagram ser ut på följande sätt:

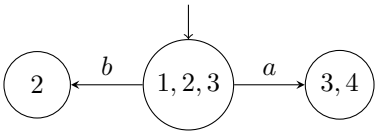


FIGURE 12.

Vi utför denna frågeställning rekursivt tills dess att mängden vi når är Ø, det vill säga nu när vi är på nod 2 så frågar vi oss ”vilka noder når vi härifrån om vi bara använder a” och samma sak för b.
Vi får då följande diagram:

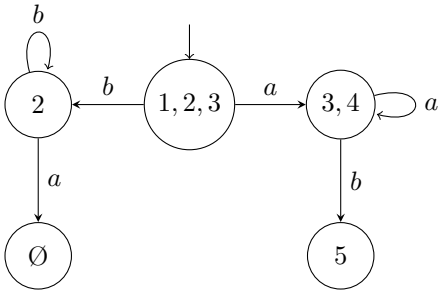


FIGURE 13.

Givetvis fortsätter vi med Ø och 5:

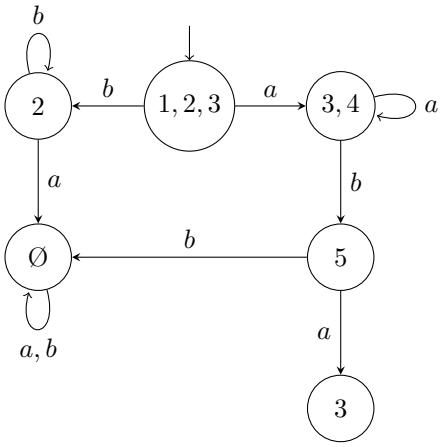


FIGURE 14.

Forstätter vi med 3 får vi slutligen:

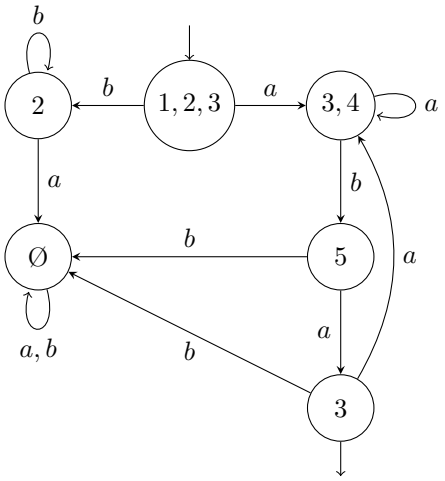


FIGURE 15.

Eftersom inga nya tillstånd (noder) har tillkommit, så är vi klara. Vi kan fråga oss varför denna algoritm alltid terminerar, och då får vi inte glömma att tillståndsmängden bara har ändligt många delmängder.

Notera att de tillstånd i M'' som innehåller något accepterande tilltånd från M' blir de accepterande tillstånden för M'' (det vill säga, i M' (Figure 10) hade nod 3 accepterande tillstånd, så de accepterande tillstånden i M'' blir de noder med enbart mängden 3).

Varför accepterar M'' då samma strängar som M'?
Jo, enligt konstruktion för en sträng w så gäller att:

- Det finns en avläsningsväg av w igenom M' så att man till slut hamnar i ett accepterande tillstånd $\Leftrightarrow$ Det finns en (unik) avläsningsväg av w igenom M'' så att man till slut hamnar i ett accepterande tillstånd.

Exempel:
Beskriv en avläsningsväg för $w = aaabaa$ igenom M' och igenom M'' som slutar med acceptans i båda fallen.

Enligt den beskrivna omvandlingsmetoden så har vi:

Definition/Sats 6.3

För varje NFA M_1 så finns en DFA M_2 som accepterar samma språk som M_1

Definition/Sats 6.4

För varje NFA M så finns ett reguljärt uttryk som beskriver $L(M)$

Vi visar nu:

Definition/Sats 6.5

Om L är ett reguljärt språk så finns en NFA (och alltså även en DFA) som accepterar L

Proof 6.1

Induktion över uppbyggnaden av reguljära språk:

- Om $L = \emptyset$ så accepteras L av startnoden men ingen accepterande nod, och vice versa (se Figur 16)
- Om $L = \{\sigma\}$ (godtycklig tecken) så accepteras L av NFA:n med 2 noder, en start och en accepterande, där det krävs just σ för att ta sig igenom (se Figure 17)
- Antag att $L = L_1 \cup L_2$ där L_1, L_2 accepteras av M_1, M_2 respektive. Då accepteras L av (Figure 18)
- Antag att $L = L_1 L_2$ och att L_1, L_2 accepteras av M_1, M_2 respektive. Då accepteras L av (Figure 19)
- Antag att $L = L_1^*$ och att L_1 accepteras av M_1. Då accepteras L av (Figure 20)
- Fallet $L = L^+$ behandlas på liknande sätt som föregående punkt
- Fallet $L = \varepsilon$ behandlas enligt Figure 21

□

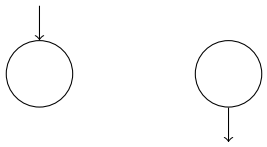


FIGURE 16.

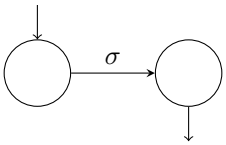


FIGURE 17.

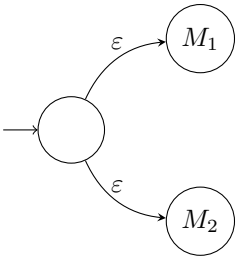


FIGURE 18. "Parallellkoppling"

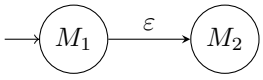


FIGURE 19. "Seriekoppling"

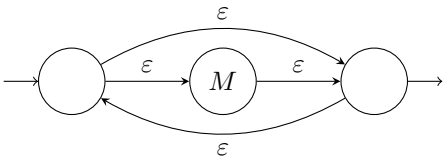


FIGURE 20.

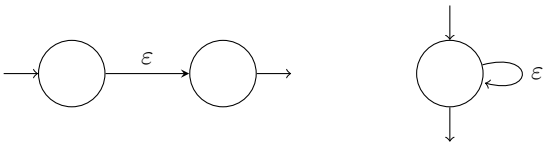


FIGURE 21.

Definition/Sats 6.6: Finit/ändlig automat

Med en **finit/ändlig automat** menar vi en NFA eller en DFA (vilken som)

7. Övning

En NFA för språket som beskrivs av $a(aba \cup aa)^*$:

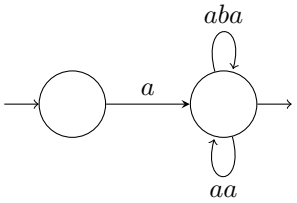


Figure 22.

Skapa en DFA för samma språk, gör först NFA:n icke-glupsk (och namnge tillstånden):

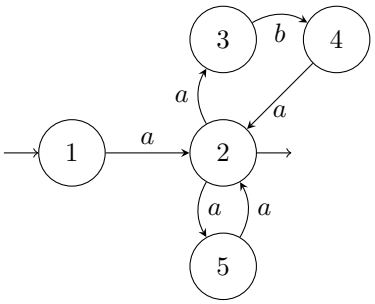


Figure 23.

Med delmängdsalgoritmen får vi följande DFA:

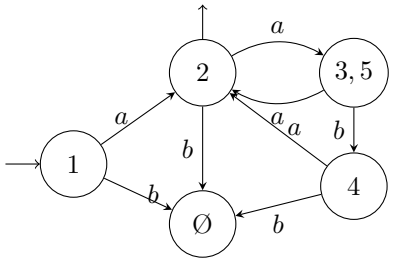


Figure 24.

8. Generaliserade Finita Automater [GFA]

En GFA ser ut (grafiskt) som en NFA *förutom* att vi tillåter att pilarna bär reguljära uttryck (istället för endast strängar)

Exempel:

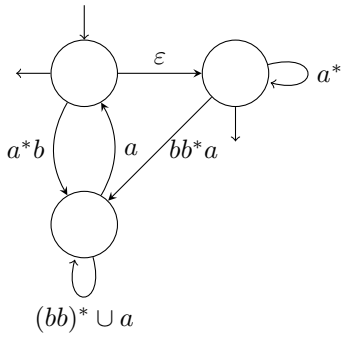


Figure 25.

Om α är ett reguljärt uttryck så tolkar vi:

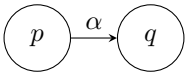


Figure 26.

som att man kan gå från tillstånd p till tillstånd q genom att avläsa en sträng i språket som α beskriver.

Vi använder GFA:er för att omvandla en given NFA/DFA M till ett reguljärt uttryck som beskriver $L(M)$, och då följer att $L(M)$ är reguljärt.

8.1. **Tillståndseliminationsalgoritmen.**

Detta långa namn är namnet på den algoritm som omvandlar en NFA/DFA M till ett reguljärt uttryck för $L(M)$:

Antag att en NFA/DFA M är given, om M saknar starttillstånd eller accepterande tillstånd så $L(M) = \emptyset$ och då beskrivs $L(M)$ av det reguljära uttrycket $\emptyset$ (och vi är klara)

(Fråga om det gäller att om en NFA/DFA inte har start/acc. tillstånd att den är isomorf med en annan NFA/DFA som inte heller har start/acc. tillstånd)

Därför kan vi anta att M har minst ett starttillstånd och minst ett accepterande tillstånd.

Algoritmen går därmed ut på följande sätt:

- Lägg till ett *nytt* starttillstånd och pilar som bär ε från detta nya starttillstånd till alla de gamla starttillstånd som nu förlorar sin status som starttillstånd (här förändras *inte* vilka strängar som accepteras)
- Lägg till ett *nytt* accepterande tillstånd och pilar som bär ε till detta nya accepterande tillstndet **från** alla de gamla accepterande tillstånden som nu förlorar status som accepterande tillstånd
- Eliminera alla de gamla tillstånden, ett för ett, samt ersätt gamla pilar med nya enligt följande mönster (där krysset markerar det tillstånd som elimineras):

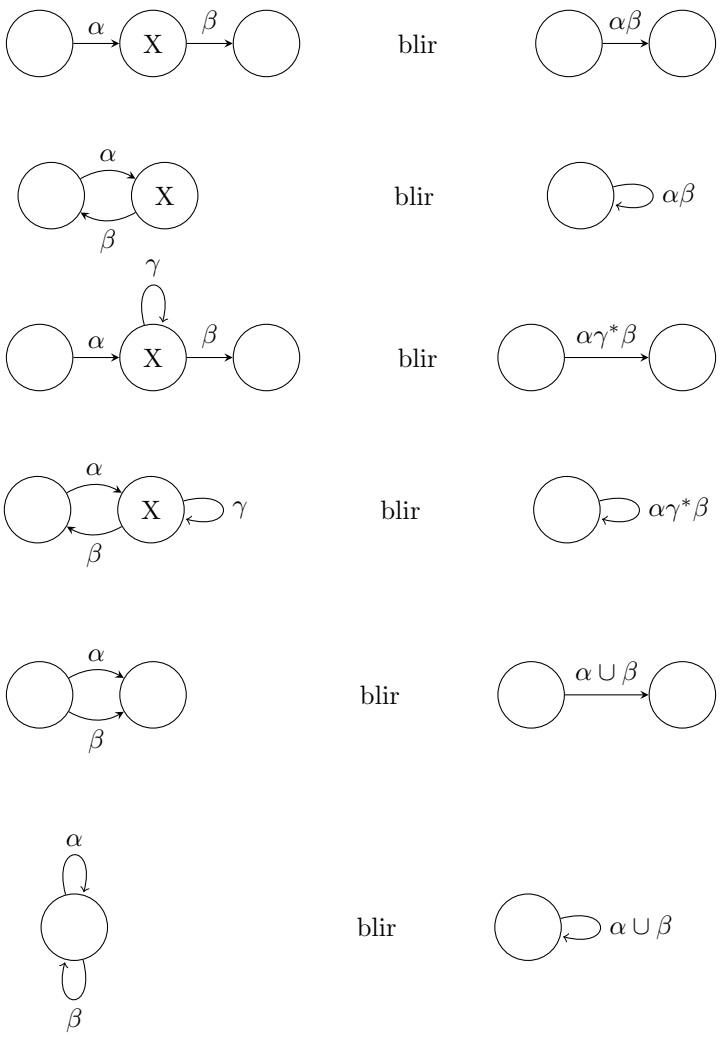


FIGURE 27.

När alla elimineringar har upprepats tills det inte går att förenkla mer, så har vi en GFA på formen:

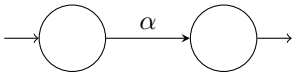


FIGURE 28.

Där α är ett reguljärt uttryck som beskriver $L(M)$

Från algoritmen följer det:

Definition/Sats 8.1

Varje språk som accepteras av någon finit automata är reguljärt.

Exempel:
Betrakta följande NFA samt omvandla till ett reguljärt uttryck:

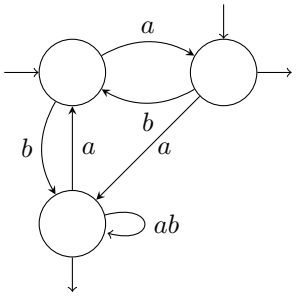


FIGURE 29.

Vi börjar med att lägga till ett nytt starttillstånd samt ett nytt accepterande tillstånd och kopplar de till de redan existerande med ε:

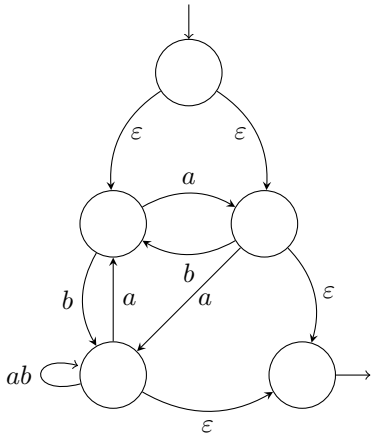


FIGURE 30.

Det finns lite olika strategier för att välja vilken nod man kan börja med att elimineras, personligen föredrar jag att välja den nod med minst antal anslutningar, det gör saker lite lättare.
Om det är så att det endast finns en nod som *inte* har en direkt koppling till det accepterande tillståndet, föredras det eftersom det blir lättare att se vilka kopplingar som måste ersättas. Väljer man en nod som är direkt kopplad till det accepterande tillståndet så finns det flera andra pilar som kommer från andra noder som man kan råka blanda ihop.

Det är många små steg som görs åt gången under elimineringsprocessen, vi ska försöka köra detta exempel med detalj.

Vi börjar med att markera alla noder, detta kommer göra saker lättare att förklara och är inget måste.

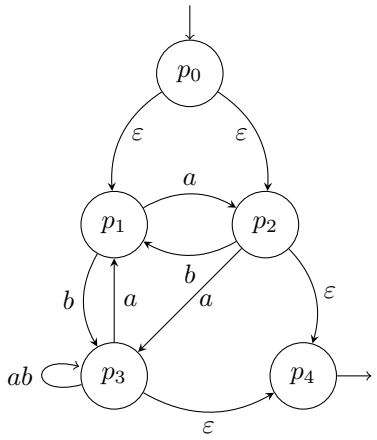


FIGURE 31.

Notera att ”den längsta vägen” till det accepterande tillståndet är via p_0, p_1, p_3, p_4
Därför väljer vi att eliminera den noden i den vägen som inte är direkt kopplad till det accepterande tillståndet, det vill säga p_1:

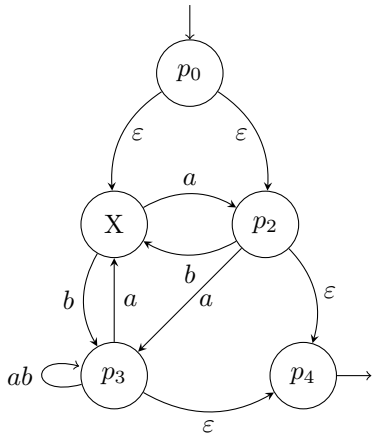


FIGURE 32.

Om vi nu ska ta bort denna nod, så måste vi fortfarande kunna avläsa samma strängar för at komma till ett accepterande tillstånd.
Strategin här är att notera vilka vägar som är möjliga att ta till de noder som är kopplade till p_1.

Tidigare kunde jag använda εa för att gå från p_0, p_1, p_2. Eftersom ε är den tomma strängen, så räcker det alltså med att säga ”tidigare kunde jag läsa a för att komma till p_2”.
Tar vi bort p_1 måste vi kunna gå från p_0 till p_2 genom att också läsa a. Vi kan lägga till den egenskapen genom följande:

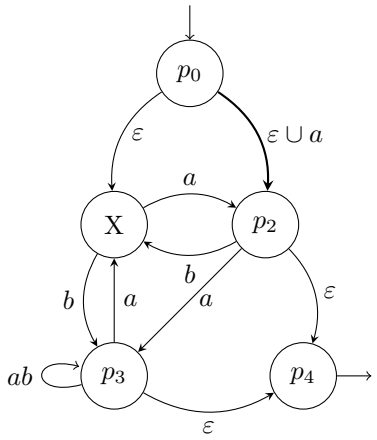


FIGURE 33.

En relevant fråga man kanske ställer sig är ”varför tar vi inte bor vägen a” från p_2 till p_1? Detta eftersom vi tidigare kunde avläsa εba för att ”loopa” runt p_2, och detta måste vi naturligtvis behålla:

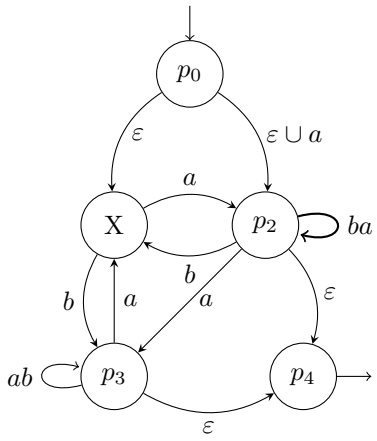


FIGURE 34.

Är vi redo att ta bort de vägarna nu? Nästan. Tidigare kunde vi komma till p_3 via vägen p_0, p_2, p_1, p_3 med strängen εbb eller εa. Vi måste alltså kunna gå från p_2 till p_3 med den strängen, vi gör därför följande:

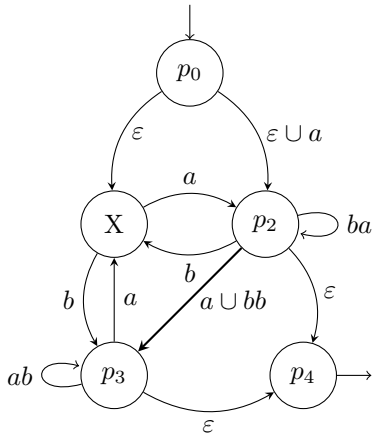


FIGURE 35.

Nu kan vi plocka bort b-pilen, eftersom vi har återskapat alla vägar:

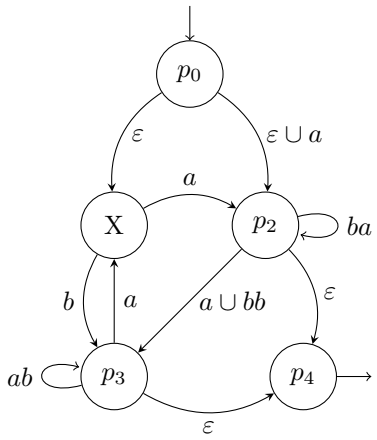


FIGURE 36.

Nu kan vi alldeles strax få bort a. Notera att vi kunde från p_3 nå p_2 genom p_1 genom att avläsa aa (gå upp ett steg, sedan till höger), eftersom vi tar bort p_1 behöver vi alltså ha en pil från p_3 till p_2 med aa. För er som undrar varför vi inte ska ta hänsyn till det a som redan står på en pil mellan p_1 och p_2 så har vi redan gjort det när vi bytte till $\varepsilon \cup a$

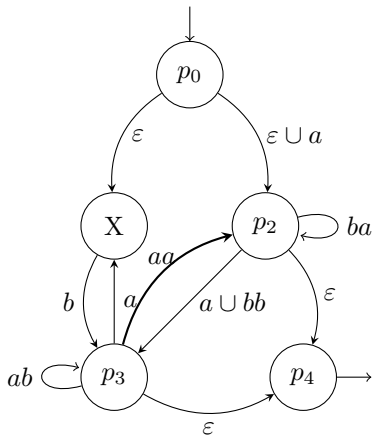


FIGURE 37.

Nästan där! Vi har kunnat läsa εb för att ta oss från p_0 till p_3 genom p_1, vilket är samma sak som att läsa b. Vi behöver alltså en koppling från p_0 till p_3 med bara b:

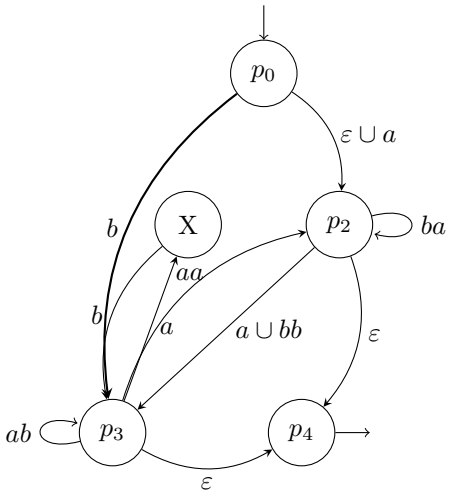


FIGURE 38.

Tidigare har vi kunnat avläsa $\varepsilon bababababab$ för att loopa mellan p_1 och p_3 (trots att p_3 redan har en loop för ab), vi har redan replikerat att vi kan ta b för att komma till p_3, och eftersom loopen ab redan finns, behöver vi inte replikera den.
Vi kan därför nu plocka bort pilarna a, b:

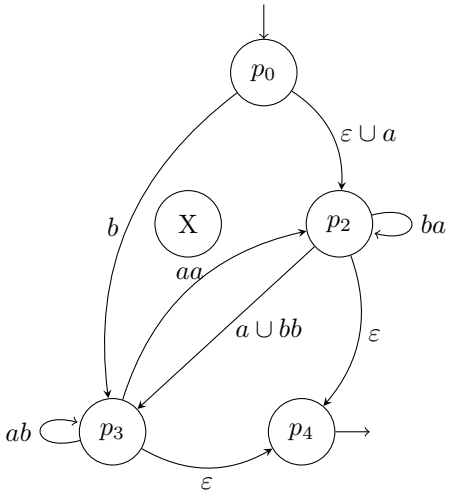


FIGURE 39.

Notera att noden vi vill ta bort är fri! Den har inga kopplingar till andra noder, och vi kan därför ta bort den:

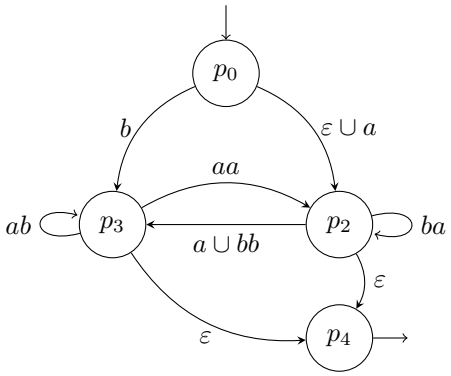


FIGURE 40.

Vi repeterar nu processen för antingen p_3 eller p_2 (det spelar ingen roll, de har precis samma antal kopplingar och kopplas med samma avstånd till det accepterande tillståndet). Vi väljer p_3.

Det som gör denna nod lite speciell är att den har en loop, och detta måste vi ta hänsyn till varje gång vi kopplar bort en koppling. Fördelen är att vi kan notera detta med $(ab)^*$ och därmed inkludera de fall då vi inte alls väljer att snurra i loopen (eftersom $\varepsilon \subseteq (ab)^*$)

Vi noterar att vi från p_2 kan avläsa $((a \cup bb)aa)^*$ för att loopa p_2. Men! Eftersom vi når p_3 (och därmed kan avläsa ab och loopa p_3) så måste vi ha med det i vår nya loop:

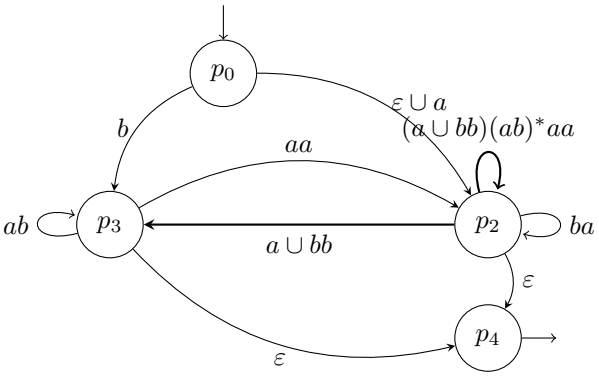


FIGURE 41.

Eftersom vi inte kan nå något mer än p_4 via ε efter att ha avläst $(a \cup bb)(ab)^*$ kan vi nu ta bort den pilen (fetmarkerad i Figure 41) och skapa en pil från p_2 till p_4:

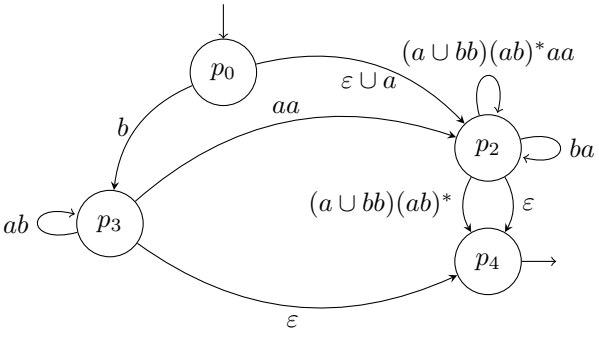


FIGURE 42.

Vi vill nu försöka få bort pilen över den vi just tog bort, den går från p_3 till p_2 så vi måste se till att de strängar vi kan läsa för att komma till p_3 leder till p_2
Vi noterar att vi kan avläsa baa för att komma till p_2 från p_3, men vi kan ju också läsa $babaa$ och oändligt antal fler loopar runt ab, så vi skapar en pil från p_0 till p_2 med $b(ab)^*aa$:

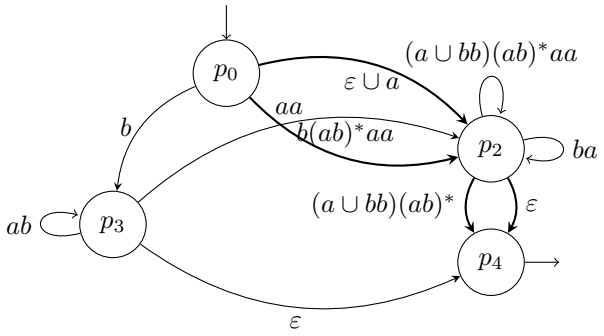


FIGURE 43.

Men vi har nu 2 olika pilar som går från samma till samma nod, vi kan slå ihop dessa till en enda:

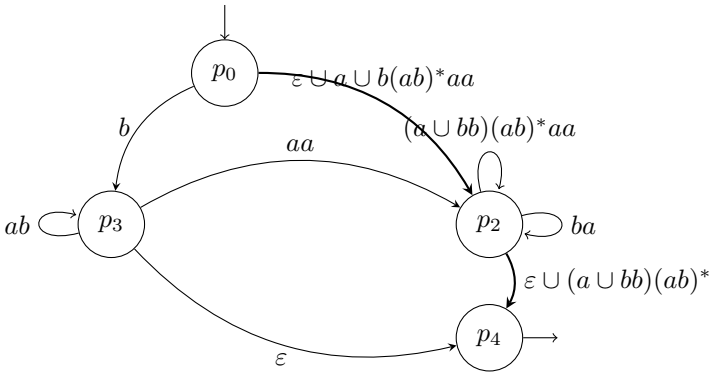


FIGURE 44.

Notera att vi kan ta vägen $b\varepsilon$ (men också givetvis loopa runt p_3 med ab hur mycket som helst eller hur lite som helst), detta bevarar vi genom att dra en pil från p_0 till p_4 med $b(ab)^*$.
Vi kan även ta bort pilarna från p_0 till p_3, pilarna från p_3 till p_2, samt pilarna från p_3 till p_4:

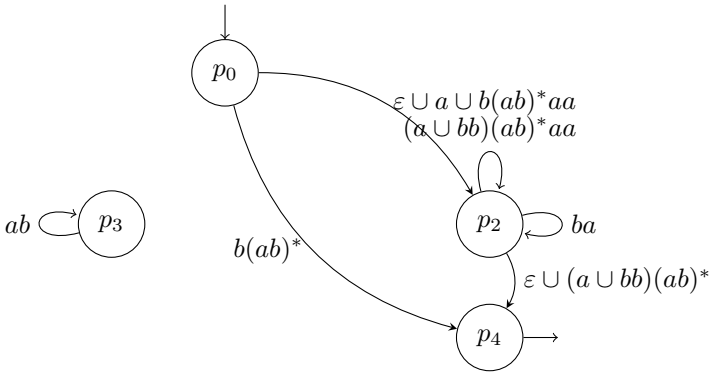


FIGURE 45.

p_3 är nu ensamt och vi kan ta bort den:

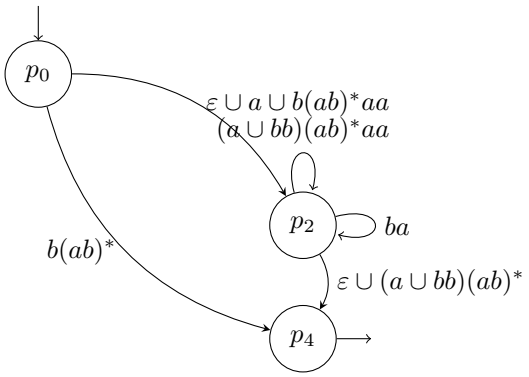


FIGURE 46.

Notera att vi har två loopar på p_2, detta kan vi förenkla till en genom att använda union:

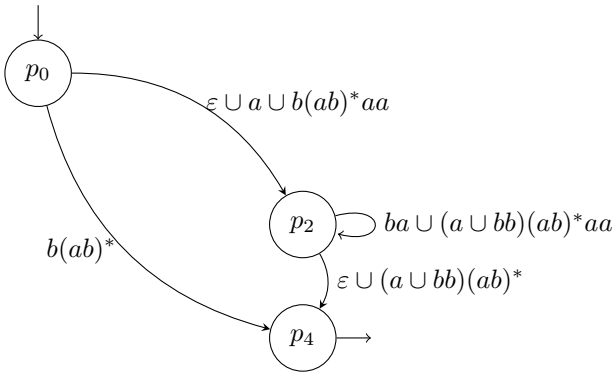


FIGURE 47.

Dags att börja eliminera p_2! De pilar som går in i p_2 är en pil från p_0, och de pilar som går ut är en pil till p_4, alltså bör vi kunna avläsa strängen $\varepsilon \cup a \cup b(ab)^*aa\varepsilon \cup (a \cup bb)(ab)^*$ för att ta oss från p_0 till p_4 via p_2, men vi har ju även en loop, alltså måste vi slänga in $(ba \cup (a \cup bb)(ab)^*aa)^*$:

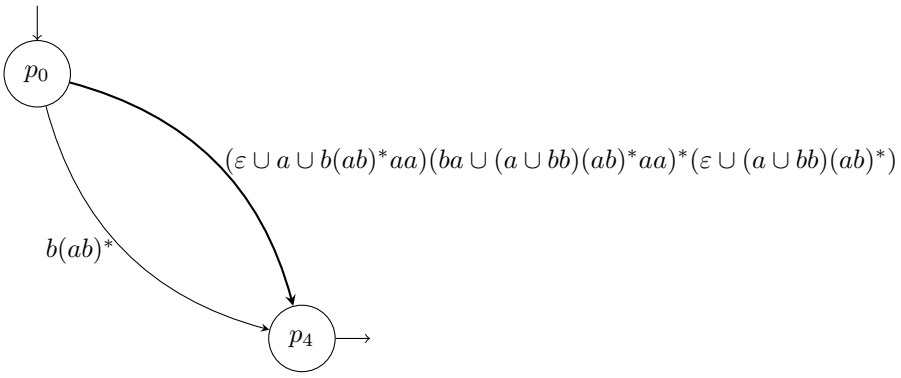


FIGURE 48.

Vi har nu 2 alternativa vägar från p_0 till p_4, antingen vägen med det långa reguljära uttrycket, eller den korta med $b(ab)^*$. Har vi en nod som går till en annan med 2 olika pilar kan dessa kombineras med union. Man kan tänka på union som ett ”eller”, det vill säga $b(ab)^* \cup Q$ kan översättas till ”$b(ab)^*$ eller Q”

Gör vi detta får vi:

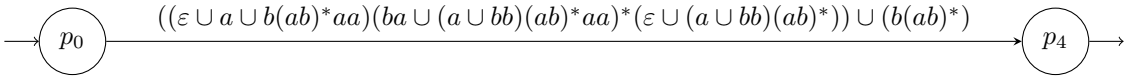


FIGURE 49.

Vi har nu ett starttillstånd, ett accepterande tillstånd, och inga andra noder. Bara en pil som kopplar de 2 tilltånden, och texten över den är det reguljära uttrycket för den ursprungliga NFA:n

9. Analys av reguljära uttryck

Om L är ett reguljärt uttryck $\Leftrightarrow$ L accpeteras av någon NFA $\Leftrightarrow$ L accpeteras av någon DFA. Detta väcker frågan, vad är den minsta DFA:n vi kan skapa givet ett reguljärt uttryck?

9.1. **Minimering av DFA.**

Definition/Sats 9.1

En DFA M är *minimal* om det inte finns en DFA som accepterar samma som M och har färre tillstånd än M

Definition/Sats 9.2

Låt M vara en DFA och w en sträng (w är en sträng över M:s alfabet).
Vi säger att w *driver* DFA:n M från ett tillstnd p till ett annat q om när M startas i tillstånd P med w på tapen så stannar M i tillstånd q

Exempel:

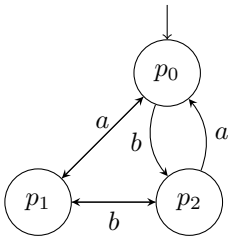


Figure 50.

Här driver strängen aaa M från p_2 till p_0. Strängen ab driver M från p_2 till p_2

Om ingen sträng driver M från starttillstånd till q så kallas q för *isolerat* tillstånd.

Exempel:

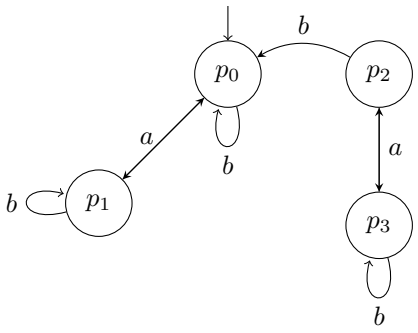


Figure 51.

Notera här att p_2 och p_3 är isolerade tillstånd! Då kan vi lika gärna plocka bort de utan att det påverkar vilka strängar som accepteras.

Definition/Sats 9.3: Reduktion av ett tillstånd (om möjligt)

Låt M vara en DFA (utan isolerade tillstånd) och antag att p och q är olika tillstånd i M.
Antag att följande villkor gäller för varje sträng w (även ε):

- w driver M från p till ett accepterande tillstånd om och endast om w driver M från q till ett accepterande tillstånd

Låt nu M' vara som M utom att q tas bort, och varje övergång i Figure 52 ersätts med Figure 53.
Då gäller $L(M') = L(M)$ och M' har färre tillstånd än M

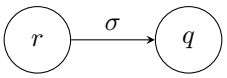


FIGURE 52.

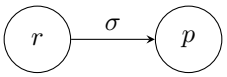


FIGURE 53.

Detta påminner lite om tillståndeliminering, men vi vill inte skapa reguljära uttryck utan det är ok att bara koppla bort pilarna från en nod till en annan.

Definition/Sats 9.4: Särskiljandealgoritmen för minimering av en DFA

Exempel:
Låt M vara följande DFA:

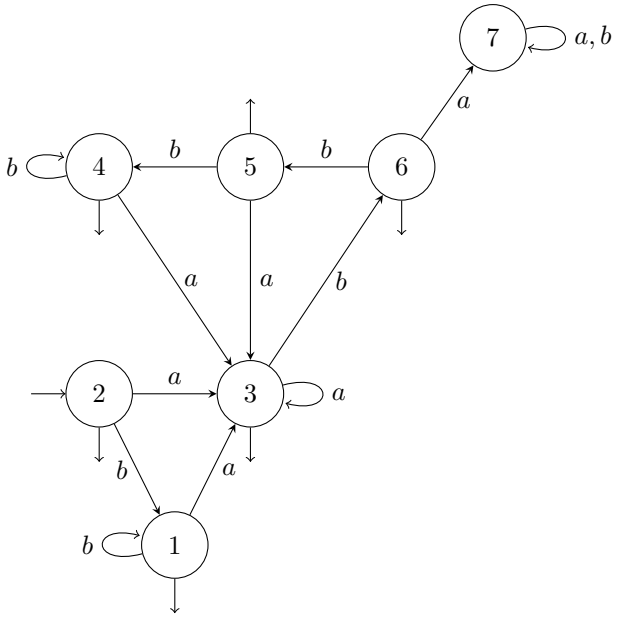


FIGURE 54.

Här är det enklare att skriva en tabell över tillståndsövergångarna:

	1	2	3	4	5	6	7
a	3	3	3	3	3	3	7
b	1	1	6	4	4	5	7

Nivå	Sönderdelningar
1	$\{7\}\,\{1,2,3,4,5,6\}$ (initial & accept. tillst)
2	$\{7\}\,\{1,2,3,4,5\}\,\{6\}$
3	$\{7\}\,\{1,2,4,5\}\,\{3\}\,\{6\}$
4	$\{7\}\,\{1,2,4,5\}\,\{3\}\,\{6\}$

Hur bildas nivå $n+1$ efter att nivå n är bildad?

- Om två tillstånd tillhör olika delar på nivå n, så gör de även det på nivå $n+1$
- Antag att p och q är tillstånd som tillhör samma del på nivå n, p och q ska särskiljas, dvs placeras i olika delar på nivå $n+1$ **om** det finns ett $\sigma \in \Sigma$ som driver DFA:n från p till en annan del på nivå n än σ som driver DFA:n från q

Efter att vi kom till nivå 4 såg vi att vi inte fick några ändringar, nu kan vi konstruera en DFA M' som är minimal.
M':s tillstånd är delarna på den sista nivån (nivå 4).

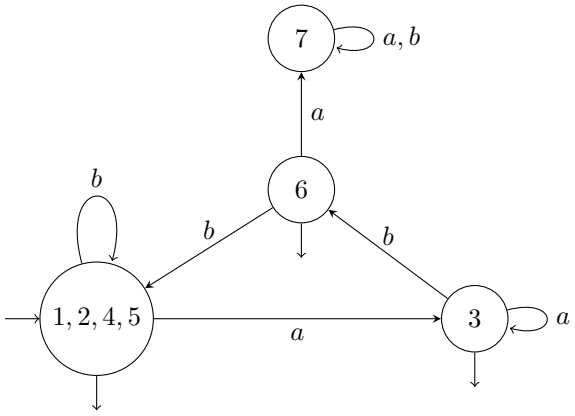


FIGURE 55.

Starttillståndet hos M' är den del/mängd som innehåller M:s starttillstånd. Samma gäller för accepterande tillstånd.
Givet en del/mängd P och tecken σ så lägg till en övergång $P \to Q$ via σ där Q är den unika del sp att det finns $p \in P$ och $q \in Q$ så att övergången från $p \to q$ via σ finns i M

Övning:
Betrakta följande tillståndsövergångstabell:

	1	2	3	4	5	6	7	8
a	1	1	2	3	5	7	8	2
b	1	4	6	5	5	5	6	6

Tillstånd $4, 6, 7$ är accepterande och tillstånd 4 är starttillståndet. Vi utför algoritmen:

Nivå	Sönderdelningar
1	$\{1,2,3,5,8\}\,\{4,6,7\}$
2	$\{1,5\}\,\{2,3,8\}$
3	$\{1,5\}\,\{2\}\,\{3,8\}\,\{4\}\,\{6\}\,\{7\}$

DFA:n blir följande:

När man väl är klar med konstruktionen så spelar det ingen roll vad tillstånden kallas (det vill säga, man behöver inte använda namnet som är mängder utan man får lov att säga p_0 osv)

Definition/Sats 9.5

Låt L vara ett språk över något alfabet Σ. Två strängar över alfabet ($x, y \in \Sigma^*$) *särskiljs* av L om det finns $z \in \Sigma^*$ så att precis en av xz och yz tillhör språket L.

Notera! Alla dessa strängar är tillåtna att vara ε

Definition/Sats 9.6

En mängd $A \subseteq \Sigma^*$ särskiljs av L om för varje par $x, y \in A$ så att $x \neq y$ så finns någon sträng z så att Definition 9.5 gäller (det finns $z \in \Sigma^*$ så att precis en av xz och yz tillhör L)

Exempel:
Låt $L_1 = \left\{w \in \{a,b\}^* : |w| \equiv_{\text{mod}3} 0\right\}$. Låt $x = aa$ och $y = a$, här är $x \neq a$ och vi påstår att den särskiljs av L eftersom vi kan hitta z så att $xz \in L_1$ men att $yz \notin L_1$.
Välj exempelvis $z = a$, då är $xz = aaa$ och $|xz| = 3 \equiv 0$, men $yz = aa$ som $|yz| \equiv 2 \neq 0$

Exempel:
Låt $L_2 = \left\{w \in \{a,b\}^* : w \text{ innehåller minst 2 } a\text{:n}\right\}$ Låt $x = aab$ och $y = ab$ Om $z = \varepsilon$ så särskiljs x och y av L_2

Exempel:
Låt $A = \{a, aa, aaa\}$. Vi påstår att den här mängden särskiljs av L_1
För att visa detta måste vi visa att varje par har ett z så att Definition 9.5 gäller.
För att underlätta gör vi en tabell:

	a	aa	aaa
a	X	a	ε
aa		X	ε
aaa			X

Notera att diagonalen inte fylls i. Tabellen visar exempel på z så att exakt en av xz och yz tillhör L_1 om $x \neq y$ och x är på lodräta axeln och y på vågräta axeln.

Definition/Sats 9.7

Antag att mängden A särskiljs av språket L och att A innehåller n strängar (de är givetvis över samma alfabet).
Då måste varje DFA M sådant att $L(M) = L$ ha minst n tillstånd.

Det följer att om $L(M) = L$ och M har exakt n tillstånd så är M minimal

Det följer då från satsen (och de föregående exempel) att varje DFA som accepterar L_1 (från föregående exempel) har minst 3 tillstånd.

Eftersom följande DFA accepterar L_1 och har 3 tillstånd, så är den minimal.

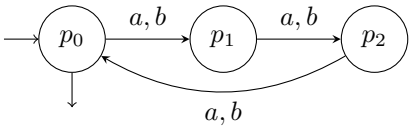


FIGURE 56.

Proof 9.1: Sats 9.7

Antag att $A \subseteq \Sigma^*$ som särskiljs av L. Antag även att A innehåller minst n strängar. Låt M vara en DFA som accepterar L.
Låt $A = \{x_1, x_2, \cdots, x_n\}$. Så om $x_i, x_j \in A$ och $i \neq j$ så finns något $z \in \Sigma^*$ så att precis en av $x_i z$ och $x_j z$ tillhör L, dvs accepteras av M

Det betyder ju att när vi läser av x_i resp. x_j så kommer vi till olika tillstånd på grund av att bara en av de accepteras av M. Kommer man till samma tillstånd skulle båda strängar accepteras eller ingen av de.

Alternativt, det betyder att x_i driver M från starttillståndet s till ett annat tillstånd än vad x_j driver M från s. Därför måste det finnas minst n tillstånd i M. □

Vad händer om det finns en oändligt stor mängd A som särskiljs av ett visst språk?

10. Slutenhetsegenskaper hos reguljära språk

Vi vet att operationerna på språken gör att vi får ett reguljärt språk, det följer ur definitionen av ett reguljärt språk.
Men! Det visar sig att om man har ett reguljärt språk så kommer komplementet vara reguljärt, det följer *inte* från definitionen!

Antag att L_1 och L_2 är reguljära språk. Från definitionen reguljära språk, följer det att även $L_1 \cup L_2$, L_1L_2, L_1^* och L_1^+ är reguljära.
Det finns dock fler slutenhetsegenskaper:

Definition/Sats 10.1

Om L_1 och L_2 är reguljära, så är $\bar{L_1}$ (på samma sätt för L_2), och $L_1 \cap L_2$, $L_1^{\text{rev}} = \{w^{\text{rev}} : w \in L_1\}$, Prefix($L_1$), Suffix($L_1$).
Alla dessa är reguljära.

Bevisskiss:
Vi tittar först på komplementet. Om L_1 är reguljärt, så finns en DFA M_1 som accepterar L_1. Vi kan därför konstruera en DFA med starttillstånd M_1 som accepterar komplementet.
Vi påminner oss om att om $w \in L_1 \Leftrightarrow M_1$ accepterar $w \Leftrightarrow w$ driver M_1 från s till ett accepterande tillstånd.
Låt M_1' vara DFA:n som fås från M_1 genom att göra alla M_1:s accepterande tillstånd till icke-accepterande och göra alla M_1:s icke-accepterande tillstånd till accepterande. Då accepteras $\bar{L_1}$ av M_1' och därmed är $\bar{L_1}$ reguljärt.

Bevisskiss:
Vi tittar nu på snittet. Antag att L_1 och L_2 är reguljära. Enligt det vi visade i bevisskiss ovan så är $\bar{L_1}$ och $\bar{L_2}$ reguljära. Enligt definitionen av reguljära språk, så är $\bar{L_1} \cup \bar{L_2}$ och då är även komplementet av unionen också reguljärt. Men, snittet $L_1 \cap L_2$ är ju komplementet till unionen.

Anmärkning:
Snittet av 2 reguljära uttryck är *inte* ett reguljärt uttryck, men snittet av 2 reguljära språk är reguljära.

10.1. **Konstruktion av en ”snitt”-DFA.**

Givet input: Två DFA:er M_1 resp. M_2:
M_1:
M_2

En DFA som accepterar $L(M_1) \cap L(M_2)$ med ordnade par från vardera DFA. Starttillståndet blir tillståndet med starttillstånd från vardera DFA. Lägg till alla kombinationer, och ta bort de isolerade tillstånden. I vårat fall får vi:

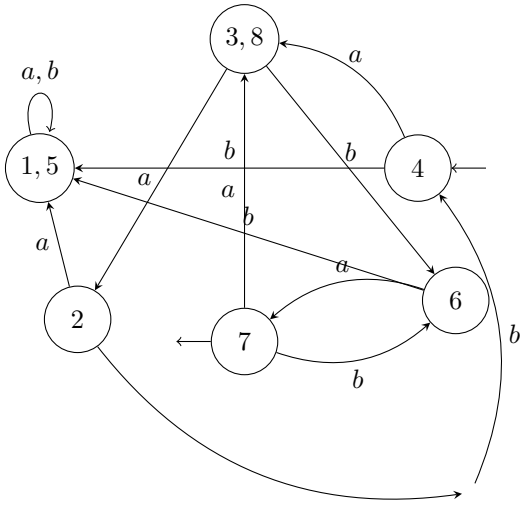


FIGURE 57.

Definition/Sats 11.1: Särskiljandesatsen

Låt L vara ett språk över alfabetet Σ och låt A vara en oändlig delmängd av strängar över Σ ($A \subseteq \Sigma^*$).
Om L särskiljer A, så kan L inte vara reguljär

Bevisskiss:
Antag att vi har en mängd $A = \{x_1, x_2, \cdots\}$ som innehåller oändligt antal strängar. Om A särskiljs av språket L och vi har en DFA som accepterar L, då skulle varje par av strängar x_i, x_j i A så måste x_i driva DFA:n från starttillståndet till ett tillstånd. Samma med x_j, men vi hamnar ju i olika tillstånd (vi antar även att $x_i \neq x_j$). En DFA har däremot bara ändligt många tillstånd. DFA:n kan inte acceptera ett språk som särskiljer en oändlig mängd.

Exempel:
Ett exempel på ett icke-reguljärt språk är $L = \{a^n b^n : n \in \mathbb{N}\}$. Notera att $a * b^*$ *inte* är ett reguljärt uttryck för språket L, eftersom med det reguljära uttrycket kan vi uttrycka abb, vilket vi inte kan per definition i L.
Detta är dock inte ett bevis, men vi kan använda satsen för att faktiskt bevisa att L ej är reguljärt.
Låt $A = \{a^n : n \in \mathbb{N}\}$, som är *oändlig*.
Vi visar nu att A särskiljs av L, då följer det från satsen att A inte är reguljärt. Detta kan vi göra genom att ta två skilda strängar $x, y \in A \quad x \neq y$
Då finns naturliga tal $i, j \in \mathbb{N}$ där $i \neq j$ så att $x = a^i$ och $y = a^j$. Välj då exempelvis $z = b^i$. Vi får då att $xz = a^i b^i \in L$ och $yz = a^j b^i \notin L$
Eftersom i, j valdes godtycklig, så särskiljs A av L. Det följer då från särskiljandesatsen att L *inte* är reguljärt.

Exempel: (och liten inledning till pumpsatsen)
Betrakta följande DFA:

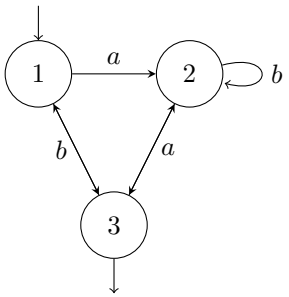


Figure 58.

Antalet kombinationer av tecken och tillstånd är $2 \cdot 3 = 6$. Följande sträng accepterars:

$$\underbrace{abbaababa}_{u}\underbrace{baaabba}_{w}\underbrace{babb}_{v}$$
$$122232231223231322313$$

Välj en delsekvens som har längd större än 6. Vi kallar den strängen w:

$$baaabba \qquad |baaabba| = 7 > 6$$

Vi kallar de övriga delarna u och v

Ur lådprincipen följer det att eftersom $|w| > 6$ så kommer någon kombination av tecken/tillstånd uppträda minst 2 gånger. Vi måste alltså ”loopa” ett tillstånd minst 2 gånger när w avläses. Exempelvis dyker kombinationen $(a, 3)$ två gånger.

Vad vi ska göra nu är att dela upp w i 3 delar baserat på detta:

$$\underbrace{ba}_{x}\underbrace{aabb}_{y}\underbrace{a}_{z}$$

Låt delen av w fram till just före 1:a förekomsten av $(a, 3)$, vi kallar den x. Låt delen av w från 1:a förekomsten av $(a, 3)$ till just före 2:a förekomsten av $(a, 3)$, vi kallar den y. Resten av w kallas för z.

På grund av att DFA:n är deterministisk så accepterars även $uxyyzv$. Faktum är att vi kan haka på hur många y som helst, vi kan även *ta bort* y och strängen kommer fortfarande att accepterars.

Definition/Sats 11.2: Pumpsatsen för reguljära språk

Antag att vi har ett oändligt reguljärt språk L. Att det är oändligt garanterar att språket innehåller strängar av hur lång längd som helst, annars hade vi kunnat hitta ett supremum.

Då finns ett naturligt tal $N \in \mathbb{N}$ sådant att om vi har en sträng $uwv \in L$ och $|w| \geq N$ så finns strängar x, y, z så att $w = xyz$ och $y \neq \varepsilon$ och $uxy^n zv \in L$ för alla naturliga tal $n \in \mathbb{N}$.
Om $n = 0$ får vi $uxzv \in L$

Mall för att visa med pumpsatsen att ett språk L inte är reguljärt:

- Visa att L är oändligt
- Antag att L är reguljärt
- Låt N vara talet som anges (för L) i pumpsatsen
- Välj lämpliga (konkreta upp till att N:s värde är okänt) strängar u, w, v sådant att $uwv \in L$ och $|w| \geq N$
- Visa att hur än w delas upp i 3 delar ($w = xyz$ där $y \neq \varepsilon$) så finns något tal $k \in \mathbb{N}$ så att $uxy^k zv$ *inte* tillhör L
- Anmärk att det motsäger pumpsatsen eftersom L antogs vara reguljärt.

Exempel:
Låt $L = \{a^n b^n : n \in \mathbb{N}\}$. Vi vill visa att L inte är reguljärt med pumpsatsen.

Vi följer mallen:

- Vi visar att L är oändligt:
  $ab, a^2b^2, a^3b^3, \cdots \in L$ så L oändlig
- Vi antar att L är reguljärt
- Låt N vara givet för L av pumpsatsen
- Vi kan tänka oss att pumpa ett a i en given sträng, eftersom vi inte pumpar b så kommer alltså vår nya sträng *inte* att finnas i L.
  Alltså, låt $u = \varepsilon$, $w = a^{N+1}$, $v = b^{N+1}$ (notera att dessa är inte konkreta eftersom vi inte vet vad N är)
  Så $uwv = a^{N+1}b^{N+1} \in L$ och $|w| \geq N$
- Antag att vi har en uppdelning av $w = xyz$ där $y \neq \varepsilon$. Då gäller att $y = a^m$ för något $m > 0$ och $uxy^0zv = uxzv = a^{N+1-m}b^{N+1} \notin L$
- Eftersom uppdelningen skedde godtyckligt, motsäger detta pumpsatsen, alltså måste L vara irreguljärt.

Exempel:
Låt L vara språket med *minst* lika många b som a, dvs $L = \left\{w \in \{a, b\}^* : \text{antalet } b \text{ i } w \text{ är minst lika stort osm antalet } a\right\}$

Språket är inte ändligt och är inte reguljärt, vi visar detta med pumpsatsen.

- Antag att L är reguljärt
- Låt N vara givet för L av pumpsatsen
- Låt $u = \varepsilon$, $w = a^{N+1}$, $v = b^{N+1}$, så $uwv = a^{N+1}b^{N+1} \in L$ och $|w| \geq N$
- Antag att $w = xyz$ där $y \neq \varepsilon$. Så $y = a^m$ för $m > 0$. Vi får nu att $uxy^2zv \notin L$
- Detta motsäger antagandet att L är reguljärt, därmed följer det från pumpsatsen att L inte är reguljärt.

12. Grammatiker

Detta är ett nytt sätt att se på språk. Tidigare har vi haft reguljära uttryck som beskriver ett språk, och finita automater som läser dessa.
När vi istället tittar på grammatiker så talar de om hur nya strängar kan skapas från tidigare strängar. Då kommer vi inte tänka oss att vi accepterar strängar utan att vi producerar strängar givet grammatik.

Definition/Sats 12.1: Produktionsregel

En *produktionsregel* är en sträng/uttryck på formen:

$$u \to v$$

där $\to$ inte förekommer i strängarna u och v.

Definition/Sats 12.2: Grammatik

En *grammatik* är en 4-tupel $G = (\Sigma_N, \Sigma_T, P, S)$ där:

- Σ_N är ändlig mängder av symboler n står för "non-terminating"
- Σ_T är ändlig mängder av symboler T står för "terminating"
- P är ändlig mängd av produktionsregler $u \to v$ där $u, v \in \Sigma_N \cup \Sigma_T$ där $u \neq \varepsilon$
- $S \in \Sigma_n$, kallas för startsymbol

12.1. Produktion av strängar.

Vad är vitsen och hur fungerar det? Vi kommer i slutet vara intresserade av strängar som enbart är terminerande

Låt $G = (\Sigma_N, \Sigma_T, P, S)$ vara en grammatik:

Definition/Sats 12.3

En sträng v kan produceras från en sträng u i ett steg, betecknas $u \Rightarrow v$ om $u = xy_1z$ och $v = xy_2z$ och det finns en produktionsregel $\in P$ som är på formen $y_1 \to y_2$

Definition/Sats 12.4

Vi säger att v kan produceras i noll eller flera steg från en annan sträng u, betecknat $u \Rightarrow^* v$, om:

- $u = v$
- $\exists n \geq 2, n \in \mathbb{N}$ och strängar $w_1, \cdots, w_n$ så att $u = w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = v$

Definition/Sats 12.5

Språket som G defninerar är $L(G) = \{w \in \Sigma_T^* : S \Rightarrow^* w\}$
Kan kallas för "G:s språk" eller "språket som G producerar/beskriver".

Exempel:
Låt $L = \{a^n b^n : n \in \mathbb{N}\}$. Vi såg att detta språk L *inte* var reguljärt. Det finns alltså inte en finit automat som accepterar språket. Det går heller inte att beskriva med ett reguljärt uttryck, men vi kan faktiskt ge en grammatik som producerar strängarna i det här språket.

Låt $G = (\Sigma_N, \Sigma_T, P, S)$ där:
$\Sigma_n = \{S\}$, $\Sigma_T = \{a, b\}$, $P = \{S \to aSb, S \to \varepsilon\}$

Vi ska nu motivera varför denna grammatik producerar bara strängar i L.
Vi ser att om $w \in \Sigma_T^*$ kan produceras av G, dvs om $S \Rightarrow^* w$, så måste $w = a^n b^n$

Säg att vi börjar med S, då kan jag antingen ta bort S och få $\varepsilon = a^0b^0$, eller så kan jag använda första regeln och få aSb. Då kan jag antingen plocka bort S och få ab, eller så kör jag första regeln igen och får $aaSbb$ och så fortsätter det...
Alltså kommer vi alltid få något på a^nb^n.

Ett induktionsbevis för att $a^nb^n \in L(G) \quad \forall n \in \mathbb{N}$:
Basfall: Om $n = 0$ så $S \Rightarrow^* S = a^0Sb^0$ och $S \Rightarrow \varepsilon$, så $\varepsilon \in L(G)$
Induktions steg: Antag att $S \Rightarrow^* a^nSb^n$. Med regeln $S \Rightarrow aSb$ följer att $a^nSb^n \Rightarrow a^{n+1}Sb^{n+1}$
Från $S \Rightarrow^* a^nSb^n \Rightarrow a^nb^n \quad \forall n \in \mathbb{N}$ följer att $a^nb^n \in L(G)$.
Notera, detta visar enbart att $L \in L(G)$
Detta är ett exempel på ett så kallat *sammanhangsfri grammatik* och språket är ett *sammanhangsfritt* språk.

12.2. **Sammanhangsfria grammatiker och språk.**

Definition/Sats 12.6

En produktionsregel kallas *sammanhangsfri* om den har formen $A \to w$ där A är en icke terminerande symbol ($A \in \Sigma_N$) och A bara är ett tecken

Definition/Sats 12.7: Sammanhangsfri grammatik

En grammatik är sammanhangsfri om alla dess regler är sammanhangsfria. (CFG, Context Free Grammar)

Definition/Sats 12.8: Sammanhangsfritt språk

Ett språk L kallas sammanhangsfritt om det kan produceras med en sammanhangsfri grammatik G. (CFL, Context Free Language)

Från föregående exempel, följer det att L är sammanhangsfritt.

Definition/Sats 12.9

Varje reguljärt språk är sammanhangsfritt

Exempel:
Antag att L är reguljärt. Då finns en DFA M så att DFA:ns språk $L(M) = L$.
Låt M vara följande DFA:

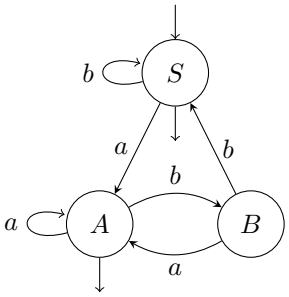


FIGURE 59.

Vi omvandlar M till en CFG: $G = (\Sigma_n, \Sigma_T, P, S)$ genom att låta:

- $\Sigma_n = \{S, A, B\}$
- $\Sigma_T =$ DFA:s alfabet $\{a, b\}$

och där varje tillståndsövergång i M motsvaras av en produktionsregel på följande sätt:
Övergången blir med produktionsregel $S \to aA$

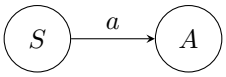


FIGURE 60.

På liknande sätt får vi också följande produktionsregler:

- $A \to bB$
- $B \to aA$
- $B \to bS$
- $\vdots$
- $S \to \varepsilon$ (accepterande tillstånd)
- $A \to \varepsilon$ (accepterande tillstånd)

Vi bör få $3 * 2$ regler, kardinaliteterna $\Sigma_N * \Sigma_T$

Betrakta nu sträng *baaba* (som accepteras av M), notera att denna sträng även produceras av G:

- *baaba* S
- *aaba* S
- *aba* A
- *ba* A
- *a* B
- A

Vilket blir, grammatiskt:

- $S \to bS \to baA \to baaA \to baabB \to baabB \to baabaA \to baaba\varepsilon = baaba$

Exempel:
Låt $L = \left\{w \in \{a,b\}^* : w^{\text{rev}} = w\right\}$
En CFG G så att $L(G) = L$
Notera att första tecknet måste vara lika med det sista tecknet i ett palindrom och att palindrom kan ha både udda och jämn längd. Vi inför följande produktionsregler:

- $S \to \varepsilon$
- $S \to aSa$
- $S \to bSb$
- $S \to a$
- $S \to b$

Produktion av $ababa$: $S \Rightarrow aSa \Rightarrow abSba \Rightarrow ababa$

13. Pushdownautomater [PDA]

13.1. En liten informell beskrivning:

Definition/Sats 13.1: PDA

En PDA har 2 taper, en för inputsträngen och en som kallas för *stacken* (ett slags minne).
En PDA har även 2 alfabet, 1 input-alfabet (det som används på input tapen) och ett stackalfabet (används på stacken)

Är icke-deterministisk, har ändligt många tillstånd, samt ändliga alfabet för både input& stackalfabet

När PDA:n jobbar så konsumerar den en delsträng, likt en NFA. Den avläser alltså inte högst ett tecken, utan kan avläsa flera.
Samtidigt som den avläser från inputtapen så byter den en delsträng överst på stacken med någon sträng.
Samtidigt gör den även en tillståndsövergång.

Om vi struntar i stacken så fungerar en PDA som en NFA.

En PDA har dock enbart 1 starttillstånd. (Vera hävdar att detta är lite onödig konvention).

13.2. Grafisk beskrivning av PDA.

Ganska lik en NFA, men på pilarna måste vi lägga till vad som görs på stacken:

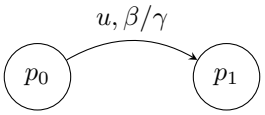


Figure 61. Byte av β mot γ överst i stacken

Såklart så är u sträng över inputalfabet och β, γ sträng över stackalfabet.

PDA:n kan gå från tillstånd p_0 till p_1 genom att avläsa strängen u på input-tapen (u måste börja där läshuvudet står på input-tapen) och ersätta β med γ överst på stacken (förutsätter att β står överst på stacken).
Det är tillåtet att u, β, γ är ε

Anmärkning:
Om β inte finns överst på stacken, så kan inte den övergången tas.

13.3. **Kriterier för acceptans hos PDA.**

En PDA M accepterar en sträng w över inputalfabet om w kan delas upp $w = v_1 \cdots v_n$ och man kan avläsa delarna $v_1, v_2, \cdots, v_n$ genom att följa övergångar:

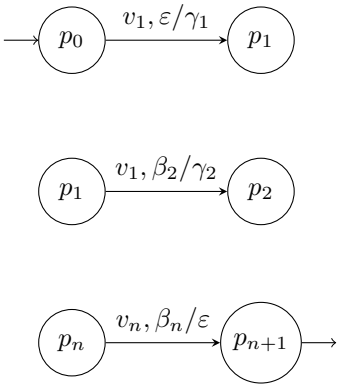


FIGURE 62.

där p_1 är starttillstånd, p_{n+1} är accepterande och stacken är tom i början och när hela w har avlästs. Det är tillåtet att $p_i = p_j$ även om $i \neq j$

13.4. **Språk hos PDA.**

Om M är en PDA så $L(M) = \{w : w$ är en sträng över M:s inputalfabet som accepteras av $M\}$
Detta är M:s språk.

Exempel:
Betrakta följande språk $L = \{a^n b^n : n \in \mathbb{N}\}$ (icke-sammanhangsfritt språk).
Konstruera en PDA som accepterar L:
Iden är att vi bara vill läsa av så många a som möjligt, och sedan för varje a lägger vi till något i stacken. Då har vi koll på *hur många* a vi har läst. Låt M vara följande PDA:

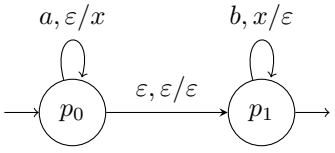


FIGURE 63.

Input-alfabet är $\{a, b\}$ och stackalfabet är $\{x\}$

Eftersom PDA:er är icke-deterministiska så finns det flera olika sätt att ”köra” PDA:n. Vi ska kolla på hur denna körning ser ut:

Tillstånd	Input-tape	stacken
p_0	$\boldsymbol{a}abb$	ε
p_0	$a\boldsymbol{a}bb$	x
p_0	$aa\boldsymbol{b}b$	xx
q	$aa\boldsymbol{b}b$	xx
q	$aab\boldsymbol{b}$	x
q	$aabb$	x
q	$aabb$	ε

Denna körning vittnar om att $aabb$ accepteras av M.
Däremot skulle $aaabb$ *inte* accepteras av följande DFA, detta eftersom det kommer finnas ett för mycket x i stacken men inget b att byta ut mot ε.

Definition/Sats 13.2

För varje CFG G så finns det en PDA M som har samma språk som G, dvs $L(M) = L(G)$

Detta kan illustreras med en så kallad ”top-down” parser metoden:
Låt $G = (\Sigma_n, \Sigma_T, P, S)$ där $P = \{A_1 \to w_1, A_2 \to w_2, \cdots, A_n \to w_n\}$ och $\Sigma_T = \{\sigma_1, \cdots, \sigma_k\}$

För att konstruera M med detta, låter vi M:s inputalfabet vara Σ_T och M:s stackalfabet vara $\Sigma_T \cup \Sigma_N$
Låt M beskrivas med (kom ihåg, S är startsymbol):

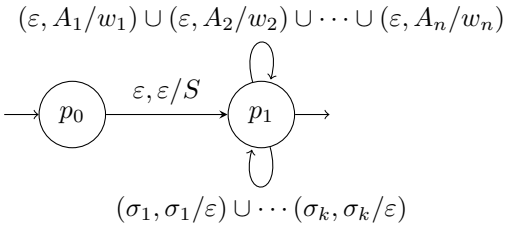


FIGURE 64.

Nu gäller att $L(M) = L(G)$

Exempel:
Låt G vara följande CFG:
$L(G) = \{a^n b^n : n \in \mathbb{N}\}$
$P = \{S \to aSb, S \to \varepsilon\}$
Konstruera en PDA med samma språk som grammatiken G m.h.a top-down parser metoden:

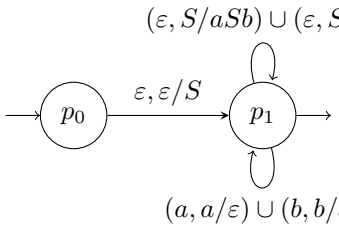


FIGURE 65.

Definition/Sats 13.3

För varje PDA M finns en CFG som har samma språk som M, dvs $L(M) = L(G)$

Från Sats 13.3 följer det att ett språk är sammanhangsfritt (CFL) $\Leftrightarrow$ någon PDA accepterar språket.

14. De sammanhangsfria språkens gränser

Vi börjar med att gå igenom pumpsatsen för sammanhangsfria språk

14.1. **Pumpsatsen för CFL.**

Vi antar först att L är ett oändligt och Sammanhangsfritt språk.
Då finns ett tal $K \in \mathbb{N}$ sådant att om vi har en sträng $w \in L$ och $|w| \geq K$, så kan w delas upp i 5 delar

$$w = uvxyz$$

så att $|vxy| \leq K$ och $vy \neq \varepsilon$, till sist ska alla pumpningar tillhöra L:

$$uv^n xy^n z \in L \quad \forall n \in \mathbb{N}$$

Bevis finns i Veras anteckningar.

Mall för användning av pumpsatsen för CFL: (för att visa att ett språk inte är CFL)

I stora drag ser den ut som använding av pumpsatsen för reguljära språk, men detaljerna (speciellt punkt 4 och 5) skiljer det sig.

Så går det ut:

- Visa att språket L är oändligt
- Antag att språket L är sammanhangsfritt (CFL)
- Låt K vara givet av pumpsatsen (för CFL) för L
- Välj lämplig sträng w (konkret upp till K, så pass konkret att om vi visste vad K var så blir det en konkret sträng med längd minst K och tillhör L)
- Visa att hur än w delas upp i 5 delar, så kommer någon av de pumpade strängarna *inte* tillhöra L
  Dvs, $w = uvxyz$ så att $|vxy| \leq K$ och $vy \neq \varepsilon$, $\exists n \in \mathbb{N}$ så att $uv^n xy^n z \notin L$
- Uttryck att föregående punkt motsäger pumpsatsen för CFL, så L kan *inte* vara en CFL

Exempel:
Låt $L = \{a^n b^n c^n | n \in \mathbb{N}\}$
Vi visar med pumpsatsen för CFL att L ej är en CFL:

- Eftersom $\varepsilon, abc, a^2b^2c^2, \cdots \in L$ så är L oändligt
- Antag att L är en CFL
- Låt K vara givet av pumpsatsen för CFL och för språket L
- Välj $w = a^K b^K c^K$, så $w \in L$ och $|w| \geq K$
- Antag att $w = uvxyz$, $|vxy| \leq K$ och $vy \neq \varepsilon$
  Då kan inte vxy börja på något a och sluta på c, eftersom det finns K stycken b i mitten vilket motsäger längden på vxy (skulle vara högst K)
  Vi kan därmed betrakta två fall, då vxy innehåller a, b eller b, c:
  - Antag att vxy är en delsträng till $a^K b^K$. Så i x har något a eller b eller båda försvunnit i jämförelse med ursprungliga vxy.
    Därför är antalet c i $uxz = uv^0xy^0z$ större än antalet a eller b (i uxz). Så $uxz \notin L$
  - Antag att vxy är en delsträng till $b^K c^K$. I x har något b eller c försvunnit i jämförelse med vxy. Då är antalet a i uxz större än antalet b eller c. så $uxz \notin L$
- Föregående punkt motsäger pumpsatsen för CFL, så L ej sammanhangsfritt.

14.2. **Slutenhetsegenskaper hos CFL.**

Definition/Sats 14.1

Om L_1 och L_2 är CFL, så är även följande sammanhangsfria:

- $L_1 \cup L_2$
- $L_1 L_2$
- $(L_1)^*$

Notera! Det är inte säkert att $L_1 \cap L_2$ eller komplementet är sammanhangsfritt! Det beror på vad L_2 och L_2 är

Partiell motivering till satsen

Antag att L_1 och L_2 är CFL. Då finns det PDA:er M_1 och M_2 så att $L(M_1) = L_1$ och $L(M_2) = L_2$ som accepterar dessa (enligt Sats 13.3)

En PDA M så att $L(M) = L_1 \cup L_2$ och då följer att $L_1 \cup L_2$ är en CFL. Vi kan konstruera följande PDA:

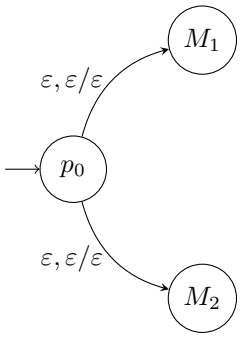


FIGURE 66.

En PDA M så att $L(M) = L_1 L_2$ och då följer att $L_1 L_2$ är en CFL:

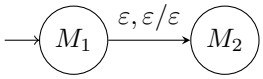


FIGURE 67.

Vi ska kika lite närmare på varför snittet av två språk inte behöver vara CFL.

Exempel:
Låt $L_1 = \{a^n b^m c^m | n, m \in \mathbb{N}\}$
Låt $L_2 = \{a^n b^n c^m | n, m \in \mathbb{N}\}$
Snittet, $L_1 \cap L_2 = \{a^n b^n c^n | n \in \mathbb{N}\}$
Notera, detta är ju samma språk som vi hade i pumpsats-exemplet som vi visade inte var CFL!

L_1 och L_2 är CFL:er för $L(G_1) = L_1$ och $L(G_2) = L_2$ om:
(Stora bokstäver betecknar icke-terminerande symboler. Lilla a, b, c är terminerande symboler. S antas vara startsymbol)

- G_1:
  - $S \to TU$
  - $T \to Ta|\varepsilon$
  - $U \to b \cup c|\varepsilon$
- G_2:
  - $S \to TU$
  - $T \to aTb|\varepsilon$
  - $U \to cU|\varepsilon$

Exempel:
Låt $\Sigma = \{a, b, c\}$
Låt $L_1 = \{ww^{rev} | w \in \Sigma^*\}$ (palindrom av jämn längd)
Låt $L_2 = \{ww | w \in \Sigma^*\}$

Det vi vill illustrera med detta exempel är att ytlig likhet inte alls innebär att språket har samma egenskaper eftersom L_1 är en CFL medan L_2 *inte* är en CFL

En CFG för L_1:

$$S \to aSa|bSb|cSc|\varepsilon$$

Vi vill nu visa att L_2 inte är sammanhangsfritt, och genomför pumpsatsen för CFL snabbt.
Vi väljer den lämpliga strängen $w = a^K b^K c^K a^K b^K c^K$, så $w \in L_2$ och $|w| \geq K$
Antag att w nu delas upp i $uvxyz$ och $|vxy| \leq K$ och $vy \neq \varepsilon$
Då måste vxy vara en delsträng av $\{(a, b), (c, a), (b, c)\}$
Om vi pumpar ut, så kommer tecken försvinna från ett av blocken men inte i de anra. Den strängen som uppstår när man pumpar ur kommer inte tillhöra L_2, vilket motsäger pumpsatsen för CFL och L_2 är därmed inte en CFL.

15. Restriktionsfria språk

En grammatik är restriktionsfri om det bara finns ett krav på produktionsreglerna:

Definition/Sats 15.1: Restriktionsfri grammatik

En grammatik $G = (\Sigma_N, \Sigma_T, P, S)$ är *restriktionsfri* (eller en *RFG*) om det enda kravet på produktionsreglerna $u \to v$ i P är att u innehåller minst en icke-terminerande symbol (Σ_N).

När man väl har producerat en sträng som inte innehåller en icke-terminerande symbol är strängen klar, man kan inte göra något mer, just på grund av det kravet att u måste innehålla minst en icke-terminerande symbol.

Definition/Sats 15.2: Restriktionsfritt språk

Ett språk L är per definition restriktionsfritt (en *RFL*) om det finns en restriktionsfri grammatik (RFG) G sådant att $L = L(G)$ (mängden av alla strängar av de terminerande symbolerna som kan produceras av G)

Anmärkning:
Om vi tänker efter, en sammanhangsfri grammatik är restriktionsfri, för i en sammanhangsfri grammatik har vi alltid exakt en icke-terminerande i vänstersträngen. Då följer direkt att varje sammanhangsfritt språk är restriktionsfritt. Vi vet också att varje reguljärt språk är sammanhangsfritt.

Alltså, om L är ett reguljärt språk, $L \Rightarrow \text{CFL} \Rightarrow \text{RFL}$

Exempel:
Låt $L = \{a^n b^n c^n | n \in \mathbb{N}\}$. Vi såg att detta språk *inte* var sammanhangsfritt m.h.a pumpsatsen. Däremot är L faktiskt restriktionsfritt.
För att visa detta behöver vi konstruera en RFG G så att $L(G) = L$, ty då följer det att L är en RFL.

Vi konsruerar regler för G:

- $S \to \varepsilon | abNSc$ (S ersätts med $abNSc$) (kan producera $(abN)^k c^k \ \forall k \in \mathbb{N}$)
- $bNa \to abN$ (följande 3 regler är icke-sammanhangsfria regler, sammahangskänsliga)
- $bNb \to bbN$
- $bNc \to bc$

Exempel:
Betrakta följande språk: $L = \left\{1^{2^n} | n \in \mathbb{N}\right\}$
Bestäm om språket är reguljärt eller inte, sammanhangsfritt eller ej, och om det är restriktionsfritt så konstruera en restriktionsfri grammatik.

Vi vet att om språket är reguljärt, så kan vi dra slutsatsen att det är sammanhangsfritt, och om det är sammanhangsfritt så är det även restriktionsfritt. Däremot, om det inte är reguljärt, kan det hända att det fortfarande är sammanhangsfritt, vilket vi kan göra genom att konsruera en korrekt PDA/CFG. Om man inte tror att det är sammanhangsfritt så kan man visa det genom sammanhangsfria pumpsatsen.

Språket är inte sammanhangsfritt och därmed inte reguljärt, men vi kan visa detta med pumpsatsen för sammanhangsfria språk:

- L är uppenbarligen oändligq
- Antag att L är sammanhangsfritt
- Låt K vara givet från den sammanhangsfria pumpsatsen
- Välj jämplig sträng med längd minst K. Låt $w = 1^{2^K}$, så $w \in L$ och $|w| \geq K$
- Antag $w = uvxyz$ och $|vxy| \leq K$ och $vy \neq \varepsilon$
  Vi får då att om vi delar in en sträng med bara ettor i en sträng med längd 5 och pumpar den, så kommer vi inte längre ha ett jämt antal ettor:
  $2^K < \left|uv^2xy^2z\right| = \underbrace{|uvxyz|}_{2^K} + \underbrace{|vy|}_{\leq K} \leq 2^K + K < 2^K + 2^K < 2^{K+1}$

- Punkt 5 motsäger pumpsatsen så L är *inte* en CFL

En RFG för L:

- $S \to Q1T|1$
- $T \to TT|P$ FÖr att bli av medd T så måste man producera strängen 1 eller strängen $Q1P^n$ för något $n \in \mathbb{N}_+$
- $1P \to P11$
- $QP \to Q$
- $Q \to \varepsilon$

Exempel:
Låt $L = \{ww|w \in \{a,b\}^*\}$. Detta är *inte* en CFL (bevis finns längre upp). Vi kan testa producera en restriktionsfri grammatik för detta språk
Grundiden är ganska fin, tänk oss att vi till början betraktar A och B, och om vi skriver en sträng typ $abba$, men om vi bäddar in stora bokstav så att en stor bokstav följer efter en liten så vi får $aAbBbBaA$. I slutändan vill vi få $abbaabba$, vi behöver regler som separerar stor och liten bokstav, dvs flyttar alla stora bokstäver till höger *utan* att ändra ordningen på de, och till sist kan vi byta ut stor bokstav mot motsvarande lilla bokstav.

16. Turingmaskiner [TM:er]

Turingmaskinerna påminner mest om "vanliga" datorer och att pogrammera i maskindatorer. Skillnaden mellan turingmaskinerna och finita automater är att de kan skriva på sin texttape, och flytta läshuvudet inte bara på ett håll, men kan hoppa och gå fram och tillbaka.
Detta motsvarar att man har minne, inte bara skriva output och läsa input.

Informell beskrivning:
En TM har en tape som är oändlig i båda riktningar (höger och vänster, likt $\mathbb{Z}$):

$\cdots$	a	b	$\#$	$\cdots$

Varje ruta innehåller ett tecken varav ett kan vara "blankteknet" $\#$
TM har en kontrollmekanism som befinner sig i ett ändligt många möjliga tillstånd
Ändligt många tecken får användas på tapen.

Mängden av tecken som får användas för att skriva inputsträngar med kallas TM:s input-alfabet
Tecknet "#" ingår *inte* i inputalfabetet
Kontrollmekanismen och läshuvudet kan:

- Byta tillstånt (inkl stanna i samma tillstånd) och skriva tecken i rutan som avläses
- Byta tillstånd och flytta läshuvudet en ruta till vänster eller till höger

Vad en Turingmaskin gör i en given situation beror bara på vilket tecken den avläses, och vilket tillstånd den befinner sig i

En TM har (exakt) ett starttilstånd och (exakt) stopptillstånd (på Engelska "Halting state", betecknas oftast med H).

En TM sätts alltid igång i starttilståndet, och fortsätter att arbeta (göra tillståndsövergångar och eventuellt skriva på tapen) tills den hamnar i stopptillståndet, om detta sker.
Det är inte säkert att en TM någonsin hamnar i stopptillstånd, precis med vanliga datorer så behöver inte varje program terminera (exvis en while-true loop)

16.1. **Tape-konfigurationer.**

Det kommer vara bekvämt att ha notation och veta vad som står på tapen. Vi kommer alltid anta att ändligt många rutor är icke-blanka. I någon mening betyder det att vi inte använt mer än ändlig del av minnet, samma med inputsträng (ändlig)

Vi vill också kunna säga vart läshuvudet står i förhållande till de icke-tomma rutorna på tapen.

Vi antar att TM:s tape bara har ändligt många icke-blanka rutor när den startas (och därmed vid varje senare tillfälle)

Om tapen ser ut så här:

$\cdots$	$\#$	σ_1	$\cdots$	σ_k	$\cdots$	σ_n	$\#$	$\cdots$

och $\sigma_1 \neq \#$, $\sigma_n \neq \#$ och alla turot till vänster om σ_1 och till höger om σ_n är blanka, så är tapekonfigugariotnen

$$\sigma_1 \cdots \sigma_k \cdots \sigma_n$$

Här är läshuvudet på σ_k

Om tapen ser ut så här:

$\cdots$	$\#$	σ_1	$\cdots$	σ_n	$\#$	$\cdots$

och $\sigma_n \neq \#$ och alla rutor till vänster om σ_1 är blanka och alla rutor till höger om σ_n är blanka, så är tapekonfigurationen:

$$\sigma_1 \cdots \sigma_n$$

Notera att läshuvudet står på σ_1. Notera även att σ_1 kan vara blankteknet

Om tapen ser ut såhär:

$\cdots$	$\#$	σ_1	$\cdots$	σ_n	$\#$	$\cdots$

och $\sigma_1 \neq \#$ och alla rutor till vänster om σ_1 och till höger om σ_n är blanka, så är tapekonfigurationen:

$$\sigma_1 \cdots \sigma_n$$

Notera att läshuvudet står på σ_n här.

Om TM:en vid ett tillfäle befinner sig i tillstånd p och har tapekonfigurationen $x\sigma y$ så är **TM:ens konfiguration** vid detta tillfälle helt enkelt paret $(p, x\sigma y)$
(x och y antas vara strängar och σ ett tecken/symbol)

Vet man TM:ens konfiguration vet vi hur den kommer arbeta n-steg framåt, konfigurationen bestämmer dess fortsatta arbete.

16.2. **Grafiska beskrivningar a TM:er.**

Vi undersöker grafisk notation vs vad det betyder:

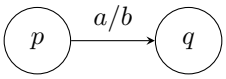


FIGURE 68. Betyder att a byts mot b vid tillståndsövergång

Betyder helt enkelt att den övergår från tillstånd p till tillstånd q och skriver b i rutan som avläses (om det stod a i den rutan innan)

Vi kan även skriva:

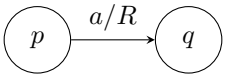


FIGURE 69. Flyttar läshuvudet till höger (Right)

Övergår från p till q och flyttar läshuvudet en ruta till höger, förutsatt att det stod a i rutan som avlästes innan.

Vi kan även skriva (för vänster):

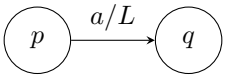


FIGURE 70. Flyttar läshuvudet till vänster (Left)

För att det inte ska bli för många pilar i diagrammen, så kan vi använda oss av förkortningar:

Följande:

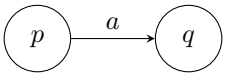


FIGURE 71.

är en förkortning för:

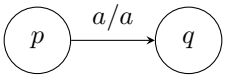


FIGURE 72.

Följande:

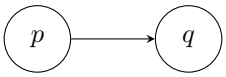


FIGURE 73.

är en förkortning för:

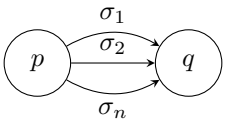


FIGURE 74.

Följande:

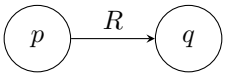


FIGURE 75.

är en förkortning för:

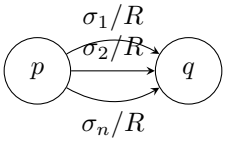


FIGURE 76.

Följande:

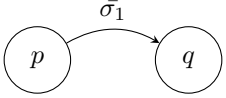


FIGURE 77.

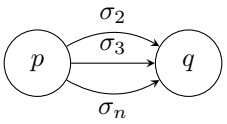


FIGURE 78.

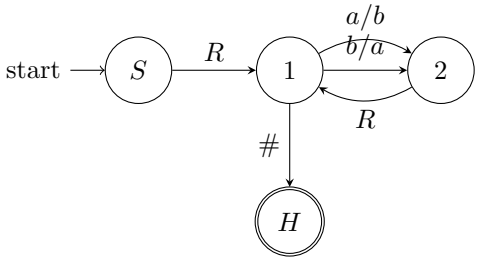


FIGURE 79.

är en förkortning för:

Exempel:
Låt TM M beskrivas av diagrammet/"grafen":

Vi gör en "körning" på följande inputsträng "$abba$":
Turingmaskinen startar alltid på S, och börjar alltid med läshuvudet på det första blanka rutan till vänster om inputsträngen (samma konvention som finita automater):

- $(S, \#abba)$
- $(1, \mathbf{a}bba)$
- $(2, \mathbf{b}bba)$
- $(1, b\mathbf{b}ba)$
- $(2, b\mathbf{a}ba)$
- $(1, ba\mathbf{b}a)$
- $(2, ba\mathbf{a}a)$
- $(1, baa\mathbf{a})$
- $(2, baa\mathbf{b})$
- $(1, baab\#)$
- $(H, baab\#)$

När TM har kommit till stopptillståndet så stannar den.
Denna TM har bytt ut a mot b.

Anmärkning:
Vi kommer bara titta på deterministiska TM i denna kurs (man måste tala om vad den ska göra i varje tillstånd i kombination med varje tecken i alfabetet).

Några standard TM med egna namn:

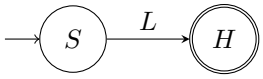


FIGURE 80. L - Flyttar läshuvudet ett steg till vänster

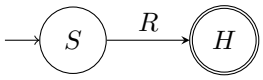


FIGURE 81. R - Flyttar läshuvudet ett steg till höger

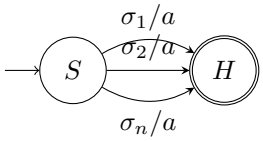


FIGURE 82. a - Skriver a i rutan som avläses

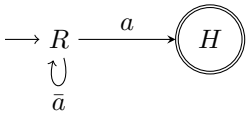


FIGURE 83. R_a - Söker efter första rutan till höger som har ett a, varpå termination

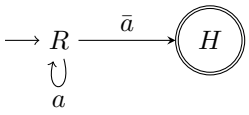


FIGURE 84. $R_{\bar{a}}$ - Söker efter första rutan till höger som inte innehåller ett a, varpå termination

Figure 83 och Figure 84 kallas för *teckenletare*.
L_a och $L_{\bar{a}}$ fungerar på samma sätt, förutom att de söker till vänster istället för till höger.

Följande kallas en "vänsterskiftare", och betecknas med S_L. Det finns givetvis motsvarande som kallas för "högerskiftare":

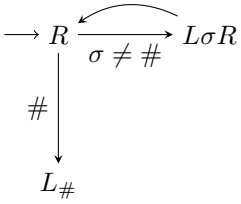


FIGURE 85.

Denna omvandlar tapekonfigurationen $ua\sigma v$, där a, σ är tecken och u, v är strängar, till tapekonfigurationen $u\sigma v\#$ (om $\sigma \neq \#$)

16.3. **Seriekoppling av TM.**

Antag att vi har två TM M_1 och M_2 som beskrivs grafiskt
Seriekopplingen ska börja på M_1 starttilstånd och gå till M_1 stopptillstånd, men istället för att stoppa så hoppar den över till M_2 starttilstånd *utan* att förändra vad som står på tapen.
Detta kallar vi för *seriekoppling* mellan TM:arna, vi får då en ny TM, som vi betecknar M_1M_2

Exempel:
Betrakta Turingmaskinen Ra, som flyttar läshuvudet till höger och skriver a, sedan stannar den. R och a är separata TM
Vi kan ha en annan, exvis RRa, som flyttar läshuvudet 2 steg till höger, och skriver a

Exempel:
Konstruera en TM som terminerar om inputsträngen (inputalfabetet innehåller a, b) innehåller *minst* 2st a:

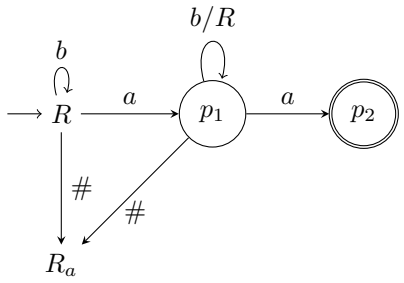


FIGURE 86.

17. Aritmetiska beräkningar i det binära systemet (1-systemet)

Talet n representeras av strängen 1^n (0 representeras av ε)

$R_\# S_L L_\#$ är en *additionsmaskin.*

Definition/Sats 17.1: Monus

Funktionen:

$$f(n,m) = \begin{cases} n-m, \text{ om } n \geq m \\ 0 \text{ annars} \end{cases}$$

kallas för *monus*

Följande är ett exempel på hur en monusmaskin ser ut:

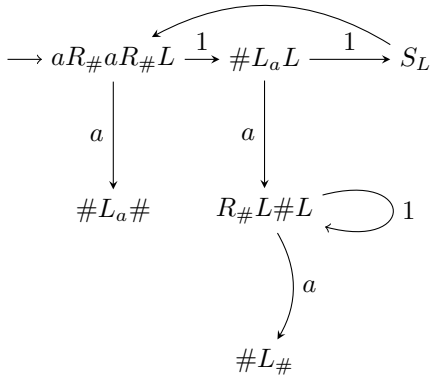


Figure 87.

Denna omvandlar $\#1^n\#1^m$ till $\#1^{n-m}$ om $n \geq m$, och annars till $\#$

Övning:
Tillverka en multiplikationsmaskin

17.1. **TM:ar och restriktionsfria grammatiker.**

Definition/Sats 17.2

Låt L vara ett språk.
Det finns en TM som accepterar L om och endast om det finns en restriktionsfri grammatik G som producerar L

17.2. **En universell TM.**

Om M är en TM och w en sträng, så låt $M(w)$ beteckna output-strängen som M ger om w ges som input.
(Om M aldrig terminerar efter start på w så är $M(w)$ odefinierad)

Låt $A = \{a_0, a_1, \cdots\}$ vara en oändlig mängd av symboler och $Q = \{q_0, q_1, \cdots\}$ en oändlig mängd av tillstånd.
Vi kan anta att varje TM:s tapealfabet är en delmängd av A och dess tillstånd utgör en delmäng av Q

Symbolerna $a_0, a_1, \cdots$ kan kodas som strängar över $\{0,1\}$ och tillstånd $q_0, q_1, \cdots$ kan också kodas som strängar över $\{0,1\}$

Då kan också varje TM:s tillståndsövergångar kodas som strängar över $\{0,1\}$

Det följer att varje TM M kan kodas som en sträng $K_M \in \{0,1\}^*$ och varje sträng w över M:s tapealfabet kan kodas som en sträng $K_w \in \{0,1\}^*$

Definition/Sats 17.3: Universell TM

Det finns en så kallad *universell TM* U med följande egenskap:

Om M är en godtycklig TM och w en sträng i M:s tapealfabet och U startas på tapekonfigurationen

$$\#K_m\#K_w$$

så stannar U på tapekonfigurationen

$$\#K_{M(w)}$$

om M stannar då w ges som input

(Annars stannar inte U då $K_M\#K_w$ ges som input)

17.3. **Funktioner.**

Låt $f : \mathbb{N}^k \to \mathbb{N}$ vara en partiell funktion. Vi säger att en TM M, vars input-alfabet är $\{1\}$ *beräknar* f om, för varje $k-tupel$ $(n_1, \cdots, n_k) \in \mathbb{N}^k$, M vid start på $\#1^{n_1}\#\cdots\#1^{n_k}$ stannar med tapekonfigurationen $\#1^{f(n_1,\cdots,n_k)}$ om $f(n_1, \cdots, n_k)$ är definierad, och M inte terminerar i annat fall

17.4. **Avgörbarhet.**

Låt L vara ett språk

Definition/Sats 17.4

Om det finns en TM M som vid start på tapekonfigurationen $\#w$ ger output

$$\#\text{ja} \quad \text{om } w \in L$$
$$\#\text{nej} \quad \text{om } w \notin L$$

så säger vi att L är *avgörbart*, annars är L *oavgörbart*

Anmärkning:
Om L accepteras av någno TM så säger vi att L är accepterbart

Definition/Sats 17.5

Om L är avgörbart så är $\bar{L}$ avgörbart och L accepterbart

Definition/Sats 17.6

Om både L och $\bar{L}$ är accepterbara så är L avgörbart

Bevisskiss:
Antag att M_1 accepterar L och M_2 accepterar $\bar{L}$
Låt M vara en TM som på input w kopierar w till $\#w\#w$ och kör igång M_1 på den första kopian av w och M_2 på den andra kopian av w

Så M_1 och M_2 arbetar "parallellt" på varsin kopia av w
Förr eller senare så stannar precis en av M_1 eller M_2 (eftersom $w \in L$ eller $w \in \bar{L}$)
Om M_1 stannar så skriver M "ja" på tapen och stannar
Om M_2 stannar så skriver M "nej" på tapen och stannar
Detta M avgör L

18. Stopp-problemet

Låt:

- $L_{\text{stopp}} = \{(M, w) \mid M \text{ är en TM och } w \text{ en sträng sådan att } M \text{ stannar vid start på } w\}$

Anmärkning:
Med (M, w) menar vi egentligen koden för M och koden för w (dvs $K_M \# K_w$)

Den universella TM:en U accepterar L_{stopp}, så L_{stopp} är accepterbart

Definition/Sats 18.1

L_{stopp} är *inte* avgörbart

Proof 18.1

Antag för motsägelse att L_{stopp} är avgörbart. Då finns en TM SUPER som vid start på $K_M \# K_w$ svarar "ja" om M stannar vid start på w och svarar "nej" annars

DÅ kan vi konsruera en ny TM SUPER$'$ som fungerar som SUPER *förutom* att när SUPER stannar och svarar "ja", så leds SUPER$'$ in i en oändlig slinga (dvs, stannar inte)

Sedan konstruerar vi en TM STUPER med följande funktion:
STUPER kopierar sitt input w så att $\#w$ övergår till $\#w\#w$ och sedan körs SUPER$'$ igång

Nu frågar vi oss vad som händer om w är koden för STUPER och SUPER får $w\#w$ som input:

- Antag att SUPER svarar "ja" vid start på $w\#w$. Då stannar STUPER vid start på w, så SUPER$'$ stannar vid start på $w\#w$ vilket betyder att SUPER stannar med svaret "nej" vid start på $w\#w$. Detta motsäger antagandet
- Antag att SUPER svarar "nej" vid start på $w\#w$. Då stannar inte STUPER vid start på w, så SUPER$'$ stannar inte vid start på $w\#w$ vilket betyder att SUPER svarar "ja" vid start på $w\#w$. Detta motsäger antagandet

Vi får alltså under alla omständigheter en motsägelse, så L_{stopp} kan inte vara avgörbart □

Anmärkning:
L_{stopp} accepterbart men ej avgörbart $\Rightarrow \bar{L}_{\text{stopp}}$ är ej accepterbart

19. Rices Sats

Definition/Sats 19.1

Antagg att Ω är en icke-tom mängd av accepterbara språk och att något accepterbart språk inte tillhör Ω

Då finns ingen TM som avgör för en godtycklig TM M om $L(M)$ tillhör Ω eller ej

Annorlunda uttryckt, språket

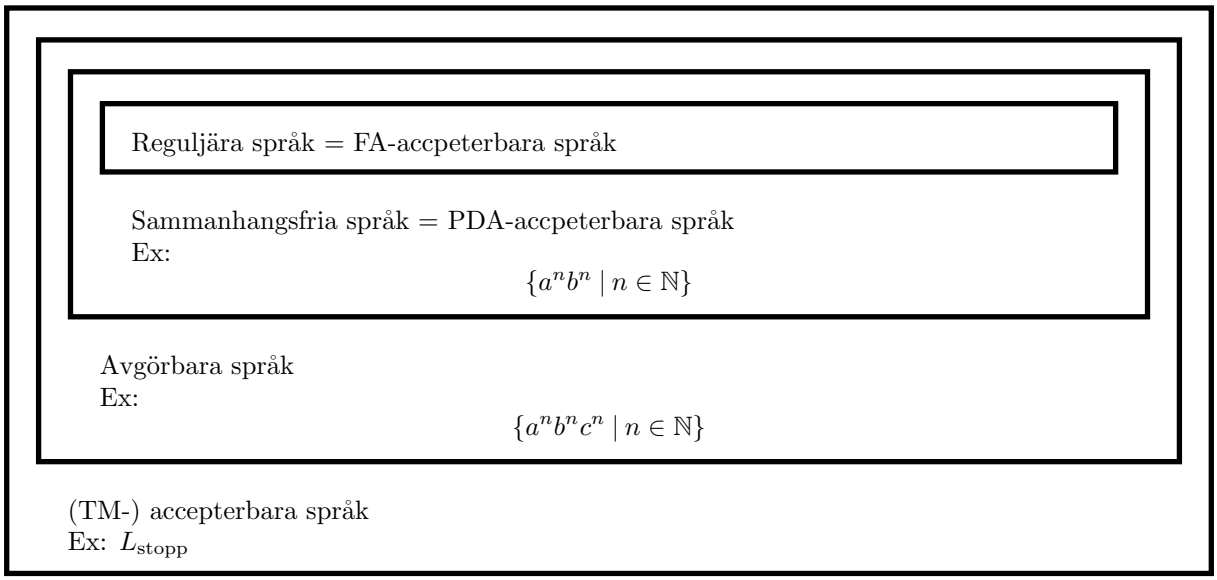
$$L_\Omega = \{K_M \mid M \text{ är en TM och } L(M) \in \Omega\}$$

är oavgörbart

20. Chomskys språkhierarki

Vi har följande inklusioner ($\subset$) och likheter ($=$) mellan de språkklasser som vi har betraktat:

Reguljära språk = FA-accpeterbara språk $\subset$ Sammanhangsfria språk = PDA-accepterbara språk $\subset$ Avgörbara språk $\subset$ (TM-) accepterbara språk = Restriktionsfria språk



Ej (TM-) accepterbara språk
Ex: $\bar{L}_{\text{stopp}}$