# 6 Formella språk

Det mänskliga språket är ett komplext system av komponenter som ljud, skrivtecken, ord, grammatik, satsmelodi med mera. Instruktionerna för en dator däremot formuleras på ett ganska enkelt språk bestående av ettor och nollor. När vi vill använda oss av datorer för att behandla mänskligt språk måste vi därför försöka hitta något ”mellanspråk” som å ena sidan är tillräckligt enkelt för att kunna översättas till maskinkod, och å andra sidan tillräckligt komplext för att fånga så många aspekter av naturliga språk som möjligt. Sådana ”mellanspråk” kallas **formella språk**. De är glasklara exempel på matematiska modeller enligt den karakteriseringen som vi såg i första föreläsningen: De måste nödvändigtvis vara abstrakta (och kan inte fånga in all komplexitet bakom det mänskliga språket); de måste vara formella (så att de kan implementeras på en dator); och de bör vara användbara i praktiska tillämpningar.

I den här delen av kursen kommer vi att höra om två typer av formella språk som kallas **reguljära språk** och **kontextfria språk**. Båda typer av språk kan beskrivas på flera olika sätt.

I samband med reguljära språk kommer vi att prata om **finita automater** och **reguljära uttryck**. Ett sätt att se på automater är att se dem som maskiner som avgör om ett visst uttryck är korrekt. Reguljära uttryck kan ses som någon form av ”minigrammatik” som beskriver vilka uttryck är korrekta. Vi kommer att se att finita automater och reguljära uttryck är ekvivalenta på det sättet att ett uttryck accepteras av en finit automat om och endast om det kan beskrivas genom ett reguljärt uttryck. I vissa sammanhang är det mera praktiskt att använda sig av automater, i andra sammanhang bör man föredra reguljära uttryck.

Kontextfria språk kan också beskrivas genom (mera avancerade) automater; men på den här kursen kommer vi att fokusera på deras beskrivning genom **kontextfria grammatiker**. Vi kommer att se att dessa grammatiker är mera expressiva än både finita automater och reguljära uttryck i det avseende att de kan beskriva alla reguljära språk, men att det tvärtom finns kontextfria språk som inte kan beskrivas genom varken automater eller reguljära uttr

Vi börjar med att lite mera precist definiera vad vi menar med ett formellt språk.

## 6.1 Grundläggande begrepp

En **sträng** är en följd av symboler tagna ur något **alfabet**. Ett exempel på en sträng är `apa`, som består av 2 symboler ur det svenska alfabetet, en av vilka (`a`) förekommer två gånger. Notera att man skiljer mellan symboler och deras förekomster.

När man skriver `a` kan man mena både `a` som symbol och den sträng som består av en enda symbol `a`.

Det är inte väsentligt att alfabetet är kopplat till något naturligt språk. Vi kommer att prata mycket om strängar där symbolerna kommer från mer eller mindre artificiella alfabet som det binära alfabetet $\{0, 1\}$. Ett exempel på en sträng över detta alfabet är `101`. Det enda som är väsentligt är att alfabetet är en mängd.

Ett viktigt specificalfall på strängar är den **tomma strängen**, strängen som inte innehåller några symboler alls. Den tomma strängen skrivs $\lambda$ (Grekiskans *lambda*; i Eriksson och Gavel) eller $\varepsilon$ (Grekiskans *epsilon*; i de flesta andra böckerna).

Mängden av alla strängar över ett alfabet $\Sigma$ betecknas med $\Sigma^*$. Ett exempel:

$$\{\mathtt{a}, \mathtt{b}\}^* = \{\lambda, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{bb}, \mathtt{ab}, \mathtt{ba}, \ldots\}$$

Vi kommer att använda variabler som $u, v, w, x, y, z$ för strängar.

En ny sträng kan bildas genom **sammansättning** av två andra strängar. Sammansättning av `01` och `001` till exempel ger `01001`. Om $x$ och $y$ betecknar strängar, så skriver vi $x \cdot y$ eller oftast bara $xy$ för att beteckna sammansättningen av $x$ och $y$. Notera att $xy$ inte nödvändigtvis är samma sak som $yx$.

**Längden** hos en sträng $u$ skrivs som $|u|$. Längden kan definieras rekursivt:

$$|u| = \begin{cases} 0 & \text{om } u = \lambda \\ 1 + |v| & \text{om } u \text{ är på formen } \mathtt{a}v \end{cases}$$

När man säger att "$u$ är på formen $\mathtt{a}v$" menar man att $u$ är en sträng som är sammansatt av en enda symbol `a` och en annan sträng $v$.

Ett **språk** modelleras som en mängd av strängar, vilka kallas för **ord**. Ett exempel på ett språk är $\{\mathtt{Per}, \mathtt{Pia}, \mathtt{Pål}\}$. Detta språk består av tre strängar över det svenska alfabetet; dessa tre strängar är språkens ord. Ett mera komplicerat exempel är det så kallade **kopispråket**:

$$\{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$$

> **Fråga:** Kan du förklara hur orden i detta språk ser ut?
>
> Orden i kopispråket består av alla strängar över alfabetet $\{\mathtt{a}, \mathtt{b}\}$ där den ena halvan är en exakt kopia av den andra halvan.

När vi använder uttrycket ”språk” menar vi oftast att det finns en grammatik eller någon annan specifikation som anger vilka strängar ingår i språket och vilka inte gör det.

# 7 Reguljära språk

## 7.1 Finita automater

Det är 1980. Ni jobbar på ett företag som säljer ett ordbehandlingsprogram vid namnet *SuperWord*. En dag kommer chefen in med en lysande idé för hur man kan göra den nästa versionen av *SuperWord* ännu mera attraktiv: Lägg till en stavningskontroll! Den ska gå genom en text som användaren har skrivit och för varje ord i texten antingen acceptera det eller refusera det som felstavat.

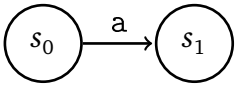
> **Fråga:** Hur ska ni gå tillväga för att implementera stavningskontrollen?

Ett sätt att implementera stavningskontrollen är att samla ihop en lista med ord i svenskan och låta datorn acceptera alla ord som finns med i listan och refusera alla andra. Ord*former*, närmare sagt, för i svenskan kan ord som substantiv, verb och adjektiv böjas enligt vissa regler – ordet *apa* till exempel har åtta olika former: *apa*, *apas*, *apan*, *apans*, *apor*, *apors*, *aporna*, *apornas*. Man kan tänka sig att listan kommer att bli väldigt lång. Ett annat skäl till att lösningen med en lista är otillfredsställande är att den inte fångar den mycket systematiska uppbyggnaden av svenska ordformer. Ord som *flicka*, *docka* och *klocka* följer ju precis samma mönster när det gäller formbildningen.

### 7.1.1 Terminologi

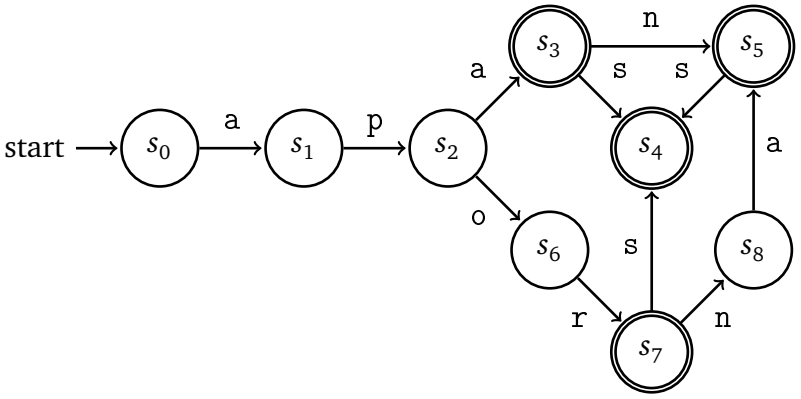
Ett bättre sätt att implementera en stavningskontroll är att använda **finita automater**. En finit automat för ordformerna hos *apa* finns i Figur 1. Figuren visar en riktad graf. Noderna i denna graf kallas **tillstånd** och bågarna kallas **övergångar**.

En övergång




kallas `a`**-övergång**. Det finns ett **starttillstånd** (tillstånd $s_0$ i diagrammet) och noll eller flera **accepterande tillstånd** (i diagrammet markerade genom en extra ring).

Automaten **accepterar** en sträng om denna leder till ett accepterande tillstånd, givet att man börjar i starttillståndet. Automaten i diagrammet accepterar till exempel teckenföljden `apas` eftersom man kan börja i tillstånd $s_0$, ta `a`-övergången till $s_1$, ta `p`-övergången till $s_2$, ta `a`-övergången till $s_3$ och avsluta med `s`-övergången till $s_4$ som är ett accepterande tillstånd. Den accepterar även `apa`, men däremot inte `ap`.




Figur 1: En finit automat som beskriver ordformerna hos *apa*.

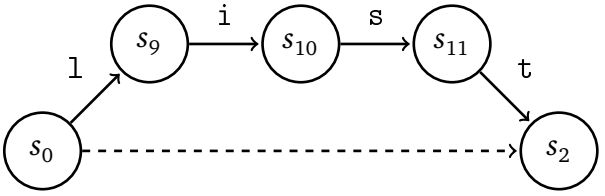
Mängden av alla strängar som accepteras av automaten kallas för automatens **språk**. Om $M$ betecknar en finit automat skriver man $L(M)$ för det av $M$ accepterade språket. Vår automat till exempel accepterar språket

$$\{\texttt{apa}, \texttt{apas}, \texttt{apan}, \texttt{apans}, \texttt{apor}, \texttt{apors}, \texttt{aporna}, \texttt{apornas}\}$$

Men på vilket sätt är finita automater bättre än listor på ordformer?

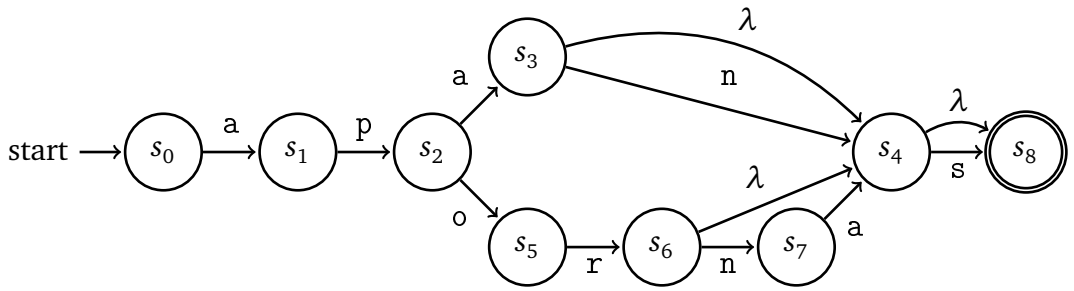
**Fråga:** Hur måste ni modifiera automaten så att den även accepterar alla ordformer hos ordet *lista*?

Man måste utvidga automaten så att teckenföljden `list` leder en från starttillståndet $s_0$ till tillståndet $s_2$:




Resten av automaten kan förbli som den är, eftersom ordformerna hos *lista* bildas på precis samma sätt som ordformerna hos *apa*. Man kan se att om man lägger till många ord av samma slag (*flicka*, *docka*, *klocka*) så ger en finit automat en mycket mera kompakt beskrivning av ordformerna än en explicit lista.

Vår automat för *apa* accepterar ett finit språk, men det finns även automater som accepterar oändligt stora språk.

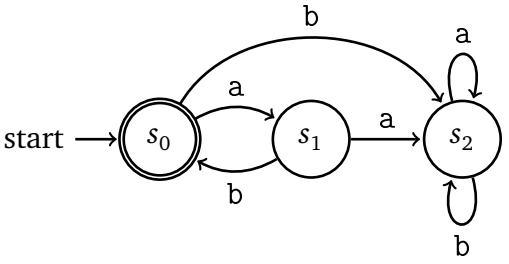



Figur 2: En annan finit automat som beskriver ordformerna hos *apa*.

**Fråga:** Kan du konstruera en automat som accepterar följande språk:

$$\{\underbrace{\texttt{ab}\cdots\texttt{ab}}_{n \text{ gånger}} \mid n \geq 0\}$$

Här är ett exempel:



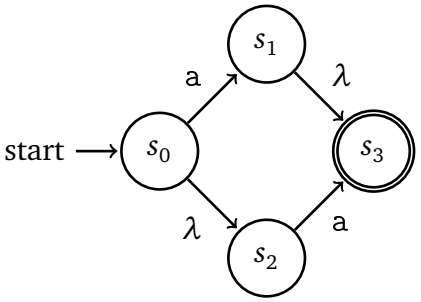

### 7.1.2 λ-övergångar

I vår automat för ordformerna hos ordet *apa* motsvarar varje övergång en symbol. Ibland vill man också tillåta en annan form av övergångar som kallas för **λ-övergångar**. Sådana övergångar får man ta utan att läsa någon symbol i strängen. En alternativ version av vår automat för *apa* som har λ-övergångar visas i Figur 2.

**Fråga:** Kan du förklara varför den nya automaten accepterar strängen `apa`?

Strängen leder till det accepterande tillståndet $s_8$: Man börjar i starttillståndet $s_0$. Sedan läser man de första tre symbolerna `apa` för att komma till tillståndet $s_3$. Nu har man läst alla symboler i strängen; men $s_3$ är inget accepterande tillståndet. För att komma till det accepterande tillståndet $s_8$ följer man två λ-övergångar: från $s_3$ till $s_4$ och från $s_4$ till $s_8$.

I en automat med $\lambda$-övergångar finns det inte nödvändigtvis en entydig väg från starttillståndet. Detta ser man i exemplet nedan:




**Fråga:** Vilket språk accepteras av denna automat?

Språket som accepteras är {a}, men för strängen a finns det två vägar genom grafen: Antingen går man den övre vägen ($s_0$–$s_1$–$s_3$) eller den undre vägen ($s_0$–$s_2$–$s_3$).

$\lambda$-övergångar gör inte finita automater mera uttrycksfulla på något sätt. För varje finit automat med $\lambda$-övergångar kan man konstruera en ny finit automat *utan* $\lambda$-övergångar sådan att den nya automaten accepterar samma språk som den gamla.

## 7.2 Reguljära uttryck

Från kursen *Introduktion till datatektik för språkvetare* känner ni igen **reguljära uttryck**. Dessa kan man använda i program som `grep` för att söka efter ett visst mönster i en textfil.

**Fråga:** Vilket mönster matchar den följande regexpen:

```
(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
```

Regexpen matchar alla strängar över alfabetet {0, . . . , 9} som representerar positiva naturliga tal. Dessa består av en siffra 1–9, följd av godtyckligt många (kanske inga!) siffror 0–9. Notera att strängen 0 matchas inte av den här regexpen.

I det här avsnittet kommer vi att behandla reguljära uttryck lite mera formellt. De uttrycken som vi behandlar är en någorlunda förenklad version av de regexpar som används i t. ex. `grep`; detta eftersom vi vill inte ha det alltför svårt. De flesta reguljära uttryck som används i praktiken kan fås som kombinationer av de enkla reguljära uttryck som vi behandlar här.

### 7.2.1 Syntax för reguljära uttryck

Vi börjar med att definiera vad vi menar med ett reguljärt uttryck:

1. Den tomma strängen $\lambda$ är ett reguljärt uttryck.
2. Varje enskild symbol i alfabetet $A$ är ett reguljärt uttryck.
3. Om $R_1$ och $R_2$ är reguljära uttryck är även $(R_1 + R_2)$ ett reguljärt uttryck.
4. Om $R_1$ och $R_2$ är reguljära uttryck är även $(R_1 \cdot R_2)$ ett reguljärt uttryck.
5. Om $R$ är ett reguljärt uttryck är även $R^*$ ett reguljärt uttryck.

Onödiga parenteser kan utelämnas.

**Fråga:** Denna definition är ett exempel på en rekursiv definition. Vad är basfallen?

De första två fallen är basfallen.

### 7.2.2 Två operationer på språk

Precis som ett uttryck som $(2+3)\cdot 4$ står för ett tal står varje reguljärt uttryck för ett språk, dvs. en mängd av strängar över ett visst alfabet. För att förklara hur detta fungerar behöver vi två operationer på språk: **sammansättning** av språk och den så kallade **Kleenestjärnan**.

### 7.2.3 Sammansättning av språk

Sammansättningen av två språk $L_1$ och $L_2$ består av alla ord som kan bildas genom att sätta ihop ett ord ur $L_1$ och ett ord ur $L_2$. Sammansättningen av $L_1$ och $L_2$ skrivs $L_1 \cdot L_2$ eller ibland bara $L_1L_2$. Med hjälp av mängdbyggaren kan vi definiera:

$$L_1 \cdot L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ och } w_2 \in L_2\}$$

Detta ska uttydas: "$L_1 \cdot L_2$ är mängden av alla strängar på formen $w_1w_2$ sådana att $w_1 \in L_1$ och $w_2 \in L_2$."

**Fråga:** Vad är $\{\texttt{auto}, \texttt{flyg}\} \cdot \{\texttt{mat}, \texttt{pilot}\}$?

$\{\texttt{auto}, \texttt{flyg}\} \cdot \{\texttt{mat}, \texttt{pilot}\} = \{\texttt{automat}, \texttt{autopilot}, \texttt{flygmat}, \texttt{flygpilot}\}$

### 7.2.4 Kleenestjärnan

**Kleenestjärnan** av ett språk $L$, som skrivs $L^*$, består av den tomma strängen och alla sammansättningar på formen $w_1 \cdots w_n$, där $n \geq 0$ och alla delsträngar $w_i$ kommer från $L$. Mera formellt kan Kleenestjärnan definieras enligt följande. Först definierar man en följd av språk $L_n$:

$$L_n = \begin{cases} \{\lambda\} & \text{om } n = 0 \\ \{ vw \mid v \in L_{n-1}, w \in L \} & \text{om } n > 0 \end{cases}$$

**Fråga:** Vad är $L_1$?

$L_1 = \{ vw \mid v \in L_0, w \in L \} = \{ vw \mid v \in \{\lambda\}, w \in L \} = \{ \lambda w \mid w \in L \} = L$

Sedan definierar man

$$L^* = L_0 \cup L_1 \cup L_2 \cup \cdots$$

**Fråga:** Vad är $\{0,1\}^*$? Kan du ange ett språk $L$ så att $L^*$ består av ett enda element?

Mängden $\{0,1\}^*$ är mängden av alla strängar över det binära alfabetet $\{0,1\}$, inklusive den tomma strängen $\lambda$. De enda språken $L$ för vilka $L^*$ består av ett enda element är $L = \emptyset$ och $L^* = \{\lambda\}$ – det senare eftersom $\{\lambda\}_n = \{\lambda\}$ för alla $n \geq 0$. Så fort $L$ innehåller någon sträng utöver $\lambda$ är $L^*$ infinit.

### 7.2.5 Semantik för reguljära uttryck

Nu kan vi säga vilket språk som beskrivs av ett reguljärt uttryck. Givet ett reguljärt uttryck $R$ skriver vi $L(R)$ för det språk som $R$ beskriver. Definitionen av $L(R)$ följer den rekursiva definitionen av $R$:

$$\begin{aligned} L(\lambda) &= \{\lambda\} \\ L(a) &= \{a\} \\ L(R_1 + R_2) &= L(R_1) \cup L(R_2) \\ L(R_1 \cdot R_2) &= L(R_1) \cdot L(R_2) \\ L(R^*) &= L(R)^* \end{aligned}$$

**Fråga:** Vilket språk beskriver det reguljära uttrycket $(0+1)^* \cdot 1$? Räkna!

Detta uttryck beskriver alla strängar över det binära alfabetet som slutar på 1:

$$\begin{aligned} L((0+1)^* \cdot 1) &= L((0+1)^*) \cdot L(1) \\ &= L(0+1)^* \cdot \{1\} \\ &= (L(0) \cup L(1))^* \cdot \{1\} \\ &= (\{0\} \cup \{1\})^* \cdot \{1\} \\ &= \{0,1\}^* \cdot \{1\} \end{aligned}$$

## 7.3 Reguljära uttryck och finita automater

Reguljära uttryck och finita automater är två sätt att karakterisera formella språk. En typisk matematisk fråga är då: Finns det språk som bara kan karakteriseras på det ena sättet, men inte på det andra? Svaret på denna fråga är ”nej”: Reguljära uttryck och finita automater är ekvivalenta på det sättet att varje språk som kan beskrivas med hjälp av ett reguljärt uttryck kan också beskrivas genom en finit automat, och vice versa. Reguljära uttryck och finita automater beskriver alltså exakt samma typ av språk. Dessa språk kallas för **reguljära språk**. De har viktiga tillämpningar inom språkteknologin. Vi har redan sett en tillämpning inom morfologin, men de är också väldigt viktiga i samband med informationssökning, *question answering* m.fl.

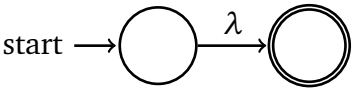
I det här avsnittet kommer vi att bevisa att varje språk som kan beskrivas med hjälp av ett reguljärt uttryck kan också beskrivas genom en finit automat. Den andra riktningen tar vi inte upp här i kursen.

Hur kan vi bevisa att varje språk som kan karakteriseras genom ett reguljärt uttryck kan också beskrivas genom en finit automat? Grundtaken är denna: För varje reguljärt uttryck ska vi bygga en finit automat som accepterar precis det språket som uttrycket representerar.

Det finns oändligt många reguljära uttryck – men de har en rekursiv definition! Vårt bevis kommer att vara en induktionsbevis som följer denna rekursiva definition. När man ser på den rekursiva definitionen av reguljära uttryck ser man att det finns två basfall och tre rekursiva fall. För basfallen kommer vi att bygga väldigt enkla automater. I de rekursiva fallen kommer vi sedan att anta att vi redan byggt automater för mindre reguljära uttryck och kombinerar dem till en automat för det komplexa uttrycket.

Då sätter vi igång!

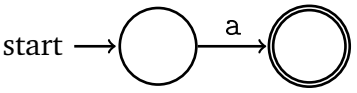
Fall 1: $\lambda$ Vi bygger en automat som accepterar språket $\{\lambda\}$. Vi tar:




Notera att denna automat har precis ett starttillstånd och ett accepterande tillstånd. I hela det här beviset kommer vi bara bygga automater med denna egenskap.

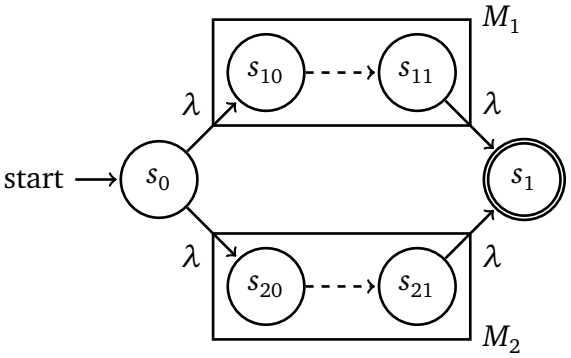
> **Fråga:** Kan du komma på en enklare automat som accepterar $\{\lambda\}$?
>
> Man kan ta automaten som består av ett enda tillstånd (som samtidigt är starttillstånd och accepterande tillstånd) och inte har några övergångar alls.

Fall 2: $a$ Vi bygger en automat som accepterar språket $\{a\}$. Vi tar:




Notera att även denna automat har precis ett starttillstånd och ett accepterande tillstånd.

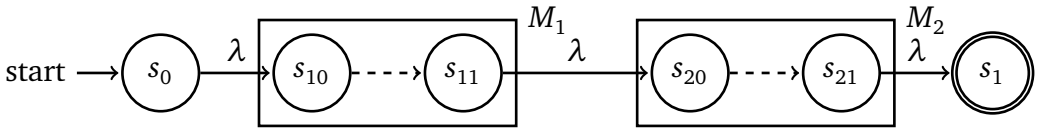
Fall 2: $R_1 + R_2$ Vi bygger en automat som accepterar språket $L(R_1) \cup L(R_2)$. Låt oss anta att vi redan har byggt en automat $M_1$ som accepterar språket $L(R_1)$ och en automat $M_2$ som accepterar språket $L(R_2)$. Vår nya automat baserar på $M_1$ och $M_2$. Låt $s_{10}$ beteckna starttillståndet i $M_1$ och låt $s_{11}$ beteckna det accepterande tillståndet i $M_1$. Låt på samma sätt $s_{20}$ och $s_{21}$ beteckna starttillståndet och det accepterande tillståndet i $M_2$. I den nya automaten kommer alla fyra tillstånd vara helt vanliga tillstånd, dvs. inte starttillstånd eller accepterande tillstånd. Istället lägger vi till två nya tillstånd $s_0$ och $s_1$, varav $s_0$ blir det nya starttillståndet och $s_1$ blir det nya accepterande tillståndet. Sedan lägger vi till fyra stycken $\lambda$-övergångar: en från $s_0$ till $s_{10}$, en från $s_0$ till $s_{20}$, en från $s_{11}$ till $s_1$ och en från $s_{21}$ till $s_1$. Schematiskt ser alltså den nya automaten ut så här:




Hur kan vi se att denna automat accepterar språket $L(R_1) \cup L(R_2)$? För att automaten ska acceptera en sträng måste det finnas en vandring i övergångsgrafen som börjar i

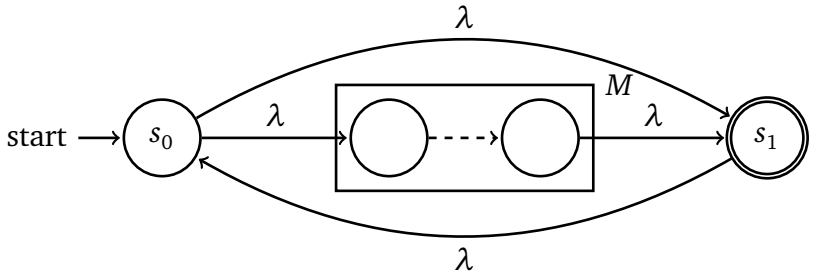
starttillståndet och slutar i det accepterande tillståndet. Det finns två sorters sådana vandringar: de som går genom automaten $M_1$ och de som går genom automaten $M_2$. Eftersom vi har antagit att språket som accepteras av $M_1$ är språket $L(R_1)$ vet vi att varje vandring som går genom $M_1$ är en accepterande vandring för ett ord i $L(R_1)$. På samma sätt är varje vandring som går genom $M_2$ en accepterande vandring för ett ord i $L(R_2)$. Vi drar slutsatsen att varje accepterande vandring genom den nya automaten är en accepterande vandring för antingen ett ord ur $L(R_1)$ eller ett ord ur $L(R_2)$. Alltså är varje accepterande vandring genom automaten en accepterande vandring för ett ord ur $L(R_1) \cup L(R_2)$.

Fall 4: $R_1 \cdot R_2$ Vi bygger en automat som accepterar språket $L(R_1) \cdot L(R_2)$. På samma sätt som i förra fallet antar vi att vi redan har byggt en automat $M_1$ som accepterar språket $L(R_1)$ och en automat $M_2$ som accepterar språket $L(R_2)$. Låt $s_{10}$, $s_{11}$, $s_{20}$ och $s_{21}$ vara definierade som i förra fallet, och låt $s_0$ och $s_1$ vara nya tillstånd som i förra fallet. Vår nya automat ska set ut så här:




På samma sätt som i förra fallet kan vi argumentera att varje accepterande vandring genom den nya automaten är en accepterande vandring för ett ord i språket $L(R_1) \cdot L(R_2)$ och att varje accepterande vandring för detta språk kan fås.

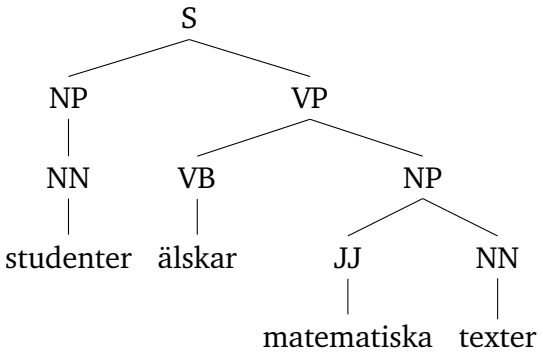
Fall 5: $R^*$ Vi bygger en automat som accepterar språket $L(R)^*$. Låt oss anta att vi redan har byggt en automat $M$ som accepterar språket $L(R)$. Vår nya automat bör se ut så här:




Med detta kan vi avsluta beviset.

## 7.4 Läsning och övningar

Eriksson och Gavel, kapitel 9 förutom 9.2.2.




Figur 3: Ett frasstrukturträd.

# 8 Kontextfria språk

Den syntaktiska strukturen hos en sats kan beskrivas genom **frasstrukturträd**. Ett exempel visas i Figur 3. Löven i ett frasstrukturträd representerar ord såsom *älskar* och *texter*; lövens föräldrarnoder representerar ordklasser såsom verbal basform (VB) och substantiv (NN); de övriga noderna representerar fraser. I exemplet representerar noden med symbolen S den fullständiga satsen *Studenter älskar matematiska texter*. Denna sats består av två delar, en nominalfras (NP) (som innehåller satsens subjekt *Studenter*) och en verbalfras (VP). Verbalfrasen förgrenar sig ytterligare i en verbal basform och en nominalfras (som innehåller satsens objekt *matematiska texter*).

Frasstruktur och den underliggande synen på syntaktisk struktur behandlas mer utförligt i mer lingvistiska kurser. I den här kursen ligger fokus på de formella aspekterna av frasstrukturgrammatiker.

## 8.1 Kontextfria grammatiker

När man tar en frasstrukturgrammatik och skalar bort alla lingvistiska intuitioner och metaforer får man en form av grammatik som kallas för **kontextfri grammatik**.

### 8.1.1 Grundläggande begrepp

Här är ett exempel på hur reglerna i en enkel kontextfri grammatik kan se ut:

$$S \to \lambda \qquad S \to aSb$$

I en kontextfri grammatik förekommer två sorters symboler (förutom den speciella symbolen $\to$ som används när man skriver ner reglerna):

- **icketerminala symboler** eller **icketerminaler** och

- **terminala symboler** eller **terminaler**

Icketerminaler brukar anges med versaler, terminaler brukar anges med gemener. I vår exempelgrammatik finns bara en enda icketerminal ($S$), men två terminaler ($a$ och $b$). När man använder kontextfria grammatiker som frasstrukturgrammatiker används terminaler för att representera orden i en mening, medan icketerminala symboler används för att representera ordklasser (såsom verbal basform och substantiv) fraser (såsom nominalfras och verbalfras).

Varje kontextfri grammatik innehåller en speciell icketerminal som kallas **startsymbol**. Det finns en konvention att alltid beteckna denna startsymbol med $S$, och denna konvention följer vi även i den här kursen.

Varje regel i en kontextfri grammatik är på formen $A \to \alpha$, där $A$ är en enskild icketerminal symbol och $\alpha$ är en sträng bestående av icketerminaler och terminaler. Observera att strängen $\alpha$ kan också vara den tomma strängen, som i regeln $S \to \lambda$ ovan.

Reglerna används som omskrivningsregler, på följande sätt. Man börjar med grammatikens startsymbol. Så länge man har en sträng $\beta$ som innehåller åtminstone en icketerminal symbol $A$ väljer man en regel som har $A$ som vänstersida och ersätter någon förekomst av $A$ i $\beta$ genom den valda regelns högersida. Till slut kommer man då till en sträng som bara innehåller terminala symboler. Mängden av alla strängar som man kan härleda på detta sätt kallas för grammatikens **språk**. Om $G$ betecknar en kontextfri grammatik skriver vi $L(G)$ för att beteckna $G$:s språk.

Det finns två sätt att se på en kontextfri grammatik: Man kan se den som en modell för att *producera* ett språk och som en modell för att *beskriva* ett språk.

> **Fråga:** Vilket språk beskriver exempelgrammatiken?
>
> Språket $\{a^n b^n \mid n \geq 0\}$, som består av strängar med jämn längd vars första halva består av bara $a$:s och vars andra halva består av bara $b$:s.

Under föreläsningen har vi pratat om det så kallade **kopispråket** $\{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$. Detta språk består av alla strängar över alfabetet $\{\mathtt{a}, \mathtt{b}\}$ som kan delas i två delsträngar såsom att den andra delsträngen är en exakt kopia på den första delsträngen. Ett språk som tillsynes är ganska likt kopispråket är det **inverterade kopispråket** eller **palindromspråket**

$$\{ww^R \mid w \in \{\mathtt{a}, \mathtt{b}^*\}\}.$$

Notationen $w^R$ står för strängen som man får genom att vända på strängen $w$ (eng. *revert*). Det inverterade kopispråket består alltså av alla strängar över alfabetet $\{\mathtt{a}, \mathtt{b}\}$ som kan delas i två delsträngar såsom att den andra delsträngen är en omvänd kopia på den första delsträngen. Sådana strängar kallas även för **palindrom**: man får ut samma sträng om man läser från vänster eller från höger.

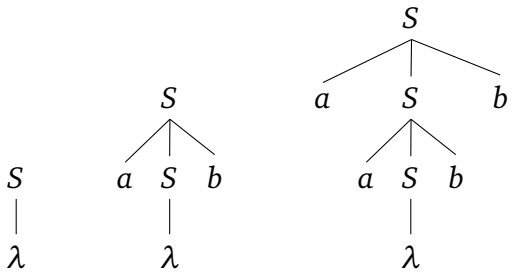
**Fråga:** Kan du ange några ord som ingår i det inverterade kopispråket? Kan du ange en kontextfri grammatik som genererar det inverterade kopispråket?

Några exempel är $\lambda$, `aa`, `bb`, `abba`, `baab`. En grammatik som genererar det inverterade kopispråket har produktionerna $S \to \lambda$, $S \to \mathtt{a}\,S\,\mathtt{a}$, $S \to \mathtt{b}\,S\,\mathtt{b}$.

Kopispråket och palindromspråket är relevanta när man modellerar bisatser i språk som tyska och holländska.

### 8.1.2 Parseträd

Ett sätt att beskriva en härledning i en kontextfri grammatik är att rita ett **parseträd**. Här är tre exempel på parseträd för exempelgrammatiken:




Parseträd är rotade träd vars noder är markerade med icketerminaler och terminaler. Rotnoden är alltid markerad med startsymbolen. Varje gång man i en härledning använder en regel på formen $A \to x_1 \cdots x_m$ för att skriva om en icketerminal $A$ till en följd av symboler $x_1, \ldots, x_m$ gör man en förgrening i parseträdet och skriver ner $x_1, \ldots, x_m$ som barn till noden som representerar $A$. I specialfallet där $A$ skrivs om till den tomma strängen får $A$ ett enda barn som är markerad med $\lambda$.

**Fråga:** Om man har ritat ett parseträd för en härledning av en sträng med $n$ symboler, hur många löv har då trädet?

Trädet har då åtminstone $n$ löv. Det kan ha fler än $n$ löv, eftersom några av löven kan vara markerade med den tomma strängen.

## 8.2 Språkhierarkier

Vi ska avrunda denna del av kursen med att titta närmare på relationen mellan reguljära och kontextfria språk. Vi ska se att alla reguljära språk kan beskrivas med hjälp av kontextfria grammatiker, men att det finns kontextfria språk som inte är reguljära, dvs. kan beskrivas varken med finita automater eller med reguljära uttryck.

### 8.2.1 Varje reguljärt språk är ett kontextfritt språk

Vi börjar med att visa att varje reguljärt språk också är ett kontextfritt språk. Ett språk är ju reguljärt om och endast om det accepteras av en finit automat av det slaget som vi sett i avsnitt 3.1. För att visa att varje reguljärt språk är ett kontextfritt språk visar vi hur man för en godtycklig finit automat kan konstruera en kontextfri grammatik som accepterar exakt samma språk som automaten.

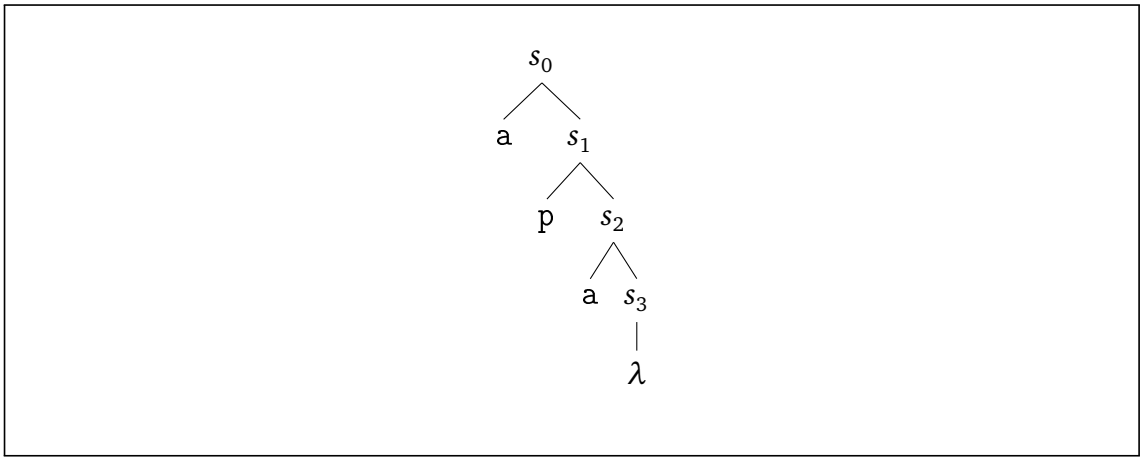
Icketerminalerna i grammatiken är automatens tillstånd. Startsymbolen är automatens starttillstånd. Reglerna i grammatiken är enligt följande: För varje $a$-övergång från ett tillstånd $s$ till ett tillstånd $s'$ i automaten lägger vi till en regel $s \to as'$ till grammatiken. För varje accepterande tillstånd $s$ lägger vi till en regel $s \to \lambda$. Intuitionen är att varje gång att automaten gör en $a$-övergång från $s$ till $s'$ så använder grammatiken regeln $s \to as'$, och vice versa.

**Fråga:** Hur ser grammatiken ut för *apa*-automaten från Figur 1? Kan du ange parseträdet för *apa* enligt denna grammatik?

Grammatiken har 9 icketerminaler, $s_0, \ldots, s_8$. Startsymbolen är $s_0$. Reglerna i grammatiken ser ut så här:

| | | |
|---|---|---|
| $s_0 \to \mathtt{a}s_1$ | $s_1 \to \mathtt{p}s_2$ | $s_2 \to \mathtt{a}s_3$ |
| $s_2 \to \mathtt{o}s_6$ | $s_3 \to \mathtt{s}s_4$ | $s_3 \to \mathtt{n}s_5$ |
| $s_5 \to \mathtt{s}s_4$ | $s_6 \to \mathtt{r}s_7$ | $s_7 \to \mathtt{s}s_4$ |
| $s_7 \to \mathtt{n}s_8$ | $s_8 \to \mathtt{a}s_5$ | $s_3 \to \lambda$ |
| $s_4 \to \lambda$ | $s_5 \to \lambda$ | $s_7 \to \lambda$ |

Parseträdet för *apa* är:




### 8.2.2 Det finns kontextfria språk som inte är reguljära språk

Vi ska nu titta på ett språk som är kontextfritt men *inte* reguljärt. Vi har redan sett detta språk: $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$, det vill säga mängden av alla strängar på formen

$$\underbrace{\mathtt{a}\cdots\mathtt{a}}_{n \text{ ggr}}\underbrace{\mathtt{b}\cdots\mathtt{b}}_{n \text{ ggr}}$$

Låt oss kalla detta språk för $\mathtt{a}^n\mathtt{b}^n$.

Notera att vi hade ett ganska liknande språk i ett tidigare exempel: $\{(\mathtt{ab})^n \mid n \geq 0\}$, dvs. mängden av alla strängar på formen

$$\underbrace{\mathtt{ab}\cdots\mathtt{ab}}_{n \text{ ggr}}$$

För detta språk konstruerade vi en finit automat, men detta är inte möjligt för det nya språket $\mathtt{a}^n\mathtt{b}^n$. Hur kan man bevisa detta?

### 8.2.3 Pumpning

Vi kommer att använda en matematisk princip som kallas **Postfacksprincipen**:

> Om man har fler brev än postfack, och om man lägger varje brev i något av postfacken, kommer något postfack att innehålla minst två brev.

Vi illustrerar denna princip genom ett bevis av följande påstående.

**Påstående:** Det finns minst två Uppsalabor som har precis lika många hår på huvudet.

**Bevis:** Det finns lite mer än 200.000 invånare i Uppsala kommun. Varje människa har mindre än 200.000 hår på huvudet (enligt Wikipedia). Det finns alltså minst ett möjligt antal hår som delas av minst två Uppsalabor.

**Fråga:** Vad är ”breven” och vad är ”postfacken” i detta exempel?

”Breven” är Uppsalaborna, ”postfacken” är möjliga antal hår.

Låt oss tänka på hur en finit automat accepterar en sträng $z$. (I det följande kommer vi att anta att automaten inte innehåller några $\lambda$-övergångar. Om man har en automat med $\lambda$-övergångar måste man först ta bort dem.)

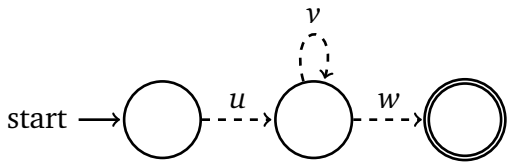
Automaten börjar i starttillståndet. Varje gång den läser en symbol i $z$ går den över till något annat tillstånd. Vi kan skriva ner alla tillstånd som automaten besöker för $z$. Om $z$ består av $n$ symboler så har denna tillståndssekvens längd $n+1$: Den består av starttillståndet och ett ytterligare tillstånd för varje symbol i strängen.

Nu använder vi Postfacksprincipen: Om $z$ består av minst lika många symboler som det finns tillstånd i automaten kommer något tillstånd att besökas minst två gånger när automaten processar $z$.

**Fråga:** Vad innebär detta för automatens struktur?

Om en finit automat accepterar en sträng genom en vandring som besöker samma tillstånd minst två gånger så måste det finnas en cykel i automatens grafdiagram.

Antag att $|z| \geq p$, där $p$ är antalet tillstånd i automaten. Då måste det finnas en cykel i automaten. Vi kan då dela in $z$ i tre delar $u, v, w$ så att $v$ är den del som automaten processar när den vandrar på cykeln och $u$ och $v$ är de delarna som automaten processar innan och efter den hamnar på cykeln. Schematiskt ser detta ut så här:




Notera att, eftersom automaten inte gör $\lambda$-övergångar måste delsträngen $v$ bestå av minst en symbol.

Antag nu att vi skär bort $v$ från $z$ och därigenom bildar en ny sträng $z_0 = uw$. Denna sträng kommer också att accepteras av automaten, eftersom det finns en accepterande vandring i grafen som ser ut precis som den vi hade för $z$, bara att den skippar cykeln som processar symbolerna i $v$.

Antag nu att vi klistra in en kopia på $v$, direkt efter den första förekomsten av $v$, och därigenom bildar en ny sträng $z_2 = uvvw$. Man säger att man har ”pumpat upp” $z$.

Även strängen $z_2$ kommer att accepteras av automaten, genom en vandring som går genom cykeln två gånger istället för bara en gång.

Mera allmänt ser man då att varje sträng $z_i$ på formen $uv^iw$ där $i \geq 0$ kommer att accepteras av automaten.

### 8.2.4 Tillämpning på språket $\mathtt{a}^n\mathtt{b}^n$

Hittills har vi pratat om godtyckliga strängar $z$. Vi har argumenterat att, om $z$ accepteras av automaten och bara är tillräckligt lång, så kan den delas upp i tre delar $u, v, w$ så att $v \neq \lambda$ och varje sträng $z_i = uv^iw$ $(i \geq 0)$ också accepteras av automaten.

Låt oss nu anta att $z$ är av den formen som orden i $\mathtt{a}^n\mathtt{b}^n$ har, och att det finns en automat $M$ som accepterar den. Låt $p$ vara antalet tillstånd i $M$ och låt

$$z = \mathtt{a}^k\mathtt{b}^k \qquad \text{så att } 2k \geq p$$

Som ett konkret exempel kan vi anta att $z = \mathtt{a}^{10}\mathtt{b}^{10}$ och att $p = 20$:

$$z = \overbrace{\mathtt{a}\cdots\mathtt{a}}^{10\text{ ggr}}\overbrace{\mathtt{b}\cdots\mathtt{b}}^{10\text{ ggr}}$$

Eftersom $|z| \geq p$ accepteras den genom en vandring som följer en cykel. Detta innebär att vi kan dela upp $z$ på det sättet som vi beskrivit: $z = uvw$. Låt oss fundera på hur strängen $v$ kan se ut. Det finns tre fall:

1. Strängen $v$ innehåller bara as. Till exempel skulle vi kunna ha:

$$z = \overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{u}}^{2\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{v}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{w}}^{2\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{w}}^{10\text{ ggr}}$$

2. Strängen $v$ innehåller bara bs. Till exempel:

$$z = \overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{u}\underbrace{\mathtt{b}\cdots\mathtt{b}}_{u}}^{10\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{v}}^{2\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{v}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{w}}^{2\text{ ggr}}$$

3. Strängen $v$ innehåller några as följd av några bs. Till exempel:

$$z = \overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{u}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{v}}^{4\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{v}}^{4\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{w}}^{6\text{ ggr}}$$

Vi har redan argumenterat att vi får ”pumpa upp” strängen $z$ och bildar en ny sträng $z_2 = uvvw$ såsom att denna sträng också accepteras av $M$. Vi kan notera det följande:

1. Om $v$ innehåller bara as innehåller den nya strängen $z_2$ fler as än bs. I exemplet har vi:

$$z = \overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{u}}^{2\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{v}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{v}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{w}}^{2\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{w}}^{10\text{ ggr}}$$

   Nu finns det alltså 16 förekomster av a, men bara 10 förekomster av b.

2. Om $v$ innehåller bara bs innehåller den nya strängen $z_2$ fler bs än bs. I exemplet:

$$z = \overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{u}}^{10\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{v}}^{2\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{v}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{v}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{w}}^{2\text{ ggr}}$$

   Nu finns det 10 förekomster av a, men 16 förekomster av b.

3. Om $v$ innehåller några as följd av några bs innehåller den nya strängen $z_2$ fler än en a-region och fler än en b-region:

$$z = \overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}}_{u}}^{6\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}\,\mathtt{b}\cdots\mathtt{b}}_{v}}^{4\text{ ggr}}\overbrace{\underbrace{\mathtt{a}\cdots\mathtt{a}\,\mathtt{b}\cdots\mathtt{b}}_{v}}^{4\text{ ggr}}\overbrace{\underbrace{\mathtt{b}\cdots\mathtt{b}}_{w}}^{6\text{ ggr}}$$

I varje fall kan vi dra slutsatsen att den nya strängen $z_2$ inte är på den formen som strängar i språket $\mathtt{a}^n\mathtt{b}^n$ har. Men vi vet att den nya strängen också accepteras av automaten! Det vill säga: Varje automat som accepterar strängar på den formen som strängarna i språket $\mathtt{a}^n\mathtt{b}^n$ har accepterar också stränger som inte är på den formen. Detta betyder att det kan inte finnas någon automat som accepterar precis de strängarna i språket $\mathtt{a}^n\mathtt{b}^n$. Alltså är detta språk inte reguljärt.

## 8.3 Läsning

Detta material. (Kontextfria språk och pumpargument tas inte upp i boken.)