

formella språk automater och beräkningar

© Lennart Salling 2001

andra upplagan

ISBN 91-630-7707-8

Förord

Abstrakta maskiner och andra algoritmiska koncept

Vad menas med *algoritm*?

Varje läsare torde ha någon sorts insikt om algoritm-begreppet, men att ge en exakt formulering – en *definition* – ställer sig icke helt självklart.

När matematikern och kodbrytaren *Alan M. Turing*, född i London och verksam huvudsakligen i Cambridge, på 30-talet gav sig i kast med Hilberts berömda *Entscheidungsproblem*:

Finns det någon algoritm som kan lösa alla matematiska problem?[†]

så tvingades han resonera om *alla* tänkbara algoritmer, både redan existerande och sådana som ingen ännu har sett. Turing tyckte sig nämligen se ett *nej* som ett rimligt svar på Hilberts *Entscheidungsproblem*, och för att kunna bevisa riktigheten av detta *nej* måste han ställa utom allt tvivel att *ingen enda algoritm* kan ha den omtalade egenskapen. Men hur skulle han kunna föra ett så-

†. För att vara mer exakt, en algoritm som för varje påstående, rörande något matematiskt problem formulerat i 1:a ordningens predikatlogik, kan avgöra (med ja eller nej-svar) om påståendet stämmer. Hilberts *Entscheidungsproblem* har av begripliga skäl kallats för den matematiska logikens principalproblem. Vid tiden för problemets formulering hade Hilbert enbart en *intuitiv* bild av algoritm-begreppet, något som inte hindrade honom och andra matematiker från att tala om algoritmer.

dant resonemang på ett trovärdigt sätt, med hjälp enbart av en intuitiv uppfattning om det algoritmiska? Det var i detta dilemma som Turing föreslog sitt numera berömda sätt att i formella (exakta) termer fånga det algoritmiska.[‡] *Turingmaskinen* var född. (Men fick sitt namn *Turingmaskin* först några år senare – av Gödel.) Trots namnet, som onekligen leder tanken till ett fysiskt objekt, är *Turingmaskinen* inget annat än en *matematisk abstraktion*^{††} – en *abstrakt maskin*.

Turingmaskinen blev den teoretiska mallen för "arkitekturen" hos de första riktiga datorerna. Men också en modell för *procedurella programmeringsspråk* som PASCAL och C samt för maskinnära assembler.

Turing hävdade 1936 att varje mekanisk beräkning kunde utföras av en *Turingmaskin*, och ännu har inget framkommit som motsäger denna tes (*Turings tes*).

I kapitel 4 presenteras *Turingmaskinen* och i kapitel 7 *Turingmaskinens* samröre med Hilberts *Entscheidungsproblem* och andra oavgörbarhetsproblem.

Ungefär tjugo år efter att Turing hade presenterat sin abstrakta maskin, föreslogs *enklare* (mindre kompetenta) abstrakta maskiner som modeller för enkla algoritmiska göromål. *Pushdownautomaten* och den *finita automaten* är de

‡. PROCEEDINGS of THE LONDON MATHEMATICAL SOCIETY, Vol. 42, 1936.

††. På motsvarande sätt som *tallinjen* är en matematisk abstraktion.

viktigaste exemplen, och spelar en stor roll i praktiska tillämpningar. Kapitel 3 och 2 tillägnas dessa enklare algoritmer.

Redan några år innan Turing presenterade sin idé hade ett par andra mycket vackra paradigmer – *rekursiva funktioner* (Gödel, Herbrand 1934) och *λ -kalkyl* (Church, Kleene 1932-33) – föreslagits men ej accepterats fullt ut som tänkbara universella algoritmiska koncept. Tvivlet hade säkert att göra med svårigheter att hitta en trovärdig avbildning från λ -kalkylens och de rekursiva funktionernas höga abstraktionsnivå till mängden av mekaniska sysslor. Men Turings visade redan i sin publikation 1936-37 att λ -kalkylen var likvärdig i uttryckskraft med Turingmaskinen, och snart visades samma sak om de rekursiva funktionerna. Dessa två koncept kom sedermera att stå modell för *funktionella programmeringsspråk* som LISP.

Oberoende av Turing presenterade amerikanen Emil Post 1936 sin idé om det algoritmiska – *system av produktionsregler* – som snart visades vara ekvivalent med Turingmaskinen, och som sedan inspirerade Chomsky att formulera algoritmiska modeller för naturligt språk. Posts algoritmiska koncept kan betraktas som en sorts modell för regelbaserad programmering (som t ex i logikprogrammeringsspråk och i *MATHEMATICA*[†]).

Sammantaget pekar dessa skilda men ekvivalenta paradigmer på existensen av ett generellt matematiskt begrepp – "det algoritmiska" eller "det beräkningsbara".

Denna skrift har kommit till under en femårsperiod (med årliga omskrivningar) i samband med författarens undervisning. Ett särskilt tack går till Anders Andersson för hans bidrag med värdefulla förslag till förbättringar. Uppmuntrande synpunkter från studenter har också haft ett avgörande inflytande på utformningen av vissa avsnitt. (Inte minst avsnittet med lösningsförslag i slutet av boken.)

Lennart Salling
Uppsala
december 1998

Tack vare noggranna korrekturlistor från bl.a. Erik Palmgren och Pontus Andersson innehåller denna nya upplaga färre felaktigheter än den förra. Dessutom har avsnitten om pumpning blivit något putsade. I övrigt är det ingen väsentlig skillnad mellan nämnda upplagor.

Lennart Salling
Uppsala
mars 2001

†. *MATHEMATICA*, WOLFRAM RESEARCH
<http://www.wolfram.com>

Förord 3

Abstrakta maskiner och andra algoritmiska koncept 3

1 Strängar och språk 9

1.1 Program, strängar, heltal och heltalsfunktioner 9

Program = tal 9

Program = heltalsfunktion 10

Syntax och semantik 10

Heltalsfunktion = oändlig sträng 11

1.2 Olika stora oändligheter 11

Hilberts hotell 12

Uppräkneligt och överuppräknligt 14

1.3 Strängar och strängoperationer 15

1.4 Språk och språkoperationer 17

1.5 Övningar 21

2 Reguljära språk och finita automater 22

2.1 Reguljära språk och reguljära uttryck 23

2.2 Deterministiska finita automater 26

2.3 Ickedeterministiska finita automater 34

Acceptans 36

2.4 För varje NFA finns det en ekvivalent DFA 38

Delmängdskonstruktionen 40

2.5 Reguljära språk är de finita automaternas språk 43

Tillståndseliminering och GFA:er 45

2.6 Några slutenhetsegenskaper 48

2.7 Minimering av en DFA 50

Minimering och särskiljande 50

Minimering genom särskiljning av tillstånd 54

Stegvisa särskiljandealgoritmen 56

Minimering genom reversering 57

Dubbla reverseringsalgoritmen 57

2.8 De reguljära språkens gränser 59

Särskiljandesatsen 59

Pumpsatsen för reguljära språk 60

Reguljära språk och periodiska mönster 62

Reguljära språk och minne 63

Finita automater och lexikal analys m.m. 64

2.9 Finita automater med output 64

Mooremaskiner 65

Mealymaskiner 66

KERMIT, en Mealymaskin 67

För varje Mooremaskin finns det en Mealymaskin och omvänt 68

2.10 Övningar 68

3 Sammanhangsfria språk och pushdownautomater 72

- 3.1 Grammatikmetoden 72
 - Strängproduktion 73
 - Produktion visavi konsumtion 73
 - Slingor, Kleenestjärnor och rekursion 73
 - Notation för strängproduktion 74
 - Terminerande och icketerminerande tecken 74
 - Grammatik 75
- 3.2 Sammanhangsfria grammatiker 75
 - Reguljära regler och reguljära grammatiker 75
 - Grammatiker för ickereguljära språk 76
 - Enkla vänsterled 78
 - Högerledens komplexitet 78
 - Produktionsträd 80
- 3.3 Pushdownautomater (PDA:er) 82
 - Push och pop 83
 - Acceptera med tom stack 83
 - Att driva 85
 - Att acceptera 86
- 3.4 För varje CFG finns det en PDA 87
 - Top-down parser 87
 - Bottom-up parser 88
- 3.5 För varje PDA finns det en CFG 90
- 3.6 De sammanhangsfria språkens gränser 93
 - Reguljära pumpsatsen i termer av produktion 93
 - Pumpsatsen för sammanhangsfria språk 94
- 3.7 Deterministiska pushdownautomater 98
 - Det finns *inte* en DPDA för varje CFG 101
- 3.8 Övningar 102

4 Restriktionsfria språk och Turingmaskiner 106

- 4.1 Restriktionsfria språk 106
- 4.2 Turingmaskiner 109
 - Tape 117
 - Starttillstånd och stopptillstånd 117
 - Atomära agerandet 118
 - Blanktecken 118
 - Konfigurationer 118
 - Grafisk beskrivning 118
 - Driva 120
 - Acceptera 121
 - Tre små ... 121
 - Seriekoppling 121
 - Shiftning 121
 - Höger- och vänstergående teckenletare 122

- Kopiering 123
- 4.3 Att acceptera respektive avgöra 123
- 4.4 Att beräkna funktioner med Turingmaskiner 125
- 4.5 Turingmaskiner är ekvivalenta med restriktionsfria grammatiker 129
- 4.6 En universell Turingmaskin 133
- 4.7 Ekvivalenta varianter av Turingmaskiner 135
 - Flera tapar 135
 - Ickedeterminism 138
 - En två-stacks-PDA har samma förmåga som en TM 139
- 4.8 Övningar 140

5 Rekursiva funktioner 142

- 5.1 Primitivt rekursiva funktioner 143
 - Den primitivt rekursiva mallen 145
 - Klassen av primitivt rekursiva funktioner 147
 - Primitivt rekursiva predikat 148
 - Funktionen Om 149
 - Delbarhet och primitivt rekursiva funktioner 151
- 5.2 Primitiv rekursion är otillräcklig 153
 - En primitivt rekursiv konstruktion av $Fib(x)$ 156
 - Rekursiva flätor 157
 - Primtal och framåtrekursion 159
- 5.3 Rekursiva funktioner – en utvidgning av primitivt rekursiva funktioner 160
 - Framåtrekursion 160
 - Klassen av rekursiva funktioner 161
- 5.4 Turings tes och Church:s tes 163
- 5.5 För varje TM finns det en rekursiv funktion 164
 - Från godtyckliga teckensträngar till tal 165
 - Några Turingmaskinoperationer tolkade som primitivt rekursiva aritmetiska beräkningar 165
 - Konfigurationerna som taltripplar 167
 - Konfigurationernas omvandlingar tolkade som primitivt rekursiva beräkningar 168
 - Den slutgiltiga funktionen 169
- 5.6 För varje rekursiv funktion finns det en TM 170
- 5.7 Övningar 171

6 Loop-program 173

- 6.1 Primitiva loop-program 174
- 6.2 För varje primitivt rekursiv funktion finns det ett primitivt loop-program 175
- 6.3 För varje primitivt loop-program finns det en primitivt rekursiv funktion 176
- 6.4 Program med sålänge-loopar 178

7 Stopp-problemet och oavgörbarhet 179

- 7.1. Stopp-problemet 180
 - Två typer av TM-igenkännbara språk 182

- Ett exempel på ett icke-algoritmiskt språk 183
- 7.2. Rices sats 184
 - Konsekvenser av Rices sats 185
 - Problemet om den flitiga bävern 187
- 7.3. Övningar 188

8 Logik 190

- 8.1 Satslogik 191
 - De satslogiska formerna som binära träd 192
 - Satslogikens semantik 192
 - Semantisk analys med sanningstabell 193
 - Några nya konnektiv skapade med hjälp av gamla 193
 - Sexton olika binära satslogiska formler 194
 - Tolkning, validitet och satisfierbarhet 195
 - Ekvivalens 195
 - Normalformerna 198
- 8.2 Validitetsproblemet och resolution 199
 - Quine:s algoritm, en bottom-up-algoritm 199
 - Två topdown-algoritmer 200
 - Semantisk tablå. (Beth, Hintikka, Smullyan) 200
 - Klausul-resolution (Davis, Putnam) 202
 - Klausul och klausulmängd 202
- 8.3 Första ordningens predikatlogik 205
 - Syntax 206
 - Symboler 206
 - Fria och bundna variabler, öppna och slutna formler 207
- 8.4 Predikatlogikens semantik 207
 - Allkvantorn 209
 - Några ekvivalensregler 210
 - Validitet och satisfierbarhet i predikatlogik 211
- 8.5 Validitetsproblemet och resolution i predikatlogik 212
- 8.6 Hornklausuler och Hornformler 215
- 8.7 Rekursiva funktioner och predikatlogik 216
 - Till varje rekursiv funktion hör en Hornformel 217
 - Framåtrekursion 219
- 8.8 Övningar 219

Lösningsförslag 221

Litteraturförteckning 260

Index 261

Beteckningar 265

Elementära mängdbegrepp 266

Strängar och språk

Program, strängar, heltal och heltalsfunktioner.....	9
Olika stora oändligheter.....	11
Uppräkneligt och överuppräknligt	14
Strängar och strängoperationer	15
Språk och språkoperationer.....	17

Att utforska “något” innebär ofta att utforska en förenklad modell av “något”. Inte säkert för att “något” är så komplicerat att det inte låter sig utforskas just som det är. Nej, ett legitimt skäl till en förenklad syn kan t ex vara att man *bara vill* utforska en *viss aspekt* av “något”, en aspekt som enklast låter sig studeras genom att man bortser från andra (i sammanhanget) ovidkommande aspekter.

■ 1.1 Program, strängar, heltal och heltalsfunktioner

■ **Program = tal** Ett program är – oavsett programmeringsspråk – en ändlig *teckensträng*, dvs en ändlig sekvens av tecken tagna ur något alfabet. Ja, ett program kan t.o.m. uppfattas som en ändlig *bitsträng* (en teckensträng vars tecken är nollor och/eller ettor). I viss mening är detta synsätt faktiskt det mest relevanta. Ty i datorns inre representeras varje program av en bitsträng.

Representerar *varje* bitsträng på detta sätt ett program?

Får ett program se ut hur som helst?

Nja, ... i varje programmeringsspråk ställs vissa krav på vilka teckenkombinationer som är tillåtna, och vad som måste finnas i ett program ..., men eftersom vi *inte* avser att diskutera något särskilt språk, väljer vi att ge programstatus åt *varje* bitsträng. Därigenom riskerar vi inte att lämna något enda program utanför vår diskussion.

En bitsträng representerar å andra sidan ett *naturligt tal*[†] skrivet i det binära positionssystemet. T ex representeras talet fem av strängen 101, men även av 0101 eftersom inledande 0:or inte har någon signifikans för talet ifråga. Man kan undvika sådan flertydighet genom att istället använda följande representation: Ordna bitsträngarna efter i första hand längd, och vid lika längd efter alfabetisk ordning (där tecknet 0 kommer före tecknet 1), och låt sedan en bitsträng representera det naturliga tal som överensstämmer med bitsträngens ord-

†. Med mängden \mathbb{N} av naturliga tal menar vi $\{0, 1, 2, \dots\}$.

ningsnummer. Se FIGUR 1.1. Därvid kommer varje naturligt tal att

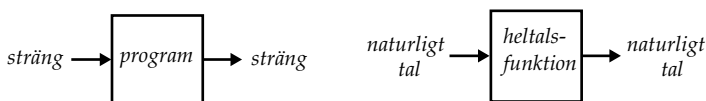
Bitsträng (= program)		Naturligt tal
0	↔	0
1	↔	1
00	↔	2
01	↔	3
10	↔	4
11	↔	5
000	↔	6
001	↔	7
...

FIGUR 1.1 Det råder 1–1 motsvarighet mellan bitsträngarna (=programmen) och de naturliga talen.

representeras av *precis en* bitsträng. Och omvänt, till varje bitsträng hör *precis ett* tal. Man brukar uttrycka detta som att det råder 1–1 motsvarighet mellan mängden av bitsträngar och mängden av naturliga tal.

Program betraktade som teckensträngar “är” således inget annat än naturliga tal.

- **Program = heltalsfunktion** Om vi komplicerar vårt perspektiv genom att ta i beaktande att ett program kan ha *input* och *output*, och samtidigt konstaterar att input och output kommer i form av teckensträngar vilka (som tidigare) kan väljas som bitsträngar, dvs som representeranter för naturliga tal, så ändras vårt synsätt till att uppfatta varje program som en *heltalsfunktion med input och output* från de naturliga talen.



FIGUR 1.2 Ett program kan uppfattas som en heltalsfunktion.

■ Syntax och semantik

Sammanfattningsvis har vi nu *två* skilda synsätt på *program*, där det första (1) anlägger enbart ett *syntaktiskt*[†] perspektiv, medan det andra (2) beskriver “vad ett program gör”, och därmed kan sägas ha ett *semantiskt*[‡] perspektiv.

Varje *program* är (representeras av) ett *heltal* i \mathbb{N} . (1)

Varje *program* är (beräknar) en *heltalsfunktion* från \mathbb{N} till \mathbb{N} . (2)

Tycker Du att det verkar konstigt att ett program både kan ses som ett

†. Syntax=ordfogningslära

‡. Semantik=betydelselära

heltal och som en heltalsfunktion? Tänk då på att vi i de två synsätten betraktar olika sidor av ett program. När vi ser ett program som ett heltal är det programmets syntaktiska form vi betraktar. Och när vi talar om ett program som en funktion är det "vad programmet uträttar" som vi intresserar oss för.

En intressant fråga är huruvida (2):s omvända synsätt är giltigt:

Kan varje funktion från \mathbb{N} till \mathbb{N} beräknas av ett program?

I nuvarande skede skulle det inte förvåna mig om Du hade ett felaktigt svar på frågan. (På sid 15 finns det rätta svaret. Och det skulle inte förvåna mig om Du just nu bläddrade fram till just denna sida.)

För att förstå varför svaret ser ut som det gör, skall vi ägna några sidor åt oändligheten. Först uppmärksamma läsaren på hur man kan representera en heltalsfunktion (från \mathbb{N} till \mathbb{N}) med en oändlig sträng.

■ Heltalsfunktion = oändlig sträng

En *totalt*[†] definierad funktion från \mathbb{N} till \mathbb{N} kan representeras av en *oändlig* sträng av naturliga tal. Ty en sådan funktion f är bestämd av sina output

$$f(0), f(1), f(2), \dots$$

vilka – placerade i följd – bildar en oändlig sträng av naturliga tal. I det fall att funktionen är ett predikat[‡] är outputsträngen en *oändlig sträng* av bitar (0:or eller 1:or).

Och en *partiellt* definierad funktion från \mathbb{N} till \mathbb{N} kan representeras av en oändlig sträng av naturliga tal eller "saknar-output"-tecken (som i den sista outputsträngen nedan).

EXEMPEL 1.1 Några heltalsfunktioners oändliga outputsträngar:

<i>input</i>	0	1	2	3	4	5	6	7	8	...
<i>output</i>	0	1	0	1	0	1	0	1	0	...
<i>output</i>	0	1	4	9	16	25	36	49	64	...
<i>output</i>	0	0	1	1	0	1	0	1	0	...
<i>output</i>	– [†]	5	2	1	1	1	0	0	0	...

†. "–" markerar att funktionen ifråga är odefinierad här.

■ 1.2 Olika stora oändligheter

Det sägs att David Hilbert, när han föreläste om Georg Cantors teori rörande olika graderingar av oändligheten, använde sig av en metafor i form av ett hotell. Följande berättelse är att betrakta som en variation på detta tema.

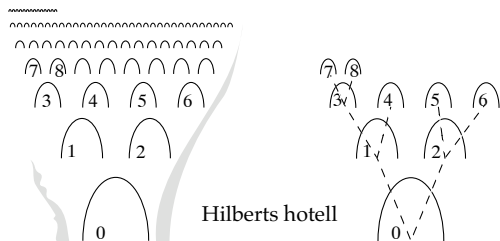
†. definierad på hela \mathbb{N} .

‡. En funktion vars output är 0 eller 1.

Hilberts hotell

Hilberts hotell har en outtömlig mängd av rum i följande mening. Till varje naturligt tal n finns det ett rum med rumsnummer n . Det finns således rum med nummer 0, 1, 2, 3, ...

Vidare har hotellet formen av ett binärt träd, med rum 0 i rotnoden och de övriga rummen i de övriga noderna numrerade som i figuren nedan.



En frågeställning som vi skall diskutera är följande. Kan detta hotell någonsin bli fullbelagt? Men innan vi försöker formulera olika tänkbara svar på frågan, måste vi göra en sak fullständigt klar: Vi har här att göra med ett hotell som inte finns annat än i tankevärlden – i vår fantasi. Likadant är det med hotellets gäster. De kan "fritt" fantiseras fram – så länge fantasierna icke resulterar i logiska motsägelser.

De första gästerna anländer. De är schackspelare – oändligt många – och skall delta i en schackturnering. Spelarna är därför numrerade efter skicklighet. Den skickligaste spelaren har nummer 1, den näst skickligaste nummer 2, osv ...

Hotellportieren Hilbert hälsar dem välkomna och säger.

"Ni skall få de bästa rummen, eftersom ni är de första gästerna. Spelare med skicklighetsgrad n , var god intag rum n ." Rum 0 – tänkte han – kan alltid vara bra att ha kvar, ifall ...

Just då kommer det ytterligare två gäster till det (till synes) fullbelagda hotellet. Hilbert som var van att lösa problem till allas belåtenhet funderade en stund. Sedan visste han ...

Via hotellets kommunikationssystem lät han sända iväg följande meddelande till hotellets samtliga rum.

"Gäst i rum n , var god flytta till rum $n + 2$."

På detta sätt blev rum 1 och rum 2 lediga. Och de två nya gästerna fick varsitt rum.

Innan Hilbert hunnit slå sig till ro, dyker det upp ytterligare en samling schackspelare. Oändligt många. Och numrerade efter skicklighet på samma sätt som de förra.

På väg ut för att möta de nya gästerna vänder han tillbaka in i hotellet igen, för att till hotellets alla rum låta sända ut ännu ett meddelande.

"Gäst i rum n , var god flytta till rum $2n$."

Därigenom blev alla rum med udda rumsnummer lediga. Och Hilbert kunde lugnt gå ut och hälsa de nya gästerna välkomna.

“Hjärtligt välkomna, jag förstår att Ni önskar varsitt rum. Det skall Ni få. Inte nog med det. Ni skall få de bästa rummen vi har. De med udda rumsnummer. Spelare med skicklighetsgrad n , var god intag rum $2n - 1$.”

Innan Hilbert hunnit sätta sig ned – nöjd över att ha ordnat rum åt alla gäster och ändå ha ett rum över – inträffar något mycket märkligt. Dagsljuset som nyss silade in genom hotellets vackert utsirade franska entrédörrar blev allt svagare. Utanför hotellet tornade något upp sig, som satte Hilberts tankar i brand. Han kände hur det hettade innanför pannloben. I sina vildaste fantasier hade han en gång drömt om att hotellet invaderades av så *förfärligt oändligt* många gäster att hotellet inte kunde ta hand dem alla. Att rummen i detta hotell icke skulle räcka trots att de var oändligt många. Med ett resonemang vars bärande idé är en diagonal hade han kommit fram till detta. Och i sin önskan att inte avvisa någon från hotellet hade tankarna kommit in på att utnyttja takterrassen.

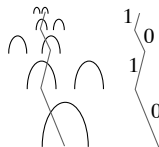
Just då viks de franska entrédörrarna undan och in i vestibulen väljer en enorm massa. En individ ur massan lyckas tränga sig fram till Hilbert och säger:

“Vi är många, men man har sagt oss att detta hotell aldrig är fullbelagt fast det synes vara det. Kan Ni ordna rum åt var och en av oss?”

När den trugande gästen böjer huvudet i en blidkande gest ser Hilbert hur en sträng av nollor och ettor dinglar fram och tillbaka runt dennes hals. En snabb blick övertygar Hilbert om att varje individ i massan har en sådan sträng runt halsen. Han granskar den trugandes sträng igen så noga han kan. Och så vitt han kan bedöma så finns det *inte* något slut på strängen.

01011000100001000001000000100000001000000001...

Då ser han med ens allting klart för sig. Varje sådan sträng kan ses som en vägbeskrivning. En karta över en oändlig väg nerifrån vestibulen och uppåt i hotellets korridorer Till takterrassen!



Den n :te siffrans värde anger vägval (0=vä och 1=hö) i hotellets binära trädstruktur när man befinner sig på hotellets n :te våningsplan. Med denna idé i bakhuvudet griper Hilbert nu tag i kommunikationssystemets mikrofon och säger med lugn och säker stämma:

“Hjärtligt välkomna, jag förstår att Ni önskar varsitt rum. Det skall Ni få. Inte nog med det. Ni skall få de bästa rummen. Ovanför detta hotell finns nämligen takterrassens rum varifrån man har en fantastisk utsikt. Dessa rum skall Ni få. Med vår accelererande hiss som för varje våning fördubblar hastigheten passerar Ni först detta hotells första våning på 1 sekund, nästa våning på $\frac{1}{2}$ sekund osv..., varför Ni efter

totalt 2 sekunder[†] har nått Edra rum. För att göra Er vistelse hos oss så personlig och säker som bara vi kan, har vi dessutom ordnat så att vår hisskonduktör Cantor tar Dig till just Ditt rum med hjälp av den sträng som Du bär runt Din hals. Ingen annan sträng leder till Ditt rum. Var så goda, tag plats i hissen.”

■ Uppräkneligt och överuppräknligt

DEFINITION En mängd kallas uppräknlig om den kan numreras på något sätt med naturliga tal så att varje element tilldelas ett unikt tal. Annars kallas mängden överuppräknlig.

✧ SATS 1.1 De ändliga bitsträngarna bildar en uppräknlig mängd.

BEVIS: Se FIGUR 1.1 på sidan 10. □

✧ SATS 1.2 De oändliga bitsträngarna bildar en överuppräknlig mängd.

BEVIS: Antag att w_0, w_1, w_2, \dots är någon uppräkning av oändliga bitsträngar. Den n :te av dessa betecknar vi med $w_n = b_{n0}b_{n1}b_{n2}\dots$.

$$\begin{array}{rcll} w_0 & = & b_{00} & b_{01} & b_{02} & \dots \\ w_1 & = & b_{10} & b_{11} & b_{12} & \dots \\ w_2 & = & b_{20} & b_{21} & b_{22} & \dots \\ & & \vdots & & & \end{array}$$

Betrakta nu $w = b_{00}b_{11}b_{22}\dots$ (se de markerade bitarna ovaför) och

$$\bar{w} = \bar{b}_{00}\bar{b}_{11}\bar{b}_{22}\dots$$

För varje n skiljer sig \bar{w} i den n :te biten från w_n .

där \bar{b}_{nn} avser ett inverterat b_{nn} , (om $w = 110\dots$ så är $\bar{w} = 001\dots$). \bar{w} är en oändlig bitsträng som faktiskt *inte* (se marginalkommentaren) finns med i uppräknningen w_0, w_1, w_2, \dots . Alltså, för varje numrering w_0, w_1, w_2, \dots av oändliga bitsträngar finns det någon oändlig bitsträng som inte kommer med i uppräknningen! □

Mängden av heltalsfunktioner är mäktigare än mängden av program.

Genom att beakta det faktum att varje oändlig bitsträng representerar en funktion från \mathbb{N} till \mathbb{N} (en funktion vars n :te output är bitsträngens n :te bit) följer av SATS 1.2 att *mängden av funktioner från \mathbb{N} till \mathbb{N} är överuppräknlig*. Och därmed att det finns heltalsfunktioner som *inte* kan beräknas med program. Ty om heltalsfunktionerna kunde beräknas med program, skulle de kunna numreras med (de naturliga tal

†. $1 + 1/2 + 1/4 + \dots = 2$

som numrerar) dessa program. Därmed har vi besvarat den fråga som vi lämnade öppen tidigare:

✧ SATS 1.3 *Det finns heltalsfunktioner från \mathbb{N} till \mathbb{N} som inte kan beräknas med program.*

⌘ Anm.1.1 Utan att utforska mängden av program särskilt djupt har vi lyckats visa på existensen av heltalsfunktioner som inte går att beräkna med program. Längre fram (i kapitel 7) skall vi ge *exempel* på sådana funktioner. En sak kan jag dock avslöja redan nu. Om alla heltalsfunktioner kunde beräknas med program, skulle det nog inte finnas så många felaktiga program på marknaden.

1.3 Strängar och strängoperationer

Alfabet Ett alfabet Σ är en ändlig mängd av ett eller flera tecken[†].

EXEMPEL 1.2 $\Sigma = \{0, 1\}$ är det binära alfabetet, $\Sigma = \{a, b, c, \dots, ä, ö\}$ är det "vanliga" svenska alfabetet.

Notera att vi här definierar en sträng som ändlig.

Sträng En sträng w över ett alfabet är en ändlig sekvens av noll eller flera tecken från alfabetet. Strängen med noll tecken kallas *tomma strängen* och betecknas ε . Antalet tecken i w betecknas $|w|$ och benämnes längden av w .

EXEMPEL 1.3 10010 är en sträng över $\{0, 1\}$ av längd $|10010| = 5$, sträng är en sträng över $\{a, b, \dots, ä, ö\}$ och $|sträng| = 6$. Strängar kommer vi oftast att beteckna med bokstäverna u, v, w, x, y, z . Och enskilda tecken oftast med a, b, c, d, e eller σ .

Strängoperationer Vi skall här ta upp tre strängoperationer: *sammanfoga*, *repetera*, *reversera* varav de första två nog får betraktas som fundamentala.

Sammanfoga ▶ Att *sammanfoga* två strängar v och w är att "slå ihop" dem till en sträng. Den sålunda sammanfogade strängen betecknas vw .

EXEMPEL 1.4 $v = ab, w = ba, vw = abba$.

⌘ Anm.1.2 Lägg märke till att tomma strängen har motsvarande egenskap vid strängsammanfogning som talet 1 har vid multiplikation

†. Ibland säger vi *symbol* istället för *tecken*.

och som talet 0 har vid addition.

$$w\varepsilon = \varepsilon w = w \quad x \cdot 1 = 1 \cdot x = x \quad x + 0 = 0 + x = x$$

Notera också *associativa* lagen vid sammanfogning:

$$(uv)w = u(vw)$$

samt att *prefix* och *suffix* kan definieras i termer av sammanfogningar:

x är *prefix* till xy

y är *suffix* till xy .

EXEMPEL 1.5 Prefixen till strängen *kalle* är $\varepsilon, k, ka, kal, kall, kalle$. Av dessa prefix är samtliga utom den första och den sista *äkta*[†]. *Suffixen* till *kalle* är $\varepsilon, e, le, lle, alle, kalle$.

Repetera ▶ Att *repetera* en sträng w är att foga noll eller flera w till den tomma strängen. Den repeterade strängen betecknas w^n , där n är antalet repetitioner. (w^n utläses *inte* " w upphöjt till n " utan " w repeterad n gånger".)

EXEMPEL 1.6 $w = ha$, $w^0 = \varepsilon$, $w^3 = hahaha$.

Reversera ▶ Att *reversera* en sträng w är att skriva w i omvänd ordning. Resultatet betecknas w^{rev} .

EXEMPEL 1.7 $w = abc$, $w^{rev} = cba$.

⌘ Anm.1.3 Operationerna *sammanfoga*, *repetera*, *reversera* kan definieras med *rekursion*[‡]:

	<i>sammanfoga</i>	<i>repetera</i>	<i>reversera</i>
<i>basfall</i>	$w\varepsilon = w$	$w^0 = \varepsilon$	$\varepsilon^{rev} = \varepsilon$
<i>rekursionssteg</i>	$w(va) = (wv)a$	$w^{n+1} = w^n w$	$(aw)^{rev} = w^{rev}a$

EXEMPEL 1.8 Vi bevisar med induktion över w 's längd att

$$(wv)^{rev} = v^{rev}w^{rev} \quad (3)$$

BEVIS: Induktionsbeviset innehåller två delar: Den ena är *basfallet* där vi visar att formeln stämmer för w av minsta längd (längd noll). Den andra är det s k *induktionssteget* där vi visar att formeln stämmer för

†. Ett prefix u till en sträng w är *äkta* om $0 < |u| < |w|$.

‡. Man säger också *induktion*. Den *rekursiva* (eller *induktiva*) principen är att med ett basfall definiera *något* i det enklaste fallet, och att sedan definiera övriga fall av *något* i termer av enklare versioner av *något*. Det *konstruktiva* i en *rekursiv* definition gör att en sådan definition kan betraktas som ett program.

varje annan sträng w (dvs längre än av kortaste längd) om formeln stämmer för strängar som är kortare än w .

Basfall: (3) stämmer om $w = \epsilon$, ty då reduceras (3) till $v^{rev} = v^{rev}$

Induktionssteg: (3) stämmer för aw om (3) stämmer för w , ty

$$\begin{aligned} ((aw)v)^{rev} &= (a(wv))^{rev} && \text{associativa lagen} \\ &= (wv)^{rev}a && \text{rekursiva def av reversera} \\ &= (v^{rev}w^{rev})a && \text{om (3) stämmer för } w \\ &= v^{rev}(w^{rev}a) && \text{associativa lagen} \\ &= v^{rev}(aw)^{rev} && \text{rekursiva def av reversera} \end{aligned}$$

✂ TEST 1.1

a) Ge en *rekursiv* definition av stränglängd.

b) Visa medelst induktionsbevis *associativa* lagen $(uv)w = u(vw)$.

LEDNING: Den rekursiva definitionen av strängsammanfogning.

Här nedan följer några formella beskrivningar av strängar över något alfabet Σ . Försök hitta enklare mer direkta beskrivningar.

c) $\{w \mid w = ww\}$, d) $\{w \mid w = w^{rev}\}$,

e) $\{w \mid w = uu^{rev} \text{ för någon sträng } u \text{ över } \Sigma\}$,

f) $\{w \mid ww = uuu \text{ för någon sträng } u \text{ över } \Sigma\}$.

1.4 Språk och språkoperationer

Vi skall ta ett mycket generellt grepp här. Och säga att ett *språk* är en mängd (vilken som helst) av strängar.

Kleenestjärna

Det *största* språket över ett alfabet Σ är därvid mängden av *alla* strängar (inklusive ϵ) över Σ . Detta språk betecknas Σ^* och kallas *Kleenestjärnatillslutningen*[†] av Σ . (Se mer om *Kleenestjärnatillslutningen* på sid 20.) Σ^* är alltså det största språket över Σ . Inget språk över Σ innehåller någon sträng som inte redan finns i Σ^* . Dvs varje annat språk över Σ är en delmängd av Σ^* .

EXEMPEL 1.9 Om $\Sigma = \{0, 1\}$ så är $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \dots\} = \{\text{alla ändliga bitsträngar}\}$.

EXEMPEL 1.10 Låt Σ bestå av alla bokstäver $\{a, b, c, \dots, \ddot{a}, \ddot{o}\}$, siffror, punkt, komma och andra skiljetecken samt blanktecken. Då innehåller Σ^* alla korrekta språkliga satser i det svenska såväl som det engelska språket – men även "nonsenssträngar" (som t ex *jschl,,ai 9sjaicb*).

†. Efter Stephen C. Kleene

EXEMPEL 1.11 Några extrema språk över ett alfabet Σ :

<i>språk</i>	<i>som innehåller</i>
\emptyset	noll strängar
$\{\epsilon\}$	en sträng (av längd noll), varför $\{\epsilon\} \neq \emptyset$
Σ	alla strängar av <i>längd ett</i> (över Σ)
Σ^*	alla strängar (över Σ)

⌘ Anm.1.4 Att vi ägnar uppmärksamhet åt det innehållslösa språket \emptyset samt språket $\{\epsilon\}$ med den innehållslösa strängen, har att göra med att dessa två språk ofta bildar en naturlig botten i klasser av intressanta språk.

⌘ Anm.1.5 Σ^* är uppräknelig, ty Σ^* :s strängar kan numreras exempelvis efter följande princip (seäven i FIGUR 1.1 på sidan 10 där dock ϵ fattas):

- först ϵ ,
- sedan alla strängar av längd 1 (tagna i alfabetisk ordning),
- sedan alla strängar av längd 2 (— „ —),
- osv ...

Hur många språk finns det? Det finns uppräkneligt många *strängar* över ett givet alfabet (se Anm.1.5), men hur många *språk* finns det?

Många! ✧ SATS 1.4 *Mängden av språk över ett alfabet är en överuppräknelig mängd.*

BEVIS: För att specificera ett språk över ett alfabet Σ behöver man ange vilka av Σ^* :s strängar som ingår i språket. Dvs för var och en av de uppräkneligt många strängarna i Σ^* skall man med *ja* (=1) eller *nej* (=0) besvara frågan om strängen tillhör språket. Detta ger oss en *oändligt uppräknelig bitsträng* som specifikation av språket. Varje oändlig bitsträng representerar på det här sättet ett språk över Σ . Och dessa bitsträngar bildar en överuppräknelig mängd (SATS 1.2). \square

■ **Språkoperationer** Eftersom språk är mängder så kan man operera på dem med alla vanliga mängdoperationer: *snitt*, *union*, *differens* och *komplement* (m.a.p. Σ^*).

De gängse mängdoperationerna. Se även sid 266.

$$L_1 \cap L_2, \quad L_1 \cup L_2, \quad L_1 - L_2, \quad \overline{L_1} = \Sigma^* - L_1 \quad (4)$$

EXEMPEL 1.12 $\Sigma = \{a, b\}$ kan ses som en union: $\Sigma = \{a\} \cup \{b\}$. Och allmännare, $L = \{w_1, w_2, \dots, w_n\} = \{w_1\} \cup \{w_2\} \cup \dots \cup \{w_n\}$.

EXEMPEL 1.13 Låt L_1 vara språket av palindromer över $\Sigma = \{a, b\}$, och L_2 språket vars strängar (över samma alfabet) har udda längd. Då består $L_1 \cap L_2$ av palindromsträngarna över $\Sigma = \{a, b\}$ som är av udda längd.

Och $\overline{L_1 \cap L_2}$ utgörs av de strängar som *inte* är palindromsträngar av udda längd (dvs som har jämn längd eller som inte är palindromer). \square

Förutom de gängse mängdoperationerna (se(4)) finns det språkoperationer som har att göra med att språk är mängder av just *strängar*. Dit hör bl.a. *sammanfogning*, *Kleenestjärnatillslutning*, *Prefix-* och *Suffix-operationerna* vilka vi nu skall presentera:

Sammanfoga

► Med *sammanfogningen* LM av två språk L, M menas alla sammanfogningar av två strängar varav den första tas från L och den andra från M :

$$LM = \{xy \mid x \in L, y \in M\}^+$$

EXEMPEL 1.14 Om $L = \{in, ut\}$ och $M = \{komst, tag\}$, så är $LM = \{inkomst, intag, utkomst, uttag\}$ och $LL = \{inin, inut, utin, utut\}$.

Inget hindrar oss från att upprepade gånger bilda sammanfogningar med ett och samma språk L :

$$LL = L^2, \quad L^2L = L^3, \quad L^3L = L^4, \quad \text{osv} \dots$$

$$L^0 = \{\epsilon\}$$

Mer formellt, genom att låta $L^0 = \{\epsilon\}$ vara basfall i en rekursiv definition kan vi definiera L^n för godtyckligt $n \in \mathbb{N}$:

Repetera

► Med det n -faldigt repeterade språket L^n avses

$$L^0 = \{\epsilon\}$$

$$L^{n+1} = L^n L$$

Till sist skall vi utvidga *Kleenestjärnans* operationsområde från ett språk som är ett alfabet (se sid 17) till ett godtyckligt språk.

†. L och M kan vara språk över olika alfabeten.

Kleenestjärna-
tillsluta

► Med *Kleenestjärnatillslutningen* L^* av ett språk L avses

L^*

$$L^* = \{x \mid x \in L^n \text{ för något } n \in \mathbb{N}\}$$

$$\text{dvs } L^* = L^0 \cup L^1 \cup L^2 \cup \dots = \{\varepsilon\} \cup L^1 \cup L^2 \cup \dots$$

Låt mig slutligen passa på att nämna notationen L^+ :

L^+

$$L^+ = L^1 \cup L^2 \cup \dots$$

EXEMPEL 1.15 Låt L vara som i EXEMPEL 1.14. Då är

$$L^* = \{\varepsilon, in, ut, inin, inut, utin, utut, ininin, \dots\},$$

$$L^+ = \{in, ut, inin, inut, utin, utut, ininin, \dots\}.$$

EXEMPEL 1.16 Eftersom varje sträng i $\{1, 10, 100, 1000, \dots\}$ är sammanfogad av två strängar: 1 och x där $x \in \{\varepsilon, 0, 00, 000, \dots\}$, så kan $\{1, 10, 100, 1000, \dots\}$ betraktas som sammanfogad av två språk:

$$\{1, 10, 100, 1000, \dots\} = \{1\}\{\varepsilon, 0, 00, 000, \dots\}$$

Det är brukligt
att låta *Kleen-
estjärna* binda
starkare än *sam-
manfogning*, an-
nars skulle (5)
skrivas
 $\{1\}(\{0\}^*)$.

Vidare är ju $\{\varepsilon, 0, 00, 000, \dots\} = \{0\}^*$, varför

$$\{1, 10, 100, 1000, \dots\} = \{1\}\{0\}^* \quad (5)$$

Således kan $\{1, 10, 100, 1000, \dots\}$ ses som ett språk skapat från de "atomära" språken $\{0\}$ och $\{1\}$ med hjälp av språkoperationerna *sammanfogning* och *Kleenestjärna*.

EXEMPEL 1.17 Språket L av strängar över $\{a, b\}$ som innehåller någon förekomst av *abba* kan på motsvarande sätt som i föregående exempel opereras fram från "atomära" språk $\{a\}$ och $\{b\}$ med hjälp av *sammanfogning*, *union* och *Kleenestjärna*:

$$L = (\{a\} \cup \{b\})^* \{a\} \{b\} \{b\} \{a\} (\{a\} \cup \{b\})^*$$

Prefixspråket

► Med *prefixspråket* till ett språk L menas mängden av alla *prefix* till L :s alla strängar:

$$Prefix(L) = \{y \in \Sigma^* \mid y \text{ är prefix till något } w \in L\}$$

Suffixspråket

► På motsvarande sätt definieras *suffixspråket*:

$$Suffix(L) = \{y \in \Sigma^* \mid y \text{ är suffix till något } w \in L\}$$

EXEMPEL 1.18 Följande tre likheter visar, att icke-tomma språk byggda med operationerna *sammanslagning*, *union* och *Kleenestjärna* har suffixspråk som är byggda med samma operationer:

För icke-tomma språk L_1, L_2, L gäller

$$\text{Suffix}(L_1 L_2) = (\text{Suffix}(L_1) L_2) \cup \text{Suffix}(L_2) \quad (6)$$

$$\text{Suffix}(L_1 \cup L_2) = \text{Suffix}(L_1) \cup \text{Suffix}(L_2) \quad (7)$$

$$\text{Suffix}(L^*) = \text{Suffix}(L) L^* \quad (8)$$

(6) följer av att om $a_1 \dots a_m \in L_1$, $b_1 \dots b_n \in L_2$ och y är ett suffix till $a_1 \dots a_m b_1 \dots b_n$, så är $y = a_p \dots a_m b_1 \dots b_n$ eller $y = b_p \dots b_n$.

⌘ Anm.1.6 Om $L_1 = \emptyset$ samtidigt som $L_2 \neq \emptyset$ är (6) falsk. Ty då är (6):s $VL = \emptyset$, medan (6):s $HL \neq \emptyset$ eftersom $HL = \text{Suffix}(L_2)$. Och om $L = \emptyset$ är (8) falsk, eftersom då (8):s $VL = \{\epsilon\}$ medan (8):s $HL = \emptyset$. (7) är däremot sann även om en eller båda av L_1, L_2 är tom.

1.5 Övningar

1.1 En *palindrom* är en sträng w sådan att $w^{rev} = w$. Palindromerna över ett alfabet bildar ett språk, *palindromspråket* över alfabetet ifråga. Teckna ned de tio kortaste palindromerna över $\{a, b\}$. Ge sedan en *rekursiv definition* av palindromspråket över $\{a, b\}$.

1.2 Visa att a) $\emptyset^n = \emptyset$ om $n > 0$ b) $\emptyset^* = \{\epsilon\}$

1.3 Beskriv $\text{Prefix}(L)$ och $\text{Suffix}(L)$ om L är som i EXEMPEL 1.14 resp. EXEMPEL 1.16.

1.4 L sägs ha *prefixegenskapen* om ingen sträng i L är ett äkta prefix (se EXEMPEL 1.5) till någon annan sträng i L . Ge ett exempel på ett oändligt språk som har *prefixegenskapen*.

1.5 Bevisa (7) och (8) på sidan 21.

Reguljära språk och finita automater

Reguljära språk och reguljära uttryck.....	23
Deterministiska finita automater	26
Deterministiska finita automater	30
En utvidgad övergångsfunktion	33
Ickedeterministiska finita automater	34
För varje NFA finns det en ekvivalent DFA.....	38
Delmängdskonstruktionen	40
Reguljära språk är de finita automaternas språk	43
Tillståndseliminering och GFA:er	45
Några slutenhetsegenskaper	48
Minimering av en DFA.....	50
Minimering och särskiljande	50
Minimering genom reversering	57
De reguljära språkens gränser.....	59
Särskiljandesatsen	59
Pumpsatsen för reguljära språk	60
Reguljära språk och periodiska mönster	62
Reguljära språk och minne	63
Finita automater och lexikal analys m.m.	64
Finita automater med output	64
Mooremaskiner.....	65
Mealymaskiner	66
För varje Mooremaskin finns det en Mealymaskin och omvänt	68
Övningar	68

Ett naturligt sätt att konstruera språk är att använda språkoperationer (se sid 18). T ex kan man – genom att “operera” med *union*, *sammanfogning* och *Kleenestjärna* på redan kända språk L_1 och L_2 – få nya (mer innehållsrika) språk

$$L_1 \cup L_2, L_1 L_2, L_1^* \quad (1)$$

på vilka man i sin tur kan tillämpa språkoperationerna för att tillverka ännu innehållsrikare språk, osv

Märkligt nog kan man utgående från mycket enkla och föga uttrycksfulla språk “operera” fram relativt komplicerade dito – *reguljära språk* – genom att använda just de tre operationerna i (1).

2.1 Reguljära språk och reguljära uttryck

Låt $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ vara ett alfabet.

DEFINITION

De tre reguljära operationerna \rightarrow

- 1 \emptyset är ett reguljärt språk över Σ .
- 2 För varje $\sigma \in \Sigma$ är $\{\sigma\}$ ett reguljärt språk över Σ .
- 3 Om L_1 och L_2 är reguljära språk över Σ ,
så är $L_1 \cup L_2$, $L_1 L_2$, L_1^* reguljära språk över Σ .

EXEMPEL 2.1 Eftersom $\{\epsilon\} = \emptyset^*$ (Se 1.2 på sidan 21) så följer av 1 och 3 att $\{\epsilon\}$ är ett reguljärt språk.

EXEMPEL 2.2 $\{0\} \cup (\{1\}(\{0\}^*))$ är ett reguljärt språk över det binära alfabetet, skapat genom att operera med alla tre reguljära operationer.

Vanligtvis beskrivs reguljära språk med en sorts förenklad mängdnotation – s.k. *reguljära uttryck*:

DEFINITION

- 1 \emptyset och ϵ är reguljära uttryck för \emptyset resp. $\{\epsilon\}$.
- 2 För varje $\sigma \in \Sigma$ är σ ett reguljärt uttryck för $\{\sigma\}$.
- 3 Om α_1 och α_2 är reguljära uttryck för L_1 respektive L_2
så är $(\alpha_1 \cup \alpha_2)$, $(\alpha_1 \alpha_2)$, (α_1^*) , (α_1^+) reguljära uttryck för
 $L_1 \cup L_2$, $L_1 L_2$, L_1^* , L_1^+ .

EXEMPEL 2.3 $(0 \cup (1(0^*)))$ är ett reguljärt uttryck för språket i EXEMPEL 2.2. Parenteserna beskriver turordningen mellan operationerna.

⌘ **Anm. 2.1** Notera att ett reguljärt uttryck är en *ändlig sträng* som beskriver hur (de ofta oändligt många) strängarna i ett reguljärt språk är uppbyggda.

⌘ **Anm. 2.2** För att reducera antalet parenteser i reguljära uttryck, brukar man använda följande turordningskonventioner, och dessutom strunta i att skriva ut de yttersta parenteserna.

De reguljära operationernas turordningskonventioner.

Först Kleenestjärna, sedan sammanfogning, sist union.

EXEMPEL 2.4 T ex kan man därmed skriva $0 \cup 10^*$ istället för $0 \cup (1(0^*))$.

EXEMPEL 2.5 Här är (utan kommentarer) några reguljära språk över $\{a, b\}$ beskrivna med reguljära uttryck.

reguljärt språk	reguljärt uttryck
$\{ab\}$	ab
$\Sigma = \{a, b\}$	$\{a\} \cup \{b\}$ $a \cup b$
Σ^*	$\{a, b\}^*$ $(a \cup b)^*$
$\{\varepsilon, a, b, a^2, b^2, a^3, b^3, \dots\}$	$\{a\}^* \cup \{b\}^*$ $a^* \cup b^*$
$\{a^m b^n \mid m, n \in \mathbb{N}\}$	$a^* b^*$

⌘ **Anm. 2.3** Av bekvämlighetsskäl talar man ofta om ett reguljärt uttryck som om det vore ett reguljärt språk (fastän det bara är en *beskrivning* av ett reguljärt språk). T ex säger man "*betrakta det reguljära språket $a^* b^*$* " istället för "*betrakta det reguljära språket som $a^* b^*$ beskriver*".

EXEMPEL 2.6 Strängar över $\{a, b\}$ utan förekomst av a erhålles om man sammanfogar noll eller flera b :n. Härav det reguljära uttrycket b^* .

EXEMPEL 2.7 Strängar över $\{a, b\}$ med *exakt en* förekomst av a byggs genom att placera noll eller flera b :n till vänster och till höger om *ett* a . Strängarna med *exakt ett* a beskrivs därför av $(b^* a) b^*$ eller $b^* (a b^*)$. Ofta struntar man i parenteserna, och skriver $b^* a b^*$. (Se regeln (4) nedanför.)

EXEMPEL 2.8 Strängar över $\{a, b\}$ med *minst ett* a är av typ $\dots a \dots$, där ... betecknar en godtycklig sträng ur $(a \cup b)^*$. Härav det reguljära uttrycket $(a \cup b)^* a (a \cup b)^*$.

EXEMPEL 2.9 Strängar över $\{a, b\}$ med *högst ett* a innehåller *inget* a eller *exakt ett* a . Härav uttrycket $b^* \cup b^* a b^*$.

EXEMPEL 2.10 Strängar över $\{a, b\}$ som börjar på a och slutar på b är av typ $a \dots b$ och beskrivs därför av $a(a \cup b)^* b$.

EXEMPEL 2.11 Strängar av längd 5 över $\{a, b\}$ byggs genom att sammanfoga 5 tecken vilka som helst ur alfabetet. Härav $(a \cup b)^5$.

EXEMPEL 2.12 Strängar över $\{a, b\}$ vars längd är delbar med 5 erhålles om man sammanfogar noll eller flera strängar ur förra språket. Härav $((a \cup b)^5)^*$.

EXEMPEL 2.13 Siffersträngar över alfabetet $\Sigma_{dec} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ representerar på gängse sätt naturligt tal. De siffersträngar som representerar

rar tal delbara med 5 kännetecknas av att de slutar på 0 eller 5, och beskrivs således av $\sum_{dec}^*(0 \cup 5)$.

EXEMPEL 2.14 Siffersträngarna som representerar *jämna* naturliga tal (de som är delbara med två) kan också beskrivas av ett reguljärt uttryck: Om talen ifråga representeras binärt, slutar deras siffersträngar på 0, dvs beskrivs av $(0 \cup 1)^*0$. (Här tillåter vi nollskilda tal att börja på 0.)

Om talen istället representeras i tiosystemet, där jämna tals siffersträngar slutar på något av tecknen 0, 2, 4, 6, 8 ges de jämna talen av $\sum_{dec}^*(0 \cup 2 \cup 4 \cup 6 \cup 8)$.

Unär representation är inte slösaktig.

Ännu enklare blir det i s.k. unär representation, där talen representeras med *en* sorts tecken, säg tecknet 1, så att talet x representeras av x stycken 1:or. Då beskrivs de jämna talen av $(11)^*$.

EXEMPEL 2.15 Naturliga tal delbara med *tre* då? I unär representation får vi helt enkelt $(111)^*$. I binär representation är det knepigare (TEST 2.5.c)), likasom i tiosystemet. Men knappast i tresystemet (TEST 2.1.e)).

⌘ **Anm.2.4** Det finns många "räkneregler" för reguljära uttryck. Regler som har mer eller mindre känd form från andra delar av matematiken. Här är några smakprov (utan bevis):

$$\alpha \cup \beta = \beta \cup \alpha \quad (2)$$

$$\alpha \cup (\beta \cup \gamma) = (\alpha \cup \beta) \cup \gamma \quad (3)$$

$$\alpha(\beta\gamma) = (\alpha\beta)\gamma \quad (4)$$

$$\alpha(\beta \cup \gamma) = (\alpha\beta) \cup (\alpha\gamma) \quad (5)$$

$$(\alpha \cup \beta)^* = (\alpha^*\beta^*)^* = \alpha^*(\beta\alpha^*)^* \quad (6)$$

Den *kommutativa* regeln (2), de *associativa* reglerna (3), (4) och den *distributiva* regeln (5) är direkta konsekvenser av definitionerna på unionsbildning och sammanfogning, medan Kleenestjärnareglerna i (6) är något mer invecklade.

✂ **TEST 2.1**

a) Ange strängarna av längd fyra i $(ab \cup ba)^*$ och i $(aa \cup bb)^*$.

b) Visa att varje ensträngsspråk $\{w\}$ är reguljärt.

c) Visa att varje ändligt språk är reguljärt.

Skriv reguljära uttryck för de strängar som

d) inleds med ett jämnt antal 1:or vilka följs av ett udda antal 0:or,

e) beskriver talen delbara med tre representerade i tresystemet (där t ex tre, fyra, fem ges av 10, 11, 12),

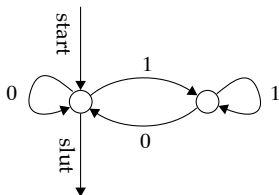
f) beskriver talen delbara med fyra representerade binärt.

g) Bevisa (6) i Anm.2.4.

■ 2.2 Deterministiska finita automater

Inledande exempel Med hjälp av riktade grafer kan reguljära språk beskrivas på ett uttrycksfullt sätt.

EXEMPEL 2.16 Betrakta följande graf:



Grafen är ett sorts "uttryck" byggd med i huvudsak pilar (bågar), noder och tecknen 0,1. Vi skall här tolka grafen som en beskrivning av vissa strängar. Närmare bestämt de binära strängar som man kan läsa när man följer de vägar i grafen som börjar i den nod där startpilen går in och slutar där slutpilen går ut. Exempel på sådana strängar är 10 och 00100010. Den senare beskrivs i grafen av att man efter start tar den inledande 0-bågen (öglan) två gånger, sedan 1-bågen över till den högra noden och 0-bågen tillbaka till den vänstra noden igen, samt upprepar dessa fyra bågar ytterligare en gång.

I själva verket beskriver grafen samtliga binära strängar som slutar på 0, och dessutom tom sträng. Dvs (utöver tom sträng) samtliga binärt representerade tal som är delbara med två. Jämför med EXEMPEL 2.14.

Nästa exempel handlar också om delbarhet.

EXEMPEL 2.17 Att dividera ett naturligt tal x med ett naturligt tal $d \neq 0$ går ut på att upprepade gånger (så länge återstoden av x är större eller lika med d) subtrahera d från x . *Antalet* upprepade subtraktioner tillsammans med *återstoden* av x utgör divisionens output och kallas för *kvoten* respektive *resten*. Den just beskrivna algoritmen går under namnet divisionsalgoritmen och kan formuleras med någon av följande fyra rader.

$$x - d - d - \dots - d = \text{resten}$$

$$x - \text{kvoten} \cdot d = \text{resten}$$

$$x = \text{kvoten} \cdot d + \text{resten}$$

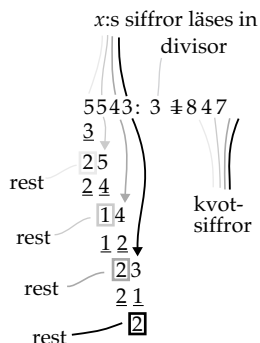
$$\frac{x}{d} = \text{kvoten} + \frac{\text{resten}}{d}$$

Divisionsalgoritmen som den beskrivs ovanför är *oberoende* av talens representation. Vad jag menar är att algoritmen är presenterad på en abstraktionsnivå som inte bryr sig om huruvida talen representeras i ett positionssystem med arabiska siffror eller i termer av stenhögar eller ...

Annorlunda är det med den sedvanliga papper-och-penna-varianten av divisionsalgoritmen. Här utföres en s.k. *lång division* i *tio-systemet* på ett sätt som utnyttjar att talen representeras i just ett positionssystem:

En lång division ...

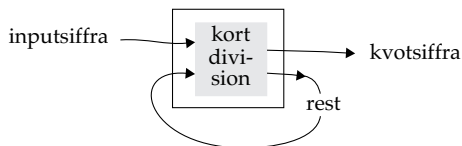
... består av flera korta.



När man dividerar $x = 5543$ med $d = 3$, tar man först x s inledande 5:a och dividerar den med d samt noterar att kvoten blir 1 och resten 2. Sedan läser man in nästa 5:a, klistrar ihop densamma med den nyss erhållna resten så att man får 25 vilken divideras med d och därvid ger upphov till kvoten 8 och resten 1. På detta sätt fortskrider arbetet med inläsning av nytt tecken, ihopklistring med förra resten, osv

Vi skall nu betrakta denna papper-och-penna-algoritm ur ett för datalogin centralt perspektiv.

Varje inläst inputsiffra driver algoritmen att göra en kort division (med d) som ger upphov till en blott *ensiffrig kvot* (*kvotsiffra*) och en *rest* vars värde är något av talen $0, 1, 2, \dots, d-1$. Den erhållna resten matas vid nästa korta division in som tilltugg tillsammans med nästa inlästa inputsiffra. Algoritmen "kör" således den korta divisionen inte enbart med hjälp av inläst siffra från x utan dessutom med hjälp av *egenproducerat* input.

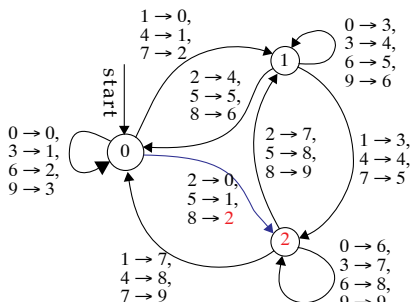


kort division
=
tillstånds-
övergång

Oavsett vilka inputsiffror som läses in från x kommer algoritmens egenproducerade input inte att ha några andra värden än $0, 1, 2, \dots, d-1$. Därför kan man säga att värdena $0, 1, 2, \dots, d-1$ på *resterna* utgör algoritmens möjliga *tillstånd*. Varje kort division blir med detta synsätt en *övergång* mellan två tillstånd. Och en lång division en sekvens av tillståndsövergångar. Det som driver algoritmen från tillstånd till tillstånd är inputströmmen av siffterecken från x .

Ett lämpligt medium för sådan tillståndsbeskrivning erbjuder grafteorin. Nedan visas en riktad graf som beskriver "division med 3":

Kvotsiffrorna returneras under tillståndsövergångarna, resterna vid tillståndsbesöken.

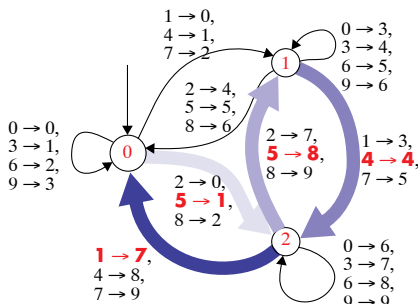


Vid "körning" i grafen skall talet som man ämnar dividera med 3 läsas in siffra för siffra, med början i grafens starttillstånd. Varje tillståndsövergång innehåller information av typen

inläst siffra → kvotsiffra

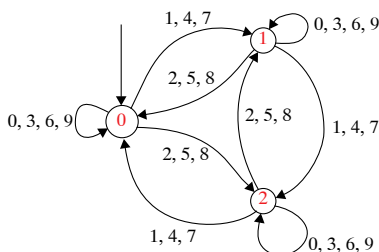
som talar om vilken kvotsiffra som tillståndsövergångens korta division ger upphov till. Det tillstånd som aktuell båge pekar på anger resten vid övergången ifråga. Nedan presenteras en "körning" med inputsträngen 5541.

Provkörning på inputsträngen 5541. Divisionens kvot avläses som en "ström" av de kvotsiffror som "kommer ut" vid tillståndsövergångarna.



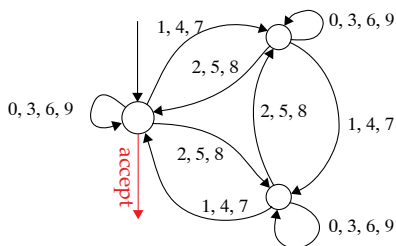
EXEMPEL 2.18 Om vi enbart vill veta *vilka* (slutgiltiga) *rester* som inlästa tal ger vid division med 3, behöver vi bara se efter vilka tillstånd vi befinner oss i efter att talen ifråga har lästs in. Önskade output finns således i tillstånden. Eftersom kvotsiffrorna saknar relevans i detta fall har vi tagit bort dem från bågarna:

Här returneras enbart resterna.

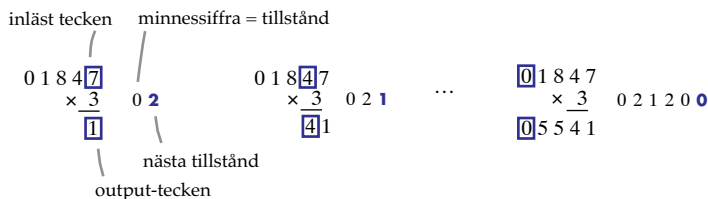


EXEMPEL 2.19 (*Mönsterigenkännare*) Betrakta grafen i EXEMPEL 2.18 igen. Antag att vi vill använda denna graf enbart för att undersöka om inlästa tal är delbara med 3. Hur kan man se ifall ett inläst tal uppfyller detta kriterium? Svaret är att det räcker att kontrollera om man hamnade i tillståndet 0 (som anger att resten blev 0) efter att talets alla siffror har lästs in. I nedanstående graf har vi tagit fast på detta, och satt en utgående båge på tillståndet 0 (utöver startbågen) för att markera att det är här som inläsningen av tal delbara med 3 kan avslutas.

Tal
delbara
med 3
"accepteras",
andra tal
"förkastas".

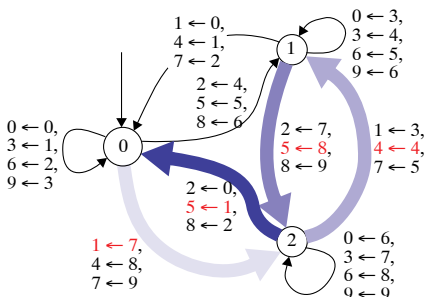


EXEMPEL 2.20 Även papper-och-penna-algoritmen för *multiplikation* med ett fixt tal (t ex 3) kan beskrivas med hjälp av tillståndsövergångar. Multiplikationens minnessiffror representerar tillstånden i detta fall.



Presenteras ovanstående algoritm med en tillståndsövergångsgraf (se nedan) finner man – kanske inte helt oväntat – att grafen överensstämmer med den *baklängeskörd*a grafen för division med samma fixa tal.

En tillståndsövergångsgraf för "multiplikation med 3" körd på inputsträngen 1847. Jämför med grafen för "division med 3" på sid 28.



Observera f.ö. att precis som vid sedvanlig multiplikationsupp-

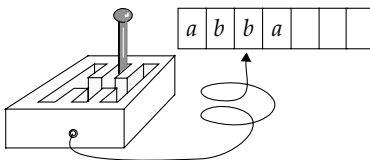
ställning skall en inputsträng (som t ex 1847) läsas in från höger.

Efter de inledande exemplen presenterar vi nu begreppet *finit automat*. Vi börjar med mönsterigenkännande automater. Och av dessa börjar vi med de *deterministiska*.

■ Deterministiska finita automater

tape = band
eller remsa

Informell beskrivning En *deterministisk finit automat*[†] (DFA) utgörs av en s.k. *tape* och en *kontrollmekanism*.



På tapen placeras strängar som DFA:n läser (konsumerar) – ett tecken i taget från vänster till höger.

ändlighet

Kontrollmekanismen kan likt en växellåda befinna sig i ett ändligt antal lägen (tillstånd).

Det här motsvarar att man kan befinna sig på ändligt många ställen i ett program. Och precis som man drivs runt från det ena stället till det andra i ett program beroende på var man befinner sig och beroende på vilka värden programmets data har på just det ställe man befinner sig, på motsvarande sätt drivs en DFA av sin kontrollmekanism runt från det ena tillståndet till det andra allt efter vilket tillstånd DFA:n befinner sig i och vilket tecken som den konsumerar just då. Detta tillståndshoppande är vanligtvis DFA:ns hela agerande[‡]. Man brukar säga att DFA:n *drivs* från tillstånd till tillstånd.

determinism

När vi säger att en automat är *deterministisk* menar vi att den måste agera på ett *bestämt* (dvs entydigt) sätt från start och framåt. Med detta menas att den bara har ett s.k. *starttillstånd*. Och att det bara finns *ett* bestämt tillstånd att hoppa till varje gång som den befinner sig i ett visst tillstånd och konsumerar ett visst tecken.

accepterande
tillstånd
(sluttillstånd)

Om DFA:n skall användas för mönsterigenkänning är den utrustad med noll eller flera s k *accepterande tillstånd* (även kallade *sluttillstånd*).

Poängen med de accepterande tillstånden är följande. Eftersom en DFA har till uppgift att känna igen ett visst mönster i inputsträngen, så måste den – efter att ha läst strängen – lämna ifrån sig ett meddelande huruvida strängen hade mönstret ifråga eller ej. Det är just det-

[†]. *Finit automat* en försvenskning av det engelska *finite automaton*.

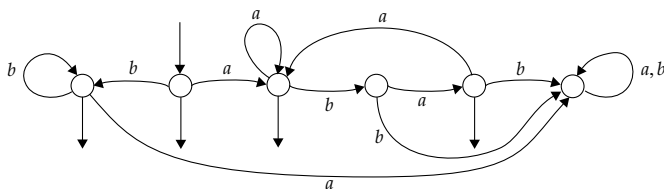
De som inte gillar svengelskan brukar säga *ändlig automat* eller *ändlig tillståndsmaskin*, där det senare är en översättning av engelskans *finite state acceptor* (FSA).

[‡]. Vissa finita automater lämnar dessutom någon form av output under arbetet.

ta meddelandeproblem som löses med accepterande tillstånd. Att DFA:n står i ett accepterande tillstånd är nämligen DFA:ns signal till användaren att hittills konsumerad sträng hade det mönster som DFA:n var specialiserad på att känna igen.

Grafisk beskrivning Vanligtvis åskådliggörs en DFA med hjälp av en *riktad graf* vars noder beskriver DFA:ns tillstånd och vars riktade bågar beskriver tillståndsövergångarna.

Bågen markerad med *två* tecken *a, b* avser egentligen två bågar med vardera ett tecken.



Från varje nod utgår bågar markerade med alfabetets olika tecken. På detta sätt kan DFA:n "läsa" strängar längs grafens vägar. *Accepterande* tillstånd markeras med utgående pil, och *starttillstånd* med en "från intet" inkommande startbåge.

Formell definition

DEFINITION En DFA M är en kvintupel

$$(Q, \Sigma, \delta, s, F)$$

där

- (i) Q är en ändlig mängd (tillstånden)
- (ii) Σ är en ändlig mängd (inputalfabetet)
- (iii) δ är en funktion (övergångsfunktionen) från $Q \times \Sigma$ till Q
- (iv) $s \in Q$ (starttillståndet)
- (v) $F \subseteq Q$ (de accepterande tillstånden)

⌘ **Anm.2.5** δ :s (*input, output*) representeras ofta som tripplar:

$$\underbrace{\text{tillstånd, tecken}}_{\text{input}}, \underbrace{\text{nästa tillstånd}}_{\text{output}}$$

δ :s hela (*input, output*)-mängd blir därmed en mängd av tripplar, vilken kan presenteras på några olika sätt. Se exemplet som följer.

EXEMPEL 2.21 Här är en formellt definierad DFA med två tillstånd:

Hur ser
DFA:ns graf
ut?

$M = (Q, \Sigma, \delta, s, F)$ där

$Q = \{s, q\}$, $\Sigma = \{a, b\}$, $F = \{q\}$,

och δ ges av $\{(s, a, s), (s, b, q), (q, a, q), (q, b, s)\}$

Övergångsfunktionen δ kan även ges i tabellform t ex som nedan:

tillstånd	tecken	nästa tillstånd
s	a	s
s	b	q
q	a	q
q	b	s

eller

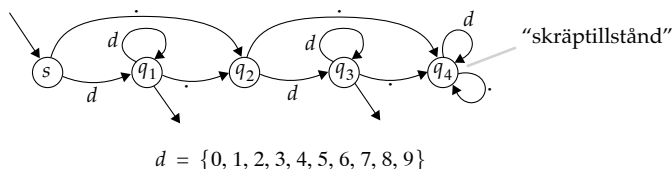
tecken	tillstånd	
	s	q
a	s	q
b	q	s

Ofta säger man att en DFA "känner igen" en sträng som den *accepterar*.

Acceptans En DFA sägs *acceptera* en sträng w , om w kan "läsas" längs någon väg från starttillståndet till något accepterande tillstånd i DFA:ns graf. Man brukar uttrycka detta som att w driver DFA:n från starttillståndet till ett accepterande tillstånd. (Se även DEFINITIONEN på sid 34.)

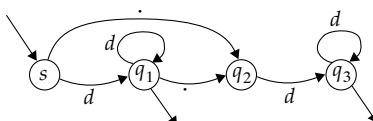
EXEMPEL 2.22 Siffersträngar som representerar decimaltal (t ex 69 eller 3.14) accepteras av nedanstående DFA, vars alfabet är $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ Strängar med flera decimalpunkter driver DFA:n till ett s.k. *skräptillstånd* – ett tillstånd från vilket den aldrig kan nå ett accepterande tillstånd.

En decimaltalsigenkännaren DFA med skräptillstånd.

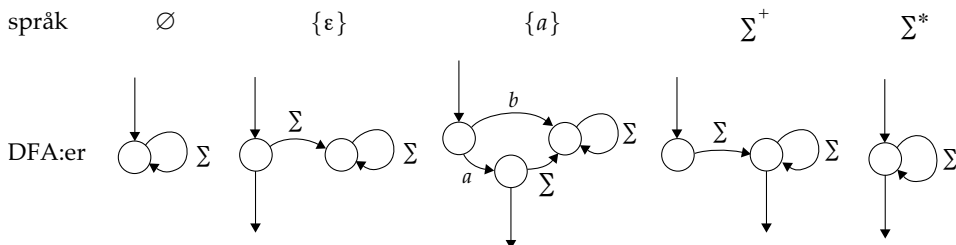


⌘ **Anm. 2.6** I syfte att göra DFA:ernas grafer mindre klottriga (och därigenom öka läsbarheten) brukar man ofta strunta i att rita ut skräptillstånd.

EXEMPEL 2.23 Decimaltalsigenkännaren utan skräptillstånd:



EXEMPEL 2.24 De små språken \emptyset , $\{\epsilon\}$, $\{a\}$ och de stora språken Σ^+ , Σ^* känns igen (accepteras) av DFA:erna nedanför.



■ En utvidgad övergångsfunktion

Övergångsfunktionen $\delta: Q \times \Sigma \rightarrow Q$ beskriver DFA:ns *atomära* agerande, dvs hur *enskilda tecken driver* DFA:n. Med hjälp av δ kan man definiera en funktion som beskriver hur *strängar vilka som helst* (noll eller flera tecken långa) driver DFA:n:

En DFA:s utvidgade övergångsfunktion.

DEFINITION För en DFA definieras funktionen $\delta^*: Q \times \Sigma^* \rightarrow Q$ av

$$\begin{cases} \delta^*(p, \epsilon) = p & \text{basfall} \\ \delta^*(p, \sigma w) = \delta^*(\delta(p, \sigma), w) & \text{rekursionssteg} \end{cases}$$

Försök – innan Du går vidare – att formulera definitionens två rader med egna ord.

EXEMPEL 2.25 Till vilket tillstånd drivs decimaltalsigenkännaren (EXEMPEL 2.22) av $w = .13$? Efter en kort blick i tillståndsgrafen kan Du givetvis ge det rätta svaret. Det intressanta just nu är emellertid att en algoritm kan detsamma:

$$\begin{aligned} \delta^*(s, .13) &= \delta^*(\delta(s, .), 13) \\ &= \delta^*(\delta(\delta(s, .), 1), 3) \\ &= \delta^*(\delta(\delta(\delta(s, .), 1), 3), \epsilon) \\ &= \delta^*(\delta(\delta(q_2, 1), 3), \epsilon) \\ &= \delta^*(\delta(q_3, 3), \epsilon) \\ &= \delta^*(q_3, \epsilon) \\ &= q_3 \end{aligned}$$

Vad menas egentligen med "att driva"?

Poängen med definitionen av δ^* är inte bara räknemässig, utan också att man med dess hjälp får en tillfredställande *formell* definition av begreppet "att driva". Och därmed blir också en DFA:s (accepterade) språk formellt definierat:

DEFINITION Om $M = (Q, \Sigma, \delta, s, F)$ är en DFA, $p, q \in Q$ och w är en sträng över Σ , så säger man att

- (i) w driver M (i noll eller flera steg) från p till q om $\delta^*(p, w) = q$
 - (ii) M 's (accepterade) språk är $L(M) = \{w \mid \delta^*(s, w) \in F\}$.
-

- ✂ **TEST 2.2** a) Hur många (*input, output*)-par har δ , om Q innehåller m stycken tillstånd och Σ innehåller n stycken tecken?
- b) Konstruera DFA:er för språken $(ab \cup ba)^*$ resp. $(aa \cup bb)^*$ över $\Sigma = \{a, b\}$.
- c) Konstruera en DFA för de strängar över $\Sigma = \{0, 1\}$ som inleds med ett jämnt antal 1:or vilka följs av ett udda antal 0:or.

■ 2.3 Ickedeterministiska finita automater

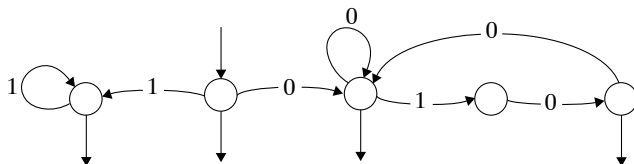
Ickedeterminism är att *inte* vara låst vid den "enda vägen".

För en given teckenkonsumtion i ett givet tillstånd skall vi nu tillåta en finit automat att agera på flera möjliga sätt. Detta kallas för *ickedeterminism*. Även igångsättning av en finit automat kommer att tillåtas ske på ett flertydigt sätt – i två eller flera starttillstånd.

Dessutom passar vi på att tillföra mer flexibilitet som inte direkt har med ickedeterminism att göra: Automaterna skall tillåtas konsumera flera tecken åt gången och dessutom kunna hänga sig.

EXEMPEL 2.26 Först en DFA:

En
deterministisk
finit automat
(skräptillståndet ej utritat).

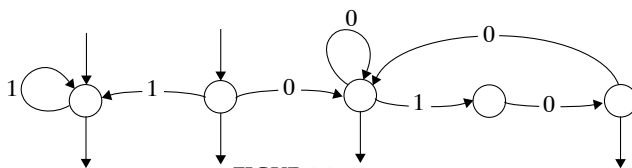


FIGUR 2.1

Genom att utrusta ovanstående automat med ytterligare ett starttillstånd – så att den kan startas i vilket som helst av två tillstånd – blir den *ickedeterministisk*. Med det synsätt som vi skall tillämpa (se avsnittet *Acceptans* på sid 36) accepterar den emellertid samma språk som

DFA:n ovan.

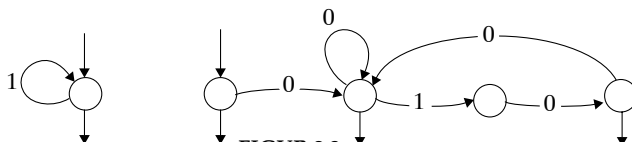
En finit automat med *icke-determinism* i starten.



FIGUR 2.2

Övergången mellan de två starttillstånden i FIGUR 2.2 fyller ingen funktion. Ty det som automaten kan göra genom att använda denna övergång kan den göra utan densamma (genom att starta i det vänstra tillståndet). Avlägsnar vi övergången ifråga får vi en *osammanhängande* automat bestående av två åtskilda komponenter:

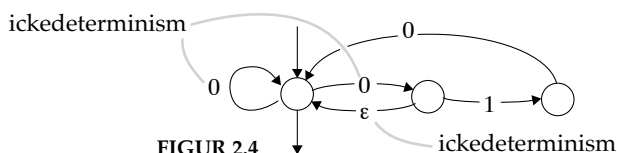
En osammanhängande finit automat. Observera att den ändå skall uppfattas som *en* automat – inte flera.



FIGUR 2.3

Den högra komponenten med sina fyra tillstånd i FIGUR 2.3 är deterministisk men ges en ekvivalent[†] framställning i FIGUR 2.4 med hjälp av en ickedeterministisk automat innehållande tre tillstånd. Innebörden i ϵ -övergången från det mittersta tillståndet till det vänstra är att om automaten står i det mittersta tillståndet kan den *utan teckenkonsumtion* endera stå kvar *eller* ta ϵ -övergången till det vänstra tillståndet – ickedeterministiskt agerande således.

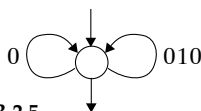
Ickedeterminism i tillståndsövergångarna.



FIGUR 2.4

Slutligen kommer en mycket *liten* automat som är ekvivalent med den förra – och vars utmärkande drag är *glupskhets*. (Den kan konsumera en tre tecken lång sträng i en enda övergång.)

En *glupsk* finit automat.



FIGUR 2.5

†. Vi säger att två finita automater är ekvivalenta om deras språk överensstämmer.

DEFINITION En NFA (ickedeterministisk finit automat) M är en kvintupel

$$(Q, \Sigma, \Delta, S, F)$$

där

Δ är en
relation!

- (i) Q är en ändlig mängd
 - (ii) Σ är en ändlig mängd
 - (iii) Δ är en ändlig delmängd (övergångsmängden) av $Q \times \Sigma^* \times Q$
 - (iv) $S \subseteq Q$ (starttillståndsmängden)
 - (v) $F \subseteq Q$
-

EXEMPEL 2.27 NFA:n i FIGUR 2.5 beskrivs av $M = (Q, \Sigma, \Delta, S, F)$ där $\Sigma = \{0, 1\}$, $Q = \{s\} = F$ och $\Delta = \{(s, 0, s), (s, 010, s)\}$.

⌘ **Anm.2.7** Tillståndsovergångarna i en NFA definieras som en mängd Δ av tripplar (tillstånd, sträng, tillstånd), dvs som en treställig relation. Att inga som helst krav, förutom ändlighet, ställs på Δ innebär

- att Δ tillåts innehålla tripplar $(p, u, q_1), (p, u, q_2), \dots, (p, u, q_n)$ med skilda q_i , dvs automaten tillåts välja mellan skilda tillstånd q_i att hoppa till,
- att Δ tillåts innehålla trippel (p, u, q) där u är tom eller där u :s längd är större än 1,
- att Δ för något tillstånd p och något tecken $\sigma \in \Sigma$ kan sakna trippel (p, σ, \dots) .

ϵ -övergångar
eller glupska
övergångar

■ **Acceptans** En NFA M sägs acceptera en sträng $w = u_1 \dots u_n$ om det finns en väg från något $s \in S$ till något $q \in F$ längs vilken M kan läsa i tur och ordning u_1, \dots, u_n . Man brukar uttrycka detta som att w driver (eller kan driva) M från ett starttillstånd till ett accepterande tillstånd. (Se även den formella DEFINITIONEN på sid 37.)

"Misslyckade"
vägar negligeras
om det finns en
väg till
acceptans.

Observera att det för en NFA M är tillåtet (ickedeterminism!) att en och samma sträng w driver M både till ett accepterande och till ett ick-accepterande tillstånd. Men det är enbart w :s möjlighet att driva M till ett accepterande tillstånd som har betydelse när det gäller frågan "Accepteras w av M ?".

⌘ **Anm.2.8** På motsvarande sätt som en DFA har sin utvidgade övergångsfunktionen δ^* så har en NFA sin utvidgade övergångsrelationen Δ^* :

En NFA:s
utvidgade
över-
gångsrela-
tion.

DEFINITION För en NFA definieras $\Delta^* \subseteq Q \times \Sigma^* \times Q$ av

$$\begin{cases} \{p, \varepsilon, p\} \in \Delta^* & \text{för varje } p \\ \{p, wu, r\} \in \Delta^* & \text{om } (p, w, q) \in \Delta^* \text{ och } (q, u, r) \in \Delta \end{cases}$$

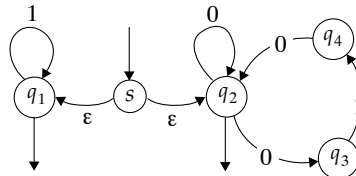
De två raderna ovanför uttrycker att $(p, u_1 \dots u_n, r) \in \Delta^*$ om det finns en väg från p till r längs vilken M konsumerar i tur och ordning u_1, \dots, u_n .

Med Δ^* :s hjälp kan vi nu formalisera begreppen *driva* och *acceptera*:

DEFINITION Om $M = (Q, \Sigma, \Delta, S, F)$ är en NFA, $p, r \in Q$ och w är en sträng över Σ , så säger man att

- (i) w kan *driva* M från p till r (i noll eller flera steg) om $(p, w, r) \in \Delta^*$
- (ii) M *accepterar* w om $(s, w, r) \in \Delta^*$ och $s \in S, r \in F$
- (iii) M 's språk är $L(M) = \{w \mid (s, w, r) \in \Delta^*, s \in S, r \in F\}$

EXEMPEL 2.28 Den utvidgade övergångsrelationen för vidstående NFA exemplifieras nedanför.



$$(s, \varepsilon, q_2) \in \Delta^* \quad \text{ty} \quad \begin{cases} \varepsilon = \varepsilon\varepsilon \\ (s, \varepsilon, s) \in \Delta^* \text{ och } (s, \varepsilon, q_2) \in \Delta \end{cases}$$

$$(s, 0, q_3) \in \Delta^* \quad \text{ty} \quad \begin{cases} 0 = \varepsilon 0 \\ (s, \varepsilon, q_2) \in \Delta^* \text{ och } (q_2, 0, q_3) \in \Delta \end{cases}$$

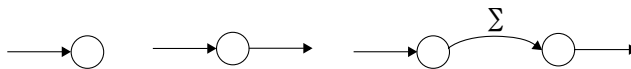
$$(s, 01, q_4) \in \Delta^* \quad \text{ty} \quad \begin{cases} 0 = \varepsilon 0 \\ \{s, 0, q_3\} \in \Delta^* \text{ och } (q_3, 1, q_4) \in \Delta \end{cases}$$

⌘ **Anm. 2.9** Varje DFA är en NFA med *ett* enda starttillstånd, och övergångstripplarna (*tillstånd*, *sträng*, *nästa tillstånd*) har följande egenskaper:

- *sträng* har *längd* = 1,
- till *varje* (*tillstånd*, *sträng*) hör *exakt ett* *nästa tillstånd*.

EXEMPEL 2.29 De tre extrema språken \emptyset , $\{\varepsilon\}$, Σ accepteras av de tre NFA:erna nedanför. (Vilka språk hör ihop med vilka automater?)

Tre extrema
språks
NFA:er.



FIGUR 2.6

✂ **TEST 2.3** Konstruera glupska NFA:er för

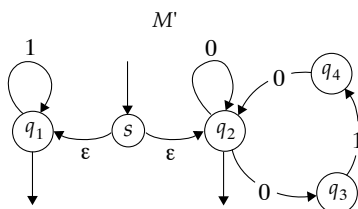
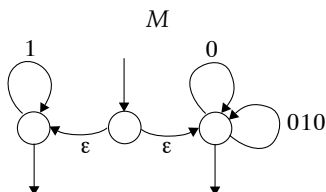
- a) $(ab \cup ba)^*$, b) $a^*a(bb)^*$.

■ 2.4 För varje NFA finns det en ekvivalent DFA

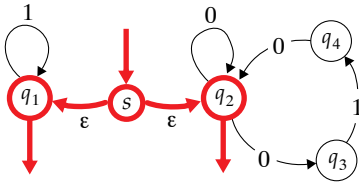
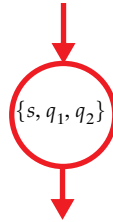
Eftersom vi har introducerat begreppet NFA som en utvidgning av begreppet DFA skulle man kunna tro att det finns NFA:er som kan känna igen mer komplicerade språk än vad någon DFA kan. Så är emellertid icke fallet. För varje NFA M kan man nämligen konstruera en DFA *ekvivalent* med M . Konstruktionen av DFA:n kallas för *deltmängdskonstruktionen* och illustreras i exemplet nedanför. Därefter ges en utförligare beskrivning.

Beträffande
ekvivalenta
automater, se
fotnot på sid 35.

EXEMPEL 2.30 Vi transformerar en NFA till en ekvivalent DFA. Notera särskilt den senares tillståndsbeteckningar.

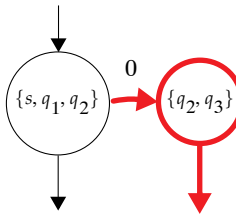
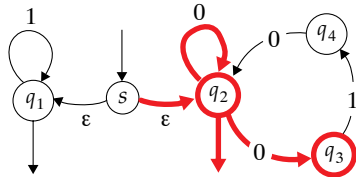


STEG 1: Först konstrueras en *ickeglupsk* NFA M' ekvivalent med M . Därefter (se nedan) inleds den egentliga delmängdskonstruktionen där en med M' ekvivalent DFA skapas. Den senare benämns M'' nedanför.

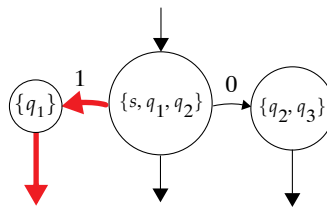
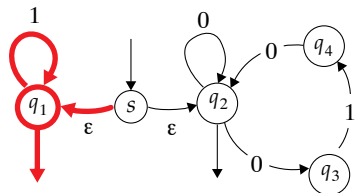
NFA M' DFA M'' 

STEG 2:

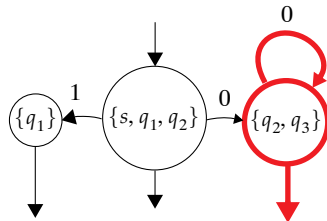
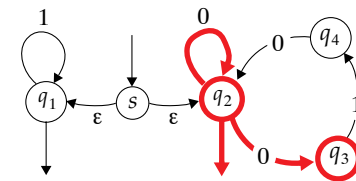
Som starttillstånd i M'' sätter vi *mängden* $\{s, q_1, q_2\}$ av tillstånd som M' vid start kan drivas till utan teckenkonsumtion. Notera f.ö. att $\{s, q_1, q_2\}$ skall åtnjuta *accepterandestatus* i M'' , eftersom q_1, q_2 har motsvarande status i M' .



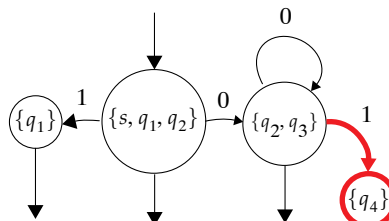
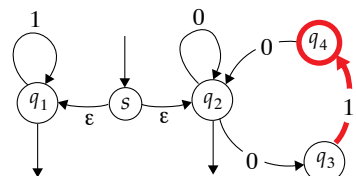
0-utgångstillstånd från $\{s, q_1, q_2\}$ blir mängden av tillstånd i M' som man kan nå från något av tillstånden s, q_1, q_2 genom konsumtion av 0.



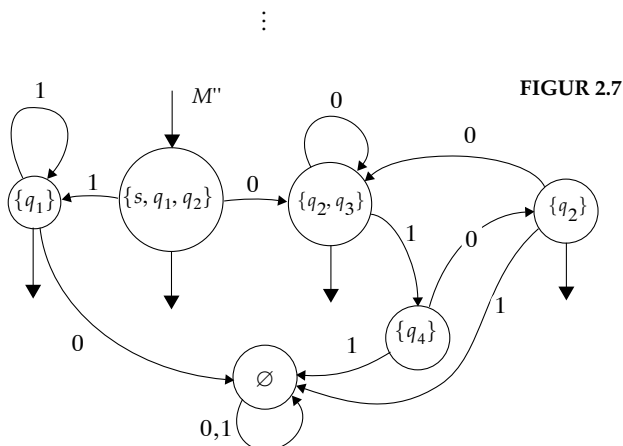
På motsvarande sätt bestäms 1-utgångstillståndet från $\{s, q_1, q_2\}$.



... liksom 0-utgångstillstånd från $\{q_2, q_3\}$



... och 1-utgångstillstånd från $\{q_2, q_3\}$.



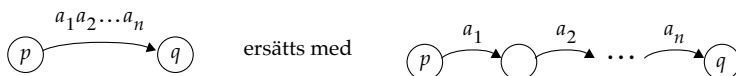
När M'' till slut blir klar, har vi en med M ekvivalent DFA vars tillstånd representeras av delmängder av M' 's tillståndsmängd.

Observera att det tillstånd som vi betecknat med \emptyset i den slutgiltiga DFA:n ovanför är ett s.k. *skräptillstånd*. (Se sid 32).

■ Delmängdskonstruktionen

Konstruktionen som sker i två steg kan beskrivas på följande sätt:

STEG 1 Ersätt varje glupsk strängkonsumtion med "ett-tecken-i-taget"-konsumtion genom att skjuta in ickeaccepterande tillstånd:



STEG 2 Om en sträng p. g. a. flertydighet i NFA:ns tillståndsovergångar kan driva NFA:n till *skilda* tillstånd, så skall vi slå ihop dessa tillstånd till *ett* tillstånd. Dvs *mängden av tillstånd* som NFA:n kan drivas till (av strängen ifråga) får representera *ett tillstånd* Q i en DFA tänkt att härma NFA:n.

För att förenkla resonemanget kallar vi den automat som STEG 1 gav upphov till för *den gamla*. Och den DFA som vi på det beskrivna sättet skall konstruera kallar vi för *den nya*.

Ett *starttillstånd* i den nya får representeras av mängden av alla tillstånd till vilka den gamla kan drivas till av ϵ från något starttillstånd, dvs vi slår ihop alla tillstånd i den gamla som man kan nå från ett starttillstånd utan att konsumera något tecken.

Antag nu att vi har konstruerat ett tillstånd P i den nya automaten såsom en delmängd av tillstånd från den gamla. Som övergångstillstånd från P vid konsumtion av ett tecken $\sigma \in \Sigma$ väljs mängden Q av alla tillstånd (i den gamla) som man genom att konsumera σ kan nå från något av de tillstånd som ingår i mängden P .

Notera att $\sigma\varepsilon = \varepsilon\sigma = \sigma$, varför eventuella tillstånd som nås (från något av P 's tillstånd) genom ε -övergång före eller efter en σ -konsumtion kommer att inkluderas i Q .

Vidare måste även ett *tomt* Q tas i beaktande som blir ett s.k. *skräp-tillstånd* i den nya, svarande mot att den gamla, för något tecken $\sigma \in \Sigma$, inte kan drivas till något enda tillstånd. (Se t.ex. tillståndet betecknat \emptyset i FIGUR 2.7.) Ty vi vill ju ha en DFA, och en sådan har i varje tillstånd specificerade övergångstillstånd för *varje* tecken.

Ett delmängdstillstånd Q i den nya automaten är *accepterande* om Q innehåller något tillstånd som i den gamla automaten är ett accepterande tillstånd. Ty den gamla accepterar en sträng om det finns något accepterande tillstånd som den kan drivas till (av strängen), och den nya skall ju härma den gamla.

Vi bevisar nu att denna konstruktion håller vad den lovar.

✧ SATS 2.1 För varje NFA M finns det en DFA ekvivalent med M .

BEVIS: Delmängdskonstruktionens första steg resulterar i en automat som uppenbarligen accepterar samma språk som ursprungsautomaten, men som inte säkert är deterministisk. (STEG 1 tvingar ju bara ursprungsautomaten att konsumera mindre glupskt.)

Vi diskuterar nu STEG 2:

Var och en som själv har utfört STEG 2 för första gången, och därvid bildat nya delmängder i en till synes aldrig sinande ström, ställer sig nog följande fråga: *Har STEG 2 alltid ett slut?*

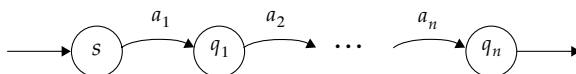
Denna fråga kan vi besvara på följande sätt. Om automaten före STEG 2 har N tillstånd, så har automaten efter STEG 2 högst 2^N tillstånd. Ty varje tillstånd hos denna senare automat är en delmängd av tillstånd tagna från den första, och det finns 2^N möjliga sådana delmängder.[†] Detta bevisar att STEG 2 har ett slut. Resulterar STEG 2 i en DFA? Javisst, ty konstruktionen är sådan att vi specificerar *ett* övergångstillstånd för varje tillståndsmängd och varje $\sigma \in \Sigma$.

Slutligen visar vi att automaten före respektive efter STEG 2 accepterar samma språk: (Vi kallar åter de två automaterna för den gamla respektive den nya.)

Att en sträng $w = a_1a_2\dots a_n$ accepteras av den gamla, innebär att det som FIGUR 2.8 nedan illustrerar finns (minst) en väg från starttillstånd till accepterande tillstånd.

En väg i den gamla automaten.

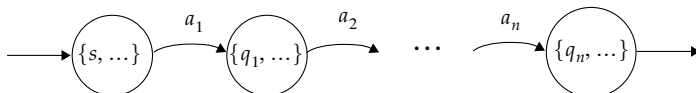
FIGUR 2.8



†. När man skall specificera en delmängd av en given mängd har man för varje element (i den senare mängden) två alternativ: att låta elementet ifråga vara med (i delmängden) eller att *inte* låta det vara med.

Men för en sådan "tillståndsväg" s, q_1, \dots, q_n i den gamla automaten finns det en motsvarande väg i den nya: En väg av delmängdstillstånd, vilka i tur och ordning innehåller tillstånden s, q_1, \dots, q_n . Ty av delmängdskonstruktionen (läs den!) ser man att det första delmängdstillståndet i denna väg innehåller s . Och det andra innehåller alla de tillstånd, däribland q_1 , som den gamla automaten drivs till när den konsumerar a_1 och står i tillståndet s . Osv...

Motsvarande
väg i den nya
automaten.



FIGUR 2.9

Det sista delmängdstillståndet (längs vägen ifråga) innehåller det accepterande tillståndet a_n från den gamles väg, och är därför ett accepterande tillstånd i den nya automaten. Dvs, en sträng som accepteras av den gamla accepteras också av den nya.

Nu vänder vi på det hela:

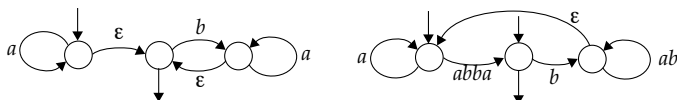
Att en sträng $w = a_1 a_2 \dots a_n$ accepteras av den nya automaten innebär att det finns en väg (precis en) från starttillståndet till ett accepterande tillstånd i den nya automaten, precis som i FIGUR 2.9. Men eftersom tillståndsövergångar i den nya automaten har sitt ursprung i existerande tillståndsövergångar i den gamla automaten, måste det finnas en väg (minst en) i den gamla, precis som i FIGUR 2.8. Och det sista tillståndet i denna väg måste vara ett accepterande tillstånd. (Varför?) Med andra ord, en sträng som accepteras av den nya automaten accepteras också av den gamla. \square

⌘ **Anm.2.10** SATS 2.1 säger att NFA:ernas språk bildar en delmängd av DFA:ernas språk. Eftersom även det omvända är sant – som en konsekvens av att varje DFA är en NFA – så följer att NFA:ernas språk överensstämmer med DFA:ernas. Vi skall i avsnitt 2.5 visa att dessa språk är just de reguljära språken.

⌘ **Anm.2.11** Eftersom NFA:er inte accepterar några språk utöver vad DFA:er accepterar, faller det sig naturligt att resa frågan vad det är för nytta med NFA:er. Ett svar är att det ofta är enklare att tänka ut en NFA än en DFA. (Vi tänker gärna i icke-deterministiska banor.)

✂ TEST 2.4

a) Transformera med hjälp av delmängdskonstruktionen följande två NFA:er till DFA:er.



- b) Konstruera en DFA för talen delbara med tre representerade binärt (där t ex tre, sex, nio ges av 11, 110, 1001).
LEDNING: Divisionsalgoritmen.

■ 2.5 Reguljära språk är de finita automaternas språk

Vi skall här visa två saker:

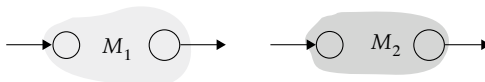
- 1 Varje reguljärt språk accepteras av en finit automat.
- 2 Varje språk som accepteras av en finit automat är reguljärt.

Punkt 1 bevisas med hjälp av följande konstaterande:

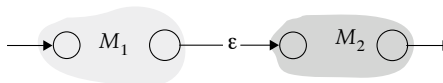
- *seriekoppling* av två automater accepterar sammanfogningen av deras språk.
- *parallellkoppling* accepterar unionen, och
- *återkoppling* av en automat accepterar Kleenestjärnatillslutningen av dess språk.

Innebörden i dessa kopplingar förklaras i nedanstående figurer[†].

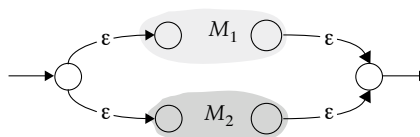
Om
 M_1 och M_2 ...



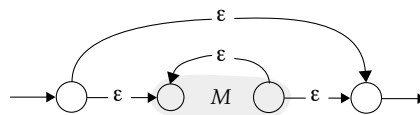
... seriekopplas
får man en finit
automat för
 $L(M_1)L(M_2)$



... parallell-
kopplas får man
en finit automat
för
 $L(M_1) \cup L(M_2)$



Om en DFA M
återkopplas
får man en finit
automat för
 $(L(M))^*$.

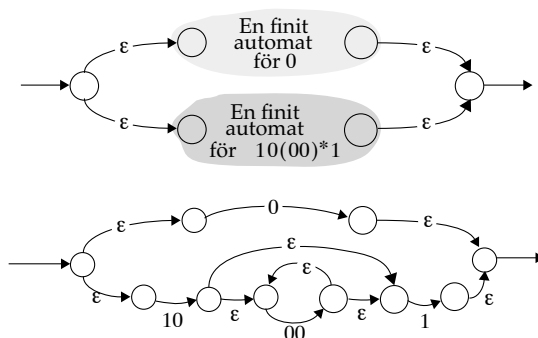


[†]. Endast starttillstånd och accepterande tillstånd är involverade i kopplingarna, varför eventuella övriga tillstånd inte ritas ut i dessa principskisser.

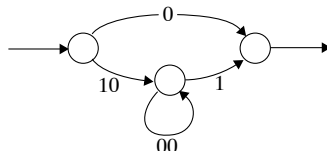
Ett induktionsbevis med formella brister.

BEVIS av **1**: De *minsta* reguljära språken är finita-automat-språk (se EXEMPEL 2.24 på sidan 33). Och eftersom varje *annat* reguljärt språk tillverkas av de minsta med ändligt många sammanfogningar, unioner eller Kleenestjärnor, så kan man med ändligt många seriekopplingar, parallellkopplingar och återkopplingar bygga en finit automat för varje reguljärt språk. \square

EXEMPEL 2.31 Här tillverkas en finit automat för $0 \cup 10(00)^*1$.



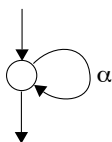
Efter viss förenkling erhålles



Innan vi tar itu med punkt **2** visar vi hur enkelt det kan vara att bilda reguljära uttryck för små finita automaters språk.

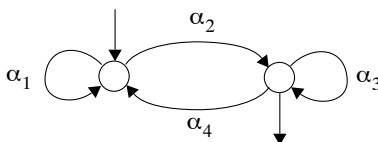
EXEMPEL 2.32

två små finita automater



deras språk

α^*



$\alpha_1^* \alpha_2 (\alpha_3 \cup \alpha_4 \alpha_1^* \alpha_2)^*$

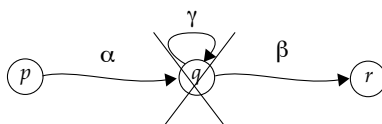
Se även EXEMPEL 2.35 på sidan 47.

Vårt bevis av punkt **2** baseras på en metod för successiv tillståndseliminering där tillstånd som inte är starttillstånd eller accepterandetillstånd avlägsnas, medan de kvarvarande tillstånden i gengäld belamras med allt utförligare notation, närmare bestämt reguljära uttryck.

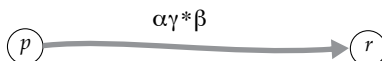
■ Tillståndseliminering och GFA:er

Låt q vara ett tillstånd i en viss finit automat, vilken kan vara deterministisk eller ickedeterministisk. Antag att q varken är starttillstånd eller accepterande tillstånd. Elimineras q samt varje till q inkommande båge från något tillstånd p , liksom varje ögla på q och utgående båge till något tillstånd r , så avlägsnas en möjlighet att av vissa strängar driva den aktuella automaten från p till r . En ersättningsbåge från p till r som i figuren återställer situationen, och är att betrakta blott som en sorts ihopdragning eller koncentrat av de just borttagna objekten.

före elimineringen



efter



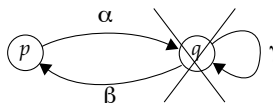
Innebörden i ersättningsbågens notationen $\alpha\gamma^*\beta$ är givetvis att automaten kan drivas från p till r av var och en av de oändligt många strängar som det reguljära uttrycket $\alpha\gamma^*\beta$ beskriver. Dvs ersättningsbågen avser egentligen oändligt många "vanliga" bågar.

GFA =
generaliserad
finit automat

Tillståndseliminering som ovan ger uppenbarligen upphov till en sorts generalisering i notationen av en finit automat. Generalisering i bemärkelsen att medan en gängse finit automat har en ändlig övergångsmängd (se definitionen på sid 36), så kan övergångsmängden nu vara *oändlig*. I fortsättningen använder vi beteckningen GFA för en finit automat med sådan övergångsnotation.

Innan eliminationsmetoden illustreras i ett exempel, uppmärksammas läsaren på att ersättningsbågen blir en *ögla* om tillstånden p och s sammanfaller, dvs om det eliminerade tillståndet står i dubbelriktad förbindelse med något tillstånd:

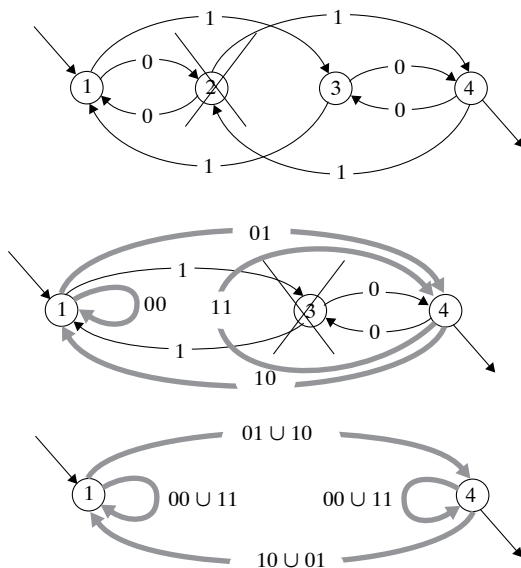
före elimineringen



efter



EXEMPEL 2.33 I nedanstående finita automat elimineras först tillståndet 2, sedan 3:

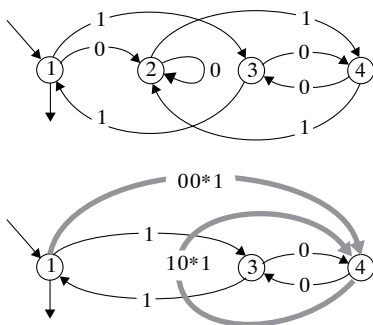


Språket som denna GFA – och därmed den ursprungliga DFA:n – accepterar är (jfr med EXEMPEL 2.32)

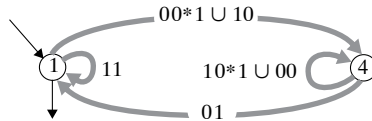
$$(00 \cup 11)^*(01 \cup 10)((00 \cup 11) \cup (10 \cup 01)(00 \cup 11)^*(01 \cup 10))^*$$

EXEMPEL 2.34 Vi eliminerar tillstånden 2, 3, 4 i nedanstående DFA.

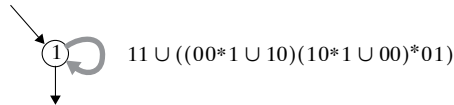
efter
elimine-
ringen av 2



efter
elimine-
ringen av 3



efter
elimine-
ringen av 4

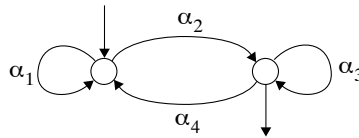


Språket för denna GFA – och därmed för den givna DFA:n – är

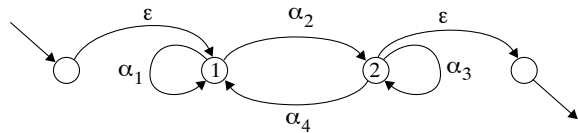
$$(11 \cup ((00^*1 \cup 10)(10^*1 \cup 00)^*01))^*.$$

EXEMPEL 2.35 Genom att "flytta över" start- och accepterandestatus till två nya tillstånd, kan även gamla start- och accepterandetillstånd elimineras.

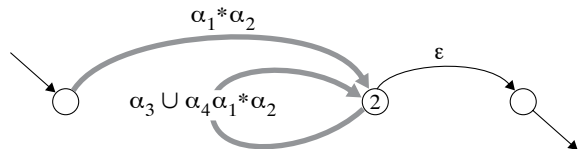
Varken starttillstånd eller accepterande tillstånd kan elimineras direkt.



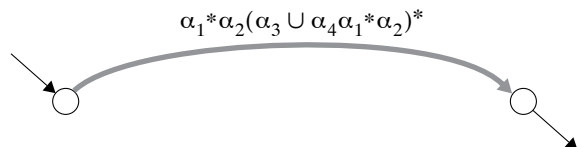
Efter insättning av ett nytt starttillstånd och ett nytt accepterande tillstånd kan de gamla (1 och 2) elimineras.



efter
elimineringen av 1



efter
elimineringen av 2

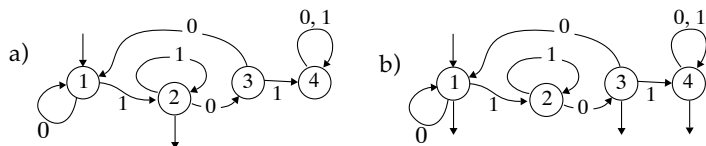


Detta visar att språket för den givna automaten är

$$\alpha_1^*\alpha_2(\alpha_3 \cup \alpha_4\alpha_1^*\alpha_2)^*.$$

BEVIS av **2**: En finit automat som saknar start- eller accepterandetillstånd accepterar det tomma språket, vilket är reguljärt. Och varje finit automat utrustad med start- och accepterandetillstånd kan, som i exemplet ovanför, transformeras till en GFA med blott två tillstånd – efter att start- och accepterandestatus har flyttats över till två nya tillstånd och alla gamla tillstånd har eliminerats. På bågen som sammanbinder det nya starttillståndet med det nya accepterandetillståndet finns därvid ett reguljärt uttryck som beskriver den givna automatens språk. \square

✂ **TEST 2.5** Skriv reguljära uttryck för de språk som ges av



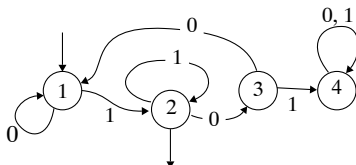
c) talen delbara med tre representerade binärt (Se TEST 2.4.b)).

■ 2.6 Några slutenhetsegenskaper

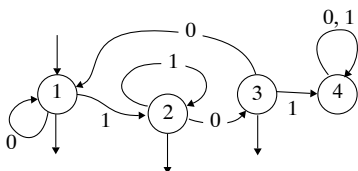
Eftersom de finita automaternas språk är just de reguljära språken, så kan vi använda resonemang om finita automater för att dra slutsatser om reguljära språk. Det här avsnittet är en utmärkt illustration till detta. Här visas hur finita automater för $Prefix(L)$, $Suffix(L)$, L^{rev} , \bar{L} och $L_1 \cap L_2$ kan konstrueras med hjälp av finita automater för L , L_1 , L_2 . Konstruktionerna ifråga bevisar att de förra språken är reguljära om de senare är det.

Låt L vara ett reguljärt språk. Då är $L = L(M)$ för någon DFA M , t ex som nedan.

En DFA M
och dess
språk L .

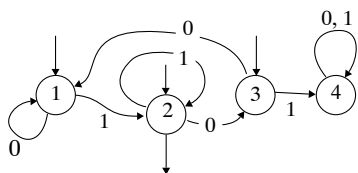


► En DFA för $\text{Prefix}(L)$:



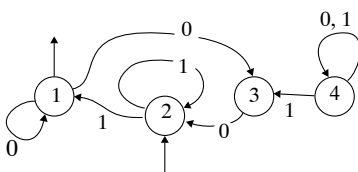
Ett prefix x till en sträng $w \in L$ driver M från starttillståndet längs en begynnande del av den väg som w driver M . Om man ger accepterandestatus till det sista tillståndet i prefixets väg kommer prefixet att accepteras. Genom att ge accepterandestatus till varje tillstånd på väg från starttillståndet till något accepterande tillstånd kommer varje prefix att accepteras och man har fått en DFA för $\text{Prefix}(L)$.

► En NFA för $\text{Suffix}(L)$:



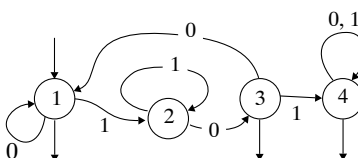
Om y är ett suffix till en sträng $w \in L$ så drivs M av y längs en avslutande del av w 's väg i M . Ger man startstatus till det första tillståndet i y 's väg accepteras y . Och genom att ge startstatus till varje tillstånd på väg från starttillståndet till något accepterande tillstånd, får man därför en NFA (med flera starttillstånd) för $\text{Suffix}(L)$.

► En NFA för L^{rev} :



Genom att kasta om riktningen på varje övergång (något som bl.a. förvandlar starttillstånd till accepterande tillstånd och vice versa), får man en NFA M^{rev} för L^{rev} .

► En DFA för \bar{L} :



Låt accepterande tillstånd och ickeaccepterande tillstånd i M byta roller. Resultatet – *komplementautomaten* till M – blir en DFA som inte accepterar någon enda sträng som M accepterar, men som accepterar varje annan sträng (över M 's alfabet förstås).

► En DFA för $L_1 \cap L_2$:

Av de Morgans regel (sid 267) $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ för mängder följer att man kan bygga en DFA för $L_1 \cap L_2$ med hjälp av DFA:er M_1 och

M_2 för L_1 och L_2 på följande sätt[†]:

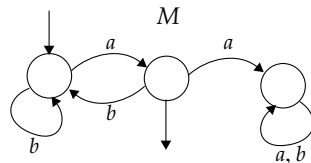
1. Bilda DFA:er för $\overline{L_1}$ och $\overline{L_2}$.
2. Bilda (genom parallellkoppling) en NFA för $\overline{L_1} \cup \overline{L_2}$.
3. Transformera (genom delmängdskonstruktionen) NFA:n i **2.** till en DFA.
4. Bilda komplementautomaten till den i **3.**

Dessa enkla resonemang bevisar följande sats.

✧ SATS 2.2 Om L_1 och L_2 är reguljära, så är även
 $Prefix(L_i)$, $Suffix(L_i)$, L_i^{rev} , $\overline{L_i}$ och $L_1 \cap L_2$ reguljära.

✧ TEST 2.6 Konstruera DFA:er för

- a) $Prefix(L(M))$ b) $Suffix(L(M))$
 c) $Prefix(L(M)) \cap Suffix(L(M))$ då M
 är som i figuren.



2.7 Minimering av en DFA

Följande optimeringsproblem bildar huvudtema för detta avsnitt.

Har en given DFA minimalt antal tillstånd? Om ej, hur kan man finna en DFA ekoivalent med den givna, men med minimalt antal tillstånd?

Minimering och särskiljande

Vi börjar med några enkla exempel, men först en definition.

DEFINITION



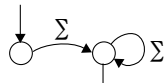
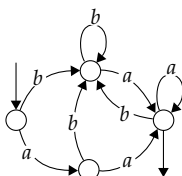
Strängar särskiljs av en DFA om de driver DFA:n till skilda tillstånd.

Av definitionen följer att en DFA alltid särskiljer en accepterad sträng

†. Det finns en genväg till "snittautomaten" som bygger på följande idé: Kör båda automaterna samtidigt, och acceptera strängar som bägge automaterna accepterar. Dvs parallellkoppla de två automaterna (ger normalt en NFA för $L_1 \cup L_2$), men strunta i accepterandestatus just nu. Transformera sedan (delmängdskonstruktionen!) NFA:n till en DFA, och sätt nu accepterandestatus på de delmängdstillstånd som innehåller accepterandetillstånd från bägge automaterna.

från en förkastad dito. Men det kan även inträffa – som i exemplet nedanför – att en DFA:er särskiljer två strängar fastän båda accepteras eller båda förkastas.

EXEMPEL 2.36 Är nedanstående DFA:er *minimala* för sina språk?

	M_1	M_2	M_3	M_4
Minimal DFA?				
dess språk	\emptyset	Σ^*	Σ^+	Σ^+a

Låt oss diskutera minimaliteten. En DFA för \emptyset skall förkasta alla strängar. Därför behöver en sådan DFA inte mer än *ett* tillstånd (och det skall vara ickeaccepterande). På motsvarande sätt klarar sig en DFA för Σ^* med ett enda tillstånd (vilket skall vara accepterande). Således är M_1 , M_2 (i figuren ovanför) minimala för sina språk.

En DFA för språket Σ^+ *måste* ha *minst två* tillstånd, ty den måste *särskilja* strängen ϵ – som ligger utanför språket – från alla strängar som tillhör språket. Eftersom den presenterade DFA:n M_3 för språket Σ^+ inte har flera tillstånd än just två, är den minimal.

M_4 kräver en utförligare diskussion. Man kan notera att M_4 använder sina fyra tillstånd till att särskilja t ex strängarna i $\{\epsilon, a, b, aa\}$. Men är det verkligen nödvändigt att särskilja alla fyra strängarna? Låt oss undersöka saken. En DFA för Σ^+a *skall* acceptera aa men inte de övriga tre. Således måste aa särskiljas från ϵ , a , b . Men måste strängarna i $\{\epsilon, a, b\}$ särskiljas?

M:s
determinism är
avgörande här.

Få se ..., antag att M är en DFA för Σ^+a som *inte* särskiljer ϵ , a . Dvs antag att M drivs av ϵ och a till ett gemensamt tillstånd. Då följer för varje z , att ϵz och az driver M till ett gemensamt tillstånd. Men det senare motsägs av att (för $z = a$)

$$\epsilon a = a \notin \Sigma^+a \quad \text{och} \quad aa \in \Sigma^+a \quad (7)$$

Alltså är det *nödvändigt* att M särskiljer paret ϵ , a .

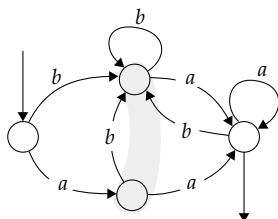
Paret ϵ , b måste också särskiljas. Ty annars skulle för varje z , de två strängarna ϵz , bz driva M till ett gemensamt tillstånd. Något som strider mot att (för $z = a$)

$$\epsilon a = a \notin \Sigma^+a \quad \text{och} \quad ba \in \Sigma^+a \quad (8)$$

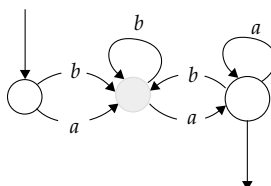
Paret a , b då? Med motsvarande resonemang som nyss: Om M vore en DFA för Σ^+a som *inte* särskiljer a , b så gäller för varje z , att az , bz driver M till ett gemensamt tillstånd – precis som nyss. Men nu finns

det inget motsägelsefullt i detta. Ty båda av az , bz tillhör Σ^+a om z slutar på a , och annars (om z inte slutar på a) gäller att ingen av az , bz tillhör Σ^+a . Därmed har vi inte funnit något skäl till att särskilja a , b . Är måhända M_4 – som särskiljer a , b – onödigt slösaktig med tillstånd? Ja faktiskt. De två tillstånd som M_4 använder för att särskilja a , b kan slås ihop till ett tillstånd utan att M_4 's språk påverkas. För

M_4 en slösaktig DFA för Σ^+a



En mindre DFA för samma språk



att klargöra skälen bakom ihopslagningen definierar vi ytterligare ett särskiljandebegrepp, samt bevisar ett lemma.

NOTERA:

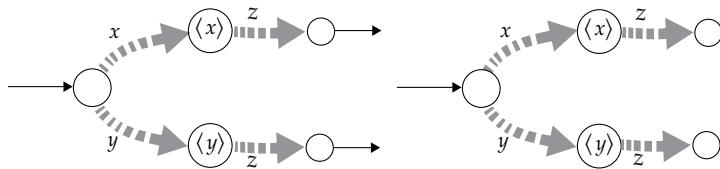
L särskiljer x , y med *tom* särskiljande sträng om exakt en av x , y tillhör L .

DEFINITION Två strängar x , y särskiljs av ett språk L om det finns någon (s.k. särskiljande) sträng z så att exakt en av strängarna xz , yz tillhör L . Och en mängd A av två eller flera strängar särskiljs av L , om för varje par x , $y \in A$, det gäller att x , y särskiljs av L .

I fortsättningen betecknar vi ibland ett tillstånd i en finit automat med $\langle x \rangle$ om strängen x driver automaten till $\langle x \rangle$.

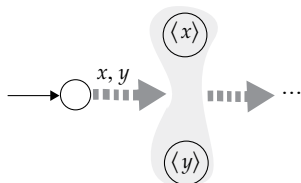
✧ lemma 2.1 Om M är en DFA och x , y särskiljs av M men inte av $L(M)$, så kan tillstånden $\langle x \rangle$, $\langle y \rangle$ slås ihop till ett tillstånd utan att M 's språk ändras.

BEVIS (av lemmat): Om x , y särskiljs av M , så drivs M till skilda tillstånd $\langle x \rangle$, $\langle y \rangle$. Och om x , y inte särskiljs av $L(M)$ gäller det att M för varje sträng z endera accepterar båda av xz , yz eller förkastar båda. Se figuren nedanför.



Om M matas med en godtycklig sträng z när M står i något av tillstånden $\langle x \rangle$, $\langle y \rangle$ så kommer således M att drivas till acceptans från båda tillstånden eller till ickeacceptans från båda.

Av detta följer, att om M håller på och läser en sträng av vilken den hittills har drivits till *ett* av tillstånden $\langle x \rangle$, $\langle y \rangle$, så skulle M kunna fortsätta från det *andra* utan att strängens acceptans eller ickeacceptans påverkas.



Tillstånden
kan slås ihop.

Därför kan $\langle x \rangle$, $\langle y \rangle$ slås ihop till *ett* tillstånd, utan att M 's språk ändras. \square

✂ **Anm.2.12** Om man slår ihop $\langle x \rangle$, $\langle y \rangle$ fastän x, y särskiljs av $L(M)$, så blir resultatet en DFA som inte är ekvivalent med M . Något som följande sats bevisar. Annorlunda uttryckt, om strängar särskiljs av en DFA:s språk, så *måste* strängarna ifråga också särskiljas av DFA:n:

✧ SATS 2.3 Om M är en DFA och A är en mängd av strängar så gäller
 A särskiljs av $L(M) \Rightarrow A$ särskiljs av M .

BEVIS: Antag att satsens påstående är falskt. Dvs att det finns två strängar, säg $x, y \in A$, som *inte* särskiljs av M (dvs så att M drivs till ett gemensamt tillstånd av strängarna ifråga). Då följer för varje z , att xz, yz driver M till ett gemensamt tillstånd (M 's determinism!) och därmed att M endera accepterar båda av xz, yz eller förkastar båda, något som motsägs av att x, y särskiljs av $L(M)$. \square

Med hjälp av SATS 2.3 är det lätt att bevisa ett samband mellan M 's tillståndsbehov och $L(M)$'s särskiljande:

✧ FÖLJDSATS 2.3

En DFA har minst N tillstånd om dess språk särskiljer N strängar.

BEVIS: Om DFA:n hade *färre* än N tillstånd skulle (minst) två av de N strängarna (som DFA:ns språk särskiljer) vara tvungna att driva DFA:n till ett och samma tillstånd. Men detta motsägs av SATS 2.3. \square

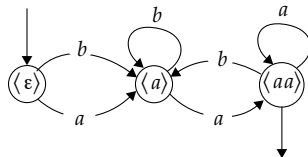
Aha, ett sätt
att bevisa att
en given DFA
är minimal.

Om en DFA M har (exakt) N tillstånd, och om N strängar särskiljs av $L(M)$, så följer att M är minimal. Ty då har *varje* DFA för $L(M)$ – enligt FÖLJDSATS 2.3 – (minst) N tillstånd. Vi formulerar detta enkla men viktiga konstaterande som en sats:

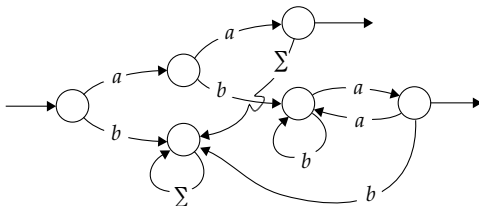
✧ SATS 2.4

En DFA med N tillstånd är minimal om dess språk särskiljer N strängar.

EXEMPEL 2.37 Vidstående DFA (en gammal bekant från EXEMPEL 2.36) är minimal. Ty de tre strängarna ϵ , a , aa (som DFA:n särskiljer med hjälp av sina tre tillstånd) särskiljs även av DFA:ns språk: paret ϵ , a särskiljs med hjälp av a , paret ϵ , aa med hjälp av ϵ , precis som paret a , aa .



EXEMPEL 2.38 DFA:n med sex tillstånd (se figuren) är minimal. De sex tillstånden används till att särskilja strängarna i $\{\epsilon, a, b, aa, ab, aba\}$.



Minimaliteten

Tabellen visar att det för varje par av $\epsilon, a, b, aa, ab, aba$ finns en särskiljande sträng z .

följer av att de sex strängarna även särskiljs (se tabellen) av automataens språk. M.a.o. att det för varje par x, y (av de sex strängarna) finns en särskiljande sträng z . Tabellen innehåller en strukturerad genomgång av alla $\binom{6}{2} = 15$ par av strängarna. Och för varje par ges ett exempel på en särskiljande sträng z .

a	a	$z = \epsilon$ för paret $(x, y) = (b, aa)$			
b	aa	a			
aa	ϵ	ϵ	ϵ		
ab	a	aaa	a	ϵ	
aba	ϵ	ϵ	ϵ	aa	ϵ
	ϵ	a	b	aa	ab

Omvändningen av den senaste satsen är också sann. Dvs en DFA med N tillstånd är minimal *endast* om dess språk särskiljer N strängar. Annorlunda uttryckt:

✧ **SATS 2.5** En DFA med N tillstånd är inte minimal om dess språk inte särskiljer N strängar.

BEVIS: Antag att en DFA M har N tillstånd, men att M :s språk *inte* särskiljer lika många strängar (N st). Av de N strängarna finns det då två stycken, säg x, y , som *inte* särskiljs av M :s språk. Därför kan man – enligt lemma 2.1 – slå ihop tillstånden $\langle x \rangle, \langle y \rangle$ till *ett* tillstånd, utan att M :s språk ändras. Dvs man kan bilda en DFA ekvivalent med M men med färre tillstånd, vilket bevisar att M *inte* är minimal. \square

■ Minimering genom särskiljning av tillstånd

I EXEMPEL 2.36 kunde vi minimera M_4 genom att slå ihop slösaktigt

använda tillstånd. Vi visar nu hur man kan utföra minimering genom att *först* slå ihop samtliga tillstånd till *en* tillståndsmängd, och *sedan* steg för steg sönderdela densamma igen. En poäng med denna metod är att den bygger en minimal DFA utan att "bry sig om" den givna DFA:ns tillståndsanvändning.

Innan vi fortsätter, uppmärksammas läsaren på att beteckningen som introducerades strax före lemma 2.1 leder till ett alternativt sätt att uttrycka "att strängar särskiljs av ett språk":

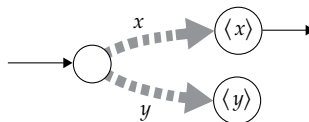
*Istället för att säga att strängarna x, y särskiljs av $L(M)$,
så säger vi att tillstånden $\langle x \rangle, \langle y \rangle$ i M särskiljs av $L(M)$.*

Ovanstående tillståndssärskiljande kan också beskrivas med rekursion:

Ibland är det bekvämare att säga att *tillstånd* särskiljs av $L(M)$.

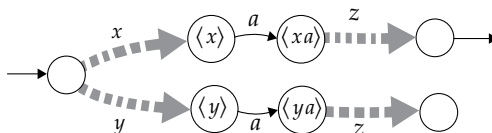
DEFINITION

Basfallet: Tillstånden $\langle x \rangle, \langle y \rangle$ särskiljs med tom sträng om exakt ett av de två är accepterande.



Rekursionssteget:

Tillstånden $\langle x \rangle, \langle y \rangle$ särskiljs med icke-tom sträng az om tillstånden $\langle xa \rangle, \langle ya \rangle$ särskiljs med hjälp av z .



Tillstånd som *icke* kan särskiljas (med någon enda sträng) är tillstånd som enligt lemma 2.1 kan slås ihop utan att DFA:ns språk ändras. Och särskiljbara tillstånd är sådana som *inte* kan slås ihop utan att DFA:ns språk ändras. (Se Anm.2.12 på sidan 53.) Att minimera en given DFA M kan därför sägas gå ut på att slå ihop tillstånd som inte är särskiljbara och hålla isär sådana som är särskiljbara. Om man inledningsvis slår ihop alla tillstånd, så återstår det "bara" att sönderdela den ihopslagna tillståndsmängden under iakttagande av att tillstånd skall hållas isär om de är särskiljbara. I den algoritm – som vi nu ska presentera – görs en sådan sönderdelning stegvis i bemärkelsen att tillstånd särskiljbara med kortare strängar behandlas före sådana som kräver särskiljning med längre strängar.

Om den aktuella DFA:n har enbart accepterande eller enbart icke-accepterande tillstånd så finns det förstås ingen anledning att sönderdela. Annars så börjar man med att dela upp tillståndsmängden i två delmängder med icke-accepterande tillstånd i den ena och accepterande i den andra. Därmed har man ordnat så att tillstånd särskiljbara med $z = \epsilon$ (sträng av längd noll!) hålls isär. Sedan söker man ef-

ter tillstånd som är särskiljbara med sträng av längd 1. Om sådana finns går man vidare och söker tillstånd särskiljbara med sträng av längd 2, osv Som värst kan särskiljning kräva särskiljande sträng av längd $N - 1$ där N är antalet tillstånd. (Kan Du förstå det?) Så proceduren har ett slut.

Sökandet efter tillstånd särskiljbara med sträng av längd $n + 1$ kan enligt rekursionssteget ovanför utföras på så sätt att man, för varje tecken a , betraktar varje par av tillstånd $\langle x \rangle, \langle y \rangle$ som *inte* är särskiljbara med sträng av längd n . Om $\langle xa \rangle, \langle ya \rangle$ för något tecken a är särskiljbara med z av längd n , så är $\langle x \rangle, \langle y \rangle$ särskiljbara med az (dvs med sträng av längd $n + 1$), och skall hållas isär. Med andra ord, den sönderdelning, säg $Q = Q_1 \cup \dots \cup Q_K$, som man har åstadkommit när tillstånden särskiljbara med strängar av längd n har hittats, kan användas i sökandet efter tillstånd särskiljbara med strängar av längd $n + 1$. Man behöver ju bara söka efter sådana tillstånd som för något tecken a har a -övergångar in i olika delmängder av $Q_1 \cup \dots \cup Q_K$!

De successiva sönderdelningarna kan med fördel presenteras som nivåer i ett träd. Se algoritmen nedanför samt EXEMPEL 2.39.

Stegvisa särskiljandealgoritmen

INPUT: En DFA M utan isolerade tillstånd.

OUTPUT: En minimal DFA som accepterar samma språk som M .

STEG 0 (nivå $n = 0$) Samla M 's alla tillstånd i *rotnoden*.

STEG 1 Bilda nivå $n + 1$ genom att splittra noder på nivå n som kan särskiljas med en sträng av längd n .

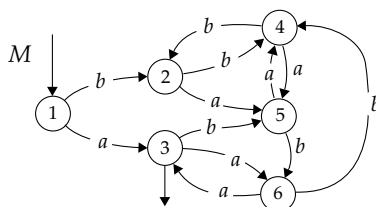
Då $n = 0$ kommer en sådan splittring (om den är genomförbar) att (på nivå 1) ge upphov till två noder, en med accepterande tillstånd och en med ickeaccepterande. Om $n \geq 1$ erhålles noderna på nivå $n + 1$ med hjälp av noderna på nivå n genom att tillstånd $\langle x \rangle$ och $\langle y \rangle$ från en gemensam nod på nivå n placeras i en gemensam nod på nivå $n + 1$ omm för varje tecken a det gäller att tillstånden $\langle xa \rangle$ och $\langle ya \rangle$ ligger i en gemensam nod på nivå n .

Om nivå $n + 1$ är identisk med nivå n avslutas STEG 1. (Detta inträffar förr eller senare. I värsta fall får man lika många nivåer som det finns tillstånd i M .)

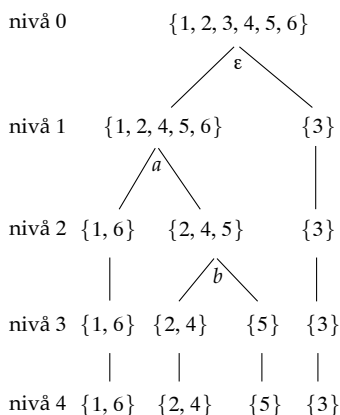
STEG 2 De av M 's tillstånd som ligger i en gemensam nod på den understa nivån slås ihop, varvid en minimal DFA ekvivalent med M erhålles.

EXEMPEL 2.39 Vi tillämpar stegvisa särskiljandealgoritmen på följande DFA.

En slösaktig
DFA



Sönderdelningar



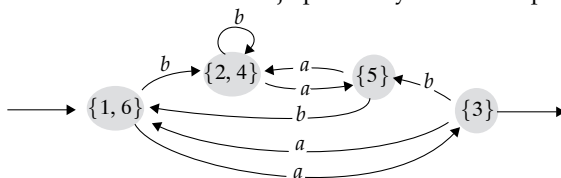
Kommentar

Tillstånden 1, 2, 4, 5, 6 är ickeaccepterande och 3 är accepterande.

Från {1, 6} drivs M av a in i {3}, men från {2, 4, 5} drivs M in i {1, 2, 4, 5, 6}.

Från {2, 4} drivs M av b in i {2, 4, 5} men från 5 in i {1, 6}.

En minimal DFA erhålles med hjälp av de fyra noderna på nivå 4.



■ Minimering genom reversering

Det finns en överaskande algoritm för tillståndsminimering som går ut på att två gånger reversera den DFA som man vill minimera. Algoritmen som är långsammare än den förra använder sig bl.a. av det faktum att M^{rev} accepterar $(L(M))^{rev}$ (se sid 49). Beviset av att algoritmen fungerar lämnas till den intresserade läsaren att själv konstruera.

Dubbla reverseringsalgoritmen

INPUT: En DFA M .

OUTPUT: En minimal DFA M'' ekvivalent med M .

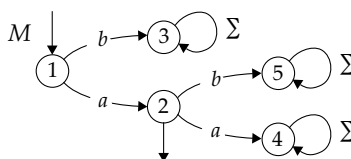
STEG 1 Bilda M^{rev} och transformera den sedan genom delmängdskonstruktionen till en DFA M' .

STEG 2 Bilda $(M')^{rev}$ och transformera den sedan genom delmängdskonstruktionen till en DFA M'' .

Efter STEG2 har man en DFA M'' för $(L(M)^{rev})^{rev}$, dvs för $L(M)$. Det märkliga är att av alla DFA:er för $L(M)$ så är M'' minimal. Förklaringen till detta mirakel står att finna i delmängdskonstruktionerna – som följer efter reverseringarna i de två stegen. De slår ihop sådana tillstånd som ursprungsautomaten M använder på ett slösaktigt sätt.

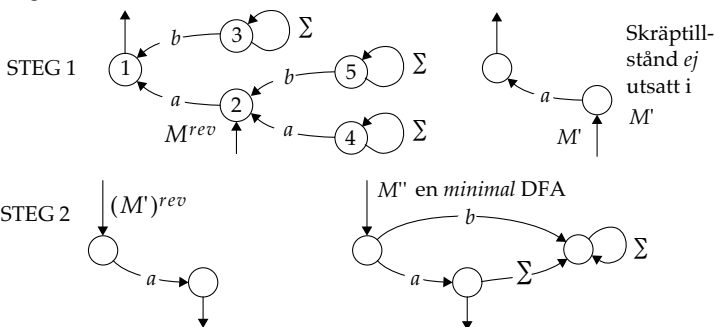
EXEMPEL 2.40 Låt oss tillämpa dubbla reverseringsalgoritmen på följande DFA.

En slösaktig
DFA

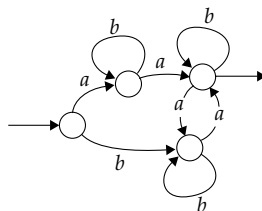


När M' bildas (med hjälp av delmängdskonstruktionen) i STEG1, så *försvinner* tillstånden 3, 4, 5, eftersom dessa tillstånd aldrig kan nås från starttillståndet i M^{rev} , vilket i sin tur beror på att tillstånden 3, 4, 5 är skräptillstånd i M , dvs att ingen sträng driver M till acceptans från något av dessa tillstånd. Notera också att vi inte har satt ut något skräptillstånd i M' , eftersom ett sådant tillstånd ändå skulle ha försvunnit i $(M')^{rev}$ (av samma skäl som att 3, 4, 5 försvann). När M'' bildas (med hjälp av delmängdskonstruktionen) i STEG2, så skapas *ett* skräptillstånd (istället för de ursprungliga tre). Här sker minimeringen!

Minimering
med dubbel
reversering.



✂ **TEST 2.7** Minimera vidstående DFA både med stegvisa särskiljandealgoritmen och med dubbla reverseringsalgoritmen.



2.8 De reguljära språkens gränser

Om man kan hitta en finit automat som accepterar ett givet språk L , eller om man kan hitta ett reguljärt uttryck som beskriver L , så vet man att L är reguljärt.

EXEMPEL 2.41 Addition av två naturliga tal kan tecknas på olika sätt och kan utföras i olika positionssystem. I 2-systemet tecknas vanligtvis addition som i marginalillustrationen invid. Bildar de sålunda tecknade korrekta additionerna ett reguljärt språk?

En korrekt och en felaktig addition

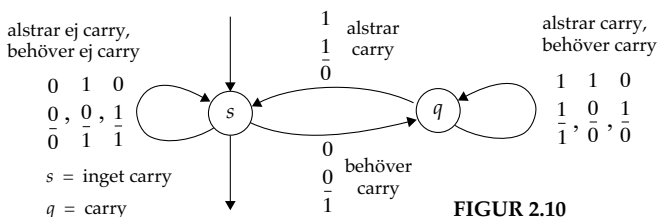
001110
010010
100000

001110
010010
110000

$\begin{matrix} a \\ b \\ c \end{matrix}$ där $a, b, c \in \{0, 1\}$

En mycket enkel DFA (FIGUR 2.10) kan känna igen sådana korrekta additionssträngar. Så dessa bildar ett reguljärt språk. Principen bakom DFA:ns konstruktion är att styra tillståndsovergångarna efter om tecknen (tripplarna) *alstrar* eller *behöver* minnessiffra (carry).

En DFA (utan skräptillstånd) som känner igen korrekta additionssträngar. (Automaten skall läsa en sträng från vänster.)



Särskiljandesatsen

Om man, trots idoga försök, inte kan hitta en passande finit automat eller ett reguljärt uttryck, hur gör man då ...? Ger upp? Nej, nej! Då försöker man bevisa att L inte är reguljärt. Följande sats kan vara behjälplig i en sådan situation.

✧ SATS 2.6 (Särskiljandesatsen)

Om oändligt många strängar särskiljs av L , så är L ickereguljärt.

BEVIS: Om oändligt många strängar särskiljs av L så måste – enligt SATS 2.3 – en DFA för L särskilja oändligt många strängar. Men en DFA kan bara särskilja så många strängar som den har tillstånd – ändligt många således. \square

EXEMPEL 2.42 $L = \{ (), (()), ((())), \dots \}$ är ett ickereguljärt språk. Ty varje par x, y av de oändligt många strängarna $(, (, ((, \dots$ särskiljs av L . Något som framgår av att om x och y består av m respektive n vänsterparenteser och $m \neq n$, så gäller det att $xz \in L$ men $yz \notin L$ för det z som består av m högerparenteser.

■ **Pumpsatsen för reguljära språk**

Särskiljandesatsen visar att ett reguljärt språk *inte* kan särskilja oändligt många strängar. Här följer nu en sats som på ett helt annat sätt uttrycker de reguljära språkens gränser.

✧ SATS 2.7 (Pumpsatsen) Antag att L är ett reguljärt språk som innehåller oändligt många strängar. Då finns det ett tal N sådant att om $w \in L$ är av längd minst N så innehåller w någon icketom delsträng y med följande egenskap:

y kan "pumpas ur" en gång och "pumpas upp" en eller flera gånger utan att w faller ur L , dvs w kan skrivas $w = xyz$ och

$$xz, xyz, xy^2z, xy^3z, \dots \in L$$

Närmare bestämt innehåller varje N -block[†] av w ett sådant y .

BEVIS: Av antagandet om L 's reguljäritet följer att det finns en DFA M som accepterar L . Likt varje DFA har M ändligt många tillstånd, säg N stycken.

Betrakta en godtycklig sträng $w \in L$ och ett godtyckligt N -block u av längd N . (Längden hos u garanterar att w är minst lika lång.) Vi ska nu visa att u innehåller en pumpdel y som omnämns i satsen.

Villkoret $w \in L$ innebär att M drivs av w till något accepterande tillstånd. Och villkoret $|u| = N$ innebär att M under konsumtion av delsträngen u drivs att besöka något tillstånd *minst två gånger*[‡]. Säg att q är ett sådant tillstånd, och låt y vara den del av u som konsumeras mellan det första och det andra besöket i q . Då är $y \neq \varepsilon$. (Ty en DFA måste ju konsumera *något* vid varje tillståndsövergång.) Vidare måste M under konsumtionen av y gå runt i någon slinga som börjar och

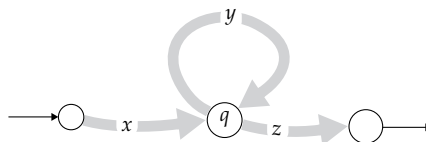
†. delsträng av längd N .

‡. Detta förklaras av att M som är en DFA (och därför konsumerar exakt ett tecken i taget vid tillståndsövergångar) gör $N + 1$ tillståndsbesök vid N tillståndsövergångar.

Existensen av w garanteras av att L är ett oändligt språk.

slutar i q . (Se FIGUR 2.11.) Såvida inte denna slinga börjar redan i starttillståndet, är M tvungen att konsumera någon "förrätt" x före inträdet i slingan. Likaså kan M eventuellt vara tvungen att konsumera någon "efterrätt" z efter att ha fullgjort ett varv i slingan, för att därigenom kunna nå ett accepterande tillstånd eller helt enkelt för att w är så lång att M måste gå runt flera varv i slingan. (I det senare fallet ingår en eller flera huvudrätter i efterrätten.)

M drivs att besöka något tillstånd två gånger.



FIGUR 2.11

Notera att x och z kan vara tomma, men *inte* y .

Således kan w skrivas som en sammansättning av tre strängar xyz , där y konsumeras *under ett första varv* i slingan, medan x och z konsumeras före respektive efter nämnda varv.

Upp-pumpning

Om vi nu kör M på någon av strängarna $xy^2z, xy^3z, xy^4z, \dots$ kommer M – p.g.a. sin determinism – att drivas runt i slingan *ytterligare* en, två, tre, ... gånger, för att i slutänden drivas till samma accepterande tillstånd som vid körning på $w = xyz$. Och vid körning på strängen xz kommer M att drivas till det accepterande tillståndet direkt efter första besöket i q .

Urpumpning

Dvs M accepterar $xy^n z$ för varje $n \in \mathbb{N}$. □

EXEMPEL 2.43 Låt L vara det oändliga reguljära språk som ges av $a(ab)^*b$, och betrakta $w = aabb$ som tillhör L . Om w 's ab -förekomst pumpas får man nya strängar i L , eller hur! Notera också att varje sträng i L förutom den kortaste kan pumpas på samma sätt. (Varför kan den kortaste strängen inte pumpas?) Närmare bestämt kan varje sträng i L av längd minst 4 pumpas inuti varje delsträng u av längd 4. (Inse det senare!). För $a(ab)^*b$ är således $N = 4$ ett N som i pumpsatsen. Rita själv en DFA för $a(ab)^*b$ och jämför med FIGUR 2.11.

☞ **Anm. 2.13** Pumpsatsen utställer för varje oändligt reguljärt språk ett *löfte*. Nämligen att det till språket ifråga hör ett tal N sådant att varje sträng i språket (av längd minst N) kan pumpas någonstans i varje N -block u . Ett oändligt språk som inte uppfyller nämnda löfte kan omöjligt vara reguljärt. Se exemplen nedanför.

EXEMPEL 2.44 I EXEMPEL 2.42 bevisade vi med hjälp av särskiljandesatsen att språket $L = \{ (, ((, (((, \dots \}$ inte är reguljärt. Vi bevisar nu samma sak med hjälp av pumpsatsen.

$((\dots (\quad) \dots))$
 N st N st

Om L vore reguljärt skulle det finnas N så att t ex strängen med N vänsterparenteser följda av lika många högerparenteser skulle kunna pumpas någonstans i blocket med de N första vänsterparenteserna.

Men sådan pumpning skulle ju resultera i en sträng med olika antal vänsterparenteser och högerparenteser, dvs pumpningen skulle leda ut ur L , vilket motsäger det som pumpsatsen lovar för reguljära språk. Alltså är L inte reguljärt.

EXEMPEL 2.45 $L = \{1^2, 1^3, 1^5, \dots\}$ (primtalsmängden representerad i 1-systemet) är inte reguljärt. Ty om så vore, skulle det för varje $w \in L$ som är tillräckligt lång finnas något $y \neq \epsilon$ så att $w = xyz$ och så att $xy^n z \in L$ för varje $n \in \mathbb{N}$. (Allt enligt pumpsatsen.) Eftersom w består av enbart 1:or är innebörden av ovanstående att det skulle finnas något $j \geq 1$ så att $w = 1^i 1^j 1^k$ och så att $1^i 1^{j \cdot n} 1^k \in L$ för varje $n \in \mathbb{N}$. Det senare betyder att $i + k + j \cdot n$ är prima för varje $n \in \mathbb{N}$. Eftersom det minsta primtalet är 2 följer att $i + k \geq 2$. Om m betecknar talet $i + k$, gäller således

$$\begin{cases} m \geq 2 \\ m + nj \text{ är prima för varje } n \in \mathbb{N} \end{cases}$$

Här kommer
 $j \geq 1$, $i + k \geq 2$
in i bilden.

Men å andra sidan är $m + nj$ inte prima för $n = m$, eller hur! Av denna motsägelse tvingas vi förkasta antagandet som ledde oss fram till motsägelsen – den om L 's reguljäritet.

☞ **Anm. 2.14** Pumpsatsen kan inte vändas om, dvs L 's reguljäritet är ett tillräckligt men inte nödvändigt villkor för pumpning. Här kommer ett s.k. motexempel mot omvändningen.

EXEMPEL 2.46 Betrakta $L = L_1 \cup L_2 \cup L_3$, över $\Sigma = \{a, b, c\}$ där

$$L_1 = \{w \in \Sigma^* \mid w \text{ har lika många förekomster av } a \text{ som av } b\}$$

$$L_2 = \{w \in \Sigma^* \mid w \text{ har någon förekomst av } aa\}$$

$$L_3 = \{w \in \Sigma^* \mid w \text{ har någon förekomst av } bb\}$$

L är inte reguljärt. Ändå kan varje sträng w i L pumpas som är av längd minst 3: Om w har någon c -förekomst, fungerar den senare som pumpdel. Och annars så kan endera ab , ba , a eller b väljas som pumpdel. Någon av ab , ba om w är av typ $(ab)^n$ eller $(ba)^n$ för något $n \geq 2$. Och annars a eller b . Ty om w inte är av typ $(ab)^n$ eller $(ba)^n$, så innehåller w någon av strängarna aaa , bba , abb , bbb , aab , baa varav de första tre tillåter att a väljs som pumpdel, och de senare tre att b väljs. Bristen på reguljäritet hos L bevisas t ex av att L särskiljer de oändligt många strängarna i $\{ac, acac, acacac, \dots\}$. Se ÖVNING 2.12 på sidan 70.

■ **Reguljära språk och periodiska mönster** Vissa enkla mönster av periodisk karaktär kan finita automater känna igen. T ex mönstret

i strängarna 1, 10, 100, ... som består i att tecknet 0 repeteras. "Primtal" och "nästlade parenteser" har emellertid en typ av mönster som finita automater inte kan känna igen.

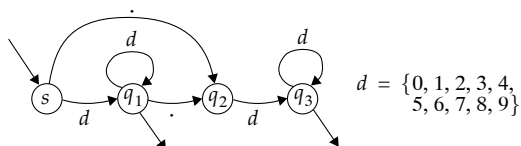
I pumpsatsen finns en naturlig förklaring till den här begränsningen hos de finita automaternas förmåga:

Först måste man konstatera – och detta har inget med pumpsatsen att göra – att en finit automat utan återkopplingar (slingor) bara kan känna igen ett *ändligt* antal strängar (hur många?).

Därför finns det bara ett sätt om man vill höja automatens förmåga så att den kan känna igen *oändligt* många strängar: Att bygga in en eller flera slingor. Men den kompetenshöjning som därvid uppstår blir begränsad till en förmåga att känna igen strängar med sådana periodiska mönster som uppstår när automaten drivs runt i en eller flera slingor. Det är denna begränsning som uttrycks i pumpsatsen.

■ **Reguljära språk och minne** Begränsningen i de reguljära språkens förmåga att uttrycka mönster kan också förklaras av att finita automatens minne är *ändligt*.

Betrakta t ex den decimaltalsigenkännande automaten från sid 32 igen. När den befinner sig i tillstånd q_3 "minns" den att den har konsumerat en av följande rätter:



1. en decimalpunkt följt av minst en siffra

2. minst en siffra följt av en decimalpunkt följt av ytterligare minst en siffra.

Däremot har den inget minne av *hur många siffror* den har konsumerat. (På motsvarande sätt som Du nog kan minnas att Du har ätit potatisar under året – men *inte hur många*.)

Till varje tillstånd hör en motsvarande ändlig "kom-ihåg-lista". Och eftersom den finita automatens tillståndsmängd är ändlig, är dess minne ändligt.

Varje finit automat utrustad med en *slinga* på väg från start mot något accepterande tillstånd kan, precis som automaten i figuren ovan, acceptera *hur långa strängar som helst*, genom att gå runt i slingan tillräckligt många gånger. Men *hur många* varv som den har gjort i slingan minns den inte. Ty den har ingen mekanism för sådan bokföring.

Ingen finit automat kan således känna igen mönster som kräver att den måste komma ihåg godtyckligt många redan konsumerade tecken. Parentesmönstret i EXEMPEL 2.42 är ett sådant. (För att kunna avgöra om en godtyckligt lång sträng av vänsterparenteser åtföljs av lika många högerparenteser krävs det någon mekanism för att bokföra godtyckligt många redan konsumerade vänsterparenteser.)

■ **Finita automater och lexikal analys m.m.** En kompilator eller en programtolk i färd med att behandla ett givet program behöver i ett första skede gå igenom programkoden för att upptäcka eventuella "reserverade ord" som `tex` `SUB` eller `main`. Sådan mönsterigenkänning – lexikal analys – gör kompilatorn eller tolken medhjälp av en finit automat.

För övrigt används finita automater flitigt i olika textbehandlingsprogram (editorer och ordbehandlare).

Ett annat användningsområde finner man bland mönsterigenkännande maskiner som nöjer sig med ett litet minne, men som har någon form av agerande i samband med mönsterigenkänningen. Myntväxlare, kaffeautomater (se 2.16 på sidan 71), tvättmaskiner och hissar är sådana exempel.

✂ **TEST 2.8** Reguljära? I a) – e) antas alfabetet vara $\{a, b\}$.

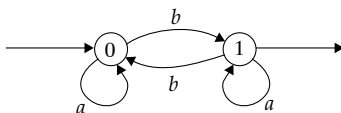
- a) $L_1 = \{w \mid w \text{ har jämnt antal av varje tecken}\},$
- b) $L_2 = \{w \mid w \text{ har lika antal av varje tecken}\},$
- c) $L_3 = \{w \mid \text{varje prefix av } w \text{ som har jämn längd tillhör } L_2\},$
- d) $L_4 = \{xyx^{rev} \mid x, y \in \Sigma^+\}$ e) $L_5 = \{yxx^{rev} \mid x, y \in \Sigma^+\},$
- f) $L_6 = \{1^2, 1^3, 1^5, 1^7, \dots\}^*,$ där 2, 3, 5, 7, ... avser primtalen.

■ 2.9 Finita automater med output

Har de hittills betraktade finita automaterna haft någon form av output?

Ja, men bara i en mycket begränsad mening. De har ju varit specialiserade på att acceptera eller förkasta sina inputsträngar. Dvs de returnerar en sorts binär outputsignal i form av "accepterar" (markerad 1 i figuren nedan) eller "accepterar ej" (markerad 0 i figuren).

binärt output



input abba
output 00100

I varje läge av strängkonsumtionen, dvs efter noll eller flera konsumerade tecken (oavsett om hela inputsträngen är konsumerad eller ej), så har den finita automaten kunnat "rapportera" huruvida hittills konsumerad sträng var till belåtenhet eller ej. (Om och endast om nuvarande tillstånd är accepterande så är hittills konsumerad sträng accepterad.)

Nu skall vi komplettera finita automater med "outputfunktioner" kompetenta att lämna outputsignaler formulerade i godtyckligt alfabet (inte bara i det binära). Och vi skall göra detta på två sätt. Automaterna ifråga kallas Mooremaskiner respektive Mealymaskiner.

Mooremaskiner

En Mooremaskin är en vanlig DFA utrustad med en funktion som lämnar output vid varje tillståndsbesök (även i starttillståndet).

DEFINITION En Mooremaskin M är en sextupel

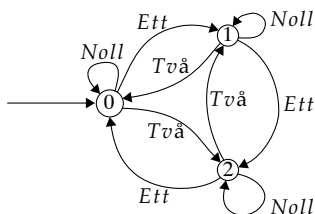
$$(Q, \Sigma_{in}, \Sigma_{ut}, \delta, \lambda, s)$$

där

- (i) Q är tillståndsmängden
- (ii) Σ_{in} är inputalfabetet
- (iii) Σ_{ut} är outputalfabetet
- (iv) δ är övergångsfunktionen från $Q \times \Sigma_{in}$ till Q
- (v) λ är outputfunktionen från Q till Σ_{ut}
- (vi) $s \in Q$ är starttillståndet

M beräknar en funktion från Σ_{in}^* till Σ_{ut}^* genom att returnera $\lambda(s)\lambda(q_1)\lambda(q_2)\dots\lambda(q_n) \in \Sigma_{ut}^*$ för en inputsträng $\sigma_1\sigma_2\dots\sigma_n \in \Sigma_{in}^*$ som driver M från s till q_1 till q_2 ... till q_n .

EXEMPEL 2.47 En Mooremaskin som returnerar resterna när ett tal x divideras med 3. Beräkningen utföres i 10-systemet. Se även EXEMPEL 2.18 på sidan 28.



$Noll = \{0, 3, 6, 9\}$

$Ett = \{1, 4, 7\}$

$Två = \{2, 5, 8\}$

input	53701
output	022001

Det output som levereras vid tillståndsbesök utgör resten $Rest(x, 3)$ för hittills konsumerad inputsträng x . Exempelvis erhålles

input	53701
output	022001

som bl.a. visar att $Rest(53, 3) = 2$ och att $Rest(537, 3) = 0$.

Mealymaskiner

En Mealymaskin är en vanlig DFA kompletterad med en funktion som lämnar output vid varje tillståndsovergång.

DEFINITION En Mealymaskin M är en sextupel

$$(Q, \Sigma_{in}, \Sigma_{ut}, \delta, \lambda, s)$$

där

- (i) Q är tillståndsmängden
- (ii) Σ_{in} är inputalfabetet
- (iii) Σ_{ut} är outputalfabetet
- (iv) δ är övergångsfunktionen från $Q \times \Sigma_{in}$ till Q
- (v) λ är outputfunktionen från $Q \times \Sigma_{in}$ till Σ_{ut}
- (vi) $s \in Q$ är starttillståndet

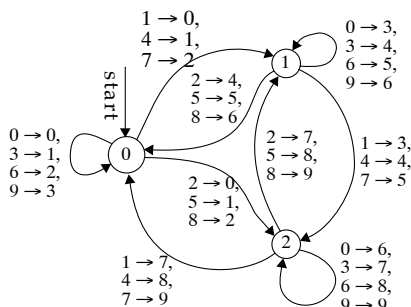
M beräknar en funktion från Σ_{in}^* till Σ_{ut}^* genom att returnera

$\lambda(s, \sigma_1)\lambda(q_1, \sigma_2)\dots\lambda(q_{n-1}, \sigma_n) \in \Sigma_{ut}^*$ för en inputsträng

$\sigma_1\sigma_2\dots\sigma_n \in \Sigma_{in}^*$ som driver M från s till q_1 till $q_2 \dots$ till q_{n-1} till q_n .

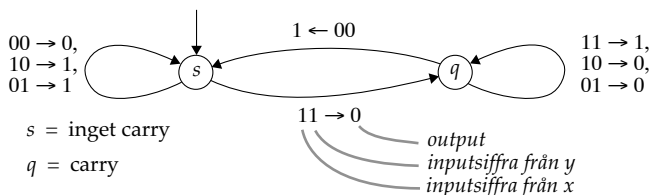
⌘ **Anm.2.15** Notera att för en Mealymaskin är outputsträngen lika lång som inputsträngen, medan för en Mooremaskin outputsträngen är ett tecken längre än inputsträngen. Och Mealymaskinen ger (till skillnad från Mooremaskinen) tomt output vid tomt input.

EXEMPEL 2.48 Redan i EXEMPEL 2.17 mötte Du en Mealymaskin (sid 28) som returnerar kvoterna när tal divideras med 3. Här är den igen:



EXEMPEL 2.49 En Mealymaskin som returnerar summan av två tal x och y . Beräkningen utföres i binära systemet. Ett par siffror åt gången, en från vardera talet, skall läsas in. Och precis som vid sedvanlig additionsupställningskalkyl börjar man plocka talens siffror från höger ände.

Jämför med
DFA:n i
EXEMPEL 2.41
på sidan 59.



Text utföres additionen

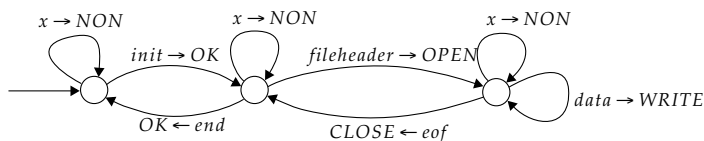
```

      111001    x
    +101101    y
    -----
    1100110  output
  
```

som i tabllån nedanför genom att ett par siffror, en från vardera x och y , konsumeras åt gången. Observera att tabllån skall läsas från höger.

steg	6	5	4	3	2	1	0
x	0	1	1	1	0	0	1
y	0	1	0	1	1	0	1
input	00	11	10	11	01	00	11
tillstånd	q	q	q	s	s	q	s
övergång	00 \rightarrow 1	11 \rightarrow 1	10 \rightarrow 0	11 \rightarrow 0	01 \rightarrow 1	00 \rightarrow 1	11 \rightarrow 0
nästa tillstånd	s	q	q	q	s	s	q
output	1	1	0	0	1	1	0

KERMIT, en Mealy-maskin Ett protokoll är ett program som ombesörjer nätverkskommunikation – textfilöverföring – mellan två datorer. Ett av de mest kända är KERMIT[†]. Här följer en grovskiss av dess mottagarkomponent. Som synes är KERMIT en DFA med output i tillståndsövergångarna, dvs en Mealy-maskin. (För att inte göra bilden onödigt klottrig har skräptillståndet utelämnats.)



input	output
x = önskat tecken	NON = "nonchalerar mottaget tecken"
init = starttecken för överföring	OK = "godkänner mottaget tecken"
end = sluttecken	
fileheader = tecken som inleder en fil	OPEN = "öppnar en fil"
data = tecken i en fil	WRITE = "skriver mottaget data"
eof = tecken som avslutar en fil	CLOSE = "stänger filen"

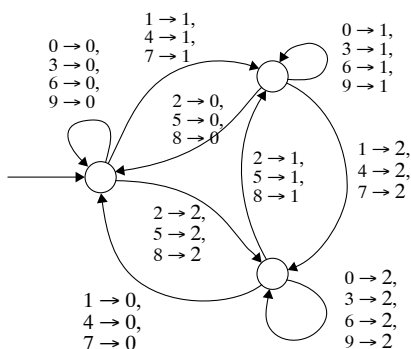
†. Protokollat KERMIT är utvecklad vid Columbia University.

- **För varje Mooremaskin finns det en Mealymaskin och omvänt**
 Givet en Mooremaskin, så finns det en Mealymaskin som beräknar samma funktion som Mooremaskinen om vi bortser från Mooremaskinens output vid start (vilket saknar motsvarighet hos Mealymaskinen). Mealymaskinens output bestäms av Mooremaskinens dito på följande sätt

$$\lambda_{Mealy}(q, \sigma) = \lambda_{Moore}(\delta(q, \sigma))$$

För övrigt är Mealymaskinen identisk med Mooremaskinen.

Text skulle nedanstående Mealymaskin motsvara Mooremaskinen i EXEMPEL 2.47.



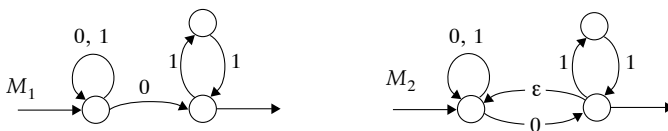
Omvänt, givet en Mealymaskin så kan man också konstruera (dock inte lika enkelt som i föregående fall) en Mooremaskin (med flera tillstånd än Mealymaskinen) som beräknar samma funktion.

- ✂ **TEST 2.9** a) Rita en Mooremaskin som returnerar resterna representerade i 10-systemet vid division med 2.
 b) Rita en Mealymaskin som returnerar kvoterna representerade i 10-systemet vid division med 2.

2.10 Övningar

2.1 Skriv ett reguljärt uttryck för det språk som accepteras av den decimaltalsigenkännande NFA:n på sid 63.

2.2 Betrakta de två NFA:erna M_1 och M_2 nedanför.



Konstruera

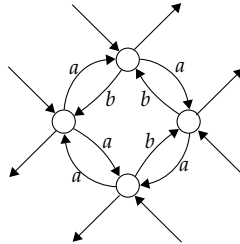
- a) reguljära uttryck för $L(M_1)$ och $L(M_2)$, b) DFA:er ekvivalenta med M_1 och M_2 ,
c) minimala DFA:er för $L(M_1) \cup L(M_2)$ och $(L(M_2))^*$.

2.3 Betrakta språket

$L = \{w \in \{a, b\}^* \mid ab \text{ förekommer som delsträng i } w \text{ och } |w| \text{ är jämn}\}$

- a) Ge ett reguljärt uttryck för L .
b) Konstruera en minimal DFA för L .

2.4 Transformera, med hjälp av delmängdskonstruktionen, följande NFA till en DFA. Hur många tillstånd får den senare?

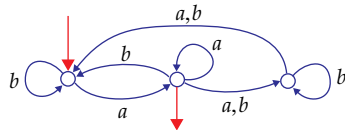


2.5 Konstruera minimala DFA:er för de språk över $\{a, b\}$ vars strängar

- a) innehåller *baab* b) inte innehåller *abba*
c) innehåller *baab* men inte *abba*
d) inleds med ett jämnt antal a :n vilka följs av ett jämnt antal b :n som i sin tur följs av ett jämnt antal a :n.

2.6 Konstruera för vidståendes NFA

- a) en minimal DFA,
b) ett reguljärt uttryck.



2.7 Konstruera en *minimal* DFA för de bitsträngar som beskrivs av $1*0(10*1 \cup 01*0)^*10*$. Glöm inte bort att motivera minimalitet.

2.8 Är följande språk över $\{a, b\}$ reguljära? Presentera en finit automat för språket om det är reguljärt. Bevisa annars med hjälp av pumpsatsen eller särskiljandesatsen att språket inte är reguljärt.

- a) $L_1 = \{w \mid w \text{ är en palindrom}\}$,

- b) $L_2 = \{w \mid w \text{ är inte en palindrom}\}$,
 c) $L_3 = \{uabbav \mid |u| = |v| = 2\}$, d) $L_4 = \{uabbav \mid |u| = |v|\}$,
 e) $L_5 = \{uvabba \mid |u| = |v|\}$, f) $L_6 = \{uvw \mid |u| = |v| = |w|\}$.

2.9 Vilka av följande språk över $\{a, b, c\}$ är reguljära?

- a) $L_1 = \{a^m b^n c^k \mid 0 \leq m \leq n \leq k\}$,
 b) $L_2 = \{xa^n y b^n z c^3 \mid x, y, z \in \{a, b\}^* \text{ och } n \geq 1\}$, c) $L_3 = L_1 \cap L_2$.

2.10 Visa att mängden av reguljära uttryck över säg alfabetet $\{a, b\}$ inte bildar ett reguljärt språk.

2.11 Ge exempel på två språk L_1, L_2 över $\{a, b\}$ sådana att

- a) L_1 är reguljär, L_2 är icke-reguljär, och $L_1 \cup L_2$ är reguljär.
 b) L_1 är reguljär, L_2 är icke-reguljär, $L_1 \cap L_2$ är oändlig och reguljär.

2.12 Visa att $\{ac, acac, acacac, \dots\}$ särskiljs av språket L i EXEMPEL 2.46.

2.13 Visa att språket L i EXEMPEL 2.46 blir reguljärt om dess alfabet ändras till $\Sigma = \{a, b\}$.

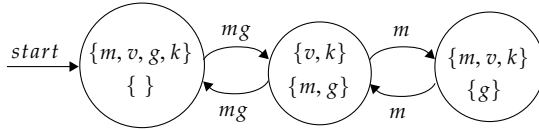
2.14 Sant eller falskt? Motivera noga.

- a) Varje delmängd av ett reguljärt språk är reguljärt.
 b) Varje oändlig union $\bigcup_{n=1}^{\infty} L_n$ av reguljära språk L_n är reguljär.
 c) Ingen oändlig union $\bigcup_{n=1}^{\infty} L_n$ av reguljära språk L_n är reguljär.

2.15 En man, en varg, en get och ett kálhuvud Har Du hört den förut? Historien om en stackars man som skulle ta sig över en flod med tre ägodelar: ett kálhuvud, en get som inte kunde motstå kál och en varg som álskade getter. Fáerden över floden skulle ske i en båt som endast rymde honom sjálv och *en* av de tre ägodelarna, varför mannen måste göra flera fárdar fram och tillbaka över floden, och vid varje fárd lámnar två av sina ägodelar kvar på stranden. Men lámnades vargen och geten ensamma på samma sida av floden skulle vargen utan tvivel áta upp geten. Och likaså skulle säkert geten sätta i sig kálhuvudet, om den lámnades ensam med sådant godis ...

Här nedan skisseras början av en finit automat som beskriver man-

nens färder över floden. Automaten tillstånd representerar "tillstånd" på ömse sidor om floden, dvs olika tudelningar av mängden $\{man, varg, get, kållhuvud\}$. Och tillståndsovergångarna är märkta med båtens innehåll.



Hur ser en korrekt fortsättning ut av automaten?

2.16 I en kaffeautomat kostar en mugg kaffe 4 kr. Automaten kan ta emot mynt av följande slag 50 öre (0.50 kr), 1 kr, 5 kr. När 4 kr har matats in returnerar automaten kaffet (utan att man behöver trycka på någon extra knapp). Om mer än 4 kr matats in (t ex 5 kr eller 1 kr + 5 kr eller ...) så returnerar automaten kaffe + överskjutande belopp. Beskriv kaffeautomatens funktion med

a) en Mealymaskin

b) en Mooremaskin

2.17 Skissera ett program (t ex i C) som implementerar tillståndsovergångsgrafen på sid 28. Programmet skall kunna läsa in ett godtyckligt långt tal och beräkna dess kvot och rest (vid division med 3).

Sammanhangsfria språk och pushdown-automater

Grammatikmetoden	72
Sammanhangsfria grammatiker	75
Produktionsträd	80
Pushdownautomater (PDA:er)	82
För varje CFG finns det en PDA	87
För varje PDA finns det en CFG	90
De sammanhangsfria språkens gränser.....	93
Pumpsatsen för sammanhangsfria språk.....	94
Deterministiska pushdownautomater.....	98
Det finns inte en DPDA för varje CFG.....	101
Övningar	102

De finita automaternas tillkortakommanden har att göra med deras *ändliga* tillståndsmängder. (Se sid 63.) Istället för att introducera automater med oändliga tillståndsmängder – vilka svårigen kan förverkligas på annat sätt än som skrivbordsprodukter – presenterade A. G. Oettinger[†] år 1961 idén om en automat med ändlig tillståndsmängd men med ett obegränsat ”yttre” minne i form av en s.k. *stack*. Innan vi ger oss i kast med Oettingers s.k. *stackautomater* (eller *pushdownautomater* som de vanligtvis kallas), skall vi ännu en gång betrakta några finita automater, närmare bestämt deras språk, men nu anlägga ett *grammatiskt* synsätt som så småningom kommer att leda oss till pushdownautomaternas språk.

3.1 Grammatikmetoden

Vad gäller reguljära språk har vi lärt oss hur de kan beskrivas av reguljära uttryck och av finita automater. I exemplet som följer introduceras en annan metod för språkbeskrivning – *grammatikmetoden*.

EXEMPEL 3.1 Nedanstående tre *regler* beskriver syntaxen hos strängarna i språket $\{1, 10, 100, \dots\}$, och bildar därmed en s.k. *grammatik*[‡] för språket.

†. Proc. Symposia in Applied Math. 12, American Mathematical Society, Providence, R.I.

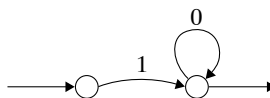
‡. Grammatik = språkregler (Svenska Akademiens ordlista)

Grammatikreg-
ler samt en finit
automat för
språket
{1, 10, 100, ...}

$$S \rightarrow 1A$$

$$A \rightarrow \varepsilon$$

$$A \rightarrow 0A$$



Den första regeln uttrycker att varje tiopotenssträng S börjar på 1 och har ett suffix A . De övriga två reglerna säger att A endera är en tom sträng eller en sträng som börjar med tecknet 0 och fortsätter med någon (kortare) sträng av samma slag (som A).

☞ Anm.3.1 Av sparsamhetsskäl introducerar vi en notation som tillåter flera produktionsregler att samsas på en rad:

Utläses
"A kan producera
 ε eller $0A$ ".

$$A \rightarrow \varepsilon | 0A$$

Strängproduktion Bruket av pilsymbolen " \Rightarrow " hänger samman med att reglerna inte bara beskriver, utan också *producerar* språkets strängar. Nedan visar vi hur det går till att producera strängen 100 med reglerna i EXEMPEL 3.1.

Sättet att
använda två
typer av pilar
kommenteras på
sid 74.

<i>produktion</i>	<i>med hjälp av regeln</i>
$S \Rightarrow 1A$	$S \rightarrow 1A$
$\Rightarrow 10A$	$A \rightarrow 0A$
$\Rightarrow 100A$	$A \rightarrow 0A$
$\Rightarrow 100$	$A \rightarrow \varepsilon$

Produktion visavi konsumtion Automaten och produktionsreglerna i EXEMPEL 3.1 beskriver tiopotenser på väsentligen likadant sätt. Skillnaden är bara att det ena konceptet talar om *konsumtion* och det andra om *produktion*:

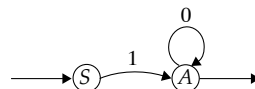
Produktion Av S kan man producera 1 följt av en ny symbol A . Och av denna kan man producera 0 följt av samma symbol A .

$$S \rightarrow 1A$$

$$A \rightarrow \varepsilon | 0A$$

Men man kan också "avbryta" produktionen här genom att sätta ut A (låt A producera ε).

Konsumtion Från starttillståndet S kan man konsumera 1, varvid man drivs till ett nytt tillstånd A . Och från detta kan man konsumera 0 och dri-



vas tillbaka till A igen. Men man kan också avsluta konsumtionen här och gå i acceptans.

Slingor, Kleenestjärnor och rekursion Lägg märke till att automaten i EXEMPEL 3.1 har en *slinga* för att beskriva de avslutande nol-lorna i en tiopotenssträng, att det reguljära uttrycket 10^* nyttjar

Kleenestjärna i samma syfte, och att grammatiken använder rekursiva regler $A \rightarrow \varepsilon$, $A \rightarrow 0A$ (med den första regeln som basfall).


 0^*
 $A \rightarrow \varepsilon \mid 0A$

En automat gör det med slingor, ett reguljärt uttryck med Kleenestjärnor och en grammatik med rekursion.

Notation för strängproduktion Vid strängproduktion använder vi pilsymbolen " \Rightarrow " som beteckning för *ett* produktionssteg. Symbolen " \rightarrow " reserveras för själva *presentationen* av produktionsreglerna. Ibland kommer vi även att nyttja symbolen " \Rightarrow^* " vilken betecknar *noll* eller *flera* produktionssteg. T ex skriver vi

$$S \Rightarrow 1A \Rightarrow 10A \Rightarrow^* 100000$$

För fullständigets skull presenteras även en *formell definition* av " \Rightarrow " och " \Rightarrow^* ":

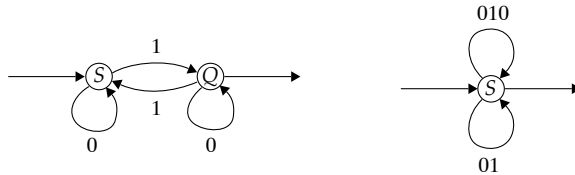
DEFINITION	$\alpha \Rightarrow \beta$ om $\begin{cases} \alpha = \lambda_1 \alpha' \lambda_2 \\ \beta = \lambda_1 \beta' \lambda_2 \\ \alpha' \rightarrow \beta' \end{cases}$	"Från α kan man i <i>ett</i> steg producera β om α och β är identiska sånär som på delsträngar α' och β' inuti α respektive β , och det finns en produktionsregel som transformerar α' till β' ".
DEFINITION	$\alpha \Rightarrow^* \alpha$ (basfall)	"Produktion i <i>noll</i> steg".
	$\alpha \Rightarrow^* \gamma$ om $\begin{cases} \alpha \Rightarrow \beta \\ \beta \Rightarrow^* \gamma \end{cases}$	"Från α kan man i <i>ett eller flera</i> steg producera γ om man kan producera β i <i>ett</i> steg, och från β producera γ i <i>noll eller flera</i> steg.

Terminerande och icketerminerande tecken Produktionsregler innehåller två sorters tecken:

- Tecken som färdigproducerade strängar innehåller – t ex 0, 1 i EXEMPEL 3.1. Dessa benämnes *terminerande*.
- Tecken avsedda att användas enbart under själva produktionen – t ex S , A i EXEMPEL 3.1 – och som *inte* skall finns med i färdigproducerade strängar. Sådana symboler kallas för *icketerminerande*.

EXEMPEL 3.2 Här kommer ytterligare två finita automater samt produktionsregler för deras språk.

automater
konsumerar



regler
producerar

$S \rightarrow 0S \mid 1Q$
 $Q \rightarrow \varepsilon \mid 0Q \mid 1S$

$S \rightarrow \varepsilon \mid 010S \mid 01S$

I nästa avsnitt formaliseras visst stoff från föregående avsnitt.

■ Grammatik

Med en *grammatik* menar vi en kvartupel

$$G = (\Sigma_N, \Sigma_T, P, S)$$

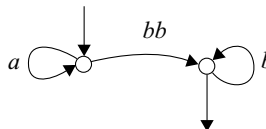
där

- (i) Σ_N är en ändlig mängd, (de icketerminerande symbolerna)
- (ii) Σ_T är en ändlig mängd, (de terminerande symbolerna)
- (iii) P är en ändlig mängd av produktionsregler $\alpha \rightarrow \beta$ där α och β är strängar över $\Sigma_N \cup \Sigma_T$
- (iv) $S \in \Sigma_N$ (startsymbolen som varje strängproduktion utgår ifrån)

Det språk $L(G)$ som G beskriver (producerar) består av de strängar w över Σ_T som man med G 's regler kan producera utgående från S :

$$L(G) = \{w \in \Sigma_T^* \mid S \xRightarrow{G}^* w\}$$

✂ **TEST 3.1** Skriv produktionsregler för det språk som accepteras av



■ 3.2 Sammanhangsfria grammatiker

Först definieras "reguljär grammatik" som sedan generaliseras till "sammanhangsfri grammatik".

■ Reguljära regler och reguljära grammatiker

De produktionsregler för *reguljära* språk som vi mött har varit av ned-

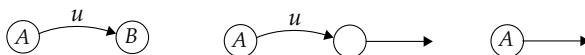
anstående slag:

reguljära
regler

$$A \rightarrow uB, \quad A \rightarrow \varepsilon, \quad A \rightarrow u \quad (1)$$

med A, B som icketerminerande symboler, och u som någon sträng av terminerande symboler. Vi säger att sådana regler är *reguljära*.

⌘ Anm.3.2 Reglerna (1) har ett intimt samband med en finit automats sätt att drivas av en sträng.



En sträng u (som kan vara tom) driver en finit automat till ett tillstånd som är accepterande eller ickeaccepterande. Och tom sträng driver alltid en finit automat till acceptans om aktuellt tillstånd är accepterande.

DEFINITION En reguljär grammatik G är en grammatik vars produktionsregler är reguljära.

Följande sats (vars bevis lämnas som övning åt den läsare som vill träna upp sin formella förmåga) är en självklar konsekvens av den korrespondens mellan *produktion* och *konsumtion* som vi pekade på redan i samband med de första exemplen och senast i detta avsnitts inledande rader.

✧ SATS 3.1 Ett språk L är reguljärt om och endast om det finns en reguljär grammatik G sådan att $L = L(G)$.

■ Grammatiker för ickereguljära språk

Vi visar här hur vissa ickereguljära språk kan beskrivas med grammatiker som är mindre restriktiva än reguljära grammatiker.

EXEMPEL 3.3 Inga *reguljära* regler förmår beskriva de "enkelt nästlade" parentessträngarna

$$\{ (), (()), ((())), \dots \}$$

eftersom dessa *inte* bildar ett *reguljärt* språk (se sid 60). Men med *ickereguljära*[†] regler går det bra:

$$S \rightarrow () | (S) \quad (2)$$

†. Regeln $S \rightarrow (S)$ har en annorlunda form på sitt högerled än vad reguljära grammatikregler har.

Nästa exempel behandlar ett parentesspråk som innehåller mer komplicerade parentessträngar än de "enkelt nästlade".

EXEMPEL 3.4 Som Du vet används parenteser i aritmetiska uttryck för att beskriva i vilken ordning aritmetiska operationer skall utföras. Sådan parentes-användning kan vara syntaktiskt korrekt eller felaktig.

Betrakta t ex följande tre parentesuttryck

$$\begin{aligned} &((x + y) \cdot (z + u) + v) \cdot (w + x) \\ &x \cdot (y - (z + u) + v) \\ &(x \cdot (y - z) + (u + v)) \end{aligned}$$

Om man klär av dessa uttryck allt utom just parenteserna, syns det tydligt vilka av de nakna parentessträngarna som är "välbildade" och vilka som inte är det:

$$\begin{array}{ll} (() ()) () & \text{välbildad} \\ (()) & \text{välbildad} \\ (() () & \text{icke välbildad} \end{array}$$

En välbildad parentessträng erhålles om man *omgärdar* ett redan välbildad parentessträng med parenteser, eller om man *sammanfogar* två välbildade parentessträngar. Följande regler formaliserar denna karaktäristik:

$$S \rightarrow () | (S) | SS \quad (3)$$

Nedanför illustreras hur parentessträngen $(() () ())$ kan produceras med reglerna (3):

" Δ " markerar vilkensymbol som är producerande i varje enskilt steg.

$$\begin{aligned} S &\Rightarrow \underset{\Delta}{SS} \\ &\Rightarrow (\underset{\Delta}{S})S \\ &\Rightarrow (\underset{\Delta}{SS})S \\ &\Rightarrow ((\underset{\Delta}{ }) \underset{\Delta}{S})S \\ &\Rightarrow ((\underset{\Delta}{ }) ()) \underset{\Delta}{S} \\ &\Rightarrow ((\underset{\Delta}{ }) ()) () \end{aligned}$$

⌘ Anm.3.3 Reglerna $S \rightarrow () | (S)S$ beskriver samma sak som (3).

EXEMPEL 3.5 Reguljära uttryck över alfabetet $\Sigma = \{a, b\}$ som t ex $a(\epsilon \cup a \cup bb)^*$ är strängar över ett större alfabet $\{a, b, \epsilon, \emptyset, \cup, *, (,)\}$. De inledande tecknen i detta utvidgade alfabet representerar som Du vet de enklaste reguljära uttrycken, medan de fyra sista tecknen används för att teckna ned mer komplicerade reguljära uttryck.

Reguljära uttryck (sid 23) över $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ kan definieras just

i termer av grammatikregler:

$$S \rightarrow \emptyset \mid \varepsilon \mid \sigma_1 \mid \dots \mid \sigma_n \mid (S \cup S) \mid (SS) \mid (S^*) \quad (4)$$

Mängden av strängar som kan produceras med hjälp av reglerna (4) bildar språket av reguljära uttryck över alfabetet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$.

Enkla vänsterled En regel

$$A \rightarrow \dots$$

sammanhang

vars vänsterled består av en ensam icketerminerande symbol A uttrycker att A får vara det som står i regelns högerled oberoende av vilka symboler som befinner sig i A :s *närhet* (oberoende av i vilket *sammanhang* som A står). Betrakta som kontrast till detta tex regeln $aAa \rightarrow abba$, vars innebörd är att A får vara bb om A har a på ömse sidor om sig. Du som längtar efter mer av det senare får vänta till nästa kapitel. I detta kapitel skall vi hålla oss till regler vars vänsterled är just *ensamma* icketerminerande tecken.

Högerledens komplexitet En reguljär regels högerled (sid 76) får innehålla högst *ett* icketerminerande tecken, som i förekommande fall ligger sist (efter en sträng av noll eller flera terminerande tecken). Men ickereguljära reglers högerled kan ha icketerminerande och terminerande tecken "lite huller om buller". En regel vars högerled tillåts vara en *godtycklig* kombination av terminerande och/eller icketerminerande symboler (noll eller flera), men vars vänsterled är *en* ensaka symbol (icketerminerande)

sammanhangs-
fria regler

$$A \rightarrow \alpha \text{ med } A \in \Sigma_N \text{ och } \alpha \in (\Sigma_T \cup \Sigma_N)^* \quad (5)$$

kallas för en *sammanhangsfri* regel, och är den minst restriktiva regeln för att uttrycka att mönstret för ett objekt är *oberoende av i vilket sammanhang* objektet står.

ContextFree
Grammar

DEFINITION

En CFG (sammanhangsfri grammatik) G är en grammatik vars produktionsregler är sammanhangsfria.

Ett CFL (sammanhangsfritt språk) L är ett språk sådant att $L = L(G)$ för någon CFG G .

⌘ Anm.3.4 Varje reguljär grammatik är sammanhangsfri, eftersom reguljära regler är specialfall av sammanhangsfria dito.

EXEMPEL 3.6 $\epsilon, ab, ba, abba, babbaaba$ är exempel på strängar i språket

$$L = \{w \in \{a, b\}^* \mid w \text{ innehåller lika många } a:n \text{ som } b:n\}$$

Nedan presenteras *tre* olika sammanhangsfria grammatiker för L , samt ett bevis för att den tredje grammatiken verkligen producerar L . Försök själv att bevisa samma sak för de övriga två grammatikerna.

<i>symboler</i>	<i>regler</i>	
$G_1 = \{\{S, A, B\}, \{a, b\}, P_1, S\}$	$P_1 = \{\{S \rightarrow \epsilon aB bA\},$ $\{A \rightarrow aS bAA\},$ $\{B \rightarrow bS aBB\}\}$	(6)
$G_2 = \{\{S\}, \{a, b\}, P_2, S\}$	$P_2 = \{S \rightarrow \epsilon aSb bSa SS\}$	(7)
$G_3 = \{\{S\}, \{a, b\}, P_3, S\}$	$P_3 = \{S \rightarrow \epsilon aSbS bSaS\}$	(8)

BEVIS (för att $L(G_3) = L$):

Vi börjar med $L(G_3) \supseteq L$, dvs att $w \in L(G_3)$ om $w \in L$. Här är det naturligt att göra ett induktionsbevis med induktion över w :s längd.

Basfall: ϵ är L :s kortaste sträng. Och $\epsilon \in L(G_3)$.

Induktionssteg: Tag nu en icke-tom sträng $w \in L$ och antag (induktionsantagandet) att varje sträng i L som är kortare än w ligger i $L(G_3)$.

Eftersom w endera börjar med a eller med b , så gäller

$$w = au \text{ där } u \text{ har "ett } b \text{ mer"}^{\dagger} \quad (9)$$

$$\text{eller } w = bv \text{ där } v \text{ har "ett } a \text{ mer"}. \quad (10)$$

Vi visar att u – som i (9) – kan skrivas w_1bw_2 , där w_1 och w_2 har lika många a :n som b :n:

Om man går från vänster till höger i u – tecken för tecken – så kommer man förr eller senare att ha "konsumerat" ett prefix u_1 (som givetvis kan vara hela u) med "ett b mer". Dvs $u = u_1w_2$ där u_1 har "ett b mer" och w_2 måste ha lika många a :n som b :n. Låt nu u_1 vara det kortaste prefixet av detta slag. Då måste u_1 sluta med b , dvs $u_1 = w_1b$ med w_1 innehållande lika många a :n som b :n. Ty om sista tecknet i u_1 vore a skulle u_1 med sista tecknet borttaget få "två b mer". Och då skulle man dessförinnan ha konsumerat en sträng med "ett b mer", eftersom man vid tecken-för-tecken konsumtion inte kan mumsa i sig "två b mer" på en gång. Dvs då skulle u_1 inte vara det kortaste

Alltså,

$$u = u_1w_2 = w_1bw_2 \text{ där } w_1 \text{ och } w_2 \text{ har lika många } a:n \text{ som } b:n. \quad (11)$$

[†]. Dvs förekomsten av a :n respektive b :n är sådan att det finns ett överskott på exakt *ett* b .

På motsvarande sätt kan man visa att

$$v = w_1 a w_2. \quad (12)$$

Av (9),(10),(11),(12) följer att

$$w = au = aw_1 b w_2 \text{ eller } w = bv = bw_1 a w_2. \quad (13)$$

Eftersom w_1 och w_2 är kortare än w , så följer av induktionsantagandet att $w_1, w_2 \in L(G_3)$. Dvs för $i = 1, 2$ har vi

$$S \xRightarrow[G_3]{*} w_i \quad (14)$$

Med hjälp av (14) och G_3 :s "förlängningsregler" $S \rightarrow aSbS \mid bSaS$ kan w produceras:

$$S \Rightarrow aSbS \xRightarrow[\Delta]{G_3^*} aw_1 bS \xRightarrow[G_3]{*} aw_1 b w_2$$

$$S \Rightarrow aSbS \xRightarrow[\Delta]{G_3^*} bw_1 aS \xRightarrow[G_3]{*} bw_1 a w_2$$

Dvs $w \in L(G_3)$.

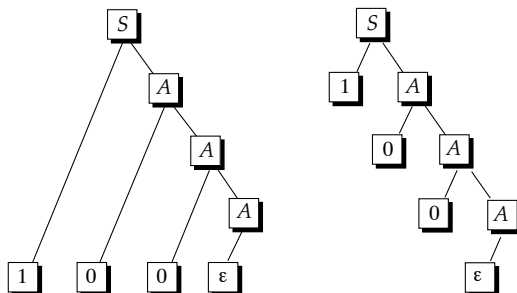
Omvänt skall vi nu visa att $L(G_3) \subseteq L$, dvs att varje sträng som produceras av G_3 innehåller "lika många a :n som b :n". Nyckeln till detta är att högerleden från G_3 :s regelmängd innehåller "lika många a :n som b :n". Då en sträng $w \in L(G_3)$ produceras, börjar man med "lika många a :n som b :n". (Ty S har noll a :n och noll b :n.) Och i varje steg av produktionen tar man sedan bort S , dvs man tar bort noll a :n och noll b :n från "arbetssträngen", och sätter istället dit en sträng som är ett högerled från regelmängden. Således förlorar arbetssträngen inga a :n eller b :n, och när den får sådana tecken (a :n och b :n), så får den lika många av varje sort. Därvid kommer den alltid att ha "lika många a :n som b :n". Speciellt gäller detta i slutänden när arbetssträngen är w . \square

■ Produktionsträd

En strängproduktion inleds alltid med att en viss icketerminerande symbol – *startsymbolen* – producerar en sträng av icketerminerande och/eller terminerande symboler. Från de icketerminerande symbolerna i denna sträng *fortskrider sedan* produktionen ...

Som Du ser finns det här en hierarkisk struktur. Men hierarkiska strukturer brukar ju representeras med träd och detta är inget undantag. Strängproduktionernas träd kallas därvid för *produktionsträd* eller *härledningsträd*.

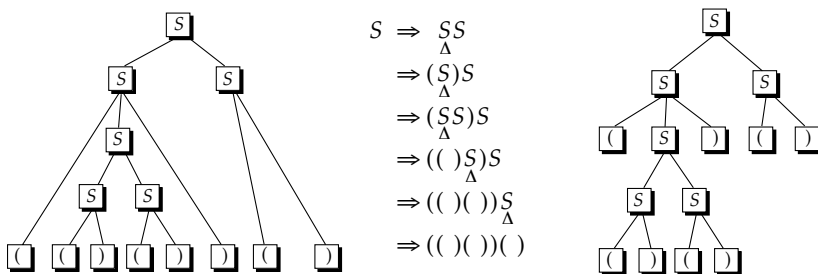
Båda träden
visar hur man
med de regul-
jära reglerna
 $S \rightarrow 1A$
 $A \rightarrow \varepsilon \mid 0A$
producerar
strängen 100.



I det vänstra trädet ligger alla *löv* på *en* rad – den undre. Då framhävs den färdigproducerade strängen. I det högra trädet ligger en nods *söner* på *en* gemensam rad. Då framhävs de enskilda reglerna.

I löven bor *terminerande* symboler och längs vägarna från roten ned till löven finns det enbart *icketerminerande* symboler innan löven. Därför kallar vi *produktionsträdens* löv för *terminerande noder* och övriga noder för *icketerminerande noder*.

Lägg märke till att *produktionsträden* i reguljära grammatiker som ovan har alla *icketerminerande* noder längs en linje. Träden i *ickereguljära sammanhangsfria grammatiker* är mer komplicerade, vilket illustreras i nedanstående figur.

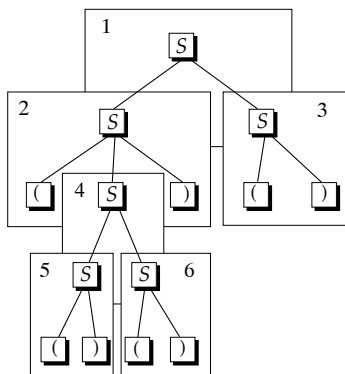


Från vänster eller höger eller ... Strängproduktionen i figuren ovanför går från *vänster* i arbetssträngen. Andra turordningar är också tänkbara. T ex från *höger*:

$$S \Rightarrow SS \Rightarrow S() \Rightarrow (S)() \Rightarrow (SS)() \Rightarrow (S())() \Rightarrow (() ())()$$

Notera att det till de två turordningarna (från vänster respektive höger) hör ett och samma *produktionsträd*, men att turordningarna representeras i trädet av olika "produktionsvägar":

Produktion
av $(() ()) ()$
från vänster
respektive hö-
ger. Numre-
ringen av de
enskilda pro-
duktionerna
visar produ-
ktionsväg i trä-
det.



$$S \xRightarrow{1} S_{\Delta} S$$

$$\xRightarrow{2} (S) S$$

$$\xRightarrow{4} (S S) S$$

$$\xRightarrow{5} (() S) S$$

$$\xRightarrow{6} (() ()) S_{\Delta}$$

$$\xRightarrow{3} (() ()) ()$$

$$S \xRightarrow{1} S S_{\Delta}$$

$$\xRightarrow{3} S ()$$

$$\xRightarrow{2} (S) ()$$

$$\xRightarrow{4} (S S) ()$$

$$\xRightarrow{6} (S ()) ()$$

$$\xRightarrow{5} (() ()) ()$$

✂ **TEST 3.2** Ge sammanhangsfria grammatiker för följande språk

- $\{a^m b^m \dots a^k b^k \mid m, \dots, k \in \mathbb{N}\}$,
- $\{w \in \{a, b\}^* \mid w \text{ innehåller flera } a:n \text{ än } b:n\}$,
- $\{w \in \{a, b\}^* \mid w \text{ innehåller inte lika många } a:n \text{ som } b:n\}$,
- $\{w \in \{a, b\}^* \mid w \text{ är en jämn palindrom}\}$,
- $\{w \in \{a, b\}^* \mid w \text{ är en palindrom}\}$,
- $\{w \in \{-1, +1\}^* \mid w:s \text{ värde är noll}\}$. Text är $+1+1-1+1-1-1$ ett sådant w .

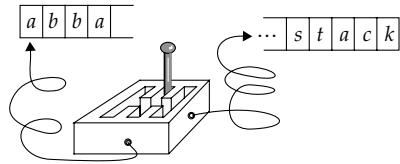
3.3 Pushdownautomater (PDA:er)

Vi har lärt känna finita automater som en typ av algoritmer specialiserade på att känna igen reguljära språk. En intressant fråga är givetvis huruvida det finns någon motsvarande beskrivning av algoritmer som är specialiserade på att känna igen de sammanhangsfria språken?

I början av 60-talet presenterade *Oettinger* en typ av automat som var en sorts utvidgning av en finit automat. Utvidgningen bestod av en extra tape där "minnesanteckningar" kunde skrivas och läsas enligt principen först-in-sist-ut. Dvs extratapan kunde användas som en stack (pushdown-lista). Sådana automater kom därför att kallas för *pushdownautomater*. *Chomsky* som hittade på begreppet sammanhangsfria språk i mitten av 50-talet bevisade någon tid efter *Oettingers* presentation av pushdownautomater att de språk som dessa automater känner igen är just de sammanhangsfria språken.[†]

Informell beskrivning

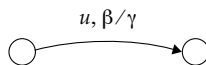
En *pushdownautomat* (PDA) består av *två tapar* och en *kontrollmekanism*. Kontrollmekanismen har ändligt antal tillstånd, men taperna kan rymma hur långa strängar som helst. På den ena tapen placeras input, dvs strängar som en PDA skall undersöka. Här läser (konsumerar) PDA:n noll eller flera tecken i taget från inputsträngens vänstra ände. På den andra tapen (stack-tapen) får PDA:n *både* läsa och skriva noll eller flera tecken. Dock endast enligt principen *först-in-sist-ut*. Vidare låter vi en PDA kunna agera ickedeterministiskt. Mer precist är dess agerande av följande slag.



Beroende på *vilket tillstånd* som kontrollmekanismen befinner sig i, *vilken delsträng* av inputsträngen och vilken *delsträng* överst på stacken som PDA:n konsumerar agerar den genom att

- göra en tillståndsövergång,
- ersätta konsumerad delsträng från stacken med ny sträng.

Grafisk beskrivning En PDA kan likt en FA beskrivas med en riktad graf. Skillnaden är att de riktade bågarna mellan noderna nu måste förse inte bara med information om vad som konsumeras från inputsträngen utan också om PDA:ns agerande på stacken:



”Om u konsumeras när β ligger överst på stacken ersätt β med γ .”

Push och pop Agerandet på stacken kan uttryckas som att en sträng β överst på stacken tas bort (*poppas*) och att en sträng γ läggs dit (*pushas*).

En variant på detta tema är att starta och sluta med en speciell ändmarkör på stacken.

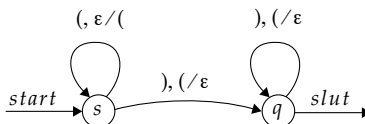
Acceptera med tom stack Vi säger att en PDA M *accepterar* en sträng om M kan konsumera hela strängen på ett sätt så att den hamnar i ett accepterande tillstånd med stacken i samma skick som från början, dvs tom. Mängden av strängar som M accepterar betecknar vi med $L(M)$.

EXEMPEL 3.7 Nedan visas en PDA M som i tillståndet s lägger upp varje läst vänsterparentes på stacken. När den första högerparentesen läses från inputtapen agerar M med att plocka av en vänsterparentes från stackens topp och övergår samtidigt till ”avplockningstillståndet” q , ett till-

†. Ungefär samtidigt och oberoende visade Schutzenberger och Evey samma sak.

stånd där varje konsumerad högerparentes från inputtappen "kvitteras" med en avplockad vänsterparentes från stacktoppen. Som du förstår innebär detta att M accepterar de enkelt nästlade parentessträngarna, $\{ (), (()), ((())), \dots \}$.

En PDA för enkelt nästlade parentessträngar.



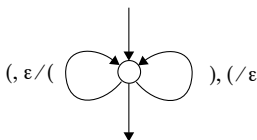
Körexempel:

Observera att stackens topp är vänster ände av stacktappen.

tillstånd	inputtappen	stacktappen
s	$((()))$	ϵ
s	$(()))$	$($
s	$()))$	$(($
s	$)))$	$((($
q	$))$	$(($
q	$)$	$($
q	ϵ	ϵ

EXEMPEL 3.8 Mängden av *alla* välbildade parentessträngar – inte bara de enkelt nästlade – accepteras av en utomordentligt enkel PDA:

En PDA för välbildade parentessträngar.



Körexempel:

Varje konsumerad vänsterparentes pushas på stacken, och poppas när motsvarande högerparentes konsumeras från inputtappen.

inputtappen	stacktappen
$((()) ())$	ϵ
$(()) ())$	$($
$()) ())$	$(($
$)) ())$	$((($
$) ())$	$(($
$())$	$($
$))$	$(($
$)$	$($
ϵ	ϵ

⌘ Anm.3.5 Många programmeringsspråk använder parenteser för att representera hierarkiska strukturer (nästlade block, listor inuti listor, ja även ett program är ett hierarkiskt objekt vars hierarki ibland representeras med parenteser). Därför är det en viktig syssla för t ex en

kompiator att kunna känna igen välbildade parentesuttryck. Likaså måste en s.k. parser för aritmetiska uttryck (ett program som undersöker ett aritmetiskt uttryck för att upptäcka dess hierarkiska form) kunna det här.

DEFINITION En PDA (*pushdownautomat*) M är en sextupel

$$(Q, \Sigma, \Gamma, \Delta, s, F)$$

där

- (i) Q är en ändlig mängd (tillståndsmängden)
 - (ii) Σ är en ändlig mängd (inputalfabetet)
 - (iii) Γ är en ändlig mängd (stackalfabetet)
 - (iv) Δ är en ändlig delmängd (övergångsmängden) av

$$(Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*)$$
 - (v) $s \in Q$ (starttillståndet)
 - (vi) $F \subseteq Q$ (mängden av accepterande tillstånd)
-

⌘ Anm.3.6 Notera att övergångsmängden Δ kan betraktas som en (femställig) *relation* definierad på $Q \times \Sigma^* \times \Gamma^* \times Q \times \Gamma^*$.

■ **Att driva** Givet en PDA $M = (Q, \Sigma, \Gamma, \Delta, s, F)$, en inputsträng $w \in \Sigma^*$, en sträng $W \in \Gamma^*$ på stacken, och två tillstånd $p, r \in Q$, så säger man att

Informellt ▶ $(w, W) \in \Sigma^* \times \Gamma^*$ kan driva M från p till r i noll eller flera steg om det finns en följd (noll eller flera) tillståndsövergångar från p till r under vilken M kan konsumera w (från inputtappen) samt på stacken ersätta något prefix av W med någon sträng $W' \in \Gamma^*$.

Formellt

DEFINITION

- (i) (ϵ, ϵ) kan driva M från p till p "M kan drivas att stanna kvar i p genom att konsumera noll tecken från inputtappen och noll tecken från stacken"

- (ii) $(w, W) = (xy, XY) \in \Sigma^* \times \Gamma^*$

kan driva M från p till r om

$$(p, x, X, q, X') \in \Delta$$

och $(y, X'Y)$ kan driva M från q till r

"Ett prefix x av w och ett prefix X av stacksträngen $W = XY$ driver här M direkt från p till q . Och resten y av w tillsammans med den nya stacksträngen $X'Y$ driver M vidare till r "

Se även
Anm.3.7

■ Att acceptera

- En PDA $M = (Q, \Sigma, \Gamma, \Delta, s, F)$ accepterar $w \in \Sigma^*$, om M kan drivas av $(w, \varepsilon) \in \Sigma^* \times \Gamma^*$ från s till något $q \in F$ med tömd stack.
- $L(M) = \{w \in \Sigma^* \mid M \text{ accepterar } w\}$

⌘ Anm.3.7 Punkt (ii) av "kan driva"-definitionen illustreras bäst med en figur:



FIGUR 3.1 En inputsträng och en stacksträng kan driva M från p till r , om ett prefix x av inputsträngen och ett prefix X av stacksträngen kan driva M från p till något tillstånd q , och resten av inputsträngen tillsammans med den nya stacksträngen (den som blir resultatet av att X har ersatts av något X') kan driva M vidare från q till r .

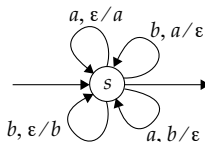
EXEMPEL 3.9 Vi presenterar nu PDA:n från EXEMPEL 3.7 på sidan 83 som en *sextupel*. (Parentessymbolerna i EXEMPEL 3.7 är här ersatta med tecknen a, b i syfte att öka läsbarheten):

$$\begin{aligned}
 M &= (Q, \Sigma, \Gamma, \Delta, s, F) \\
 Q &= \{s, q\} \\
 \Sigma &= \{a, b\} \\
 \Gamma &= \{a\} \\
 F &= \{q\} \\
 \Delta &= \{(s, a, \varepsilon, s, a), (s, b, a, q, \varepsilon), (q, b, a, q, \varepsilon)\}
 \end{aligned}$$

EXEMPEL 3.10 $L = \{w \in \{a, b\}^* \mid w \text{ har lika många } a\text{:n} \text{ som } b\text{:n}\}$ accepteras av en PDA med *ett* enda tillstånd:

En PDA för
strängarna
med lika
många a :n
som b :n.

$$\begin{aligned}
 M &= (Q, \Sigma, \Gamma, \Delta, s, F) \\
 Q &= \{s\} \\
 \Sigma &= \{a, b\} \\
 \Gamma &= \{a, b\} \\
 F &= \{s\} \\
 \Delta &= \{(s, a, \varepsilon, s, a), (s, b, a, s, \varepsilon), (s, b, \varepsilon, s, b), (s, a, b, s, \varepsilon)\}
 \end{aligned}$$



✂ **TEST 3.3** Konstruera PDA:er för

- a) $\{w \in \{a, b\}^* \mid w \text{ är en palindrom}\},$
- b) $\{w \in \{a, b\}^* \mid w \text{ innehåller flera } a\text{:n än } b\text{:n}\},$
- c) $L = \{a^m b^m \dots a^k b^k \mid m, \dots, k \in \mathbb{N}\}.$

3.4 För varje CFG finns det en PDA

Vi presenterar två metoder att till given sammanhangsfri grammatik G konstruera en PDA M sådan att $L(M) = L(G)$.

Top-down parser

Här konstruerar vi en PDA för en CFG G så att PDA:ns agerande på stacken härmar strängproduktion *uppifrån och nedåt* i G 's produktionssträd (närmare bestämt längs vägar som motsvarar turordning "från vänster" (se sid 81)). Härav namnet *top-down parser*.

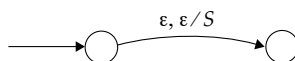
*Top-down parser*ns tillvägagångssätt när den undersöker en given inputsträng är att först lägga upp G 's startsymbol på stacken och därvid göra en tillståndsövergång till ett accepterande tillstånd varifrån *top-down parser* sedan upprepar att sysselsatt med att endera poppa en icke-terminerande symbol från stacken och ersätta den med något högerled från G 's regeluppsättning (det är här som den härmar G 's strängproduktion), eller att poppa en terminerande symbol från stacken och samtidigt konsumera densamma från inputtappen. Om *top-down parser* på detta sätt kan mumsa i sig hela inputsträngen så att stacken därvid töms accepteras strängen ifråga.

Två tillstånd är således allt vad en *top-down parser* behöver, ett accepterande tillstånd och ett icke-accepterande starttillstånd.

► I starttillståndet skall *top-down parser* göra blott en sak:

Utan att konsumera något från inputsträngen skall den pusha G 's startsymbol S på den från början tomma stacken, och därefter göra en tillståndsövergång till det accepterande tillståndet.

Parsern börjar
med att pusha
 G 's startsymbol
 S på stacken.

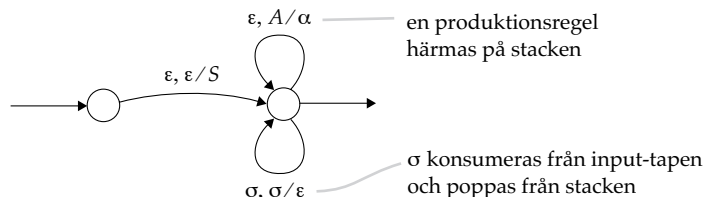


► I det accepterande tillståndet skall *top-down parser* kunna två saker:

(i) För varje regel $A \rightarrow \alpha$ i G 's regeluppsättning skall den kunna härma G 's sätt att använda denna regel genom att ersätta A på stacktoppen med α . Och detta utan att konsumera något från inputtappen.

(ii) För varje terminerande symbol σ skall den kunna konsumera σ från inputsträngen och poppa samma tecken från stacktoppen.

En *top-down parser*.



PDA:ns härminning följer produktions-trädet nerifrån och uppåt.

till- stånd	input	stacken	kommentar
s	$(((\)))$	ϵ	
s	$((\)))$	$($	pushar konsumerat $($
s	$(\)))$	$(($	pushar konsumerat $($
s	$)))$	$((($	pushar konsumerat $($
s	$))$	$(((($	pushar konsumerat $($
s	$))$	$S(($	härmar $S \rightarrow (\)$ på stacken
s	$)$	$)S(($	pushar konsumerat $)$
s	$)$	$S($	härmar $S \rightarrow (S)$ på stacken
s	ϵ	$)S($	pushar konsumerat $)$
s	ϵ	S	härmar $S \rightarrow (S)$ på stacken
q	ϵ	ϵ	poppar S

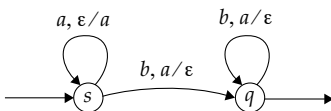
- ✂ **TEST 3.4** a) Konstruera både en top-down och en bottom-up parser för det språk som ges av $S \rightarrow \epsilon | aSaa | bSbb$ b) Visa med provkörning hur de två accepterar $abbbaa$.

■ 3.5 För varje PDA finns det en CFG

Vi illustrerar nu i ett exempel hur man kan konstruera en CFG som genom att *härma* en PDA:s tillståndsövergångar producerar precis de strängar som PDA:n accepterar. Konstruktionens idé går att tillämpa på varje PDA.

EXEMPEL 3.13 PDA:n M i figuren nedan känner Du igen sedan tidigare, eller hur.

En PDA för de enkelt nästlade "parentessträngarna".



Det språk som den accepterar utgörs av de enkelt nästlade parentessträngarna, fastän vänster och högerparentes här noteras med a respektive b :

$$L(M) = \{ab, aabb, aaabbb, \dots\}$$

M accepterar (likt varje PDA) sådana strängar som driver M från starttillståndet s till ett accepterande tillstånd (i detta fall q) på ett sådant sätt att stacken är tom såväl i början som i slutet. Låt oss uttrycka detta som att *målet* inledningsvis är att *driva* M från starttillståndet s till det accepterande tillståndet q med oförändrad stack.

En blick på M 's tillståndsövergångar visar att ett inledande tecken a driver M från s till s . Vidare att a pushas på stacken. Efter detta är målet fortfarande att *driva* M från s till q men dessutom att *poppa* a från stacken, ty stacken skall ju vara tom i slutet. Således gör M 's a -konsumerande övergång att det inledande målet transformeras till ett nytt mål.

Om nästa tecken som konsumeras också är a , agerar M likadant som nyss, dvs M drivs från s till s och pushar a på stacken. Därefter är målet (det nya) att *driva* M från s till q och att *poppa* aa från stacken.

	MÅL		INPUT	NYTT MÅL
(i)	att driva M från s till q med oförändrad stack	→	a	att driva M från s till q och att poppa a från stacken
(ii)	att driva M från s till q och att poppa a från stacken	→	a	att driva M från s till q och att poppa aa från stacken

Men strängen aa är för lång för att kunna poppas vid en enda tillståndsövergång. (Den här PDA:n kan enbart poppa enstaka tecken!)

Det nya målet måste därför sönderdelas i *delmål*. Nedanstående omskrivning av (ii) ordnar med sådan sönderdelning:

	MÅL	INPUT	DELMÅL 1	DELMÅL 2
(ii)	att driva M från s till q och att poppa a från stacken	\rightarrow a	att driva M från s till q och att poppa a från stacken	att driva M från q till q och att poppa a från stacken

Om M står i tillståndet s , har a på stacktoppen samt konsumerar ett b från inputtapen förverkligas det första av delmålen. Och genom ytterligare en b -konsumtion kan det andra delmålet förverkligas (under förutsättning att tecknet a ligger på stacktoppen).

Den här diskussionen visar hur M :s agerande vid konsumtion av en accepterad sträng kan beskrivas i termer av *måltransformationer*, där varje mål är av typen

Ett typiskt mål.
 att driva M
 från *tillstånd* till *tillstånd*
 och att poppa *en sträng* från stacken

Vi ska nu se hur man, genom att betrakta sådana *mål* som *icketerminerande symboler* och *måltransformationerna* som *produktionsregler*, kan bilda en CFG som härmar M :s agerande.

Det inledande målet, betecknar vi förslagsvis med $M_{s,q}^\varepsilon$:

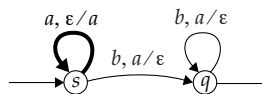
$$M_{s,q}^\varepsilon = \begin{cases} \text{att driva } M \text{ från } s \text{ till } q \\ \text{med oförändrad stack} \\ \text{(att poppa } \varepsilon) \end{cases} \quad (15)$$

Vi introducerar också en särskild icketerminerande symbol S att användas som startsymbol. Vår första produktionsregel blir därmed

$$S \rightarrow M_{s,q}^\varepsilon \quad (16)$$

Övriga regler och icketerminerande symboler kan vi formulera utgående från M :s övergångar och redan introducerade mål:

► Den a -konsumerande tillståndsövergången



transformerar $M_{s,q}^\varepsilon$ enligt (i) på sidan 90:

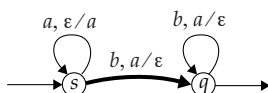
$$M_{s,q}^\varepsilon \rightarrow aM_{s,q}^a \quad (17)$$

och $M_{s,q}^a$ enligt (ii) på sidan 90:

$$M_{s,q}^a \rightarrow aM_{s,q}^aM_{q,q}^a \quad (18)$$

(Poängen med att placera konsumerade tecken i reglernas *högerled* är att reglerna därmed kan *producera* sådant som PDA:n *konsumerar*.)

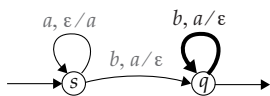
► Den b -konsumerande övergången mellan de två tillstånden



ger upphov till regeln

$$M_{s,q}^a \rightarrow b \quad (19)$$

Ty målet att *driva* M från s till q och att *poppa* a från *stacken* kan *uppfyllas* vid denna b -konsumerande övergång.

► Den b -konsumerande öglan  leder på motsvarande sätt till regeln

$$M_{q,q}^a \rightarrow b \quad (20)$$

Med förenklad notation kan de fem reglerna skrivas

$$S \rightarrow E \quad (21)$$

$$E \rightarrow aA \quad (22)$$

$$A \rightarrow aAB \quad (23)$$

$$A \rightarrow b \quad (24)$$

$$B \rightarrow b \quad (25)$$

Insätts (22) i (21), liksom (25) i (23) erhålles

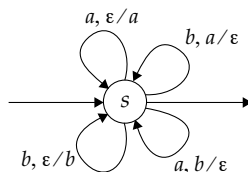
$$S \rightarrow aA$$

$$A \rightarrow aAb$$

$$A \rightarrow b$$

som bildar en ganska naturlig regeluppsättning för L , eller hur? \square

✂ **TEST 3.5** Konstruera på motsvarande sätt som i EXEMPEL 3.13 en CFG som härmar vidstående PDA.



3.6 De sammanhangsfria språkens gränser

Som Du säkert kommer ihåg kunde de *reguljära* språkens gränser stakas ut med hjälp av finita automaters begränsade minnesförmåga (ändligt många tillstånd). Närmare bestämt konstaterade vi att en DFA återuppsöker något tillstånd två eller flera gånger efter att ha mumsat i sig lika många tecken som den har tillstånd. Detta bildade stommen i beviset av den reguljära *pumpsatsen*.

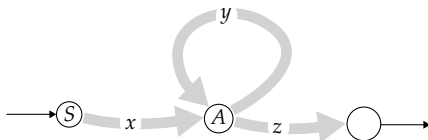
Tyvärr ställer det sig inte helt självklart hur denna bevisidé skulle kunna överföras på ett fruktbart sätt till *sammanhangsfria* språk eftersom en PDA:s agerande i ett tillstånd *inte* enbart styrs av vad den har att konsumera från input-täpn, utan också av vad som finns på stacken.

Märkligt nog kan emellertid resonemanget generaliseras till de sammanhangsfria språken på ett mycket naturligt sätt genom att tala i termer av produktion istället för konsumtion.

Reguljära pumpsatsen i termer av produktion

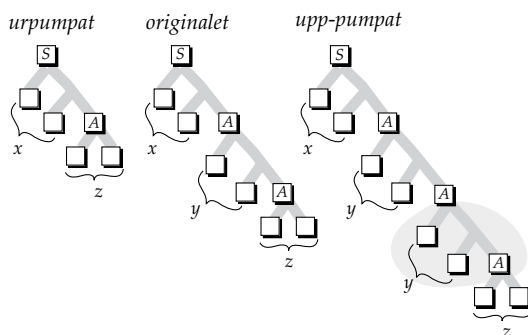
Om en DFA accepterar en sträng w innehållande lika många tecken som det finns tillstånd (eller flera) så tvingas den – under konsumtionen av någon delsträng y av w – att gå runt i en slinga där den besöker något tillstånd A minst två gånger innan den slutligen går i acceptans. (Se figuren nedanför och även sid 61.) Därför driver xy^nz , för varje $n \in \mathbb{N}$, DFA:n till acceptans via n genomlöpningar i samma slinga.

y driver
DFA:n från A
runt i en
slinga tillbaka
till A .



Pumpning av
ett produktionsträd.

Ovanstående DFA:s konsumtion av $w = xy^n z$ motsvaras av en reguljär grammatiks produktion av samma sträng (se sid 73). En trädbeskrivning av nämnda produktion presenteras i figuren till höger. Notera att *en* rundtur i DFA:ns slinga motsvaras i trädet av produktionen $A \Rightarrow^* yA$. Och *upprepade* rundturer i slingan motsvaras av att låta A producera yA *upprepade* gånger. Se det upp-pumpade trädet! DFA:ns konsumtion av $w = xz$ (noll rundturer i slingan), representeras av det urpumpade trädet.



■ Pumpsatsen för sammanhangsfria språk

Här följer nu en pumpsats för de sammanhangsfria språken, med ett bevis som föres just i termer av produktion.

- ✧ SATS 3.2 (*Pumpsatsen för CFL*) Antag att L är ett sammanhangsfritt språk med oändligt många strängar. Då finns det ett tal K sådant att om $w \in L$ är av längd minst K så innehåller w något block vxy av längd högst K som kan pumpas på följande sätt:
 w kan skrivas $w = uvxyz$ med $vy \neq \epsilon$,
 och $uxz, uvxyz, uv^2xy^2z, uv^3xy^3z, \dots \in L$

BEVIS: Först konstaterar vi att om G är en sammanhangsfri grammatik som beskriver L , så finns det i princip två faktorer som begränsar möjligheten att bygga ett produktionsträd med *många* löv, dvs producera en *lång* sträng $w \in L$, utan att använda någon icketerminerande symbol flera gånger:

Den ena faktorn är *tillgången* på icketerminerande symboler.

Den andra är *längden* hos produktionsreglernas högerled (som ju bildar "avkomor" till produktionsträdets noder, och därmed avgränsar trädets tillväxt på bredden).

För att tala mer precist, säg att m är längden av det längsta högerledet hos G :s produktionsregler. Då har produktionsträdets noder *högst* m söner (ty sönerna till en nod A är ju tecknen i högerledet hos en produktionsregel $A \rightarrow \dots$).

Speciellt har rotenoden i ett produktionsträd högst m söner, vilka i sin tur (var och en) har högst m söner, varför trädets 2:a nivå nedanför roten innehåller högst $m \cdot m = m^2$ noder. På motsvarande sätt inne-

†. Med icketerminerande nod i trädet menar vi en nod som inte är ett löv. Tänk på att i sådana noder bor icketerminerande symboler.

$$vy \neq \varepsilon, \quad (27)$$

$$\text{och varför sönderdelningen } w = uvxyz \text{ kan göras så att} \\ \text{längden av pumpblocket } vxy \text{ är högst } K = m^{N+1}. \quad (28)$$

vxy kallar vi för pumpblock. Inuti detta block finns pumpdelarna v och y .

Vi börjar med (27):

Låt oss först konstatera att varje "värdelös" produktion $A \Rightarrow^* A$ som eventuellt förekommer kan *klippas bort* ur trädet utan att slutprodukten – strängen w – ändras. Vi rensar därför trädet från alla värdelösa produktioner. Observera att sådan rensning inte ändrar på det faktum att trädet måste innehålla en väg som repeterar någon icketerminerande symbol. Ty rensningen ändrar icke trädets slutprodukt w , speciellt inte w 's längd som var orsaken till denna repetition.

Nu kan vi konstatera att (27) är sann för det träd som är rensat från värdelösa produktioner. Ty i annat fall (om $vy = \varepsilon$), så är $v = y = \varepsilon$. Och då innehåller trädet den värdelösa produktionen $A \Rightarrow^* vAy$.

Inför beviset av (28) ber jag Dig tänka på följande. Valet av de fem delarna u, v, x, y, z i sönderdelningen av w beror på

- vilken symbolrepeteterande *väg* man har valt att markera (det kan ju finnas flera vägar att välja på),
- vilken repeterande *symbol* som man valt att markera (det kan finnas flera symboler som repeteras),
- vilka *förekomster* av denna symbol längs vägen man har valt att markera (symbolen kan finnas i flera noder än två).

I syfte att bevisa (28) låter vi den markerade vägen vara en *längsta väg* av alla vägar uppifrån och ned. (En sådan längsta väg måste givetvis repetera någon icketerminerande symbol. Ty då trädet har någon väg som besöker flera noder än vad det finns icketerminerande symboler, måste ju en längsta väg besöka minst så många noder.)

Vidare, låter vi de två markerade A :na vara de *sista förekomsterna* (som man träffar på när man vandrar uppifrån och nedåt längs vägen) av en repeterande symbol. Dvs så att när denna längsta väg går igenom i omvänd riktning (nerifrån och upp) ingen symbol påträffas flera gånger innan dessa två A :n påträffas.

Med dessa val av väg, symbol och symbolförekomster är (28) sann. Ty, den del av vägen som går från det övre av de två A :na ned till ett löv (lövet är ett tecken i strängen vxy) har högst $N + 1$ icketerminerande noder. Annars skulle nämligen endera A repeteras flera gånger än två längs denna del av vägen, eller så skulle någon annan symbol repeteras, vilket strider mot vårt val av symbol och vårt val av symbolförekomster längs vägen. Vidare kan ingen annan väg från övre A och ned till något löv (lövet ligger i strängen vxy) ha fler noder än vår markerade väg. Ty en sådan väg skulle om den fortsattes uppåt till S (längs samma väg som vår markerade väg) bli en längre väg än vår markerade vilket strider mot att denna är en längsta väg.

Av det faktum att ingen väg från övre A och nedåt har flera icketerminerande noder än $N + 1$ följer nu av (26) på sidan 95 att längden av

strängen vxy är högst m^{N+1} . Därmed har vi bevisat (28). \square

EXEMPEL 3.14 Låt L vara språket av udda palindromsträngar över $\{a, b\}$, och betrakta $w = ababa$ som tillhör L . Då w s b -förekomster pumpas får man $aaa, abbabba, abbbabba, \dots$ som också är palindromsträngar av udda längd. Notera också att *varje* sträng i L av längd minst 3 kan pumpas på motsvarande sätt, dvs med pumpblock vxy av typ aaa, aba, bab eller bbb av längd 3 placerade mitt i strängen. För det betraktade palindromspråket är således $K = 3$ ett K som i pumpsatsen. Lägg förresten märke till att $w = ababa$ inte kan pumpas i något annat 3-block än det nämnda, tvärt emot vad som gäller vid reguljär pumpning.

EXEMPEL 3.15 Betrakta språket $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$.

L^\dagger är *ej* sammanhangsfritt. Ty i så fall skulle det finnas ett tal K sådant att alla strängar i L av längd $\geq K$ kunde pumpas som i pumpsatsen utan att falla ur L . T ex skulle $a^K b^K c^K$ kunna pumpas. Och pumpningen skulle kunna utföras i något pumpblock av längd $\leq K$. Men för att den sålunda pumpade strängen skall ligga kvar i L efter pumpning måste samtliga tre tecken a, b, c påverkas av pumpningen, vilket är omöjligt med ett pumpblock som inte är längre än K .

EXEMPEL 3.16 Inte heller $L = \{uu \mid u \in \{a, b\}^*\}$ är sammanhangsfritt.

Om L vore sammanhangsfritt skulle, för något K , varje sträng i L av längd $\geq K$ kunna pumpas utan att falla ur L .

$w = a^K b^K a^K b^K$ är en sådan sträng. Men vi skall strax se att w inte uppför sig på "utlovat" sätt. Vår slutsats blir därefter att L inte är sammanhangsfritt.

Visserligen kan pumpning i vänster halva av w balanseras med pumpning i höger halva på ett sätt så att w efter pumpning fortfarande har två identiska halvor, dvs utan att w faller ur L . (Låt t ex de understrukna delarna i $a^{K-1} \underline{a} b^K \underline{a} a^{K-1} b^K$ vara pumpdelar.) Detta faktum bevisar naturligtvis inte att L är sammanhangsfritt[‡], blott att w kan pumpas utan att falla ur L om de två pumpdelarna väljs på nämnda sätt åtskilda från varandra med K tecken.

Pumpsatsen utlovar emellertid att pumpdelarna skall kunna väljas åtskilda från varandra med högst K tecken. Och det löftet kommer att visa sig vara falskt. Se nedanför.

Om pumpdelarna ligger i en delsträng av längd högst K inuti vänster halva, så kommer urpumpning att leda till att mittpunkten förskjuts till höger, dvs in i den gamla högerhalvan. Och om det överhuvudtaget finns någon mittpunkt efter urpumpning (jämnt antal teck-

†. Vissa editorer kodar understrukna ord som strängar byggda av tre lika långa delar. T ex så att DU representeras av " $DU \leftarrow _ _$ ". Därför brukar man säga att $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ står modell för "understrukna ord".

‡. Pumpsatsen kan **aldrig** användas för att bevisa att ett språk är sammanhangsfritt. Varför?

en i nya w krävs för detta) så slutar vänster halva med a , medan höger halva slutar med b , och då har den pumpade w fallit ur L .

Om pumpdelarna ligger i höger halva så faller w av motsvarande skäl ur L vid urpumpning.

Slutligen, om pumpdelarna innehåller tecken från w :s båda halvor, så kommer urpumpning att få följande ödesdigra konsekvens. Om mittpunkten ligger kvar, så har vänster halvans slut färre b än vad höger halva har i slutet (och vänster halvans början har flera a än vad som finns i höger halvans början). Om mittpunkten har förskjutits åt ena eller andra hållet, så slutar de två halvorna med olika tecken eller börjar med olika tecken.

Alltså, oavsett var ett pumpblock av längd $\leq K$ placeras, leder urpumpning ut ur L . Detta bevisar bristen på sammanhangsfrihet.

✂ **TEST 3.6** a) Betrakta strängarna över alfabetet $\{a, b, c\}$ som har flera a -förekomster än b -förekomster, och flera b -förekomster än c -förekomster. T ex är $aaabb$, $aaabbc$ och $ababac$ sådana strängar. Bevisa med pumpning att strängarna ifråga inte bildar ett sammanhangsfritt språk.

b) Låt tecknen i alfabetet $\Sigma = \{v, h, u, n\}$ representera de fyra riktningarna vänster, höger, uppåt, nedåt. En sträng över Σ kan då ses som en vägbeskrivning. T ex så att $uuhnnv$ beskriver "två steg upp, ett steg till höger, två steg nedåt, ett steg till vänster".

Visa att om L är mängden av alla strängar över Σ som likt $uuhnnv$ (eller t ex likt $uuhnnvvnhu$) beskriver vägar som leder tillbaka till ursprungspunkten, så är L inte sammanhangsfritt.

3.7 Deterministiska pushdownautomater

Med en *deterministisk* PDA (DPDA) menar vi en PDA som i *varje läge* måste agera på *ett bestämt* sätt, dvs som med en given sträng på stacken och en given sträng kvar att konsumera från input-strängen *aldrig* kan välja mellan *flera olika* tillståndsovergångar.

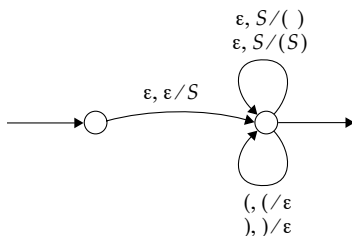
Även om de flesta av våra tidigare exempel på PDA:er har varit ickedeterministiska, så har vi faktiskt mött en deterministisk PDA – nämligen en som accepterar de enkelt nästlade parentessträngarna (se EXEMPEL 3.7 på sidan 83). Vi skall här tillverka ännu en DPDA för de enkelt nästlade parentessträngarna. Syftet är att illustrera en viss metod med vilken man ibland kan tillverka en DPDA för ett språk beskrivet av en CFG.

Slutligen skall vi se att det deterministiska kravet sätter ned pushdownautomaternas kompetens – att de deterministiska PDA:erna inte förmår känna igen (acceptera) alla sammanhangsfria språk.

Nedan ser Du en välbekant PDA (från EXEMPEL 3.11 på sidan 88) som känner igen de enkelt nästlade parentessträngarna. Den är *ickedeterministisk* eftersom den i det högra tillståndet har två möjliga till-

ståndsövergångar i ett läge då S ligger på stackens topp.

En *top-down* parser som känner igen de enkelt nästlade parentessträngarna $\{ (), (()), \dots \}$ genom att härma $S \rightarrow (S) \mid ()$.

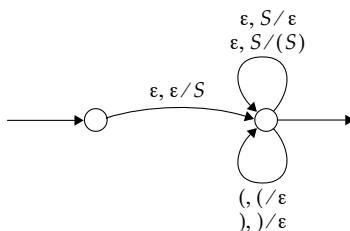


FIGUR 3.3

Du förstår nog att ett "korrekt" agerande av denna parser är att välja den övergång som härmar $S \rightarrow (S)$ så länge inputtapens läshuvud har "(" framför sig (såvida S ligger på stacktoppen förstås), och den som härmar basfallsproduktionen $S \rightarrow ()$ först när läshuvudet har "(" framför sig. Ty endast på detta sätt kan parsern producera något på stacktoppen som *stämmer överens* med vad som finns på inputtapen. Av denna insikt anar vi att bara top-down parsern hade förhandsinformation om vilka *två tecken* som ligger *framför* läshuvudet, så skulle den slippa "gissa" vilken av de två regelhärmande övergångarna som är korrekt. Och därvid skulle ickedeterminismen kunna elimineras. Vi skall snart presentera en top-down parser som har sådan förhandsinformation inbyggd.

Men först en enklare variant på samma tema ...

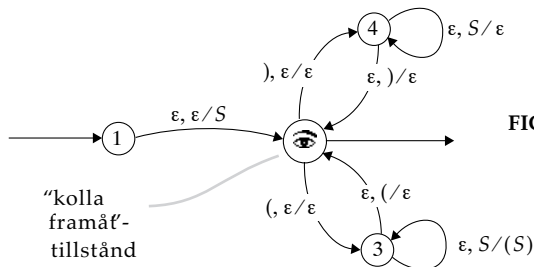
En ickedeterministisk top-down parser för $\{ \epsilon, (), (()), \dots \}$ som härmar reglerna $S \rightarrow (S) \mid \epsilon$



FIGUR 3.4

Top-down parsern som härmar $S \rightarrow (S) \mid \epsilon$ (se FIGUR 3.4) behöver bara förhandsinformation om *det första tecknet framför* läshuvudet: Om detta tecken är en vänsterparentes är det korrekt att härma $S \rightarrow (S)$, och om det är en högerparentes skall istället $S \rightarrow \epsilon$ härmas. Nästa PDA (i FIGUR 3.4) skaffar sig förhandsinformation om detta tecken genom att (i "kolla framåt"-tillståndet) helt sonika konsumera tecknet ifråga. De två regelhärmande övergångarna görs sedan i tillstånden 3 respektive 4. Och på tillbakavägen därifrån till "kolla framåt"-tillståndet utförs den kvitterande avplockningen från stacken.

En deterministisk top-down parser för $\{\epsilon, (,), (()), \dots\}$ som genom att "kolla framåt" ett tecken eliminerar ickedeterminismen i FIGUR 3.4.

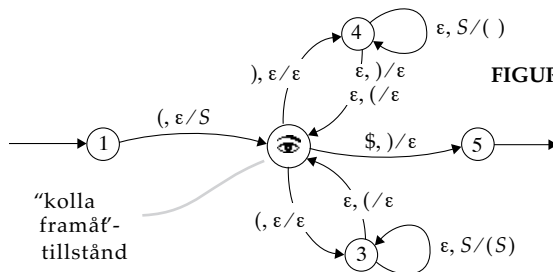


FIGUR 3.5

	tillstånd	input	stacken	
	1	(())	ϵ	
	2	(())	S	
har läst "(" ...	3	())	S	... och S ligger på stacktoppen
	3	())	(S)	
	2	())	S	
har läst "(" igen ...	3))	S	... och S ligger på stacktoppen
	3))	(S))	
	2))	S))	
har läst ")" ...	4)	S))	... och S ligger på stacktoppen
	4)))	
	2))	
har läst ")" igen ...	4	ϵ)	... och ")" ligger på stacktoppen
	2	ϵ	ϵ	

Åter till top-down parsern i FIGUR 3.3 på sidan 99, den som skulle behöva se *två* tecken framåt på inputtapen för att korrekt kunna välja regelhärmande övergång. Kompetens att *se två tecken framåt* kan inte nås på annat sätt än att ändra agerandet för PDA:n så att dess teckenkonsumtion kommer att ligga *två* tecken före valet av regelhärmande övergång, se FIGUR 3.6 nedan.

En deterministisk top-down parser för $\{(, (()), ((()), \dots)\{\$, \}$ som "kollar framåt" två tecken ...



FIGUR 3.6

Ovanstående PDA konsumerar en vänsterparentes redan *på väg* till "kolla framåt"-tillståndet, och eftersom den sedan konsumerar *ytterligare* ett tecken på väg till något av de regelhärmande tillstånden så har den ju sett två tecken framåt när den skall till att göra en regelhärmande övergång. En konsekvens av denna "två-tecken-i-förväg-konsumtion" blir att den kvitterande avplockningen från stacktoppen (som avser att kvittera den senaste teckenkonsumtion från inputs-trängen) kommer att ha en avplockning ogjord i slutänden (om en korrekt parentessträng har behandlats). På stacken ligger nämligen en överbliven högerparentes – den som motsvarar den inledande vänsterparentesen. Därför avslutar PDA:n en ändmarkerande symbol \$ som måste placeras i slutet på varje inputsträng *w* innan mumsandet på *w* sätts igång. När PDA:n har satt i sig hela *w* kommer därvid läshuvudet att peka på \$, och den överblivna högerparentesen på stacktoppen kan kvitteras mot just \$. Här följer en provkörning.

	tillstånd	input	stacken	
	1	(()\$	ε	
	2	()\$	S	
har nu läst "("	3)\$	S	... och S ligger på stacktoppen
	3)\$	(S)	
har nu läst "("	2)\$	S)	
	4)\$	S)	... och S ligger på stacktoppen
	4)\$	()	
	2)\$)	
har läst ")"	4	\$)	... och ")" ligger på stacktoppen
läser \$	2	\$)	... och ")" ligger på stacktoppen
	5	ε	ε	

Lägg också märke till att PDA:n hänger sig för sådana parentessträngar som börjar med högerparenteser, och att den får skräp kvar på stacken om den bjuds på obalanserade parentessträngar som t ex $()$. \square

Vad beträffar *finita automater* vet vi att ickedeterminism är ekvivalent med deterministism i den mening att varje NFA kan göras om till en DFA som accepterar samma språk (och att varje DFA är en NFA). För pushdownautomater är emellertid motsvarande resultat *inte giltigt*, vilket vi skall visa med ett exempel.

EXEMPEL 3.17 Palindromspråket $L = \{w \in \{a, b\}^* \mid w = w^{rev}\}$ är ett sammanhangsfritt språk (TEST 3.2 på sidan 82), varför man kan konstruera en PDA för L .

Men en DPDA då?

Nej, en DPDA M för L skulle vara tvungen att särskilja *oändligt*

många strängar, t ex $\{aa, aba, abba, abbba, \dots\}$ med hjälp av *olika* tillstånd vilket inte låter sig göras, eftersom den bara får ha ändligt många tillstånd.

Se här: Låt palindromerna x och y vara $x = ab^n a$ och $y = ab^m a$ med $n \neq m$. Om M skulle drivas till *ett och samma* tillstånd p av palindromerna x och y , hur skulle då M kunna komma ihåg *vilken* av x och y som hade konsumerats när M efter en sådan konsumtion står i p ? Med hjälp av *stacken*? Möjligen ..., men eftersom x och y är palindromer (och M antogs vara deterministisk) måste M tömma stacken efter konsumtion av x och y . Om stacken inte redan är tom i tillståndet p , måste M därför tömma stacken endera på stället, eller genom att gå till något annat tillstånd. Man skulle kunna tänka sig att M går till skilda tillstånd och utför sådan tömning om stacken har olika innehåll (efter konsumtion av x respektive y). Men då M har ändligt många tillstånd, kan M *inte* gå till skilda tillstånd för *varje* par av x, y . Dvs det finns något par x, y sådant att M vid konsumtion av x och y tömmer sin stack och går till ett gemensamt tillstånd. Därmed skulle M (p.g.a. sin determinism) drivas vidare från detta tillstånd på *ett bestämt sätt* av strängen $z = b^n a$, vilket strider mot att xz men *inte* yz skall accepteras. \square

3.8 Övningar

3.1 Var och en av följande grammatiker beskriver ett reguljärt språk. Presentera reguljära grammatiker för språken ifråga.

- a) $S \rightarrow A01B$ b) $S \rightarrow 01|001|AAS$ c) $S \rightarrow 0|01|SSS$
 $A \rightarrow \epsilon|0A|1A$ $A \rightarrow \epsilon|01|10$
 $B \rightarrow 1|01|B01|B1$

3.2 Skriv sammanhangsfria grammatiker för

- a) $\{w \in \{a, b\}^* \mid w \text{ är inte en palindrom}\}$,
 b) $\{a^{|x|}xb^{|x|} \mid x \in a^*b^*\}$,
 c) $\{a^m b^n a^{m+n} \mid m, n \text{ naturliga tal}\}$.

3.3 Konstruera både en CFG och en PDA för språket

$$L = \{a^m b^n a^k b^l \mid m + n = k + l \text{ och } m, n, k, l \geq 1\} \text{ över } \{a, b\}.$$

3.4 a) Beskriv alla enkelt nästlade parentessträngar där *två* typer av parenteser används. Ett exempel på en sådan sträng är $\{((\{ \}))\}$.

b) Betrakta enkelt nästlade parentessträngar där en enda typ av

parenteser används, men där *olika* storlek på parenteserna tillåts – dock med förbehållet att varje högerparentes är lika stor som den vänsterparentes den hör ihop med. Ett exempel på en sådan sträng är

$$(((((())))))$$

Antag nu att vi *inte* vill begränsa antalet storlekar. Utan kodning kan man inte gå iland med detta projekt utan att använda ett oändligt alfabet och ett oändligt antal regler:

$$\{ (,), (,), (,), \dots \} \quad S \rightarrow \epsilon | (S) | (S) | (S) | (S) | \dots$$

Men det vill vi inte. Därför väljer vi att koda parentesstorlek enligt tabellen

vänsterparentes	kod	högerparentes	kod
(())
(1())1
(11())11
(111())111
...

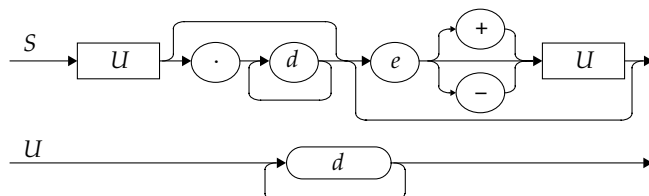
Exempel är 11(1(1111(1111)1)1)11 en kod för *en* sådan här parentessträng. Din uppgift är att konstruera en grammatik för alla (på detta sätt) kodade parentessträngar.

3.5 Låt L var det språk över $\{a, b\}$ som beskrivs av $\{S \rightarrow a|Ab|aSa, A \rightarrow B|\epsilon, B \rightarrow b|Ab\}$.

Ge exempel på

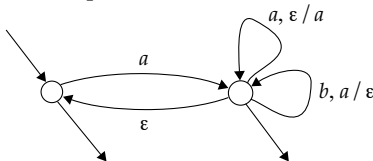
- en CFG för L som saknar regel av typ $C \rightarrow \epsilon$,
- en CFG för L som bara har regler av typ $C \rightarrow DE$, $C \rightarrow a$, $C \rightarrow b$.

3.6 Nedan visas två s.k. syntaxdiagram (vars innebörd torde vara intuitivt klar). Konstruera en CFG och en PDA som beskriver samma sak som diagrammen.



3.7 Konstruera

a) först en CFG för det språk som nedanstående PDA accepterar.



b) sedan både en top-down och en bottom-up parser baserad på CFG:n i a). Visa också med provkörning hur dessa fungerar.

3.8 Här är en CFG G för enkla aritmetiska uttryck

$$S \rightarrow (S + T) \mid T$$

$$T \rightarrow T \cdot F \mid F$$

$$F \rightarrow a \mid S$$

a) Ge en top-down parser och en bottom-up parser som härmar G .

b) Visa med provkörning hur dessa två PDA:er uppför sig för att acceptera $(a + a \cdot a) \cdot a$.

3.9 Visa hur man med hjälp av CFG:er G_1 , G_2 för L_1 och L_2 kan konstruera CFG:er för

a) $L_1 \cup L_2$

b) $L_1 L_2$

c) L_1^*

3.10 Visa hur man med hjälp av PDA:er M_1 , M_2 för L_1 och L_2 kan konstruera PDA:er för ovanstående språk (de i 3.9).**3.11** Formulera och bevisa

a) ett lemma av typen

*Om ett språk L har två strängar x och y med egenskapen ...,
så måste en DPDA för L drivas av x och y
med tömd stack till skilda accepterande tillstånd.* (29)

b) en sats av typen

*Om ett språk L har egenskapen ...,
så finns det ingen DPDA för L .* (30)

LEDNING: Jfr EXEMPEL 3.17 på sidan 101.

Att w är identiskt noll innebär att w har värdet noll oavsett vilket värde x har.

3.12 Visa att aritmetiska uttryck av typen

$\{w \in \{-1, +1, -x, +x\}^* \mid w:s \text{ värde är identiskt noll}\}$
inte bildar ett sammanhangsfritt språk. Jämför med TEST 3.2 f).

3.13 Visa att följande språk *inte* är sammanhangsfria.

- a) $L = \{1^{n^2} \mid n \in \mathbb{N}\}$ b) $L = \{1^p \mid p \text{ är ett primtal}\}$
 c) $\{1^{0+1+\dots+x} \mid x \in \mathbb{N}\}$, d) $\{(a^m b^m)^n \mid m, n \in \mathbb{N}\}$,
 e) $\{(a^n b^n)^n \mid n \in \mathbb{N}\}$.

3.14 Betrakta de enkelt nästlade parentesträngarna med olika stora parenteser (kodade på samma sätt som i **3.4**), *men* så att vi endast tillåter parentessträngar som har större parenteser utanför mindre.

$((()))$ ~~$((()))$~~ ~~$((()))$~~

Visa att det sålunda beskrivna parentesspråket *inte* är sammanhangsfritt.

3.15 Låt L och G_1, G_2 vara som i EXEMPEL 3.6 på sidan 79. Bevisa att $L = L(G_1)$ och att $\bar{L} = L(G_2)$.

3.16 Om Σ är ett alfabet med minst två tecken och x, y, z är tre strängar i Σ^* så är

$$L = \{xuyuz \mid u \in \Sigma^*\}$$

inte sammanhangsfritt för något enda val av x, y, z . Bevisa detta (med pumpning) för det fall att $\Sigma = \{a, b\}$.

⌘ Anm.3.8 Ett programmeringsspråk som tillåter *godtyckligt långa variabelnamn* men kräver att *variabler deklarerar* innan de används, tvingas innehålla godtyckligt långa strängar som de i ovanstående L (och är därmed *icke* sammanhangsfritt). Här är ett exempel från språket C:

$$\text{main()}\{\text{int } u; u = 1\} \quad (31)$$

dvs

$$\text{main()}\{\text{int}\square u; u = 1\} \quad (32)$$

där "blanktecken" representeras av \square .

Strängen (32) är således av typen $\alpha\beta u\gamma$ med

$$x = \text{"main()}\{\text{int}\square", \quad y = ";", \quad z = "=1\}"$$

Restriktionsfria språk och Turingmaskiner

Restriktionsfria språk.....	106
Turingmaskiner	109
Att acceptera respektive avgöra.....	123
Att beräkna funktioner med Turingmaskiner.....	125
Turingmaskiner är ekvivalenta med restriktionsfria grammatiker.....	129
En universell Turingmaskin	133
Ekvivalenta varianter av Turingmaskiner.....	135
Ickedeterminism.....	138
En två-stacks-PDA har samma förmåga som en TM	139
Övningar.....	140

Noam Chomsky presenterade 1959 fyra typer av formella språk såsom teoretiska modeller för naturligt språk. De fyra typerna bildar en hierarki av alltmer restriktiva grammatiker:

	Typ 0 (restriktionsfri)	Typ 1 (CSG) [†]	Typ 2 (CFG)	Typ 3 (reguljär)
regler	$\alpha \rightarrow \beta$	$\alpha \rightarrow \beta$	$A \rightarrow \beta$	$A \rightarrow uB \mid v$
krav på reglerna	α, β godtyckliga, dock så att α innehåller någon icke-terminerande symbol.	Som typ 0 men β minst lika lång som α . (Dvs ingen förkortningsregel.)	β godtycklig sträng, A icketerminerande symbol.	u, v godtyckliga strängar av terminerande symboler och A, B icketerminerande symboler.

†. Context Sensitive Grammar (sammanhangskänslig). Denna typ har ännu inte väckt något större intresse inom datalogin. Notera förresten att utan förkortningsregler kan inte tom sträng produceras.

4.1 Restriktionsfria språk

En produktionsregel av typ 0 kallar vi för en *restriktionsfri* regel. Den har en *godtycklig* sträng av terminerande och/eller icketerminerande symboler i högerledet såväl som i vänsterledet. Vänsterledet måste emellertid innehålla någon icketerminerande symbol[†]. En grammatik vars regler är av sådant allmänt slag kallas (förstås) för *restriktionsfri*. Och det språk som en restriktionsfri grammatik beskriver sägs vara *restriktionsfritt*.

†. Om man från tom sträng eller från enbart terminerande symboler tillåts producera något – skulle man vid strängproduktion inte kunna se på arbetssträngen om produktionen var färdig eller ej.

DEFINITION En grammatik G med restriktionsfria produktionsregler sägs vara restriktionsfri, och ett språk L sådant att $L = L(G)$ för någon restriktionsfri grammatik G , sägs vara restriktionsfritt.

Varje sammanhangsfritt (och därmed varje reguljärt) språk är restriktionsfritt, eftersom en restriktionsfri grammatik får se ut "hur som helst". Vidare är det lätt att konstruera (se EXEMPEL 4.1 nedan) en restriktionsfri grammatik för $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ som vi känner igen (från EXEMPEL 3.15 på sidan 97) som ett icke sammanhangsfritt språk. Således bildar de restriktionsfria språken en *strikt* större mängd än de sammanhangsfria språken.

Sammanhang
(se även sid 78)

I och med utvidgningen från sammanhangsfria till restriktionsfria språk bereds vi möjlighet att uttrycka "sammanhang". Ty med en produktionsregel vars vänsterled innehåller två eller flera symboler kan man formulera "att en icketerminerande symbol kan producera något (bara) om den har vissa symboler i sin närmsta omgivning", dvs om den står i ett visst sammanhang. (Ex.vis $bNc \rightarrow bc$ som uttrycker att N kan suddas bort om N står mellan b och c .)

EXEMPEL 4.1 Språket $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ beskrivs av följande regler:

regler	kommentar
$S \rightarrow \varepsilon \mid abNSc$	producerar $(abN)^n c^n$
$bNa \rightarrow abN$	transporterar bN till höger förbi a
$bNb \rightarrow bbN$	transporterar bN till höger förbi b
$bNc \rightarrow bc$	suddar N mellan b och c

Som illustration får Du strängen $aabbcc$ producerad:

$$\begin{aligned}
 S &\Rightarrow abNSc \\
 &\quad \Delta \\
 &\Rightarrow abNabNScc \\
 &\quad \Delta \\
 &\Rightarrow abNabNcc \\
 &\quad \Delta\Delta\Delta \\
 &\Rightarrow abNabcc \\
 &\quad \Delta\Delta\Delta \\
 &\Rightarrow aabNbcc \\
 &\quad \Delta\Delta\Delta \\
 &\Rightarrow aabbNcc \\
 &\quad \Delta \\
 &\Rightarrow aabbc
 \end{aligned}$$

EXEMPEL 4.2 Följande regler (med fyra icketerminerande symboler $S, D, [,]$) beskriver $\{1, 1^2, 1^4, 1^8, 1^{16}, 1^{32}, \dots\}$.

regler	kommentar
$S \rightarrow [1]$	
$[\rightarrow [D$	med dessa två produceras $[DD\dots D1]$
$D1 \rightarrow 11D$	när D går till höger förbi 1 tillverkas en extra 1:a
$D] \rightarrow]$	suddar D om D står omedelbart till vänster om $]$.
$[\rightarrow \varepsilon$	suddar $[$
$] \rightarrow \varepsilon$	suddar $]$

T ex produceras strängen $11111111 = 1^8$ sålunda:

$$\begin{aligned}
 S &\Rightarrow [1] \Rightarrow [D1] \Rightarrow^* [DDDD1] \\
 &\quad \Delta \quad \Delta \quad \Delta \Delta \\
 &\quad \Rightarrow [DD11D] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [D11D1D] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [D1111DD] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [11D111DD] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [1111D11DD] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [11111D1DD] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [1111111DDDD] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [11111111DD] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [11111111D] \\
 &\quad \quad \Delta \Delta \\
 &\quad \Rightarrow [11111111] \\
 &\quad \quad \Delta \\
 &\quad \Rightarrow [11111111] \\
 &\quad \quad \Delta \\
 &\quad \Rightarrow 11111111
 \end{aligned}$$

⌘ Anm.4.1 Grammatiken i EXEMPEL 4.2 kan *hänga sig* under ett försök att producera. T ex

$$S \Rightarrow [1] \Rightarrow [D1] \Rightarrow [D1 \Rightarrow D1$$

$\Delta \quad \Delta \quad \Delta$

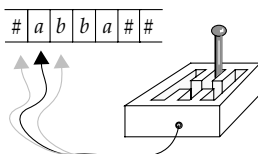
som *inte kan* komma vidare ...

✂ TEST 4.1

- a) Regeln $bNc \rightarrow bc$ i EXEMPEL 4.1 används enbart för att sudda ut N . Kanske man kunde ersätta denna regel med den enklare suddningsregeln $N \rightarrow \epsilon$? Vad säger Du?
- b) Presentera en restriktionsfri grammatik för
- $$L = \{xx \mid x \in \{a, b\}^*\}.$$

4.2 Turingmaskiner

Vi har redan sett att finita automater och pushdownautomater är specialiserade på typ 3- respektive typ 2- språk. Nu skall vi presentera en sorts automater, *Turingmaskiner*, kompetenta att känna igen språk av typ 0.



Abstrakta maskiner såsom finita automater och pushdownautomater var okända begrepp när *Alan Turing*, formulerade sitt koncept för beskrivning av "det algoritmiska". Året var 1936 och konceptet, som några år senare döptes av Gödel till *Turingmaskin*, kom sedermera att bli den teoretiska prototypen för de första datorerna. Finita automater och pushdownautomater som är senare påhitt (från 1950 och 1960-talen) är inget annat än Turingmaskiner som frångått vissa befogenheter och därför har mindre beräkningsförmåga.

En Turingmaskin (TM) har – precis som en PDA eller en FA – en *kontrollmekanism* med *ändligt* många tillstånd, och en *oändlig tape* från vilken inputsträngen läses. Men medan en PDA tvingas föra sina minnesanteckningar på en *annan* tape och enbart enligt principen sist-in-först-ut, så använder en TM inputtappen för minnesanteckningar – och detta utan restriktioner.

Dvs en TM får skriva och läsa var som helst på sin inputtape, något som möjliggörs genom att dess läs- och skriv-huvud kan förflytta sig i tapens båda riktningar. På detta sätt kan en TM (precis som en PDA) sägas ha *oändligt stort minne* trots sin ändliga tillståndsmängd, eftersom den har möjlighet att göra godtyckligt långa minnesanteckningar på sin oändliga tape. Dessutom kan en Turingmaskin, tack vare sitt restriktionsfria skrivande och läsande på tapen, göra mer komplicerade minnesanteckningar än en PDA och därigenom känna igen språk som ligger utom räckhåll för PDA:er.

Turing hävdade 1936 att varje mekanisk beräkning kunde utföras

"Restriktionsfritt" klottrande på tapen.

Turings tes

av en TM, och ännu har inget framkommit som motsäger denna tes (Turings tes). Som stöd för sin tes gav Turing bl. a. en mycket trovärdig motivering av hur det mekaniska agerandet hos en person sysselsatt med beräkningar kan sönderdelas i små enheter som motsvarar just en Turingmaskins atomära agerande.



Alan M. Turing, (1912 – 1954) född i London och verksam huvudsakligen i Cambridge inledde sina matematiska studier inom sannolikhetsteorin, men kom – inspirerad av Gödels sensationella ofullständighetssats – att intressera sig mer för matematikens grunder, och speciellt för Hilberts berömda ENTSCHEIDUNGSPROBLEM (se sid 3 och sid 117). För att lösa detta problem skapade Turing en abstrakt maskin – *Turingmaskinen* – som modell för det algoritmiska. 1936-1937 publicerade han sin maskin och (med dess hjälp) ett negativt svar på Hilberts problem (se sid 117).

På nästkommande sidor återges några valda avsnitt ur Turings banbrytande artikel[†]. Notera att där Turing skriver "computer" menar han en *mänsklig* "beräknare" sysselsatt med att beräkna decimaldelen i ett reellt tal mellan 0 och 1. Turingmaskinens tape representerar därvid papperen som beräkningarna antecknas på och tillstånden motsvarar beräknarens mentala tillstånd.

[†]. PROCEEDINGS of THE LONDON MATHEMATICAL SOCIETY, Vol. 42, 1936.

230

A.M.TURING

[Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936. — Read 12 November, 1936.]

The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π , e , etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel[†]. These results

†. Gödel, “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I”, *Monatshefte Math. Phys.*, 38 (1931), 173–198.

1936.]

ON COMPUTABLE NUMBERS.

231

have valuable applications. In particular, it is shown (§11) that the Hilbertian Entscheidungsproblem can have no solution.

In a recent paper Alonzo Church[†] has introduced an idea of “effective calculability”, which is equivalent to my “computability”, but is very differently defined. Church also reaches similar conclusions about the Entscheidungsproblem[‡]. The proof of equivalence between “computability” and “effective calculability” is outlined in an appendix to the present paper.

1. *Computing machines.*

We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. No real attempt will be made to justify the definitions given until we reach §9. For the present I shall only say that the justification lies in the fact that the human memory is necessarily limited.

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q_1, q_2, \dots, q_R which will be called “ m -configurations”. The machine is supplied with a “tape” (the analogue of paper) running through it, and divided into sections (called “squares”) each capable of bearing a “symbol”. At any moment there is just one square, say the r -th, bearing the symbol $\mathfrak{S}(r)$ which is “in the machine”. We may call this square the “scanned square”. The symbol on the scanned square may be called the “scanned symbol”. The “scanned symbol” is the only one of which the machine is, so to speak, “directly aware”. However, by altering its m -configuration the machine can effectively remember some of the symbols which it has “seen” (scanned) previously. The possible behaviour of the machine at any moment is determined by the m -configuration q_n and the scanned symbol $\mathfrak{S}(r)$. This pair $q_n, \mathfrak{S}(r)$ will be called the “configuration”: thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (*i.e.* bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m -configuration may be changed. Some of the symbols written down

[†]. Alonzo Church, “An unsolvable problem of elementary number theory”, *American J. of Math.*, 58 (1936), 345–363.

[‡]. Alonzo Church, “A note on the Entscheidungsproblem”, *J. of Symbolic Logic*, 1 (1936), 40–41.

232

A.M.TURING

[Nov. 12,

will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to “assist the memory”. It will only be these rough notes which will be liable to erasure. It is my contention that these operations include all those which are used in the computation of a number. The defence of this contention will be easier when the theory of the machines is familiar to the reader. In the next section I therefore proceed with the development of the theory and assume that it is understood what is meant by “machine”, “tape”, “scanned”, etc.

2. Definitions.

Automatic machines.

If at each stage the motion of a machine (in the sense of §1) is *completely* determined by the configuration, we shall call the machine an “automatic machine” (or *a-machine*).

For some purposes we might use machines (choice machines or *c-machines*) whose motion is only partially determined by the configuration (hence the use of the word “possible” in §1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems. In this paper I deal only with automatic machines, and will therefore often omit the prefix *a-*.

Computing machines.

If an *a-machine* prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. If the machine is supplied with a blank tape and set in motion, starting from the correct initial *m*-configuration, the subsequence of the symbols printed by it which are of the first kind will be called the *sequence computed by the machine*. The real number whose expression as a binary decimal is obtained by prefacing this sequence by a decimal point is called the *number computed by the machine*.

At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the *m*-configuration will be said to describe the *complete configuration* at that stage. The changes of the machine and tape between successive complete configurations will be called the *moves* of the machine.

Nu hoppar vi ett tiotal sidor framåt i Turlings artikel, till ett avsnitt där Turing mer utförligt motiverar maskinens relevans som modell för "det algoritmiska"

Ett ändligt alfabet!

1936.]

ON COMPUTABLE NUMBERS.

249

9. *The extent of the computable numbers.*

No attempt has yet been made to show that the "computable" numbers include all numbers which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is "What are the possible processes which can be carried out in computing a number?"

The arguments which I shall use are of three kinds.

(a) A direct appeal to intuition.

(b) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

(c) Giving examples of large classes of numbers which are computable.

Once it is granted that computable numbers are all "computable", several other propositions of the same character follow. In particular, it follows that, if there is a general process for determining whether a formula of the Hilbert function calculus is provable, then the determination can be carried out by a machine.

I. [Type (a)]. This argument is only an elaboration of the ideas of §1.

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent[†]. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as

[†]. If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leq x \leq 1$, $0 \leq y \leq 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. If these sets are restricted to be measurable, we can define the "distance" between two symbols as the cost of transforming one symbol into the other if the cost of moving unit area of printer's ink unit distance is unity, and there is an infinite supply of ink at $x=2$, $y=0$. With this topology the symbols form a conditionally compact space.

250

A.M.TURING

[Nov. 12,

17 or 9999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 9999999999999999 are the same.

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape. Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always "observed" squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognizable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

In connection with "immediate recognizability", it may be thought that there are other kinds of square which are immediately recognizable. In particular, squares marked by special symbols might be taken as imme-

En ändlig
tillstånds-
mängd!

1936.] ON COMPUTABLE NUMBERS. 251

diately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognise a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find "... hence (applying Theorem 157767733443477) we have...". In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other "immediately recognisable" squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. This idea is developed in III below.

The simple operations must therefore include:

(a) Changes of the symbol on one of the observed squares.

(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(A) A possible change (a) of symbol together with a possible change of state of mind.

(B) A possible change (b) of observed squares, together with a possible change of state of mind.

The operation actually performed is determined, as has been suggested on p. 250, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an " m -configuration" of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares.

Agerandet
vid tillstånd-
övergång

1936.]

ON COMPUTABLE NUMBERS.

259

11. *Application to the Entscheidungsproblem.*

The results of § 8 have some important applications. In particular, they can be used to show that the Hilbert Entscheidungsproblem can have no solution. For the present I shall confine myself to proving this particular theorem. For the formulation of this problem I must refer the reader to Hilbert and Ackermann's *Grundzüge der Theoretischen Logik* (Berlin, 1931), chapter 3.

I propose, therefore, to show that there can be no general process for determining whether a given formula \mathcal{A} of the functional calculus \mathbf{K} is provable, *i.e.* that there can be no machine which, supplied with any one \mathcal{A} of these formulae, will eventually say whether \mathcal{A} is provable.

It should perhaps be remarked that what I shall prove is quite different from the well-known results of Gödel. Gödel has shown that (in the formalism of Principia Mathematica) there are propositions \mathcal{A} such that neither \mathcal{A} nor $\neg \mathcal{A}$ is provable. As a consequence of this, it is shown that no proof of consistency of Principia Mathematica (or of \mathbf{K}) can be given within that formalism. On the other hand, I shall show that there is no general method which tells whether a given formula \mathcal{A} is provable in \mathbf{K} , or, what comes to the same, whether the system consisting of \mathbf{K} with $\neg \mathcal{A}$ adjoined as an extra axiom is consistent.

If the negation of what Gödel has shown had been proved, *i.e.* if, for each \mathcal{A} , either \mathcal{A} or $\neg \mathcal{A}$ is provable, then we should have an immediate solution of the Entscheidungsproblem. For we can invent a machine \mathcal{M} which will prove consecutively all provable formulae. Sooner or later \mathcal{M} will reach either \mathcal{A} or $\neg \mathcal{A}$. If it reaches \mathcal{A} , then we know that \mathcal{A} is provable. If it reaches $\neg \mathcal{A}$, then, since \mathbf{K} is consistent (Hilbert and Ackermann, p. 65), we know that \mathcal{A} is not provable.

Skillnaden
mellan Hilberts
Entscheidungs-
problem och
Gödels ofull-
ständighetssats.

Den Turingmaskin som vi nu mer i detalj skall bekanta oss med är i huvudsak lik Turings ursprungliga maskin.

Tape Vi låter Turingmaskinens *tape* ha oändlig utsträckning både till vänster och till höger.

Starttillstånd och stopptillstånd En Turingmaskins kontrollmekanism har ett *starttillstånd* och ett *stopptillstånd*. Turingmaskinen stannar *ovillkorligt* i stopptillståndet. Accepterande tillstånd i den mening som finita automater och pushdownautomater har, behöver inte Turingmaskiner använda sig av. Ty en Turingmaskin kan ju "rapportera" acceptans eller ickeacceptans genom att skriva ett meddelande därom på tapen.

Atomära agerandet Vid tillståndsövergång passar en Turingmaskin på att agera i följande bemärkelse

- ersätter[†] läst tecken med nytt eller samma tecken eller
- flyttar läshuvudet ett steg (en cell) till vänster eller höger.

Blanktecken De celler på tapen som icke innehåller något tecken sägs vara *blanka*. Eftersom vi vill att en TM skall

- kunna upptäcka att en cell är blank,
- kunna sätta ett tecken så att dess cell blir blank

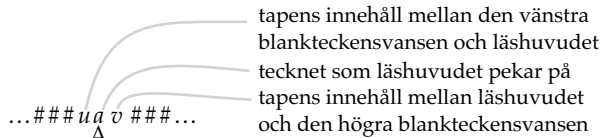
så inför vi en särskild beteckning # (blanktecken) för "innehållet" i en blank cell. Att sätta ett tecken σ blir därmed detsamma som att ersätta σ med #. Och en s.k. blank tape är av typen ... ### ...

Tillräckligt långt till vänster såväl som till höger på tapen finns det alltid en *oändlig blankteckensvans* (oavsett vilken sträng w som är skriven på tapen) ... ## w ### ...

Konfigurationer För att beskriva "nuläget" i Turingmaskinens beräkningar brukar man använda en särskild notation $(q, u \underset{\Delta}{a} v)$ kallad *Turingmaskinens konfiguration*, där

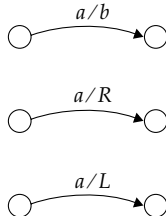
- q är nuvarande tillstånd och
- $u \underset{\Delta}{a} v$ är nuvarande tapekonfiguration

Tape-
konfigurationen



Grafisk beskrivning Finita automater och pushdownautomater kan beskrivas med hjälp av riktade grafer. Detsamma gäller Turingmaskiner:

En Turingmaskins tillståndsövergångar. Jfr avsnittet **atomära agerandet** ovanför.



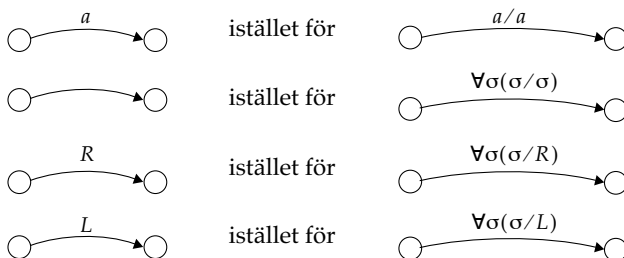
"Om läshuvudet pekar på a , ersätt a med b ."

"Om läshuvudet pekar på a , flytta läshuvudet till höger."

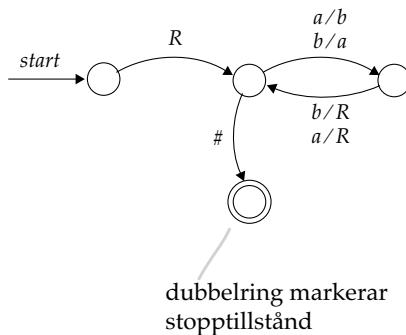
"Om läshuvudet pekar på a , flytta läshuvudet till vänster."

[†]. För denna syssla tänker vi oss att läshuvudet "byter skepnad" till ett skrivhuvud.

⌘ Anm.4.2 Av bekvämlighetsskäl ritar vi



EXEMPEL 4.3 Nedanstående TM “inverterar” varje tecken (byter a mot b och omvänt) under inspektion av tapen i höger riktning, och stannar när $\#$ påträffas. Till höger ser Du en provkörning.



abba#
 Δ
 # abba#
 Δ
 # bbba#
 Δ
 # bbba#
 Δ
 # baba#
 Δ
 # baba#
 Δ
 # baaa#
 Δ
 # baaa#
 Δ
 # baab#
 Δ
 # baab#
 Δ

Efter denna mer eller mindre informella inledning kan det väl vara dags för en mängdteoretisk definition.

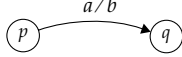
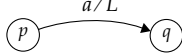
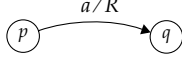
DEFINITION En TM M är en sextupel

$$(Q, \Sigma, \Gamma, \delta, s, h)$$

där

- (i) Q är en ändlig mängd (tillståndsmängd)
- (ii) Σ är en ändlig delmängd (input-alfabet) av Γ sådan att $\# \notin \Sigma$
- (iii) Γ är en ändlig mängd (tape-alfabetet) sådan att $\# \in \Gamma$
- (iv) δ är en funktion (övergångsfunktion) från $(Q - \{h\}) \times \Gamma$ till $Q \times (\Gamma \cup \{L, R\})$
- (v) $s \in Q$ (starttillstånd)
- (vi) $h \in Q$ (stopptillstånd)

⌘ Anm.4.3 Punkt (iv) i definitionen säger att tillståndsövergångarna är av deterministiskt slag (dvs *funktionsstyrda*), och beskriver vilka input-output som kommer ifråga för δ . Här är en svulstigare redogörelse av det senare:

$\delta: \text{input} \rightarrow \text{output}$	kommentar	grafiskt (jfr sid 118)
$(p, a) \rightarrow (q, b)$	hoppas från p till q och ersätter a med b	
$(p, a) \rightarrow (q, L)$	hoppas från p till q och flyttar läshuvudet till vänster	
$(p, a) \rightarrow (q, R)$	hoppas från p till q och flyttar läshuvudet till höger	

Notera också att mängdnotationen $(Q - \{h\}) \times \Gamma$ på andra raden i (iv) uttrycker att M saknar övergångar från stopptillståndet, vilket betyder att M 's agerande är avslutat om M hamnar i stopptillståndet – m.a.o. att M stannar där.

Risk för
hängning!

I varje annat tillstånd än stopptillståndet skall M i princip ha en fullständig övergångsuppsättning (utgång för varje tecken). För att inte göra Turingmaskinernas grafer onödigt klottriga struntar man dock ofta i utgångar till skräptillstånd (dvs tillstånd från vilka stopptillståndet inte kan nås), något som vid körning kan resultera i s.k. "hängning".

■ **Driva** Om $M = (Q, \Sigma, \Gamma, \delta, s, h)$ är en TM, $p, r \in Q$ och $u_{\Delta} q v$ är en tapekonfiguration, så säger man att

informellt ...

► $u_{\Delta} q v$ driver M från p till r (i noll eller flera steg) om givet tapekonfiguration $u_{\Delta} q v$, det finns en följd av (noll eller flera) tillståndsövergångar som gör att M övergår från tillståndet p till r .

formellt ...

► Här är en formell definition av att $u_{\Delta} q v$ driver M från p till r :

- (i) $(p, u_{\Delta} q v)$ driver M från p till p ("inget agerande")
- (ii) $(p, u_{\Delta} q v)$ driver M från p till r om något av följande tre villkor är uppfyllda

1. $\delta(p, a) = (q, b)$ och $(q, u_{\Delta} b v)$ driver M från q till r



2. $\delta(p, a) = (q, L)$ och $(q, \overline{xcav})^u_\Delta$ driver M från q till r



3. $\delta(p, a) = (q, R)$ och $(q, uad\overline{y})^v_\Delta$ driver M från q till r



■ **Acceptera** Om $M = (Q, \Sigma, \Gamma, \delta, s, h)$ är en TM och $w \in \Sigma^*$, så säger man att

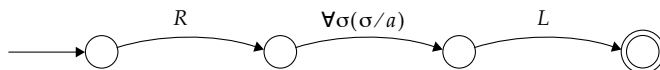
- M accepterar w , om $\# w$ driver M från s till h
- M 's accepterade språk $L(M)$ är $\{w \in \Sigma^* \mid M \text{ accepterar } w\}$

Tre små ... Vi presenterar här tre enkla Turingmaskiner a , L , R som är till god hjälp när man vill konstruera mer komplicerade dito. Lägg märke till att vi döper dem efter deras sysslor.

	TM	namn	syssla
De tre små Turingmaskinerna a, L, R .		a	skriver tecknet "a" i den cell där läshuvudet står
		L	går till vänster ett steg
		R	går till höger ett steg

Seriekoppling Turingmaskiner kan – likt finita automater och pushdownautomater – *seriekopplas*. Den inledande maskinens stopptillstånd byts därvid ut mot nästa maskins starttillstånd. T ex kan de tre små Turingmaskinerna ovan seriekopplas som nedan. Seriekopplingen betecknas förslagsvis RaL

seriekoppling

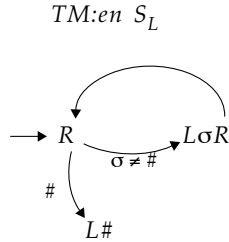


Shiftning Ibland vill man flytta en sträng w ett steg till höger eller

till vänster på tapen. Detta kallar vi *höger-* respektive *vänstershiftning* av strängen w .

Här presenteras en Turingmaskin vars specialitet är *vänstershiftning*.

Vänstershiftaren S_L
(Var är dess stopp-tillstånd?)



Beträffande övergången med beteckningen $\sigma \neq \#$, se Anm.4.4.

dess syssla

Vänstershiftar (ett steg) den sträng som ligger mellan läshuvudet och första blanktecknet till höger (om läshuvudet) inklusive detta blanktecken, varefter läshuvudet ställs till höger om den shiftade strängen.

Notera att det tecken som läshuvudet stod på innan shiftningen skrivs över.

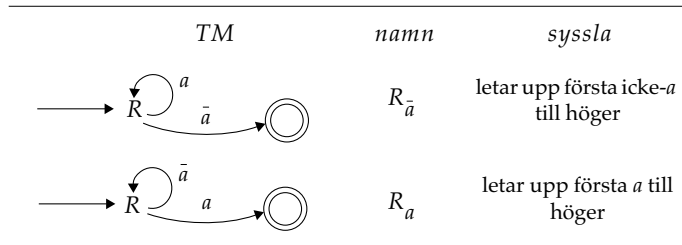
Ex.vis kommer vänstershiftning att omvandla $\underset{\Delta}{a}babba\#$ till $aabba\underset{\Delta}{\#}$.

Höger- och vänstergående teckenletare När en TM inspekterar sin tape är det ofta med avsikt att leta efter

- ett *visst* tecken, t ex a
- något tecken som skiljer sig från ett *visst* tecken, t ex "icke- a ", (betecknad \bar{a} nedan)

Här följer två teckenletare byggda med hjälp av Turingmaskinen R .

Teckenletarna
 $R_{\bar{a}}$ och R_a

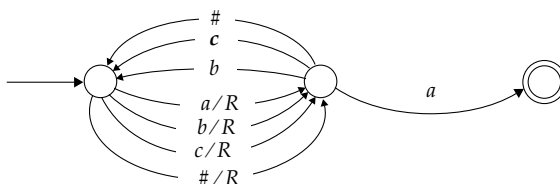


⌘ Anm.4.4 Tänk på att bakom bekväma övergångsbeteckningar som t ex

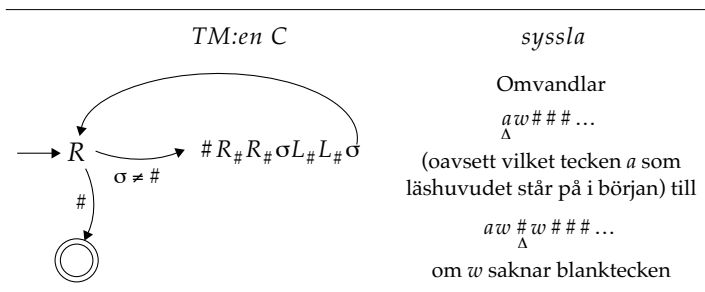


gömmar sig ofta en bunt av övergångar – som om de sätts ut kan göra en graf hopplöst svårläst. Den kompakt betecknade R_a skulle t ex för $\Sigma = \{a, b, c\}$ få nedanstående klottriga utseende ifall man satte ut, inte bara alla övergångar som \bar{a} avser, utan dessutom samtliga övergångar i "delmaskinen" R .

En klottrig
bild av teck-
enletaren R_a



Kopiering Följande TM C kopierar en blankteckenfri sträng mellan läshuvudet och den oändliga blankteckensvansen till höger därom. Kopian läggs efter det första blanktecknet till höger om originalet.



✂ TEST 4.2 Konstruera

- Högershiftaren S_R
- a) S_R den högershiftande motsvarigheten till S_L . Text skall S_R omvandla $ababba\#$ till $ab\#abba\#$.
- b) vänstergående teckenletare $L_{\bar{a}}$ respektive L_a .

■ 4.3 Att acceptera respektive avgöra

Redan i definitionen av en TM och dess agerande (sid 121) har vi formulerat vad som menas med att *acceptera*. Men för jämförelsens skull tar vi det igen – nu tillsammans med en definition av att *avgöra*:

acceptera

DEFINITION En TM M sägs acceptera ett språk L om

$(s, \# \Delta w)$ driver M till stopptillståndet då $w \in L$, och

$(s, \# \Delta w)$ inte driver M till stopptillståndet då $w \notin L$.

avgöra

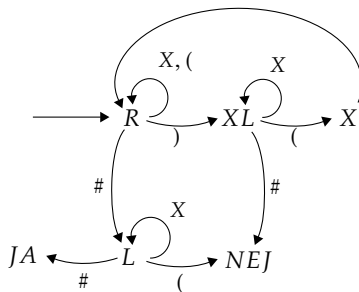
DEFINITION En TM M sägs avgöra ett språk L om

$(s, \# \Delta w)$ driver M till $(h, \# \Delta "ja")$ då $w \in L$, och

$(s, \# \Delta w)$ driver M till $(h, \# \Delta "nej")$ då $w \notin L$.

En TM som
avgör språket
av välbildade
parentes-
strängar.

En Turingmaskin med uppgift att *avgöra* de välbildade parentessträngarna kan därför lämpligen börja med gå till höger på tapen för att leta efter en högerparentes. Om sådan hittas ersätts den med ett kryss, varefter Turingmaskinen vänder på klacken och inleder en sökning i vänsterled efter en motsvarande vänsterparentes. Om sådan hittas ersätts även denna med ett kryss. Detta höger- och vänsterparentesletande upprepas tills endera högerparenteserna eller vänsterparenteserna tar slut. Om vänsterparenteserna tar slut först, dvs om man efter att ha funnit en högerparentes icke hittar någon motsvarande vänsterparentes vet man att parentesuttrycket icke är välformat. Om å andra sidan högerparenteserna sinar först, kan vi icke uttala oss förrän vi undersökt om det eventuellt finns någon överflödig vänsterparentes.



✂ **TEST 4.3** Konstruera dels en TM som accepterar dels en TM som avgör språket över $\{a, b\}$ av strängar som innehåller någon förekomst av *abba*.

4.4 Att beräkna funktioner med Turingmaskiner

Ett predikat är en funktion vars outputmängd består av två element som t ex $\{0, 1\}$, $\{nej, ja\}$, eller $\{falskt, sant\}$.

En Turingmaskin som avgör ett språk returnerar *ja* eller *nej* och beräknar därmed ett *predikat*.

Att beräkna *funktioner* med Turingmaskiner faller sig förstås helt naturligt, eftersom en TM med sin förmåga att läsa och skriva på sin tape både kan läsa input från tapen och skriva output på densamma innan den stannar. Mer formellt:

DEFINITION Givet en funktion F med n argument och en TM M , så sägs M beräkna F om

$(s, \# \underset{\Delta}{x_1} \# x_2 \# \dots \# x_n)$ driver M till $(h, \# \underset{\Delta}{y})$ för varje

(x_1, x_2, \dots, x_n) sådan att $F(x_1, x_2, \dots, x_n) = y$,

och $(s, \# \underset{\Delta}{x_1} \# x_2 \# \dots \# x_n)$ inte driver M till stopptillståndet för

(x_1, x_2, \dots, x_n) där F är odefinierad.

Vid TM-beräkningar av funktioner från \mathbb{N} till \mathbb{N} är det bekvämt att re-

presentera de naturliga talen i 1-systemet. Exempelvis blir *addition* därvid mycket enkel, vilket illustreras i exemplet nedanför.

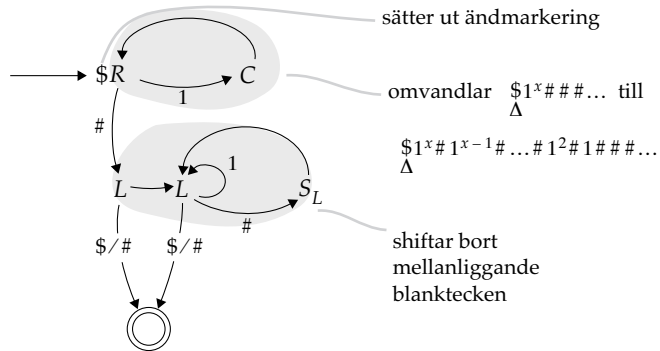
EXEMPEL 4.7 En Turingmaskin med uppdrag att beräkna $x + y$ för naturliga tal x, y representerade i 1-systemet behöver bara shifta bort det blanktecken som åtskiljer argumenten – så att $\#1^x\#1^y$ övergår i $\#1^x1^y$:

En TM som adderar naturliga tal.

$$R_{\#} S_L L_{\#}$$

EXEMPEL 4.8 En Turingmaskin, *Tri*, som returnerar triangeltal:

Turingmaskinen *Tri* beräknar $x + (x - 1) + \dots + 2 + 1$ för $x \in \mathbb{N}$.



Då ovanstående TM körs igång på $\#1^x$ stannar den i konfigurationen $\#1^{x+(x-1)+\dots+2+1}$ dvs den beräknar $Triangel(x) = x + (x - 1) + \dots + 2 + 1$ för $x \in \mathbb{N}$.

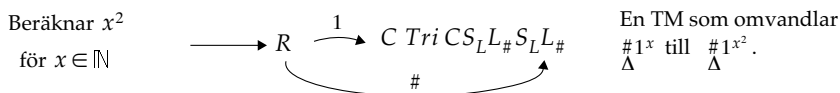
Provkörning:

$\#111\#\#\dots$	
Δ	
$\$111\#\#\dots$	\$
Δ	
$\$111\#\#\dots$	R
Δ	
$\$111\#11\#\dots$	C
Δ	
$\$111\#11\#\dots$	R
Δ	
$\$111\#11\#1\#\dots$	C
Δ	
$\$111\#11\#1\#\dots$	R
Δ	
$\$111\#11\#1\#\dots$	C
Δ	
$\$111\#11\#1\#\#\dots$	R
Δ	
$\$111\#11\#1\#\#\dots$	$LL \curvearrowright^1$
Δ	
$\$111\#111\#\#\dots$	S_L
Δ	
$\$111\#111\#\#\dots$	$L \curvearrowright^1$
Δ	
$\$111111\#\#\#\dots$	S_L
Δ	
$\$111111\#\#\#\dots$	$L \curvearrowright^1$
Δ	
$\#111111\#\#\#\dots$	$\$/\#$

EXEMPEL 4.9 Betrakta följande samband mellan kvadrattal och triangelstal (Försök bevisa dess giltighet!):

$$\begin{aligned} 4 &= 2 + 1 + 1 \\ 9 &= 3 + 2 + 1 + 2 + 1 \\ x^2 &= x + \text{Triangelstal}(x-1) + \text{Triangelstal}(x-1) \end{aligned}$$

Med hjälp av detta samband och Turingmaskinen *Tri* är det förstås lätt att konstruera en Turingmaskin som returnerar kvadratiske naturliga tal, dvs som beräknar $Kvad(x) = x^2$ för $x \in \mathbb{N}$. Figuren nedanför illustrerar detta.



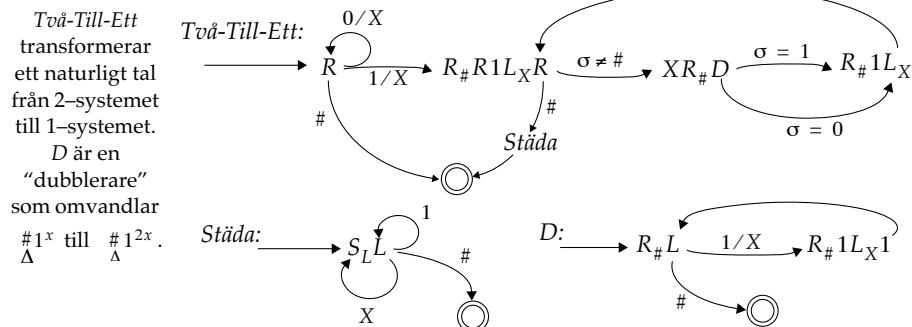
EXEMPEL 4.10 Vi konstruerar här en Turingmaskin *Två-Till-Ett* med kompetens att transformera representationer av naturliga tal från 2-systemet till 1-systemet. Dvs omvandla $\#w$ till $\#1^x$ för $w \in \{0, 1\}^*$ som representerar det naturliga talet x .

Två-Till-Ett skall således beräkna funktionen

$$f(w) = y$$

för $w \in \{0, 1\}^*$ och $y \in \{1\}^*$ sådana att w (i 2-systemet) och y (i 1-systemet) representerar ett och samma naturliga tal x .

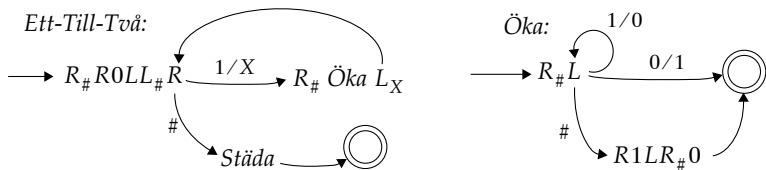
Vår konstruktion av *Två-Till-Ett* bygger på det faktum att om w är en binär representation av ett naturligt tal x , så representeras $2x$ av $w0$ och $2x + 1$ av $w1$. Närmare bestämt undersöker *Två-Till-Ett* sin inputsträng w från vänster till höger och för varje gång som en 1:a eller 0:a påträffas (inledande 0:or oräknade) uppdateras den till höger om w skrivna outputsträngen $y \in \{1\}^*$. Uppdateringen går ut på att dubblera antalet 1:or i y om 0 påträffas i w och att både dubblera antalet 1:or och tillfoga en extra 1:a om 1 påträffas i w . Figuren nedanför avslöjar detaljerna.



Provkörning:	#0101###...	
	Δ #0101###...	R
	Δ #X101###...	0/X
	Δ #X101###...	R
	Δ #XX01###...	1/X
	Δ #XX01#1##...	$R_{\#}R1L_XR$
	Δ #XXX1#1##...	$XR_{\#}$
	Δ #XXX1#11##...	D
	Δ #XXX1#11##...	L_X
	Δ #XXX1#11##...	R
	Δ #XXXX#11##...	$XR_{\#}$
	Δ #XXXX#1111##...	D
	Δ #XXXX#1111##...	$R_{\#}1L_X$
	Δ #XXXX#1111##...	R
	Δ #11111#...	Städa

EXEMPEL 4.11 Följande TM är omvändningen till den i EXEMPEL 4.10 och fungerar så att den börjar med att skriva en 0:a som resultatsträng till höger om inputsträngen. (Mindre än så kan inte resultatet bli.) Sedan går den tecken för tecken igenom inputsträngen (den med 1:or) och för varje påträffad 1:a ilar läshuvudet iväg till höger på tapen och ökar resultatet.

Ett-Till-Två använder sig av Öka, som ökar det binärt representerade talet med en enhet, t ex 10111 till 11000.



✂ **TEST 4.4** Konstruera Turingmaskiner som beräknar nedanstående funktioner från \mathbb{N} till \mathbb{N} (med de naturliga talen representerade i 1-systemet precis som i exempen ovanför):

- $F(x) = x - 1$ för $x > 0$, $F(0) = 0$,
- $F(x, y) = x - y$ för $x > y$, $F(x, y) = 0$ annars.

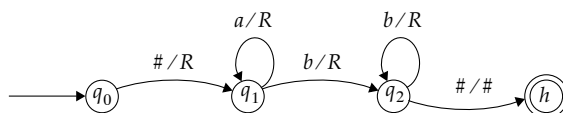
4.5 Turingmaskiner är ekvivalenta med restriktionsfria grammatiker

Som Du säkert kommer ihåg, har vi sagt att en TM *accepterar* de strängar som den stannar för, oavsett läshuvudets position och oavsett tapens innehåll när den stannar. Givetvis kan man korrigera en TM så att den utför någon form av städning på tapen omedelbart innan den stannar, vilket illustreras i följande exempel.

EXEMPEL 4.12 Först presenteras en TM M_1 som stannar på första blanktecknet till höger om sådana strängar som ligger i språket a^*b^+ och som hänger sig eller vägra stanna för strängar som ligger utanför a^*b^+ .

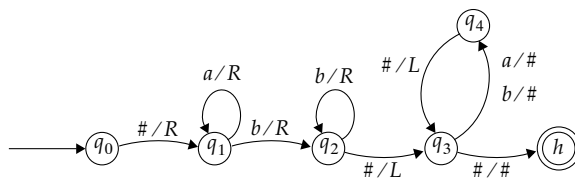
Sedan följer en TM M_2 som är en korrigering av M_1 . Korrigeringen består i att M_2 gör hela tapen blank och ställer läshuvudet på första blanktecknet till vänster innan den stannar. Det är allt.

En TM M_1 som accepterar a^*b^+ utan att städa på tapen.



$(q_0, \#aab\#)$
 $(q_1, \#aab\#)$
 $(q_1, \#aab\#)$
 $(q_1, \#aab\#)$
 $(q_2, \#aab\#)$
 $(h, \#aab\#)$

En TM M_2 som accepterar a^*b^+ , och som gör hela tapen blank innan den stannar.



$(q_0, \#aab\#)$
 $(q_1, \#aab\#)$
 $(q_1, \#aab\#)$
 $(q_1, \#aab\#)$
 $(q_2, \#aab\#)$
 $(q_3, \#aab\#)$
 $(q_4, \#aab\#)$
 $(q_3, \#aab\#)$
 $(q_4, \#a\#\#\#)$
 $(q_3, \#a\#\#\#)$
 $(q_4, \#\#\#\#)$
 $(q_3, \#\#\#\#)$
 $(h, \#\#\#\#)$

På liknande sätt som i exemplet ovan kan varje TM korrigeras så att en blanktape-städare läggs till i slutet. Vi ska nu visa hur man utgående från en sådan blanktape-städande Turingmaskin kan konstrue-

ra en grammatik som från intet kan producera varje sträng som Turingmaskinen accepterar.

Grammatikens uppgift är att härma Turingmaskinens omvandlingar av konfigurationerna. Därmed måste förstas grammatiken ha ett sätt att representera Turingmaskinens konfigurationer. För detta ändamål låter vi Turingmaskinens konfigurationer

$$(q, u \underset{\Delta}{a} v)$$

representeras av strängar

$$[uqav]$$

där q :s placering till vänster om a uttrycker att a är det tecken som läshuvudet står på (Ett enkelt sätt att lösa problemet med representationen av läshuvudsplaceringen, eller hur!). Här är en illustration:

	$(q_0, \underset{\Delta}{\#}aab\#)$	$[q_0\#aab\#]$	
	$(q_1, \underset{\Delta}{\#}qab\#)$	$[\#q_1aab\#]$	
	$(q_1, \underset{\Delta}{\#}aqb\#)$	$[\#aq_1ab\#]$	
Turing- maski- nens konfigu- rations- omvan- dlingar	$(q_1, \underset{\Delta}{\#}aab\#)$	$[\#aaq_1b\#]$	Gram- mati- kens sätt att repre- sentera dem
	$(q_2, \underset{\Delta}{\#}aab\#)$	$[\#aabq_2\#]$	
	$(q_3, \underset{\Delta}{\#}aab\#)$	$[\#aaq_3b\#]$	
	$(q_4, \underset{\Delta}{\#}aaa\#)$	$[\#aaq_4\#\#]$	
	$(q_3, \underset{\Delta}{\#}aaq\#\#)$	$[\#aaq_3a\#\#]$	
	$(q_4, \underset{\Delta}{\#}a\#\#\#)$	$[\#aaq_4\#\#\#]$	
	$(q_3, \underset{\Delta}{\#}q\#\#\#)$	$[\#q_3a\#\#\#]$	
	$(q_4, \underset{\Delta}{\#}\#\#\#\#)$	$[\#q_4\#\#\#\#]$	
	$(q_3, \underset{\Delta}{\#}\#\#\#\#)$	$[q_3\#\#\#\#\#]$	
	$(h, \underset{\Delta}{\#}\#\#\#\#)$	$[h\#\#\#\#\#]$	

FIGUR 4.1 M_2 :s konfigurationer under provkörning på aab .

Lägg märke till att grammatiken i sitt härmande arbete skall utföra Turingmaskinens konfigurationsomvandlingar *baklänges*. Ty om w är en sträng som Turingmaskinen accepterar, dvs om Turingmaskinen drivs av $(q_0, \underset{\Delta}{\#}w)$ till $(h, \underset{\Delta}{\#}\#)$, så skall ju grammatiken kunna göra en härledning som börjar från *intet* – dvs från någon startsymbol – och som slutar just med strängen w .

Nu är det ju så att en Turingmaskins arbete med konfigurationsomvandlingarna baseras på Turingmaskinens tillståndsövergångar.

Därför synes det lämpligt att på något sätt låta reglerna i grammatiken härma Turingmaskinens övergångar.

Nedan visas vilka omvandlingar som tillståndsovergångar orsakar, hur grammatikens härmande produktioner ser ut (baklänges noterade) och vilka regler (också baklänges noterade) som skulle orsaka samma produktioner:

		(1)	(2)	(3)
härmande grammatik	övergång	$\delta(q_i, \sigma) = (q_j, \sigma'')$	$\delta(q_i, \sigma) = (q_j, R)$	$\delta(q_i, \sigma) = (q_j, L)$
	konfigurationsomvandling	$(q_i, u \underset{\Delta}{\sigma} v) \Rightarrow (q_j, u \underset{\Delta}{\sigma'} v)$	$(q_i, u \underset{\Delta}{\sigma} \sigma' y) \Rightarrow (q_j, u \underset{\Delta}{\sigma} \sigma' y)$	$(q_i, x \underset{\Delta}{\sigma} \sigma' v) \Rightarrow (q_j, x \underset{\Delta}{\sigma} \sigma' v)$
	produktion	$[u q_i \sigma v] \Leftarrow [u q_j \sigma' v]$	$[u q_i \sigma \sigma' y] \Leftarrow [u \sigma q_j \sigma' y]$	$[x \sigma' q_i \sigma v] \Leftarrow [x q_j \sigma' \sigma v]$
	regel	$q_i \sigma \leftarrow q_j \sigma''$	$q_i \sigma \leftarrow \sigma q_j$	$\sigma' q_i \sigma \leftarrow q_j \sigma' \sigma$

Med vårt sätt att representera Turingmaskinens konfigurationer kommer uppenbarligen tillståndsovergångar att ge upphov till produktionsregler enligt:

tillståndsovergång	produktionsregel	
$\delta(q_i, \sigma) = (q_j, \sigma'')$	$q_j \sigma'' \rightarrow q_i \sigma$	(1)
$\delta(q_i, \sigma) = (q_j, R)$	$\sigma q_j \rightarrow q_i \sigma$	(2)
$\delta(q_i, \sigma) = (q_j, L)$	$q_j \sigma' \sigma \rightarrow \sigma' q_i \sigma$ för varje tecken σ'	(3)

T ex kommer M_2 :s tillståndsovergång $(q_2) \xrightarrow{\# / L} (q_3)$ dvs

$\delta(q_2, \#) = (q_3, L)$ att ge upphov till reglerna:

$$q_3 \# \# \rightarrow \# q_2 \#$$

$$q_3 a \# \rightarrow a q_2 \#$$

$$q_3 b \# \rightarrow b q_2 \#$$

Vidare måste vi ha en startsymbol S som kan sätta igång grammatikens strängproduktion på ett sätt som motsvarar Turingmaskinens avslutande agerande. Härav följande regel:

$$S \rightarrow [h\#] \quad (4)$$

Och med hjälp av regeln

$$\#] \rightarrow \# \#] \quad (5)$$

kan vi uttrycka att tapen innehåller oändligt många blanktecken.

Slutligen behövs produktionsregler som kan avlägsna de *icketerminerande* symbolerna $q_0, \#, [,]$ när den producerade strängen har en form $[q_0 \# w \# \# \dots \#]$ som motsvarar att Turingmaskinen står i *startkonfiguration* $(q_0, \# w)$. Ty vi vill ju att t ex $[q_0 \# aab \# \# \#]$ skall kunna rensas från icketerminerande symboler och övergå i strängen *aab*. Följande regler blir bra för detta ändamål:

$$[q_0 \# \rightarrow \varepsilon \quad (6)$$

$$\# \#] \rightarrow \#] \quad (7)$$

$$\#] \rightarrow \varepsilon \quad (8)$$

Nedan illustreras hur strängen *aab* skulle produceras med hjälp av reglerna (1) – (8) i en grammatik som härmar M_2 .

produktion	produktionsregel
$S \Rightarrow [h\#]$	(4)
$\Rightarrow [h\#\#]$	(5)
$\Rightarrow [h\#\#\#]$	(5)
$\Rightarrow [h\#\#\#\#]$	(5)
$\Rightarrow [h\#\#\#\#\#]$	(5)
$\Rightarrow [q_3\#\#\#\#\#]$	$h\# \rightarrow q_3\#$ (1)
$\Rightarrow [\#q_4\#\#\#\#]$	$q_3\#\# \rightarrow \#q_4\#$ (3)
$\Rightarrow [\#q_3a\#\#\#]$	$q_4\# \rightarrow q_3a$ (1)
$\Rightarrow [\#aq_4\#\#\#]$	$q_3a\# \rightarrow aq_4\#$ (3)
$\Rightarrow [\#aaq_3a\#\#]$	$q_4\# \rightarrow q_3a$ (1)
$\Rightarrow [\#aaq_4a\#\#]$	$q_3a\# \rightarrow aq_4\#$ (3)
$\Rightarrow [\#aaq_3b\#]$	$q_4\# \rightarrow q_3b$ (1)
$\Rightarrow [\#aabq_2\#]$	$q_3b\# \rightarrow bq_2\#$ (3)
$\Rightarrow [\#aaq_1b\#]$	$bq_2 \rightarrow q_1b$ (2)
$\Rightarrow [\#aq_1ab\#]$	$aq_1 \rightarrow q_1a$ (2)
$\Rightarrow [\#q_1aab\#]$	$aq_1 \rightarrow q_1a$ (2)
$\Rightarrow [q_0\#aab\#]$	$\#q_1 \rightarrow q_0\#$ (2)
$\Rightarrow aab\#]$	(6)
$\Rightarrow aab$	(8)

För varje TM M kan man teckna ned reglerna (1) – (3) utgående från M :s tillståndsövergångar. Sedan kan man tillfoga reglerna (4) – (8), vilka är oberoende av Turingmaskin M , varefter man har en restriktionsfri grammatik som genom att härma M :s konfigurationsomvandlingar (baklänges) kan producera precis de strängar som M stannar för. Härav följande sats.

- ✧ SATS 4.1 *Om L är ett Turingaccepterbart språk så kan L beskrivas av en restriktionsfri grammatik.*

Omvänt gäller det, att för varje restriktionsfri grammatik så kan man konstruera en TM som härmar grammatikreglernas strängmanipulationer. Detta bevisar vi inte här, utan nöjer oss med konstaterandet att Turingmaskinens kompetens att gå fram och tillbaka på tapen och efter vissa regler (Turingmaskinens uppsättning av tillståndsövergångar) byta ut tecken mot nya tecken synes vara precis vad som behövs för sådan härmning.

- ✧ SATS 4.2 *Om L kan beskrivas av en restriktionsfri grammatik, så är L ett Turingaccepterbart språk.*

- ⌘ Anm.4.5 En komplikation i sammanhanget är visserligen att en grammatiks strängmanipulationer inte behöver vara deterministiska, medan vår Turingmaskin per definition agerar deterministiskt. Dock kan man visa att varje ickedeterministisk TM kan härmas av en deterministisk dito (sid 138).

4.6 En universell Turingmaskin

När man kör igång en Turingmaskin M på en tape med given input-sträng w startas ett arbete som är helt mekaniskt. Steg för steg utföres Turingmaskinens tillståndsövergångar med åtföljande agerande på tapen.

Redan i den berömda publikationen från 1936 föreslog Alan Turing hur en Turingmaskin skulle kunna utföra allt detta arbete, dvs hur en (universell) Turingmaskin U skulle kunna ta M och w såsom input och lämna $M(w)$ som output:

$$U(M, w) = M(w)$$

Turings idé var väsentligen följande.

Tillståndsövergångarna tillsammans med en upplysning om vilket tillstånd som är starttillstånd utgör M :s fulla beskrivning. Därför kan M representeras av en lista innehållande M :s starttillstånd och M :s tillståndsövergångar:

$$((q_{i_0})(S_{11})(S_{12})\dots(S_{mn}))$$

S_{ik} representerar här en tillståndsövergång från tillstånd q_i när tecknet a_k läses. Närmare bestämt kan vi låta S_{ik} vara en kvadrupel

$$S_{ik} = q_i, a_k, q_j, b \quad (9)$$

där

- q_i betecknar nuvarande tillstånd,
- a_k läst tecken,
- q_j nästa tillstånd
- b betecknar någon av riktningsangivelserna L , R eller det tecken som M skall skriva på läshuvudets plats (där a_k står).

Om w är en inputsträng till M så representerar uppenbarligen

$$(((q_{i_0})(S_{11})(S_{12})\dots(S_{mn}))(w)) \quad (10)$$

ett lämpligt input till den tänkta Turingmaskinen U .

För att slippa använda parentestecken och kommatecken i U 's alfabet, kan man ersätta (10) med följande teckensträng:

$$W = q_{i_0} 00 S_{11} 00 S_{12} 00 \dots 00 S_{mn} \# w$$

där M 's kod åtskiljs från M 's inputsträng med blanktecken, tillståndsövergångarna från varandra och från starttillståndet med dubbla 0:or, och där de fyra komponenterna i varje S_{ik} åtskiljs med enstaka 0:or.

Sådana här strängar W vill vi således ge som input till en Turingmaskin U . För att U inte skall behöva arbeta med olika alfabeten när den kör olika W låter vi koda alla Turingmaskiners alfabeten och tillståndssymboler till ett enda alfabet, och för att göra det hela så enkelt som möjligt väljer vi ett alfabet med en enda symbol, säg symbolen "1", utöver blanktecknet #.

Eftersom det finns uppräknligt antal Turingmaskiner (men inte fler), så behöver vi koda uppräknligt många q_i och uppräknligt många a_k förutom de två symbolerna L , R och stopptillståndssymbolen h . Sådan kodning kan göras på t ex följande sätt.

$$\begin{array}{lll} h = q_1 = 1 & L = 1 & a_1 = 111 \\ q_2 = 11 & R = 11 & a_2 = 1111 \\ q_3 = 111 & & a_3 = 11111 \\ \dots & & \dots \end{array}$$

Om man som ovan låter q_1 vara stopptillstånd måste alla S_{1k} strykas från W , eftersom ingen TM har någon övergång från sitt stopptillstånd.

T ex får kvadrupeln q_2, a_2, q_4, L koden 1101110111101 (efter insättning av åtskiljande 0:or istället för kommatecken). Lägg märke till att sådan kodning förmår skilja L , R från varje tecken a_k . Detta är nödvändigt med tanke på hur tillståndsövergångarna ser ut, eller hur. Däremot finns det tillstånd och tecken med likadan kod. (Varför

finns det inget problem förbundet med detta?) Hela W 's kod för givet W kan Du säkert föreställa Dig. Notera speciellt att W 's kod blir ett strängpar[†] med vardera av de två strängarna i $\{0, 1\}^*$.

När vi nu har preparerat en lämplig inputsträng åt vår universella Turingmaskin U , är det dags att fundera över U 's recept. Vi nöjer oss med en informell beskrivning i sex steg:

STEG 0 Placera W (dess kod) på U 's tape: $\#W \# \# \dots$
 Δ

STEG 1 Låt läshuvudet gå *ett* steg till höger och läsa av vilket *starttillstånd* 1^i som M har.

STEG 2 Låt läshuvudet gå till höger i W ända tills w 's första teckensymbol påträffas. Läs av denna symbol 1^k .

STEG 3 Låt läshuvudet gå till vänster i W och leta rätt på *tillståndsövergången* S_{ik} (börjar med $1^i 0 1^k 0 \dots$). Om S_{ik} saknas hänger[‡] det sig.

STEG 4 I S_{ik} hittar man *nästa tillstånd* samt *symbol att skriva* eller *rörelseriktning*.

STEG 5 Låt läshuvudet gå till höger i W och *agera* på w (med symbolskrivande eller med att gå *ett* steg till vänster eller höger på tapen).

STEG 6 Om nästa tillstånd är 1 ($=h$) är vi färdiga. Annars, gå till STEG 3 med 1^i och 1^k betecknande *nuvarande tillstånd* respektive *nuvarande tecken* (det tecken som läshuvudet står på).

4.7 Ekvivalenta varianter av Turingmaskiner

Många varianter av standard-Turingmaskinen – både generaliseringar och specialiseringar – har genom åren bevisats vara ekvivalenta med den förra. Och ingen enda generalisering har hittills visats vara mer kompetent än standard-varianten vilket kan ses som ett stöd för Turings tes (sid 109).

Vi ska här lite skissartat presentera några sådana varianter.

Flera tapar

Tänk Dig en TM utrustad med flera tapar så att varje tape har ett läshuvud som på "vanligt" sätt är kopplat till kontrollmekanismen, och vidare så att agerandena på de olika taparna är oberoende av varandra.

Medan vår ursprungliga TM har tillståndsövergångar av typ

†. Om man önskar kan man representera W 's kod med *en* binär sträng, t ex genom att ersätta blanktecknet mellan de två strängarna med tre 0:or.

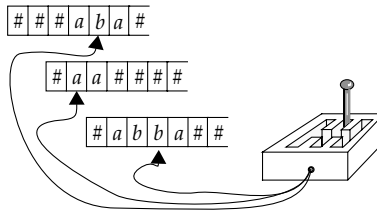
‡. Se sid 120.

$$\delta(p, a) = (q, \text{tapeagerande})$$

där *tapeagerande* är att *skriva* ett tecken eller att *flytta* läshuvudet till vänster eller till höger, så skulle en Turingmaskin med N tapar ha övergångar av typ

$$\delta(p, a_1, \dots, a_N) = (q, \text{tapeagerande}_1, \dots, \text{tapeagerande}_N)$$

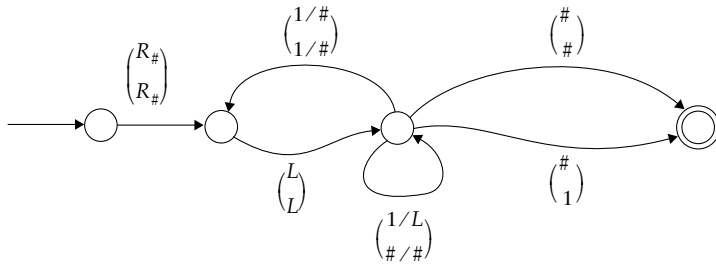
där a_i är det tecken som den i :te tapens läshuvud står på, och tapeagerande_i är tapeagerandet på den i :te tapen.



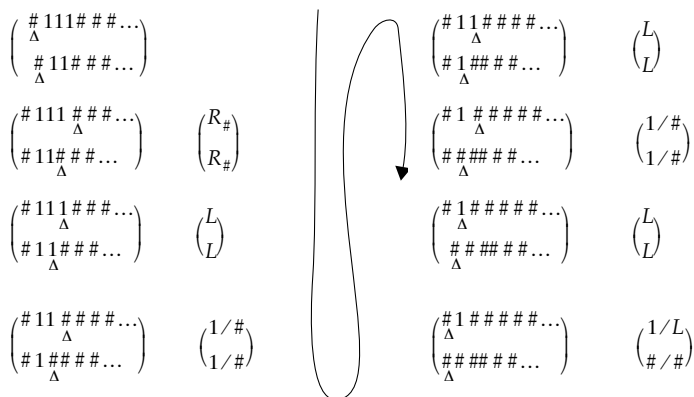
EXEMPEL 4.13 I figuren nedanför beskrivs en två-tape-TM som beräknar funktionen

$$F(x, y) = x - y \text{ för } x > y, F(x, y) = 0 \text{ annars}$$

Vid start placerar man på de två taparna de två argumentsvärden som man vill beräkna funktionens output för. Vid avslutad körning har TM:en skrivit sitt output på den första tapen.



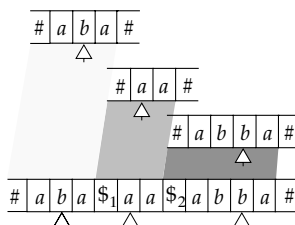
Provkörning:



Hur kan vår gamla *en*-tape-TM härma en sådan här *multi*-tape-TM?

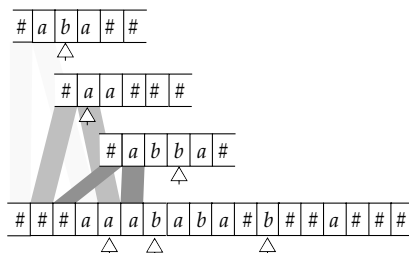
Ett sätt är att tapen hos *en*-tape-TM:en delas upp i olika delar t_1, \dots, t_N motsvarande de olika taparna hos en *multi*-tape-TM och så att *en* sådan del t_i samt delarna till höger därom dynamiskt högershiftas ifall t_i behöver mer utrymme till höger. Dessutom måste man förstås lista ut hur man kan representera N stycken läshuvuden.

Tre taper härmas med *en* tape genom uppdelning av den senare i tre åtskilda delar och dynamisk högershiftning av dessa delar.



Ett annorlunda sätt vore att dela upp tapen hos *en*-tape-TM:en cell för cell enligt nedanstående figur.

Tre tapar härmas med *en* tape genom att var tredje cell hos denna senare tape tillordnas celler från en av de tre taparna.



Ickedeterminism

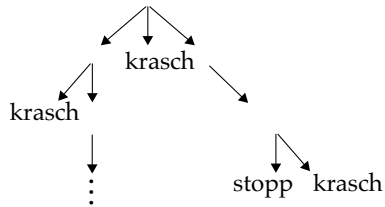
En *ickedeterministisk* TM kan – på motsvarande sätt som en ickedeterministisk FA eller PDA – agera flertydigt när den befinner sig i något tillstånd och läser något tecken. Med tanke på att en ickedeterministisk PDA har högre potentiell förmåga än en deterministisk dito ligger det nära till hands att föreställa sig att motsvarande skillnad skulle finnas mellan deterministiska och ickedeterministiska Turingmaskiner. Så är dock icke fallet, vilket vi skall försöka övertyga läsaren om genom att skissera en deterministisk TM (utrustad med tre tapar) som har kompetens att härma en ickedeterministisk TM (den senare försedd med en enda tape).

Notera först att skillnaden mellan en deterministisk och en ickedeterministisk TM kan beskrivas som att den förra, vid körning på en inputsträng, agerar med *en* följd av övergångar, medan den senare kan agera med flera sådana övergångsföljder, där denna mångfald av övergångsföljder kan representeras av ett träd som i figuren nedanför.

En deterministisk TM agerar på *ett* sätt, vid körning på en inputsträng.



En ickedeterministisk TM kan, vid körning på en inputsträng, agera på flera olika sätt. Vissa sådana sätt kan sluta i hängning (krasch). Andra kan fortsätta utan slut – representerande att TM:en aldrig stannar. Och ytterligare ett eller flera sätt kan sluta lyckligt (i stopptillståndet).



Antag nu att M är en *ickedeterministisk* TM med *en* tape.

I syfte att härma M låt M' vara en deterministisk *tre-tape*-TM, med tape 1 tänkt att användas enbart som behållare för inputsträngen så att denna när som helst kan kopieras härifrån till tape 2 vilken skall brukas som provbänk för testning av inputsträngen längs de olika övergångsföljder som ickedeterminismen hos M ger upphov till. Tape 3 är tänkt att användas som databas innehållande följder av tillståndsovergångar som är möjliga att utföra i M .

I princip kan M' härma M på följande sätt:

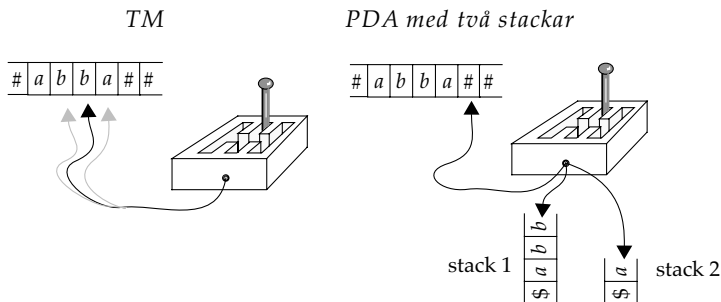
- STEG 0 Kopiera inputsträngen w från tape 1 till tape 2.
- STEG 1 Beräkna på tape 3 en kortaste följd av övergångar (*en* övergång första gången) av alla M :s icke redan testade övergångar.
- STEG 2 Provkör, på tape 2, M för det agerande som beräknats i STEG 1.
- STEG 3 Om därvid M skulle stanna, låt M' stanna. Annars, låt M' sätta ickeblanka tecken utom ändmarkeringen på tape 2, bokför på tape 3 den just testade följd av övergångar och upprepa sedan STEG 1– STEG 3.

■ En två-stacks-PDA har samma förmåga som en TM

En *vanlig* PDA (med en inputtape och en stacktape) kan ses som ett specialfall av en *två-tape-TM*, med restriktioner på sättet att använda de två taparna. En utvidgad PDA som har *två* stacktar istället för en, kan på samma sätt betraktas som ett specialfall av en *tre-tape-TM*. En sådan utvidgad PDA kan med lätthet känna igen t ex $\{a^n b^n c^n \mid n \in \mathbb{N}\}$, eller hur!

Ett intressant faktum är att en deterministisk sådan här PDA med *två* stacktar kan allt som en TM kan. Den kan nämligen härma en standard-TM genom att inledningsvis pusha inputsträngen tecken för tecken på en av stacktarna, och sedan övergå till ett annat tillstånd varifrån det egentliga TM-härmandet sätter igång. De två stacktarna får nu härma Turingmaskinens tapekonfiguration. Den stack som inputsträngen pushades på (med dess sista tecken på stacktoppen) får hela tiden innehålla den del av Turingmaskinens tape som ligger till vänster om läshuvudet och dessutom tecknet som läshuvudet pekar på. Och den andra stacken får hela tiden innehålla det som finns till höger om Turingmaskinens läshuvud.

En PDA utvidgad med en extra stack har kompetens att härma en TM.



4.8 Övningar

4.1 Visa att nedanstående restriktionsfria grammatik i själva verket beskriver ett sammanhangsfritt språk.

$$S \rightarrow aA$$

$$A \rightarrow Sa$$

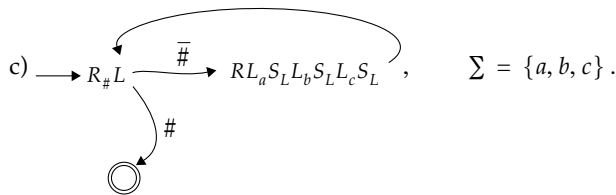
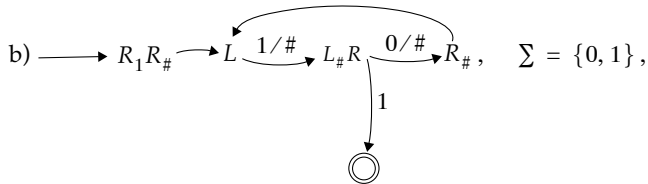
$$aAa \rightarrow b$$

4.2 Låt L vara språket över $\{v, h, u, n\}$ i TEST 3.6 b) på sidan 98. Konstruera en restriktionsfri grammatik för L .

4.3 Presentera en restriktionsfri grammatik för "triangeltalsspråket" $\{1^{0+1+\dots+n} \mid n \in \mathbb{N}\}$.

4.4 Beskriv de språk som följande Turingmaskiner accepterar.

a) $R_1 0 R_0 1 L_1 L_1$, $\Sigma = \{0, 1\}$,



4.5 Ge exempel på Turingmaskiner med inputalfabet $\Sigma = \{a, b\}$ som

a) *accepterar* språken \emptyset , $\{\epsilon\}$, Σ^* b) *avgör* samma språk.

4.6 Konstruera Turingmaskiner som beräknar följande funktioner från \mathbb{N} till \mathbb{N} med de naturliga talen representerade i 1-systemet:

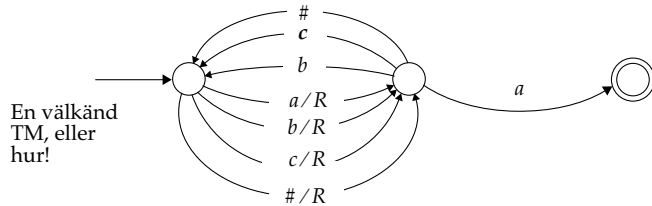
a) $F(x) = \text{Rest}(x, 2)$ b) $F(x) = \text{Kvot}(x, 2)$

4.7 Konstruera Turingmaskiner som

a) avgör "tvåpotensspråket" $\{1, 1^2, 1^4, 1^8, 1^{16}, 1^{32}, \dots\}$,

- b) beräknar $F(x) = 2^x$, $x \in \mathbb{N}$,
- c) beräknar $F(w) = w^{rev}$, $w \in \{a, b\}^*$,
- d) avgör "triangelstalsspråket" $\{1^0 + 1 + \dots + n \mid n \in \mathbb{N}\}$.

4.8 Betrakta den välkända Turingmaskinen nedan. Konstruera en grammatik som genom att *härma* denna TM (jfr med EXEMPEL 4.12 på sidan 129) kan producera precis sådana strängar w som TM:en accepterar. TM:en antas startas på tapekonfigurationen $\#w$.



4.9 Om $F(y_1, \dots, y_m)$ är en funktion från $\mathbb{N}^m = \mathbb{N} \times \dots \times \mathbb{N}$ till \mathbb{N} och om $G_1(\vec{x}), \dots, G_m(\vec{x})$ med $\vec{x} = x_1, \dots, x_n$ är funktioner från \mathbb{N}^n till \mathbb{N} kan man bilda sammansättningen $H(\vec{x}) = F(G_1(\vec{x}), \dots, G_m(\vec{x}))$ som då blir en funktion från \mathbb{N}^n till \mathbb{N} .

Antag nu att M_F och M_{G_1}, \dots, M_{G_m} är Turingmaskiner som beräknar F och G_1, \dots, G_m .

Konstruera med hjälp av M_F och M_{G_1}, \dots, M_{G_m} en TM M_H som beräknar H .

Rekursiva funktioner

Primitivt rekursiva funktioner	143
Primitivt rekursiva predikat	148
Funktionen Om	149
Delbarhet och primitivt rekursiva funktioner	151
Primitiv rekursion är otillräcklig	153
En primitivt rekursiv konstruktion av $\text{Fib}(x)$	156
Rekursiva flätor	157
Primtal och framåtrekursion	159
Rekursiva funktioner – en utvidgning av primitivt rekursiva funktioner	160
Turings tes och Church:s tes	163
För varje TM finns det en rekursiv funktion	164
För varje rekursiv funktion finns det en TM	170
Övningar	171

Målet för detta kapitel kan sägas vara att ge en “maskinoberoende” beskrivning av de *funktioner* som kan beräknas mekaniskt, dvs en beskrivning som inte är bunden till någon särskild fysisk eller abstrakt automat tänkt att utföra beräkningarna.

Vårt projekt kan synas vara vagt formulerat och alltför allmänt hållet, ja kanske rent av gigantiskt. Nåväl, låt oss avgränsa projektet till att omfatta enbart sådana funktioner som returnerar *naturliga tal* och som har *naturliga tal* i sina argument. Denna avgränsning förenklar vårt projekt, men begränsar på intet sätt dess giltighet. Ty oavsett vad input och output egentligen är för slags teckensträngar, så kan man alltid koda dem till strängar över det binära alfabetet. Och binära strängar kan ju tolkas som naturliga tal.

Först skall vi se på en klass av funktioner – *primitivt rekursiva funktioner* – som studerades framförallt under 30-talet av bl a Ackermann, Church, Gödel, Hilbert, Kleene och Péter och som kom att bli en av inspirationskällorna för funktionell programmering. Poängen med en sådan funktion är dels att den är definierad på samtliga naturliga tal, dels att dess output för ett givet input kan beräknas med ett på förhand känt antal beräkningssteg.

Innan Ackermann publicerade sin funktion (se EXEMPEL 5.14 på sidan 153) var det en öppen fråga huruvida de primitivt rekursiva funktionerna fullständigt fångade idén om “det algoritmiska” eller ej. Ackermanns funktion var nämligen det första exemplet på en algoritmisk funktion som inte rymdes inom klassen av primitivt rekursiva funktioner.

Lägg förresten märke till att vi presenterar rekursiva funktioner på

ett rekursivt sätt i den mening att vi beskriver

- den *enklaste* funktionen (*basfunktionen*), samt
- mer komplexa funktioner i termer av enklare.

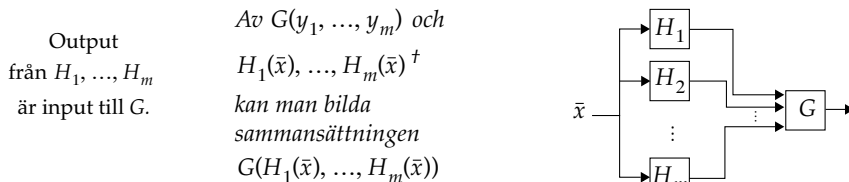
5.1 Primitivt rekursiva funktioner

■ **Basfunktionen** *Öka* Följande funktion är så enkel att den utan tvivel kan beräknas mekaniskt.

x	0	1	2	...	x	...
$\text{Öka}(x)$	1	2	3	...	$1+x$...

Funktionen *Öka* bildar "basen" inte bara för de primitivt rekursiva funktionerna (sid 148) utan också för de (allmänt) rekursiva funktionerna (sid 161).

■ **Sammansättning** Att bilda *sammansättningar* är ett mekaniskt sätt att skapa nya funktioner av gamla:



EXEMPEL 5.1 Anlita ökaren två gånger i följd[‡]:

$$\text{Öka}(\text{Öka}(x)) \quad x \rightarrow \boxed{\text{Öka}} \rightarrow \boxed{\text{Öka}} \rightarrow 2+x$$

■ **Primitiv rekursion** Vi börjar med några enkla exempel ...

EXEMPEL 5.2 Funktionen vars output är identiskt med input

En tabell-beskrivning

x	0	1	2	...
$\text{Identitet}(x)$	0	1	2	...

kan beskrivas på följande sätt med rekursion:

En rekursiv beskrivning

$$\begin{cases} \text{Identitet}(0) = 0 \\ \text{Identitet}(\text{Öka}(x)) = \text{Öka}(\text{Identitet}(x)) \end{cases} \quad \begin{matrix} \text{basfallet} \\ \text{rekursionssteget} \end{matrix} \quad (1)$$

†. \bar{x} betecknar en multipel x_1, \dots, x_n

‡. Med funktionen *Öka* och metoden *sammansättning* kan vi svårligen tillverka några mer upphetsande funktioner än denna.

Håller Du med mig, om jag säger att den rekursiva beskrivningen av identitetsfunktionen är av enklare karaktär än tabellbeskrivningen ovanför?

Vad jag menar är följande:

Om man vill använda *tabellen* för att få fram t ex det output som hör till $x = 100$ eller något annat output som inte finns med i tabellen, måste man få en insikt om ett mönster som upprepar sig i all oändlighet. Dvs man måste kunna abstrahera – från enstaka observationer (tabellens tre input-output) dra allmänna slutsatser.

Visserligen är tabellens mönster enkel att upptäcka för Dig – men knappast för en maskin. Däremot är det lätt att föreställa sig att en maskin skulle kunna använda den rekursiva beskrivningen för beräkningar som i följande provkörning.

$ \begin{aligned} & \text{Identitet}(3) \\ &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Identitet}}(2)) \\ &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Identitet}}(1))) \\ &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Identitet}}(0)))) \\ &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(0))) \\ &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(1)) \\ &= \underset{\Delta}{\text{Öka}}(2) \\ &= 3 \end{aligned} $	<p>Först tillämpas rekursionssteget om och om igen – varvid den med pilspets markerade textsträngen varje gång ersätts med motsvarande text från rekursionsstegets högerled ...</p> <p>Till slut tillämpas basfallet, och därefter återstår bara att utföra de upprepade ökningarna ...</p>
---	---

EXEMPEL 5.3 Detta exempel behandlar addition. Om man redan har adderat ett tal x till ett annat tal y , och vill addera ett ökat x till y , så behöver man bara *öka* resultatet av den förra additionen med en enhet. Med traditionell notation uttrycks detta som $(1 + x) + y = 1 + (x + y)$. Med funktionell notation:

$$\begin{cases} \text{Add}(0, y) = y & \text{basfallet} \\ \text{Add}(\underset{\Delta}{\text{Öka}}(x), y) = \underset{\Delta}{\text{Öka}}(\text{Add}(x, y)) & \text{rekursionssteget} \end{cases}$$

Här kommer en provkörning:

$$\begin{aligned}
 \text{Add}(3, 5) &= \underset{\Delta}{\text{Öka}}(\text{Add}(2, 5)) \\
 &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Add}}(1, 5))) \\
 &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Add}}(0, 5)))) \\
 &= \underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(\underset{\Delta}{\text{Öka}}(5))) \\
 &= \dots \\
 &= 8
 \end{aligned}$$

EXEMPEL 5.4 Med hjälp av det x :te triangeltalet (Se EXEMPEL 4.8 på sidan 126.)

$$x + \dots + 1 + 0$$

erhålles nästa triangeltal

$$(x + 1) + x + \dots + 1 + 0$$

genom att addera $x + 1$ till det förra. Dvs,

$$\begin{cases} \text{TriangelTal}(0) = 0 \\ \text{TriangelTal}(\text{Öka}(x)) = \text{Add}(\text{Öka}(x), \text{TriangelTal}(x)) \end{cases}$$

T ex beräknas $4 + 3 + 2 + 1$ sålunda:

$$\begin{aligned} \text{TriangelTal}(4) &= \text{Add}(4, \text{TriangelTal}(3)) \\ &= \text{Add}(4, \text{Add}(3, \text{TriangelTal}(2))) \\ &= \text{Add}(4, \text{Add}(3, \text{Add}(2, \text{TriangelTal}(1)))) \\ &= \text{Add}(4, \text{Add}(3, \text{Add}(2, \text{Add}(1, \text{TriangelTal}(0))))) \\ &= \text{Add}(4, \text{Add}(3, \text{Add}(2, \text{Add}(1, 0)))) \\ &= \text{Add}(4, \text{Add}(3, \text{Add}(2, 1))) \\ &= \text{Add}(4, \text{Add}(3, 3)) \\ &= \text{Add}(4, 6) \\ &= 10 \end{aligned}$$

■ Den primitivt rekursiva mallen

Vi har i några exempel sett hur man på ett fullständigt mekaniskt sätt kan beräkna output till en funktion F med hjälp av enklare output till densamma. Exempelen har följt mallen nedanför.

Den primitivt rekursiva mallen

Noll passiva
argument hos F

$$\begin{cases} F(0) = b \\ F(\text{Öka}(x)) = G(x, F(x)) \end{cases}$$

nuvarande input

nuvarande output

nästa output

n passiva
argument hos F

$$\begin{cases} F(0, y_1, \dots, y_n) = B(y_1, \dots, y_n) \\ F(\text{Öka}(x), y_1, \dots, y_n) = G(x, y_1, \dots, y_n, F(x, y_1, \dots, y_n)) \end{cases}$$

Ett enda re-
kurserande
argument!

Den *primitiva* rekursionens kännetecken är dels att rekursionen äger rum enbart i *ett* av argumenten – eventuella övriga argument är “passiva” – dels att det rekurserande argument *löper bakåt en enhet* varje gång som rekursionssteget används. Tack vare basfallet bottnar (upphör) rekursionen.

Det
rekurerande
argumentet
löper bakåt
från 5 till 4 till
3 till ...
... till 0
där basfallet
sätter stopp.

$$\begin{aligned}
 F(5, y) &= G(4, y, F_{\Delta}(4, y)) \\
 &= G(4, y, G(3, y, F_{\Delta}(3, y))) \\
 &= G(4, y, G(3, y, G(2, y, F_{\Delta}(2, y)))) \\
 &= G(4, y, G(3, y, G(2, y, G(1, F_{\Delta}(1, y))))) \\
 &= G(4, y, G(3, y, G(2, y, G(1, G(0, y, F_{\Delta}(0, y)))))) \\
 &= G(4, y, G(3, y, G(2, y, G(1, G(0, y, B(y))))))
 \end{aligned}$$

⌘ Anm.5.1 I den primitivt rekursiva mallen har vi valt att låta F 's första argument vara det rekurerande argumentet, men egentligen kan vilket som helst av argumenten spela denna roll.

⌘ Anm.5.2 Vi skall tolka mallen på ett mindre strängt sätt än vad som är gängse, och säga att en funktionsbeskrivning följer mallen även om funktionerna B och G i mallen *inte* använder samtliga argument som mallen tillåter.

EXEMPEL 5.5 I EXEMPEL 5.3 där Add konstrueras är $G = \ddot{O}ka$ vilken enbart nyttjar ett av de tre argument som den primitivt rekursiva mallen tillåter. Men i EXEMPEL 5.4 där G -funktionen ges av $G(x, z) = Add(\ddot{O}ka(x), z)$, använder G båda de argument som mallen bjuder på.

EXEMPEL 5.6 Följande primitivt rekursiva konstruktion beskriver en konstant funktion:

En konstant
funktion.

$$\begin{cases} Noll(0) = 0 \\ Noll(\ddot{O}ka(x)) = Noll(x) \end{cases}$$

$$\text{Provkörning: } Noll(3) = Noll(2) = Noll(1) = Noll(0) = 0$$

Med hjälp av funktionerna $\ddot{O}ka$ och $Noll$ är det lätt att bygga andra konstanta funktioner:

Tre
konstanta
funktioner.

$$\begin{aligned}
 Ett(x) &= \ddot{O}ka(Noll(x)) & x \longrightarrow \boxed{Noll} \longrightarrow \boxed{\ddot{O}ka} \longrightarrow 1 \\
 Två(x) &= \ddot{O}ka(Ett(x)) & x \longrightarrow \boxed{Ett} \longrightarrow \boxed{\ddot{O}ka} \longrightarrow 2 \\
 Tre(x) &= \ddot{O}ka(Två(x)) & x \longrightarrow \boxed{Två} \longrightarrow \boxed{\ddot{O}ka} \longrightarrow 3
 \end{aligned}$$

⌘ Anm.5.3 I fortsättningen skriver vi ofta 0 istället för $Noll(x)$, 1 istället för $Ett(x)$, 2 istället för $Två(x)$, osv ...

EXEMPEL 5.7 Funktionen som beskrivs i input-output-tabellen nedan är en sorts motsats till funktionen *Öka* och döps lämpligen till *Minska*.

Funktionen <i>Minska</i>	<i>input</i>	0	1	2	3	4	...	$x + 1$...
	<i>output</i>	0	0	1	2	3	...	x	...

Input-output-tabellen leder oss till den vänstra konstruktionen nedanför:

$$\begin{cases} Minska(0) = 0 \\ Minska(\ddot{O}ka(x)) = x \end{cases} \quad \begin{cases} Minska(0) = 0 \\ Minska(\ddot{O}ka(x)) = G(x) \end{cases}$$

Vid en jämförelse med den primitivt rekursiva mallen till höger ovanför, ser man att vår konstruktion följer ur mallen om rekursionsstegets G sätts till identitetsfunktionen.

Minska är således en primitivt rekursivt konstruerad funktion fastän den aldrig "anropar sig själv".

EXEMPEL 5.8 Den primitivt rekursiva konstruktionen av addition (se EXEMPEL 5.3 på sidan 144) bygger på att addition är upprepad ökning. Eftersom subtraktion är upprepad minskning kan vi konstruera en funktion för subtraktion på motsvarande sätt:

Här har vi valt att placera det rekurerande argumentet på andra plats.

$$\begin{cases} Sub(y, 0) = y \\ Sub(y, \ddot{O}ka(x)) = Minska(Sub(y, x)) \end{cases}$$

EXEMPEL 5.9 *Multiplikation* är upprepad addition, vilket ger oss

$$\begin{cases} Mult(0, y) = 0 \\ Mult(\ddot{O}ka(x), y) = Add(y, Mult(x, y)) \end{cases}$$

Och *potensupphöjning* är upprepad multiplikation, varför

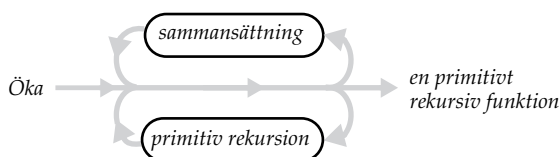
$$\begin{cases} Pot(y, 0) = 1 \\ Pot(y, \ddot{O}ka(x)) = Mult(y, Pot(y, x)) \end{cases}$$

■ Klassen av primitivt rekursiva funktioner

Vi har nu i några exempel (och flera följer) illustrerat hur man genom att tillämpa operationerna *sammansättning* och / eller *primitiv rekursion* på basfunktionen *Öka* kan tillverka mer komplicerade funktioner. Vidare, hur man genom att åter tillämpa operationerna på dessa funktioner kan tillverka ännu mer komplicerade funktioner.

DEFINITION Med mängden (eller klassen) av primitivt rekursiva funktioner (eller helt kort de primitivt rekursiva funktionerna) menas mängden av funktioner som kan tillverkas från basfunktionen *Öka* genom att operera noll eller flera (ändligt många) gånger med sammansättning och/eller primitiv rekursion.

Så här tillverkas primitivt rekursiva funktioner.



⌘ ANM.5.4 Lägg märke till att en primitivt rekursiv funktion kan tillverkas *utan* att den primitivt rekursiva mallen används. (Basfunktionen *Öka* är ju t ex primitivt rekursiv, precis som sammansättningar av *Öka*.) Men utan den primitivt rekursiva mallen kan man bara få triviala primitivt rekursiva funktioner.

■ Primitivt rekursiva predikat

Ett *predikat* (eller *Boolsk funktion*) är en funktion med output i mängden $\{0, 1\}$, där 1 tolkas som "ja", och 0 som "nej".

Följande exempel visar hur några kända predikat kan konstrueras inom klassen av primitivt rekursiva funktioner.

EXEMPEL 5.10 Predikaten *Noll?* och *IckeNoll?*

Nedan följer en beskrivning av ett par enkla predikat, båda konstruerade som primitivt rekursiva funktioner.

x	0	1	2	3	...
$Noll?(x) = Sub(ETT(x), x)^{\dagger}$	1	0	0	0	...
$IckeNoll?(x) = Noll?(Noll?(x))$	0	1	1	1	...

\dagger . Vi sätter i allmänhet tecknet "?" som ett suffix på de funktionsnamn som avser predikat.

EXEMPEL 5.11 Flera primitivt rekursiva predikat

Följande predikat – med väl genomtänkta namn – är mycket användbara i "logiska" konstruktioner.

	$Och(x, y)$	$Eller(x, y)$	$Mindre(x, y)$	$Icke(x)$
retur- 1 om	$(x \neq 0) \text{ och } (y \neq 0)$	$(x \neq 0) \text{ eller } (y \neq 0)$	$x < y$	$x = 0$
nerar 0 om	$(x = 0) \text{ eller } (y = 0)$	$(x = 0) \text{ och } (y = 0)$	$x \geq y$	$x \neq 0$

Icke är densamma som *Noll?* och de övriga tre kan konstrueras som primitivt rekursiva funktioner t ex på nedanstående sätt:

$$\begin{aligned} Och(x, y) &= Icke(Noll?(Mult(x, y))) \\ Eller(x, y) &= Icke(Noll?(Add(x, y))) \\ Mindre(x, y) &= Icke(Noll?(Sub(y, x))) \end{aligned}$$

EXEMPEL 5.12 Huruvida två naturliga tal x, y är *lika* eller *olika* reds ut av

$$\begin{aligned} Lika(x, y) &= Och(Icke(Mindre(x, y)), Icke(Mindre(y, x))) \\ Olika(x, y) &= Icke(Lika(x, y)) \end{aligned}$$

■ **Funktionen** *Om* Vi skall nu introducera en mycket användbar funktion som (efter en rekommendation från John McCarthy[†]) finns med i de flesta programmeringsspråk för att hantera vägval. Funktionen som har tre argument returnerar sitt andra eller tredje argument beroende på om det första argumentet är nollskilt eller ej. Den här funktionen är kanske mest känd under namnet **if-then-else**, men vi betecknar den med *Om*. Så här ser en formell definition ut:

$$Om(x, y, z) = \begin{cases} z & \text{om } x = 0 \\ y & \text{om } x > 0 \end{cases}$$

Att *Om* kan tillverkas inom klassen av primitivt rekursiva funktioner framgår av vår konstruktion nedan:

Vår konstruktion av *Om* Den primitivt rekursiva mallen

$$\begin{cases} Om(0, y, z) = z \\ Om(\ddot{O}ka(x), y, z) = y \end{cases} \quad \begin{cases} Om(0, y, z) = B(z) \\ Om(\ddot{O}ka(x), y, z) = G(y) \end{cases}$$

med $B = G = \text{Identitet}^{\dagger}$.

†. Se EXEMPEL 5.2 på sidan 143.

Typiska användningar av *Om*-funktionen är att beräkna falldefinierade funktioner:

Dessa *Om*-konstruktionen är primitivt rekursiva ifall de ingående funktionerna är det.

$$Om(P(\bar{x}), F(\bar{x}), G(\bar{x})) \quad \text{vilken beräknar} \quad \begin{cases} F(\bar{x}) & \text{om } P(\bar{x}) > 0 \\ G(\bar{x}) & \text{om } P(\bar{x}) = 0 \end{cases}$$

$$Om(P(\bar{x}), Om(Q(\bar{x}), F(\bar{x}), G(\bar{x})), H(\bar{x}))$$

Den senare som kanske hellre skrivs

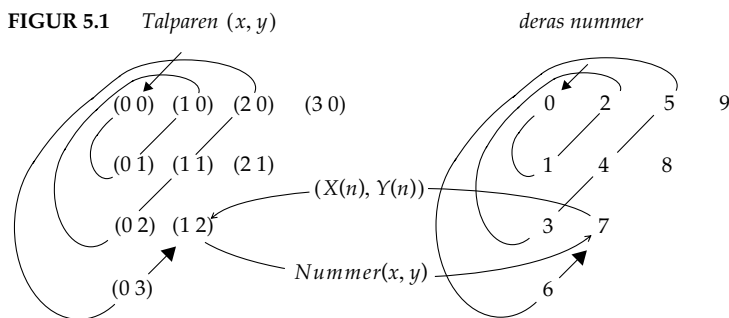
†. John McCarthy introducerade på 60-talet denna formalismen med innebörden "*Om* $P(x)$ är sann returnera $F(x)$ annars $G(x)$ ". Se McCarthy (1963), *A basis for a mathematical theory of computation* Computer Programming and formal system, Braffort and Hirschberg, NorthHolland, Amsterdam.

$$\begin{aligned}
 &Om(P(\bar{x}), \\
 &\quad Om(Q(\bar{x}), F(\bar{x}), G(\bar{x})), \\
 &\quad H(\bar{x}))
 \end{aligned}$$

beräknar en trefallsdefinierad funktion:

$$\begin{cases} F(\bar{x}) & \text{om } P(\bar{x}) > 0 \text{ och } Q(\bar{x}) > 0 \\ G(\bar{x}) & \text{om } P(\bar{x}) > 0 \text{ och } Q(\bar{x}) = 0 \\ H(\bar{x}) & \text{om } P(\bar{x}) = 0 \end{cases}$$

EXEMPEL 5.13 De pilmarkerade vägarna nedan visar ett klassiskt sätt att numrera talparen. Den omvändbara funktionen *Nummer* och dess omvändning funktionsparet (X, Y) (se figuren) kan beräknas primitivt rekursivt.



Notera att $x + y = d$ för alla talpar (x, y) som ligger på den d :te diagonalen i FIGUR 5.1 samt att den pilmarkerade vägen vid inträde i den d :te diagonalen har gått igenom ett *triangelformat* område med d rader. Genom att utnyttja dessa observationer kan man komponera följande funktion vilken tar *talparen* som input och returnerar deras *nummer* precis som i figuren ovanför.

$$Nummer(x, y) = Add(x, TriangelTal(Add(x, y)))$$

Omvänt tar nedanstående funktionspar $(X(n), Y(n))$ naturliga tal n som input och returnerar det *talpar* (x, y) vars *nummer* är just n .[†]

$$\begin{cases} X(0) = 0 \\ X(\ddot{O}ka(n)) = Om(TriangelTal?(\ddot{O}ka(n)), 0, \ddot{O}ka(X(n))) \end{cases}$$

†. *TriangelTal?* avser ett predikat som returnerar 1 om dess argument är ett triangelnummer, 0 annars. Försök själv konstruera detta predikat inom klassen av primitivt rekursiva funktioner.

$$\begin{cases} Y(0) = 0 \\ Y(\text{Öka}(n)) = \text{Om}(\text{Noll?}(Y(n)), \text{Öka}(X(n)), \text{Minska}(Y(n))) \end{cases}$$

■ Delbarhet och primitivt rekursiva funktioner

Att *dividera* ett naturligt tal x med ett (nollskilt) naturligt tal d innebär som bekant (sid 27) att hitta naturliga tal q och r sådana att

$$x = q \cdot d + r \text{ där } r < d$$

Talen q och r betecknas $Kvot(x, d)$ respektive $Rest(x, d)$ och uppför sig enligt ett enkelt mönster när ett växande x divideras med ett fixt d , se nedan. När x ökar från 0 och uppåt så gör $Rest(x, d)$ likadant, men istället för att nå värdet d faller $Rest(x, d)$ tillbaka till 0, och får börja om igen. Och $Kvot(x, d)$ som inledningsvis är 0 ökar med 1 varje gång som $Rest(x, d)$ faller till 0.

	$x = q \cdot d + r$	x	$q = Kvot(x, 4)$	$r = Rest(x, 4)$
0 dividerat med 4:	$0 = 0 \cdot 4 + 0$	0	0	0
1 dividerat med 4:	$1 = 0 \cdot 4 + 1$	1	0	1
2 dividerat med 4:	$2 = 0 \cdot 4 + 2$	2	0	2
3 dividerat med 4:	$3 = 0 \cdot 4 + 3$	3	0	3
4 dividerat med 4:	$4 = 1 \cdot 4 + 0$	4	1	0
5 dividerat med 4:	$5 = 1 \cdot 4 + 1$	5	1	1
6 dividerat med 4:	$6 = 1 \cdot 4 + 2$	6	1	2
7 dividerat med 4:	$7 = 1 \cdot 4 + 3$	7	1	3
8 dividerat med 4:	$8 = 2 \cdot 4 + 0$	8	2	0

Tabellen avslöjar hur man kan konstruera $Rest$ och $Kvot$ med hjälp av rekursion som löper d steg i taget:

$$\begin{aligned} Rest(x, d) &= \text{Om}(\text{Mindre}(x, d), \\ &\quad x, \\ &\quad Rest(\text{Sub}(x, d), d)) \end{aligned} \quad \begin{aligned} Kvot(x, d) &= \text{Om}(\text{Mindre}(x, d), \\ &\quad 0, \\ &\quad \text{Öka}(Kvot(\text{Sub}(x, d), d))) \end{aligned}$$

men också med primitivt rekursion:

$$\begin{cases} Rest(0, d) = 0 \\ Rest(\text{Öka}(x), d) = \text{Om}(\text{Mindre}(\text{Öka}(Rest(x, d)), d), \\ \quad \text{Öka}(Rest(x, d)), \\ \quad 0) \end{cases} \quad \begin{cases} Kvot(0, d) = 0 \\ Kvot(\text{Öka}(x), d) = \text{Add}(\text{Noll?}(Rest(\text{Öka}(x), d)), Kvot(x, d)) \end{cases}$$

Frågan "Är x delbar med d ?" besvaras av predikatet

$$\text{Delbar}(x, d) = \text{Noll?}(\text{Rest}(x, d))^{\dagger}$$

Och eftersom ett primtal x kännetecknas av att

x är lika med 2 eller att x är större än 2 och icke delbar med något enda av talen $2, 3, 4, \dots, x-1$,

så kan vi tillverka en primtalstestare på följande sätt:

$$\begin{aligned} \text{Prima}(x) = & \text{Eller}(\text{Lika}(x, 2), \\ & \text{Och}(\text{Större}(x, 2), \\ & \text{Icke}(\text{DelbarMedNågotTal}(x, 2, x-1)))) \end{aligned}$$

där $\text{DelbarMedNågotTal}(x, y, z)$ som skall svara "ja" omm x är delbar med något tal större eller lika med y men mindre eller lika med z , ges av nedanstående konstruktion[‡]:

$$\begin{aligned} \text{Delbar}(x, y) & \quad \text{DelbarMedNågotTal}(x, y, y) = \text{Delbar}(x, y) \\ \text{eller} & \\ \text{Delbar}(x, y+1) & \quad \text{DelbarMedNågotTal}(x, y, \text{Öka}(z)) = \\ \text{eller} & \\ \text{Delbar}(x, y+2) & \quad \text{Eller}(\text{DelbarMedNågotTal}(x, y, z), \\ \dots & \quad \text{Delbar}(x, \text{Öka}(z))) \\ \text{eller} & \\ \text{Delbar}(x, z) & \end{aligned}$$

✂ **TEST 5.1** Beräkna inom klassen av primitivt rekursiva funktioner

a) $x + y + z$, b) $\text{Udda}(x)$ resp. $\text{Jämn}(x)$,

c) $\text{Min}(x, y)$, (minimum av x, y) d) $\text{MinstAvTre}(x, y, z)$,

e) $\text{Fak}(x) = \prod_{n=1}^x n$ (Hitta på ett lämpligt basfall då $x = 0$.)

f) $F(x) = \prod_{n=0}^x G(x, n) = G(x, 0) \cdot G(x, 1) \cdot \dots \cdot G(x, x)$, där G är en godtycklig primitivt rekursiv funktion med två argument.

g) $F(x, y) = \overbrace{G(\dots(G(G(y))))}^{x \text{ stycken}} \dots$ där G är en godtycklig primitivt rekursiv funktion med ett argument.

†. $\text{Rest}(x, 0)$ och $\text{Kvot}(x, 0)$ är väldefinierade i vår konstruktion, den förstränkta med värde 0, varför $\text{Delbar}(x, 0) = 1$, något som *inte* står i samklang med den "gängse" uppfattningen om division med 0.

‡. Konstruktionen följer inte strikt den primitivt rekursiva mallen eftersom rekursionens botten *inte* ligger i 0. Kan Du korrigera så att konstruktionen blir strikt primitivt rekursiv?

■ 5.2 Primitiv rekursion är otillräcklig

EXEMPEL 5.14 År 1928 publicerade Ackermann ett recept på en sorts rekursiv funktion som besvarade en för den tiden aktuell fråga. Ackermann hade nämligen – med ett mer invecklat rekursionsförfarande än primitiv rekursion – lyckats konstruera en märklig funktion som han bevisade var omöjlig att konstruera inom klassen av primitivt rekursiva funktioner. Ackermann noterade att eftersom potensupphöjning är upprepade multiplikation som i sin tur är upprepade addition, så kan man genom att numrera *Add*, *Mult*, *Pot* :

$$Add = A_0$$

$$Mult = A_1$$

$$Pot = A_2$$

tillverka A_{n+1} med hjälp av A_n på följande sätt

$$\begin{cases} A_1(y, 0) = 0 \\ A_1(y, Öka(x)) = A_0(y, A_1(y, x)) \end{cases}$$

$$\text{ty} \quad \begin{cases} Mult(y, 0) = 0 \\ Mult(y, Öka(x)) = Add(y, Mult(y, x)) \end{cases}$$

$$\begin{cases} A_2(y, 0) = 1 \\ A_2(y, Öka(x)) = A_1(y, A_2(y, x)) \end{cases}$$

$$\text{ty} \quad \begin{cases} Pot(y, 0) = 1 \\ Pot(y, Öka(x)) = Mult(y, Pot(y, x)) \end{cases}$$

Nu undrar Du kanske vad man får om man upprepar potensupphöjning, och sedan upprepar "upprepade potensupphöjning" osv Dvs om man med hjälp av A_2 tillverkar A_3 :

$$\begin{cases} A_3(y, 0) = ? \\ A_3(y, Öka(x)) = A_2(y, A_3(y, x)) \end{cases}$$

Och sedan med hjälp av A_3 tillverkar A_4 :

$$\begin{cases} A_4(y, 0) = ? \\ A_4(y, Öka(x)) = A_3(y, A_4(y, x)) \end{cases}$$

Osv ...

$$\begin{cases} A_{\ddot{O}ka(n)}(y, 0) = ? \\ A_{\ddot{O}ka(n)}(y, \ddot{O}ka(x)) = A_n(y, A_{\ddot{O}ka(n)}(y, x)) \end{cases} \quad (2)$$

Få se ...

$$\begin{aligned} A_3(y, x) &= A_2(y, A_3(y, x-1)) \\ &= A_2(y, A_2(y, A_3(y, x-2))) \\ &= A_2(y, A_2(y, A_2(y, A_3(y, x-3)))) \\ &\dots \\ &= A_2(y, A_2(y, \dots A_2(y, A_3(y, x-x)) \dots)) \\ &= A_2(y, A_2(y, \dots A_2(y, A_3(y, 0)) \dots)) \\ &= y^{y^{\dots^y A_3(y, 0)}} \end{aligned}$$

På motsvarande sätt blir

$$\begin{aligned} A_4(y, x) &= A_3(y, A_4(y, x-1)) \\ &\dots \\ &= A_3(y, A_3(y, \dots A_3(y, A_4(y, 0)) \dots)) \end{aligned}$$

Ackermanns idé var att behandla funktionsindexet n i (2) som ett argument på jämbördig fot med funktionens två argument x, y :

$$\begin{cases} A(\ddot{O}ka(n), y, 0) = ? \\ A(\ddot{O}ka(n), y, \ddot{O}ka(x)) = A(n, y, A(\ddot{O}ka(n), y, x)) \end{cases} \quad (3)$$

I (3):s undre rad rekurerar argumenten n och x . Och i (3):s övre rad finns ett basfall skisserat för argumentet x . Ackermann preciserade detta basfall samt tillfogade ett basfall för n . (Här utelämnas de exakta uttrycken för dessa basfall):

$$\begin{cases} A(0, y, x) = ? \\ A(\ddot{O}ka(n), y, 0) = ? \\ A(\ddot{O}ka(n), y, \ddot{O}ka(x)) = A(n, y, A(\ddot{O}ka(n), y, x)) \end{cases} \quad (4)$$

Sedan bevisade han att den dubbelt rekursivt definierade funktionen A inte kunde beräknas med en primitivt rekursivt definierad funktion. Något senare presenterade den ungerska matematikern *R. Peter* en något förenklad version av Ackermanns originalfunktion. Här är receptet på *Peters* version:

$$\begin{cases} \text{Ack}(0, x) = \text{Öka}(x) \\ \text{Ack}(\text{Öka}(n), 0) = \text{Ack}(n, 1) \\ \text{Ack}(\text{Öka}(n), \text{Öka}(x)) = \text{Ack}(n, \text{Ack}(\text{Öka}(n), x)) \end{cases}$$

Man kan visa, att en sådan enorm tillväxt på output som Ackermanns funktion har, det kan ingen primitivt rekursiv funktion uppvisa – och knappast någon människa föreställa sig. När man tittar på de tre ekvationerna ovan, och ser ett enda ställe (den första ekvationen) där output ökar, och då bara med en futtig enhet. Och när man vidare konstaterar att startvärdet $\text{Ack}(0, 0)$ är 1, ja då är det svårt att föreställa sig att $\text{Ack}(4, 4)$ är så stort, att om vi fick använda alla pappersark som finns att uppbringa på vår jord för att skriva ut talet, så skulle de icke räcka till. Kosmos alla elementarpartiklar, som har uppskattats till cirka 10^{80} , räcker inte ens till att räkna antalet siffror i det här jättetalet.

$\text{Ack}(2, 2)$ är dock inte särskilt stor:

$$\begin{aligned} & \text{Ack}(2, 2) \\ &= \text{Ack}(1, \text{Ack}(2, 1)) \\ &= \text{Ack}(1, \text{Ack}(1, \text{Ack}(2, 0))) \\ &= \text{Ack}(1, \text{Ack}(1, \text{Ack}(1, 1))) \\ &= \text{Ack}(1, \text{Ack}(1, \text{Ack}(0, \text{Ack}(1, 0)))) \\ &= \text{Ack}(1, \text{Ack}(1, \text{Ack}(0, \text{Ack}(0, 1)))) \\ &= \text{Ack}(1, \text{Ack}(1, \text{Ack}(0, 2))) \\ &= \text{Ack}(1, \text{Ack}(1, 3)) \\ &= \text{Ack}(1, \text{Ack}(0, \text{Ack}(1, 2))) \\ &= \text{Ack}(1, \text{Ack}(0, \text{Ack}(0, \text{Ack}(1, 1)))) \\ &= \text{Ack}(1, \text{Ack}(0, \text{Ack}(0, \text{Ack}(0, \text{Ack}(1, 0))))) \\ &= \text{Ack}(1, \text{Ack}(0, \text{Ack}(0, \text{Ack}(0, \text{Ack}(0, 1))))) \\ &= \text{Ack}(1, \text{Ack}(0, \text{Ack}(0, \text{Ack}(0, 2)))) \\ &= \text{Ack}(1, \text{Ack}(0, \text{Ack}(0, 3))) \\ &= \text{Ack}(1, \text{Ack}(0, 4)) \\ &= \dots \\ &= 7 \end{aligned}$$

EXEMPEL 5.15 Det finns intressanta funktioner som bara med stort besvär låter sig fångas med primitiv rekursion, men som kan beskrivs mycket enkelt med annan typ av rekursion. Fibonaccifunktionen är en sådan,

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$Fib(x)$	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	...

Funktionens två första output är 0 respektive 1, och övriga output beräknas som summan av de två föregående.

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3+2 \quad \dots$$

Härav följande enkla rekursiva – men ej primitivt rekursiva – presentation.

$$\begin{cases} Fib(0) = 0 \\ Fib(1) = 1 \\ Fib(x+2) = Add(Fib(x+1), Fib(x)) \end{cases} \quad (5)$$

■ En primitivt rekursiv konstruktion av $Fib(x)$

Konstruktionen av $Fib(x)$ i EXEMPEL 5.15 följer inte den primitivt rekursiva mallen eftersom ett Fibonaccital i detta exempel *inte* beräknas med hjälp av ett *enbart* det föregående Fibonaccitalet *utan* med hjälp av *ett par* av två föregående Fibonaccital.

Men, om vi istället för att betrakta enstaka Fibonaccital,

$$0, 1, 1, 2, 3, 5, \dots$$

betraktar *par* av dem

$$(0, 1), (1, 1), (1, 2), (2, 3), (3, 5), \dots$$

så kan man med hjälp av ett sådant par $(Fib(t), Fib(t+1))$ beräkna nästa par som

$$(Fib(t+1), Add(Fib(t), Fib(t+1)))$$

Dvs, med hjälp av ett objekt kan nästa objekt beräknas, vilket är den primitiva rekursionens kännetecken.

Genom att betrakta objekt som är par blir således själva rekursionen primitiv. För att få konstruktionen att passa in i mallen för primitiv rekursion återstår därför "bara" att göra en transformation från objekt som är par av naturliga tal till objekt som är enstaka naturliga tal. Vi skall här göra en sådan transformation med hjälp av den primitivt rekursiva funktionen $Nummer(x, y)$ och dess omvändning, funktionsparet $(X(n), Y(n))$ från EXEMPEL 5.13 på sidan 150:

$$Nummer(X(n), Y(n)) = n, \quad \begin{cases} X(Nummer(x, y)) = x \\ Y(Nummer(x, y)) = y \end{cases} \quad (6)$$

Bilda funktionen

$$F(t) = \text{Nummer}(\text{Fib}(t-1), \text{Fib}(t)) \quad (7)$$

Av (6) och (7) följer att *Fib* kan beräknas som

$$\text{Fib}(t) = Y(F(t)) \quad (8)$$

vilket visar att om *F* är primitivt rekursiv så gäller (eftersom funktionen *Y*(*n*) är primitivt rekursiv) samma sak om *Fib*. Därför försöker vi nu visa att *F* är primitivt rekursiv.

Av (6) och (7) följer

$$\begin{aligned} X(F(t)) &= \text{Fib}(t-1) \\ Y(F(t)) &= \text{Fib}(t) \end{aligned} \quad (9)$$

Med hjälp av (5) och (7) får vi

$$F(t+1) = \text{Nummer}(\text{Fib}(t), \text{Add}(\text{Fib}(t), \text{Fib}(t-1))) \quad (10)$$

som med hjälp av de två likheterna i (9) övergår i

$$F(\text{Öka}(t)) = \text{Nummer}(Y(F(t)), \text{Add}(Y(F(t)), X(F(t)))) \quad (11)$$

Som synes är (11) ett rekursionssteg som följer den primitivt rekursiva mallen. Kompletterad med ett lämpligt basfall (tänk efter vilket tal som passar som värde åt *F*(0)) får vi därför en *primitivt rekursiv* konstruktion av funktionen *F*:

$$\begin{cases} F(0) = 2 \\ F(\text{Öka}(t)) = \text{Nummer}(Y(F(t)), \text{Add}(Y(F(t)), X(F(t)))) \end{cases} \quad (12)$$

vilket var allt vi behövde ha som komplement till (8).

■ Rekursiva flätor

En rekursiv fläta är en rekursiv konstruktion där två eller flera funktioner anropar varandra (och kanske också sig själva) under rekursionen på ett sätt som gör att ingen av dem står på egna ben.

Här är ett enkelt (och vackert) exempel:

$$\begin{cases} \text{Jämn}(0) = 1 \\ \text{Udda}(0) = 0 \\ \text{Jämn}(\text{Öka}(x)) = \text{Udda}(x) \\ \text{Udda}(\text{Öka}(x)) = \text{Jämn}(x) \end{cases}$$

Bortsett från ihopflätandet av de två funktionerna, som inte är tillåtet i den primitivt rekursiva mallen, rekurserar argumentet bakåt *ett* steg – precis som det skall vara vid primitiv rekursion.

Mer allmänt (men inte alltför allmänt), betrakta följande rekursiva fläta av två funktioner F_1 och F_2 :

En rekursiv
fläta där re-
kursionen lö-
per ett steg
bakåt.

$$\begin{cases} F_1(t, 0) = B_1(t) \\ F_2(t, 0) = B_2(t) \\ F_1(t, \ddot{O}ka(y)) = G_1(F_1(t, y), F_2(t, y)) \\ F_2(t, \ddot{O}ka(y)) = G_2(F_1(t, y), F_2(t, y)) \end{cases} \quad (13)$$

Vi skall nu visa hur man – åter med hjälp av den primitivt rekursiva *Nummer*-funktionen och det primitivt rekursiva funktionsparet (X, Y) – kan lösa upp flätan så att de två funktionerna blir oberoende av varandra och placerade inom klassen av primitivt rekursiva funktioner.

Sätt

$$F(t, y) = \text{Nummer}(F_1(t, y), F_2(t, y)) \quad (14)$$

Då är p.g.av (6)

$$F_1(t, y) = X(F(t, y)) \quad (15)$$

$$F_2(t, y) = Y(F(t, y)) \quad (16)$$

vilket visar att om F är primitivt rekursiv så är F_1 och F_2 primitivt rekursiva.

Resten av vårt arbete går nu ut på att ge en primitivt rekursiv konstruktion av F , något som går mer eller mindre automatiskt bara vi "pusslar ihop alla delar" rätt ...

Med hjälp av (15), (16) omformas nu de två sista raderna i (13) till

$$F_1(t, \ddot{O}ka(y)) = G_1(X(F(t, y)), Y(F(t, y))) \quad (17)$$

$$F_2(t, \ddot{O}ka(y)) = G_2(X(F(t, y)), Y(F(t, y))) \quad (18)$$

Slutligen ger (17), (18) och (14):

$$\begin{aligned} F(t, \ddot{O}ka(y)) &= \text{Nummer}(F_1(t, \ddot{O}ka(y)), F_2(t, \ddot{O}ka(y))) \\ &= \text{Nummer}(G_1(X(F(t, y)), Y(F(t, y))), \\ &\quad G_2(X(F(t, y)), Y(F(t, y)))) \end{aligned} \quad (19)$$

och vi har ett perfekt rekursionssteg i en primitivt rekursiv konstruktion av F . Ett basfall dyker upp ur (14) och de övre två raderna i (13):

$$F(t, 0) = \text{Nummer}(B_1(t), B_2(t)) \quad (20)$$

Därmed är den primitivt rekursiva konstruktionen av F klar, och av (15), (16) följer, som vi redan har påpekat, att F_1 och F_2 blir primitivt rekursiva.

■ Primtal och framåtrekursion

Antag att vi vill ha en funktion $Primtal(x)$ som returnerar primtalen i tur och ordning, dvs så att

$$Primtal(0) = 2, \quad Primtal(1) = 3, \quad Primtal(2) = 5, \quad \dots$$

Här kommer en rekursiv algoritm som beräknar denna funktion. Algoritmens idé är att utgående från föregående primtal söka nästkommande primtal genom att stega framåt med en enhet i taget tills ett primtal påträffas.

$$\begin{cases} Primtal(0) = 2 \\ Primtal(\ddot{O}ka(x)) = H(\ddot{O}ka(Primtal(x))) \end{cases}$$

$$H(y) = Om(Prima(y), y, H(\ddot{O}ka(y)))$$

I hjälpfunktionen H rekurserar y framåt (istället för bakåt). En rekursiv konstruktion skild från primitivt rekursion således. (Det går att konstruera en primitivt rekursiv primtalsuppräknare, dock inte lika enkelt.)

⌘ Anm.5.5 Eftersom en framåtrekursion aldrig bottenar, så kan $H(y)$ inte beräknas med beräkningsförfarandet *inifrån och utåt*. Beräkningen skulle nämligen därvid aldrig bli färdig:

$$\begin{aligned} H(4) &= Om(Prima(4), 4, H_{\Delta}(5)) \\ &= Om(Prima(4), 4, Om(Prima(5), 5, H_{\Delta}(6))) \\ &= Om(Prima(4), 4, Om(Prima(5), 5, Om(Prima(6), 6, H_{\Delta}(7)))) \\ &\vdots \end{aligned}$$

Dock kommer beräkningsförfarandet *utifrån och inåt* att ge upphov till en ändlig beräkning av $H(y)$:[†]

$$\begin{aligned} H(4) &= Om_{\Delta}(Prima(4), 4, H(5)) = Om_{\Delta}(Prima(4), 4, H(5)) = Om_{\Delta}(0, 4, H(5)) \\ &= H(5) \\ &= Om_{\Delta}(Prima(5), 5, H(6)) = Om_{\Delta}(Prima(5), 5, H(6)) = Om_{\Delta}(1, 5, H(6)) \\ &= 5 \end{aligned}$$

†. Notera att i detta beräkningsförfarande Om -funktionen *inte* skall behöva utföra några "onödiga" argumentsberäkningar, dvs ifall dess första argument är positivt, så behöver det tredje argumentet aldrig beräknas (och ifall det första argumentet är noll, så behöver enbart det tredje argumentet beräknas).

■ 5.3 Rekursiva funktioner – en utvidgning av primitivt rekursiva funktioner

I förra avsnittet mötte vi funktioner – $Fib(x)$ och $Primtal(x)$ – som visserligen kan konstrueras med primitiv rekursion, men som konstrueras naturligare med annan rekursion. Detta visar på ett praktiskt behov av annan rekursion än primitiv rekursion.

Men vi träffade också på en funktion $Ack(x, y)$ som de facto är omöjlig att konstruera inom klassen primitivt rekursiva funktioner, vilket visar att de primitivt rekursiva funktionerna bara ofullständigt fångar idén om det algoritmiska.

För att fånga det algoritmiska måste vi tydligen *utvidga* klassen av primitivt rekursiva funktioner. Dvs tillföra mer uttryckskraft. Att rekursera bakåt ett steg är inte nog!

En extrem lösning (som tex funktionella programmeringsspråk brukar välja) är att tillåta funktioners argument att rekursera "fritt", dvs att låta funktioner beräkna sina output med vilka som helst av sina egna och andra funktioners output.

Men i själva verket räcker det med en mycket måttfull utvidgning.

■ Framåtrekursion

Med mallen nedan kan man beräkna $F(\bar{x})$ på ett helt mekaniskt sätt, givet att $G(\bar{x}, y)$ och $P(\bar{x}, y)$ kan beräknas mekaniskt.

Framåtrekursion för F med ett eller flera argument.

$$F(\bar{x}) = H(\bar{x}, 0)$$

$$H(\bar{x}, y) = Om(P(\bar{x}, y), G(\bar{x}, y), H(\bar{x}, Öka(y)))$$

⌘ Anm.5.6 Beräkningsförfarandet för Om -funktionen skall här följa filosofin "*inga onödiga argumentsberäkningar*", dvs ifall dess första argument är positivt, så skall det andra argumentet returneras och det tredje aldrig beräknas. Och ifall det första argumentet är noll, så skall det tredje argumentet returneras varvid det andra aldrig beräknas. Slutligen, ifall det första argumentet inte har något värde p.g. av att det innehåller en *partiellt* definierad funktion, (angående sådana funktioner se Anm.5.8 på sidan 161), så kan Om -funktionen inte heller returnera något värde (beräkningen *hänger sig*).

⌘ Anm.5.7 Notera att y rekurserar *framåt* så länge $P(\bar{x}, y) = 0$. Om $P(\bar{x}, y) = 0$ för alla $y < y_0$, men $P(\bar{x}, y_0) \neq 0$ så upphör rekursionen just när $y = y_0$ varvid $G(\bar{x}, y_0)$ returneras som $F(\bar{x})$'s värde:

$$\begin{aligned}
 F(\bar{x}) &= H(\bar{x}, 0) \\
 &= H(\bar{x}, 1) \text{ om } P(\bar{x}, 0) = 0 \\
 &= H(\bar{x}, 2) \text{ om } P(\bar{x}, 1) = 0 \\
 &= H(\bar{x}, 3) \text{ om } P(\bar{x}, 2) = 0 \\
 &\vdots \\
 &= G(\bar{x}, y_0) \text{ om } P(\bar{x}, y_0) \neq 0
 \end{aligned}$$

⌘ Anm.5.8 Genom *framåtrekursion* introduceras *partiellt definierade* F . Ty om det finns något \bar{x} sådant att $P(\bar{x}, y) = 0$ för alla y så kommer framåtrekursionen aldrig att upphöra. Därmed kommer H aldrig att kunna ge F något värde. Man säger att F är *odefinierad* för sådant \bar{x} .

Om för något \bar{x} , $P(\bar{x}, y_0)$ är odefinierad, men $P(\bar{x}, y) = 0$ för alla $y < y_0$, så blir $F(\bar{x})$ också odefinierad. Ty då rekurerar y fram till y_0 där beräkningen "hänger sig".

⌘ Anm.5.9 I det specialfall att $G(\bar{x}, y) = y$, så får den framåtrekurerande mallen följande form:

$$\begin{aligned}
 F(\bar{x}) &= H(\bar{x}, 0) \\
 H(\bar{x}, y) &= Om(P(\bar{x}, y), y, H(\bar{x}, Öka(y)))
 \end{aligned}$$

Obegränsad
minimering

vilket innebär att F returnerar *det minsta* y :et sådant att $P(\bar{x}, y) \neq 0$, om det finns något sådant y .

EXEMPEL 5.16 *Primtalsuppräknaren* konstruerades på sid 159 med en sorts framåtrekursion som *inte exakt* följer mallen för framåtrekursion. Genom att ändra lite i konstruktionen kan man dock få *primtalsuppräknarens* framåtrekursion att bli "regelmässig":

$$\begin{aligned}
 Primtal(0) &= 2 \\
 Primtal(Öka(n)) &= F(Primtal(n)) \\
 F(x) &= H(x, 0) \\
 H(x, y) &= Om(P(x, y), y, H(x, Öka(y))) \\
 P(x, y) &= Och(Prima(y), Mindre(x, y))
 \end{aligned}$$

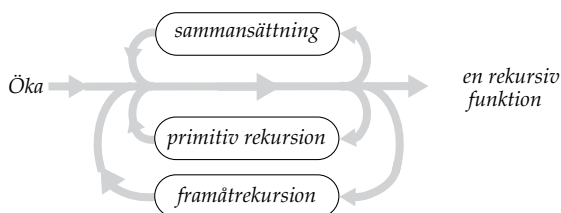
Lägg märke till att vi i denna konstruktion använder *samtliga* hittills introducerade operationer – *sammansättning*, *primitiv rekursion* och *framåtrekursion*.

■ Klassen av rekursiva funktioner

DEFINITION Med klassen av rekursiva funktioner menas mängden av funktioner som kan tillverkas från basfunktionen $Öka$ genom att operera noll eller flera (ändligt många) gånger med *sammansättning*, *primitiv rekursion*

och/eller framåtrekursion.

Så här
tillverkas
rekursiva
funktioner.



EXEMPEL 5.17 Som påpekats i Anm.5.8 kan framåtrekursion ge upphov till partiellt definierade funktioner. Här är ett trivialt exempel.

$$F(x) = H(x, 0)$$

$$H(x, y) = \text{Om}(\text{Ett?}(\text{Mult}(x, y)), y, H(x, \text{Öka}(y)))$$

$F(x) = 1$ för $x = 1$, men odefinierad för övriga x .

(För $x \neq 1$ finns det *ingen* naturligt tal y sådant att $x \cdot y = 1$.)

EXEMPEL 5.18 Vi har tidigare visat (se på sidan 151) hur heltalsdivision kan utföras inom klassen av primitivt rekursiva funktioner. $Kvot(x, y)$ blev därvid definierad för alla x, y (speciellt för nämnaren $y = 0$) vilket, som vi då påpekade, inte stämmer överens med det gängse sättet att se på division. Med hjälp av framåtrekursion kan man dock konstruera en funktion som beräknar $Kvot(x, y)$ på vanligt sätt, dvs så att kvoten blir odefinierad för nämnaren 0:

$$Kvot(x, y) = H(x, y, 0)$$

$$H(x, y, z) = \text{Om}(\text{Mindre}(x, \text{Mult}(y, z)), z - 1, H(x, y, z + 1))$$

För givna x, y letar funktionen $Kvot$ (med H 's hjälp) upp det minsta naturliga tal z sådant att $x < y \cdot z$, (om sådant z finns) varefter detta z minskas ett steg. Eftersom det *inte* finns något naturligt tal z sådant att $x < 0 \cdot z$, blir emellertid $Kvot(x, 0)$ odefinierad.

$$\begin{array}{ll} Kvot(5, 2) = H(5, 2, 0) & Kvot(5, 0) = H(5, 0, 0) \\ = H(5, 2, 1) & = H(5, 0, 1) \\ = H(5, 2, 2) & = H(5, 0, 2) \\ = H(5, 2, 3) & = H(5, 0, 3) \\ = 3 - 1 \text{ ty } 5 < 2 \cdot 3 & = H(5, 0, 4) \\ = 2 & \vdots \end{array}$$

EXEMPEL 5.19 Vi konstruerar här en rekursiv funktion F så att dess *värdeområde* är skärningen mellan två givna rekursiva funktioners värdeområden. Vår första konstruktion bygger på att de två givna funktionerna G_1 och G_2 är definierade på hela \mathbb{N} :

$$F(x) = H(x, 0)$$

$$H(x, y) = Om(Lika(G_1(x), G_2(y)), G_2(y), H(x, Öka(y)))$$

Idén med ovanstående konstruktion är att för varje output $G_1(x)$ från G_1 , leta med hjälp av framåtrekursion bland G_2 :s värden efter samma output. Om G_2 också har det aktuella talet som output stannar rekursionen och talet ifråga returneras. Annars stannar aldrig rekursionen, och F blir odefinierad för input x .

Tyvärr kommer den här konstruktionen inte att fungera tillfredsställande om G_2 är odefinierad för något y . Ty vid ett sådant y hänger sig framåtrekursionen, varför efterkommande värden inte kan undersökas. För *partiellt definierade* G_2 måste vi därför försöka hitta på något annat...

Om man analyserar föregående konstruktion finner man att det bristfälliga hänger samman med det sätt som framåtrekursionen används: att för ett fixt värde på den första funktionen söka bland den andra funktionens värden efter ett likadant värde. På detta sätt blir nämligen varje beräkning av $Lika(G_1(x), G_2(y))$ beroende av att föregående beräkning, den av $Lika(G_1(x), G_2(y-1))$ inte hänger sig.

Följande konstruktion använder framåtrekursionen på ett helt annat sätt. Med hjälp av funktionerna $X(n)$ och $Y(n)$ från EXEMPEL 5.13 beräknas nu $Lika(G_1(X(n)), G_2(Y(n)))$ för varje par $X(n), Y(n)$ av naturliga tal, varmed ingen enda likhetsundersökning blir ogjord. Framåtrekursionen är här enbart ett slags "skräptillstånd" som gör funktionen odefinierad när de jämförda värdena icke överensstämmer.

$$F(n) = H(n, 0)$$

$$H(n, z) = Om(Lika(G_1(X(n)), G_2(Y(n))), G_2(Y(n)), H(n, Öka(z)))$$

✂ TEST 5.2 Konstruera

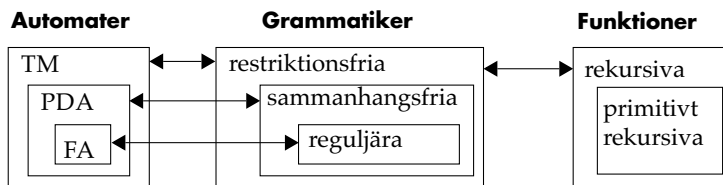
$$F(x) = \text{det minsta naturliga tal } y \text{ sådant att } x \cdot y > x + y$$

inom klassen av rekursiva funktioner. Vilka x är F odefinierad för?

5.4 Turings tes och Church:s tes

Vi har sett exempel på tre olika algoritmiska koncept: *automater*, *grammatiker* och *rekursivt konstruerade funktioner*. Inom var och en av de tre koncepterna har vi presenterat en hierarki av allt kraftfullare typer av algoritmer.

Tre
algoritmiska
koncept.



Dessutom har vi sett hur de olika automaterna och grammatikerna är relaterade till varandra. Speciellt att Turingmaskiner accepterar precis sådana språk som kan beskrivas med de kraftfullaste grammatikerna – de restriktionsfria – vilket kan ses som ett stöd för att Turingmaskiner representerar det *algoritmiskt* optimala. Denna uppfattning har kommit att kallas för *Turings tes*.

I själva verket är Turingmaskiner också ekvivalenta med rekursiva funktioner i en mening som vi skall formalisera snart. Att rekursiva funktioners kompetens är likvärdig med Turingmaskiners är ett stöd för *Church:s tes*: att de rekursiva funktionerna representerar det *algoritmiskt* mest uttrycksfulla.

Lägg märke till att det *inte* är bevisat att *Turings* och *Church:s teser* skulle vara sanna, vilket har att göra med svårigheten (omöjligheten?) att definiera begreppet *algoritmisk* på ett matematiskt användbart sätt? En *intuitiv känsla* för vad ordet representerar har väl varje programmerare och matematiker. Men att precis definiera begreppet är något helt annat. Det enda vettiga är måhända att tala i termer av specifika koncept som Turingmaskiner, restriktionsfria grammatiker, rekursiva funktioner eller något annat av alla de koncept som alltsedan 1930-talet presenterats av olika forskare och som alla visat sig vara ekvivalenta.

Att så många olika forskare har formulerat "*det algoritmiska*" i skilda men ekvivalenta koncept är faktiskt det enda stödet för *Turings* och *Church:s teser*.

5.5 För varje TM finns det en rekursiv funktion

Vi har tidigare sett hur en TM kan fungera som en *språkaccepterare*, eller som en *funktionsberäknare*.[†]

I själva verket kan man se *varje* TM som en funktionsberäknare genom att uppfatta tapens innehåll om och när Turingmaskinen stannar som output. (Om Turingmaskinen för ett input aldrig stannar är motsvarande funktion odefinierad för detta input.)

[†]. Bland de *funktionsberäknande* Turingmaskinerna finns även de *språkavgörande*, vilka är funktionsberäknare som returnerar sanningsvärden.

Vidare kan man, se nedan, tolka tapens innehåll som tal, och därmed uppfatta Turingmaskiner som beräknande heltalsfunktioner. Vi ska visa att varje sådan Turingmaskinberäknad heltalsfunktion tillhör klassen av rekursiva funktioner.

■ Från godtyckliga teckensträngar till tal

Det är tillräckligt att betrakta Turingmaskiner vars taper har oändlig utsträckning till höger men inte till vänster, eftersom varje sådan TM kan härmas med en TM utrustad med gängse tape. (Halvoändliga taper gör att det blir enklare att identifiera tapeinnehåll med naturliga tal.) För en TM som arbetar med k tecken (blanktecknet # inkluderat) låter vi # identifieras med talet 0, och övriga tecken med talen 1, 2, ..., $k-1$. En godtycklig sträng $w = a_1 a_2 \dots a_m$ till vänster om den oändliga blankteckensvansen kan därmed identifieras med ett naturligt tal skrivet i k -systemet:

teckensträng		naturligt tal
$a_1 a_2 \dots a_m \# \dots$	\leftrightarrow	$a_1 + a_2 k^1 + \dots + a_m k^{m-1}$

Lägg märke till att i $a_1 a_2 \dots a_m \# \dots$ representerar a_n ett tecken i TM:ens tapealfabet, men i $a_1 + a_2 k^1 + \dots + a_m k^{m-1}$ representerar a_n ett naturligt tal. Eftersom den här identifieringen är *entydig*, (*olika* strängar identifieras med *olika* tal), kan man med hjälp av talet som representerar en teckensträng återskapa teckensträngen ifråga.

EXEMPEL 5.20 För en TM med två tecken a , b i sitt alfabet blir tapeinnehållet identifierat med ett naturligt tal skrivet i 3-systemet:

teckensträng		naturligt tal
$aab \# \dots$	\leftrightarrow	$1 + 1 \cdot 3^1 + 2 \cdot 3^2 = 22$
$\#aaa \# \dots$	\leftrightarrow	$0 + 1 \cdot 3^1 + 1 \cdot 3^2 + 1 \cdot 3^3 = 39$
$\#aba \# \dots$	\leftrightarrow	$0 + 1 \cdot 3^1 + 2 \cdot 3^2 + 1 \cdot 3^3 = 48$
$\#aab \# \dots$	\leftrightarrow	$0 + 1 \cdot 3^1 + 1 \cdot 3^2 + 2 \cdot 3^3 = 66$
$\#bab \# \dots$	\leftrightarrow	$0 + 2 \cdot 3^1 + 1 \cdot 3^2 + 2 \cdot 3^3 = 69$

■ Några Turingmaskinoperationer tolkade som primitivt rekursiva aritmetiska beräkningar

Intressant att notera är hur ovanstående identifikation mellan teckensträngar och naturliga tal leder till att några fundamentala Turingmaskinoperationer tolkas som lika fundamentala aritmetiska beräkningar inom klassen av primitivt rekursiva funktioner. Se här ...

► **Shifta** Om läshuvudet står längst ut till vänster kommer *vänstershiftning* att tolkas som division, och *högershiftning* som multiplikation:

$$\begin{aligned} w = a_1 a_2 a_3 \dots a_m \# \dots &\Leftrightarrow a_1 + a_2 k^1 + a_3 k^2 + \dots + a_m k^{m-1} = w \\ S_L(w) = a_2 a_3 \dots a_m \# \dots &\Leftrightarrow a_2 + a_3 k^1 + \dots + a_m k^{m-2} = Kvot(w, k) \\ S_R(w) = \# a_1 a_2 \dots a_m \# \dots &\Leftrightarrow a_1 k^1 + a_2 k^2 + \dots + a_m k^m = Mult(w, k) \end{aligned}$$

EXEMPEL 5.21 Vänstershiftningen $S_L(\#bab\# \dots) = bab\# \dots$ tolkas som beräkningen $Kvot(69, 3) = 23$, och högershiftningen $S_R(bab\# \dots) = \#bab\# \dots$ som $Mult(23, 3) = 69$.

Lägg också märke till att shiftning n steg motsvarar division respektive multiplikation med k^n . Förresten, kan Du svara på följande fråga: Blir resultatet detsamma om man vänstershiftar $n-1$ steg som om man först vänstershiftar n steg och sedan högershiftar *ett* steg? Dvs är det någon skillnad mellan $S_L^{n-1}(w)$ och $S_R(S_L^n(w))$?

Få se ..., i första fallet vänstershiftar vi bort de $n-1$ första tecknen, i det andra fallet börjar vi med att vänstershifta bort de n första tecknen varefter vi genom högershiftning tillfogar ett blanktecken i strängens början. Uppenbarligen blir resultaten olika. (Se nedan.) Syntaktiskt sett skiljer sig de resulterande strängarna åt i vänstra ändpositionen. De tal som strängarna identifieras med skiljer sig på motsvarande sätt:

$$\begin{aligned} S_L^{n-1}(w) = a_n a_{n+1} \dots a_m \# \dots &\Leftrightarrow a_n + a_{n+1} k^1 + \dots + a_m k^{m-n} = Kvot(w, Pot(k, n-1)) \\ S_R(S_L^n(w)) = \# a_{n+1} \dots a_m \# \dots &\Leftrightarrow a_{n+1} k^1 + \dots + a_m k^{m-n} = Mult(Kvot(w, Pot(k, n)), k) \end{aligned}$$

Skillnaden mellan $Kvot(w, Pot(k, n-1))$ och $Mult(Kvot(w, Pot(k, n)), k)$ är a_n , dvs innehållet i w 's n :te cell (tolkad som ett naturligt tal). Detta visar att *symbolen i w 's n :te cell (tolkad som ett naturligt tal) är*

$$\begin{aligned} Symbol(w, n) = Sub(Kvot(w, Pot(k, n-1)), \\ Mult(Kvot(w, Pot(k, n)), k)) \end{aligned} \quad (21)$$

► **Skriva** Turingmaskinernas förmåga att kunna skriva tecken var som helst på tapen är givetvis helt avgörande för deras kompetens. Genom identifikationen mellan strängar och naturliga tal (Se på sidan 165), kommer sådant tapeklottrande att tolkas som aritmetiska beräkningar. Vilken typ av aritmetisk beräkning det rör sig om kan man förstå efter att ha betraktat tabellen nedan.

$$\begin{aligned} w = a_1 \dots a_n \dots a_m \# \dots &\Leftrightarrow a_1 + \dots + a_n k^{n-1} + \dots + a_m k^{m-1} = w \\ w' = a_1 \dots a \dots a_m \# \dots &\Leftrightarrow a_1 + \dots + a k^{n-1} + \dots + a_m k^{m-1} = w' \end{aligned}$$

Som synes är $w' = w - a_n k^{n-1} + a k^{n-1}$ vilket visar att "skriva a i tapens n :te cell" har följande aritmetiska tolkning:

$$\text{Skriv}(a, w, n) = \text{Add}(\text{Sub}(w, \text{Mult}(\text{Symbol}(w, n), \text{Pot}(k, n-1))), \text{Mult}(a, \text{Pot}(k, n-1))) \quad (22)$$

EXEMPEL 5.22 Betrakta tapen # $aba\# \dots$, som identifieras med talet $0 + 1 \cdot 3 + 2 \cdot 9 + 1 \cdot 27 = 48$. "Skriva a i tapens 3:e cell" (denna cell hyser för närvarande tecknet b) tolkas som

$$\begin{aligned} \text{Skriv}(1, 48, 3) &= \text{Add}(\text{Sub}(48, \text{Mult}(\text{Symbol}(48, 3), \text{Pot}(3, 2))), \\ &\quad \text{Mult}(1, \text{Pot}(3, 2))) \\ &= 39 \end{aligned}$$

Kontroll: Den nyskrivna tapen # $aaa\# \dots$ identifieras således med talet 39.
 $0 + 1 \cdot 3 + 1 \cdot 9 + 1 \cdot 27 =$

■ Konfigurationerna som taltriplar

En Turingmaskins konfiguration beskriver som Du vet nuläget i en Turingmaskins beräkningar.

Det här avsnittets egentliga konstaterande är att den information som en konfiguration bär på finns att hämta i en taltrippel (q, w, n) , där q är numret på Turingmaskinens *nuvarande tillstånd*[†], w är tapens *innehåll* till vänster om den oändliga blankteckensvansen *tolkad som ett naturligt tal* och n är *läshuvudets position*[‡]. Att tecknet som läshuvudet står på kan beräknas utifrån taltrippeln bevisas av vår formel (21) på sidan 166 som ju beskriver hur symbolen i tapens n :te cell kan beräknas.

EXEMPEL 5.23 För t ex en Turingmaskin med inputalfabet $\{a, b\}$ och konfigurationen $(q_3, \#aba)$ är motsvarande taltrippel $(3, 48, 2)$. Här representeras q_3 av talet 3, $\#aba$ av talet 48, och läshuvudets position av talet 2. T ex kan den 2:a cellens tecken " a ", som identifieras med det naturliga talet 1, i beräknas via formeln (21) på följande sätt:

$$\begin{aligned} \text{Symbol}(48, 2) &= \text{Sub}(\text{Kvot}(48, \text{Pot}(3, 1)), \text{Mult}(3, \text{Kvot}(48, \text{Pot}(3, 2)))) \\ &= \text{Sub}(\text{Kvot}(48, 3), \text{Mult}(3, \text{Kvot}(48, 9))) \\ &= \text{Sub}(16, \text{Mult}(3, 5)) \\ &= 1 \end{aligned}$$

†. Turingmaskinens tillstånd antas vara numrerade på något sätt.

‡. Vi säger att tapens första cell har position 1. Position 0 reserverar vi som en markering av att läshuvudet har fallit av tapen.

■ Konfigurationernas omvandlingar tolkade som primitivt rekursiva beräkningar

Konfigurationernas omvandlingar

$$(\text{tillstånd}, \text{tape}, \text{position}) \rightarrow (\text{nästa tillstånd}, \text{nästa tape}, \text{nästa position})$$

beskriver Turingmaskinens förehavanden.

Positionen ändras endast om Turingmaskinens *läshuvud* flyttar sig. Och *tapen* ändras enbart om Turingmaskinen *skriver något*.

För att kunna beräkna *nästa position* och *nästa tape* måste man således veta *huruvida Turingmaskinen flyttar läshuvudet eller skriver*. Det som avgör det senare är Turingmaskinens övergångsfunktion

$$\delta(\text{tillstånd}, \text{symbol}) = (\text{nästa tillstånd}, \text{agerande})$$

där *agerande* är någon av riktningssymbolerna L, R om Turingmaskinen flyttar läshuvudet, och en tapesymbol a om den skriver.[†] Även beräkningen av *nästa tillstånd* måste gå via δ .

Eftersom δ är definierad för ändligt antal tillstånd och ändligt antal tecken, kan *nästa tillstånd* och *agerandet* beräknas primitivt rekursivt av falldefinierade funktioner vars exakta form bestäms av den aktuella Turingmaskinens tillståndsövergångar:

$$\begin{aligned} \text{NästaTillstånd}(q, w, n) = & \text{Om}(\text{Lika}(q, 0), \\ & \text{Om}(\text{Lika}(\text{Symbol}(w, n), 0), \\ & \dots) \end{aligned}$$

$$\begin{aligned} \text{Agerande}(q, w, n) = & \text{Om}(\text{Lika}(q, 0), \\ & \text{Om}(\text{Lika}(\text{Symbol}(w, n), 0), \\ & \dots) \end{aligned}$$

Sammanfattningsvis har vi följande primitivt rekursiva funktioner som tar hand om konfigurationernas omvandlingar:

$$\begin{aligned} \text{NästaPosition}(q, w, n) = & \text{Om}(\text{Lika}(\text{Agerande}(q, w, n), L), \\ & \text{Minska}(n), \\ & \text{Om}(\text{Lika}(\text{Agerande}(q, w, n), R), \\ & \text{Öka}(n), \\ & n)) \end{aligned}$$

$$\begin{aligned} \text{NästaTape}(q, w, n) = & \text{Om}(\text{Eller}(\text{Lika}(\text{Agerande}(q, w, n), L), \\ & \text{Lika}(\text{Agerande}(q, w, n), R)) \\ & w, \\ & \text{Skriv}(\text{Agerande}(q, w, n), w, n)) \end{aligned}$$

[†]. Förslagsvis kan L, R representeras med tal större än de tal som representerar Turingmaskinens tapesymboler.

$$\text{Skriv}(a, w, n) = \text{Add}(\text{Sub}(w, \text{Mult}(\text{Symbol}(w, n), \text{Pot}(k, n - 1))), \\ \text{Mult}(a, \text{Pot}(k, \text{Minska}(n))))$$

$$\text{Symbol}(w, n) = \text{Sub}(\text{Kvot}(w, \text{Pot}(k, \text{Minska}(n))), \\ \text{Mult}(\text{Kvot}(w, \text{Pot}(k, n)), k))$$

■ Den slutgiltiga funktionen

Nu återstår bara att konstatera att konfigurationernas omvandlingar avbryts om Turingmaskinens nuvarande tillstånd är stopptillstånd. Därför beskrivs Turingmaskinens arbete ytterst av

$$\text{Turing}(q, w, n) = \text{Om}(\text{Lika}(q, \text{halt}), \\ w, \\ \text{Turing}(\text{NästaTillstånd}(q, w, n), \\ \text{NästaTape}(q, w, n), \\ \text{NästaPosition}(q, w, n)))$$

som körs igång med $q = 0$ (starttillståndet) och $n = 1$ (läshuvudet längst till vänster på tapen).

Lägg märke till att samtliga funktioner utom *Turing*, den yttersta, är konstruerade inom klassen av primitivt rekursiva funktioner. Vad gäller funktionen *Turing*, så noterar Du säkert att den är konstruerad på ett sätt som i hög grad liknar framåtrekursion. En funktion med *åkta* framåtrekursion låter sig inte konstrueras lika naturligt (tycks det mig). Men låt oss göra ett försök ...

Paketera varje konfiguration i ett naturligt tal x :

$$x = 2^{\text{tillstånd}} \cdot 3^{\text{tape}} \cdot 5^{\text{position}}$$

Introducera sedan ett "*klockslagsargument*" y , dvs ett framåtrekurse-
rande argument med uppgift att hålla räkning på antalet konfigura-
tionsomvandlingar som Turingmaskinen gör innan den stannar.

Konstruera sedan en funktion $\text{Konfig}(x, y)$ med uppgift att retur-
nera det värde som x omvandlas till efter y omvandlingar.

$$\begin{cases} \text{Konfig}(x, 0) = x \\ \text{Konfig}(x, \text{Öka}(y)) = \text{NästaKonfig}(\text{Konfig}(x, y)) \end{cases}$$

Den här konstruktionen som ger

$$\text{Konfig}(x, 1) = \text{NästaKonfig}(\text{Konfig}(x, 0)) = \text{NästaKonfig}(x)$$

$$\begin{aligned}\text{Konfig}(x, 2) &= \text{NästaKonfig}(\text{Konfig}(x, 1)) \\ &= \text{NästaKonfig}(\text{NästaKonfig}(x))\end{aligned}$$

$$\begin{aligned}\text{Konfig}(x, 3) &= \text{NästaKonfig}(\text{Konfig}(x, 2)) \\ &= \text{NästaKonfig}(\text{NästaKonfig}(\text{NästaKonfig}(x)))\end{aligned}$$

använder sig av

$$\text{NästaKonfig}(x) = 2^{\text{nästaTillstånd}} \cdot 3^{\text{nästaTape}} \cdot 5^{\text{nästaPosition}}$$

vilken går att konstruera primitivt rekursivt (jfr utredningen på sid 168 angående *nästa tillstånd*, *nästa tape* och *nästa position*). Därmed är funktionen $\text{Konfig}(x, y)$ primitivt rekursiv.

Man kan också visa (gör det!) att *tillstånd*, *tape* och *position* efter y steg går att beräkna primitivt rekursivt utgående ifrån värdet på dåvarande konfigurationstal $\text{Konfig}(x, y)$.

$\text{Tillstånd}(\text{Konfig}(x, y))$ och $\text{Tape}(\text{Konfig}(x, y))$ som förekommer i $H(x, y)$ nedan, antas kunna beräkna värdena på *tillstånd* och *tape* efter y steg.

Turingmaskinens beräkningar startas nu med

$$\text{Turing}'(2^0 \cdot 3^{\text{tape}} \cdot 5^1)$$

där

$$\begin{aligned}\text{Turing}'(x) &= H(x, 0) \\ H(x, y) &= \text{Om}(\text{Lika}(\text{Tillstånd}(\text{Konfig}(x, y)), \text{halt}), \\ &\quad \text{Tape}(\text{Konfig}(x, y)), \\ &\quad H(x, \text{Öka}(y)))\end{aligned}$$

Den diskussion vi här har presenterat bevisar (om än något ofullständigt) satsen:

✧ SATS 5.1 (Kleene) Om F kan beräknas av en TM så kan F beräknas av en rekursiv funktion.

5.6 För varje rekursiv funktion finns det en TM

Vi har just visat att de funktioner som Turingmaskiner kan beräkna tillhör klassen av rekursiva funktioner. Omvänt gäller att varje rekursiv funktion kan beräknas av en TM. Detta bevisas av nedanstående fyra punkter som Du uppmanas att ge Dig i kast med.

1. Basfunktionen Öka kan beräknas av en TM.
2. Om F och G_1, \dots, G_m kan beräknas av TM:er, så gäller detsamma om sammansättningen $F(G_1(\bar{x}), \dots, G_m(\bar{x}))$.

3. Om B och G kan beräknas av TM:er, så gäller detsamma för F given av den *primitivt rekursiva* mallen:

$$\begin{cases} F(0, \bar{y}) = B(\bar{y}) \\ F(\bar{O}ka(x), \bar{y}) = G(x, \bar{y}, F(x, \bar{y})) \end{cases}$$

4. Om P och G kan beräknas av TM:er, så gäller detsamma för F given av den *framåtrekursiva* mallen:

$$\begin{aligned} F(\bar{x}) &= H(\bar{x}, 0) \\ H(\bar{x}, y) &= Om(P(\bar{x}, y), G(\bar{x}, y), H(\bar{x}, \bar{O}ka(y))) \end{aligned}$$

5.7 Övningar

- 5.1 Konstruera F inom klassen av primitivt rekursiva funktioner.

x	0	1	2	3	4	5	6	7	8	9	...
$F(x)$	1	0	3	2	5	4	7	6	9	8	...

- 5.2 Ge med hjälp av den primitivt rekursiva mallen exempel på en funktion som beräknar $1^y + 2^y + 3^y + \dots + x^y$.

- 5.3 Visa hur följande "avrundningsfunktion" kan beräknas inom klassen av primitivt rekursiva funktioner.

x	0	1	...	5	6	...	9	10	...	15	...
$Avrunda(x)$	0	0	...	5	5	...	5	10	...	15	...

- 5.4 Visa att om F är en periodisk funktion från \mathbb{N} till \mathbb{N} (För något p är $F(x) = F(x + p)$ för alla x) så kan F beräknas inom klassen av primitivt rekursiva funktioner.

- 5.5 Visa hur följande funktioner kan beräknas både inom klassen av primitivt rekursiva funktioner och med hjälp av framåtrekursion.

a)	x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
	$TvåLog(x)$	0	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3	4	...
b)	x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
	$KvadratRot(x)$	0	1	1	1	2	2	2	2	2	3	3	3	3	3	3	3	4	...

- c) $TriangelTal?(x)$ (Se fotnoten på sid 150.)

d) $TvåPotens?(x)$ (Skall returnera 1 om $x = 2^y$ för något $y \in \mathbb{N}$, och 0 annars.)

e) $KvadratTal?(x)$ (Skall returnera 1 om $x = y^2$ för något $y \in \mathbb{N}$.)

5.6 Konstruera en funktion som

a) inom klassen av primitivt rekursiva funktioner beräknar största gemensamma delare till två naturliga tal,

b) med hjälp av framåtrekursion beräknar minsta gemensamma multipel till två naturliga tal.

5.7 Låt $F(x) = x$:s längd (antalet siffror), där de naturliga talen x och $F(x)$ skrivs i 10-systemet. T ex är $F(9) = 1$, $F(5230) = 4$. Visa hur F kan beräknas

a) medelst framåtrekursion,

b) inom klassen av primitivt rekursiva funktioner.

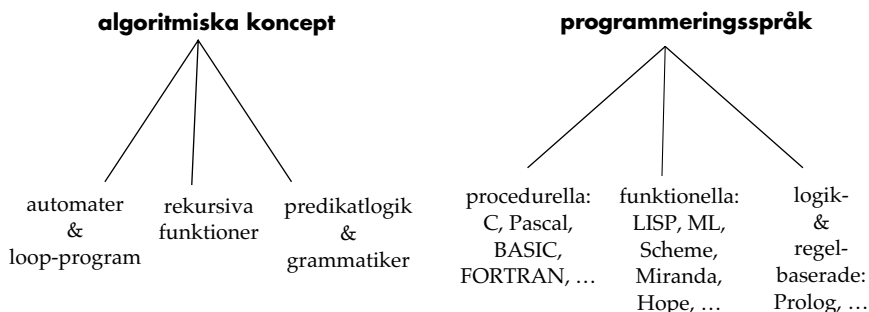
5.8 Konstruera ett *rekursivt predikat* som härmar en DFA genom att returnera 1 för de strängar som DFA:n accepterar och 0 för varje annan sträng.

LEDNING: Givetvis måste man här på lämpligt sätt representera en DFA:s inputsträngar med naturliga tal. Hur detta kan göras och hur det rekursiva predikatet sedan kan konstrueras kan Du säkert få inspiration till genom att läsa om hur en rekursiv funktion kan härma en TM i avsnittet 5.5

Loop-program

Primitiva loop-program.....	174
För varje primitivt rekursiv funktion finns det ett primitivt loop-program.....	175
För varje primitivt loop-program finns det en primitivt rekursiv funktion.....	176
Program med sålänge-loopar	178

Gängse programmeringsspråk brukar klassificeras som *procedu-
rella, funktionella* eller *logik- & regelbaserade*. Denna indelning är
inget annat än en sortering av programmeringsspråken efter vilka al-
goritmiska koncept som ligger till grund för språken ifråga.



Program skrivna i procedurella språk har måhända inte så tydliga spår av automater i själva *uttryckssättet* (när man skriver ett procedurellt program, så uttrycker man sig inte i termer av tillståndsovergångar). Men sådana programs sätt att lagra och manipulera data kan beskrivas just i termer av tillstånd och tillståndsovergångar.

Om ett visst procedurellt programs variabler är x_1, x_2, \dots, x_n , så utgörs programmets *tillstånd* av de värden som tupeln (x_1, x_2, \dots, x_n) har i olika skeden av programkörningen.

Procedurella program å ena sidan och kompetenta automater å den andra (just nu avses PDA:er och TM:er) har också det gemensamt att båda typerna bygger sin beräkningsförmåga på möjligheten att göra "minnesanteckningar". Dock finns det en skillnad i sättet att göra minnesanteckningar. Medan ett procedurellt programspråk använder variabler såsom lätt åtkomliga "behållare" för minnesanteckningarna tvingas en automat, för att komma åt en cell i "tapeminnet", gå igenom alla celler mellan läshuvudet och den aktuella "minnescellen".

Vi ska nu presentera ett algoritmiskt koncept – *loop-program* – som på ett mer “verklighetstroget” sätt modellerar de procedurrella språkens minnesanteckningar. Inom konceptet ifråga finns det en underklass av *primitiva loop-program* – precis som det inom konceptet *rekursiva funktioner* finns en underklass av *primitivt rekursiva funktioner*. I själva verket har dessa två underklasser samma algoritmiska uttryckskraft.

6.1 Primitiva loop-program

Procedurrella språk har en stomme av instruktioner där följande tre komponenter ingår. (x betecknar en *variabel* med värden i \mathbb{N} .)

<i>instruktioner</i>	<i>innebörd</i>
1. $x \leftarrow 0$	“Nollställ x ”
2. $x++$	“Öka x en enhet”
3. $x \text{ ggr}\{\dots\}$	“Slingans block $\{\dots\}$ – som får innehålla godtyckliga sekvenser av 1, 2, 3 men inget annat – skall <i>repeteras</i> x gånger. [†] Vidare får x <i>inte</i> ändras inuti slingans block.”

†. Kanske känner Du igen en sådan här loop som en *for-loop* eller *ovillkorlig loop*.

DEFINITION Ett primitivt loop-program är en ändlig sekvens av instruktioner av typ 1, 2, 3.

Vi skall visa att *primitiva loop-program* har exakt samma uttryckskraft som *primitivt rekursiva funktioner*. Men först ett par exempel.

EXEMPEL 6.1 Följande primitiva loop-program kommer att visa sig vara användbart. Programmet, som betecknas $y \leftarrow x$, tilldelar y det värde som x har. Annorlunda uttryckt, programmet *kopierar* x :s värde och lägger kopian i variabeln y .

```

y ← x =
      y ← 0
      x ggr{y++}

```

EXEMPEL 6.2 Detta primitiva loop-program *minskar* en variabls värde med *en* enhet, dock inte längre ned än till 0. Dvs $0--$ är 0.

```

x-- =
      z ← x
      y ← 0
      z ggr{x ← y
            y++}

```

■ 6.2 För varje primitivt rekursiv funktion finns det ett primitivt loop-program

Vi visar här att varje primitivt rekursiv funktion kan beräknas av ett primitivt loop-program. Men först några rader om *hur* ett loop-program vanligtvis används vid funktionsberäkning.

Vissa loop-program, som t ex $x++$, använder *samma* variabel (eller variabler) för såväl input som output.

Andra, som t ex $y \leftarrow x$, har *skilda* variabler för input och output.

I syfte att göra läsaren fullständigt klar över vad som är input och output i ett loop-program, tillämpas notationen

$$P \ x \ i \ y$$

för det fall att programmet P har x som inputvariabel och y som outputvariabel, men notationen

$$P \ x$$

om P har x både som inputvariabel som outputvariabel.

EXEMPEL 6.3 I denna anda passar

Öka x , Minska x , Nollställ x , Kopiera x i y

som namn på loop-programmen $x++$, $x--$, $x \leftarrow 0$, $y \leftarrow x$

Inför beviset av satsen nedan, påminner jag om att en funktion F ligger i klassen av *primitivt rekursiva funktioner* om F kan tillverkas utgående från basfunktionen *Öka* genom att *operera noll* eller *flera gånger* med sammansättning och/eller primitiv rekursion.

✧ SATS 6.1 *Varje primitivt rekursiv funktion kan beräknas med ett primitivt loop-program*

BEVIS: Vi bevisar satsen med hjälp av induktion över antalet operationer med vilka en primitivt rekursiv funktion är byggd.

Basfall: *Basfunktionen Öka* kan beräknas med ett primitivt loop-program:

<i>primitivt rekursiv funktion</i>	<i>primitivt loop-program</i>
$\text{Öka}(x) = x + 1$	$\text{Öka } x$

Induktionssteg: En primitivt rekursiv funktion skild från basfunktionen är av typ:

$$F(x) = H_1(H_2(x)) \quad \text{eller} \quad \begin{cases} F(0) = b \\ F(\text{Öka}(x)) = G(x, F(x)) \end{cases} \quad (23)$$

En primitivt rekursiv funktion byggd med *en* eller *flera* operationer.

(För F med flera argument gäller motsvarande formler.)

Om F i (23) är byggd med totalt n stycken operationer utgående från basfunktionen, så är H_1, H_2, G byggda med färre än n operationer,

och kan antas vara beräkningsbara med primitiva loop-program (*induktionsantagande*):

$$P_{H_1} \ x \ i \ z, \ P_{H_2} \ x \ i \ y, \ \text{och} \ P_G \ x, y \ i \ u \quad (24)$$

Med hjälp av programmen i (24) kan då de två funktionerna F i (23) beräknas av i tur och ordning nedanstående två primitiva loop-program vilket bevisar satsen.

<i>primitivt rekursiva funktioner</i>	<i>primitiva loop-program</i>
$F(x) = H_1(H_2(x))$	$P_F \ x \ i \ z =$ $P_{H_2} \ x \ i \ y$ $P_{H_1} \ y \ i \ z$
$\begin{cases} F(0) = b \\ F(\text{Öka}(x)) = G(x, F(x)) \end{cases}$	$P_F \ x \ i \ z =$ $x' \leftarrow 0$ $z \leftarrow b$ $x \ \text{ggr} \{ z' \leftarrow z$ $P_G \ x', z' \ i \ z$ $x' ++ \}$

✎ **TEST 6.1** Konstruera ett primitivt loop-program

Om x så y annars z i r
 som beräknar funktionen $Om(x, y, z)$.

■ 6.3 För varje primitivt loop-program finns det en primitivt rekursiv funktion

Nu visar vi att det som primitiva loop-programs kan göra, det kan också primitivt rekursiva funktioner göra.

Låt oss se ett loop-programs samtliga variabler *både* som input- och outputvariabler, på så sätt att variablernas värden vid start utgör input, och så att variablernas värden när programmet har stannat utgör output. Därvid tar vi hänsyn till ett loop-programs alla eventuella sidoeffekter som ju i någon mening också är beräkningsresultat (eftertraktade eller inte). Med detta synsätt menar vi t ex att ett primitivt loop-program som har *två* variabler x, y beräknar *två* funktioner, säg $F_x(x, y)$ och $F_y(x, y)$, där F_x :s och F_y :s output utgörs av de värden som variabeln x respektive variabeln y har när loop-programmet har stannat.

T ex beräknar programmet *Minska x* (se EXEMPEL 6.2 på sidan 174) *tre* funktioner. En returnerar det minskade x -värdet, och de två andra returnerar y :s och z :s värden efter att programmet har stannat.

<i>primitivt loop-program</i>	<i>primitivt rekursiva funktioner</i>
	$F_x(x, y, z) = x - 1$
Minska x	$F_y(x, y, z) = x$
	$F_z(x, y, z) = x$

✧ SATS 6.2 *Varje primitivt loop-program kan beräknas med en primitivt rekursiv funktion*

BEVIS: Precis som i beviset av föregående sats gör vi ett induktionsbevis – nu med induktion över antalet instruktioner i ett loop-program.

Basfall: Det som de tre enklaste[†] primitiva loop-programmen

$$x \leftarrow 0, \quad x++, \quad x \text{ ggr} \{ \}$$

uträttar kan beräknas med följande primitivt rekursiva funktioner

$$\text{Noll}(x) = 0, \quad \text{Öka}(x) = x + 1, \quad \text{Identitet}(x) = x$$

Induktionssteg: Ett godtyckligt primitivt loop-program P byggt med $n \geq 2$ stycken instruktioner är endera en sekvens av två loop-program vart och ett byggt med färre än n instruktioner, eller en repetitionslinga vars repetitionsblock likaså har färre än n instruktioner:

Ett primitivt loop-program P byggt med $n \geq 2$ instruktioner måste ha någon av dessa två former.

$$\begin{array}{ll}
 P = & P = \\
 \begin{array}{c} P_1 \\ \text{var och en med färre} \\ \text{än } n \text{ instruktioner} \end{array} & \begin{array}{c} y \text{ ggr} \{ P' \} \\ \text{en instruktion} \end{array} \quad \begin{array}{c} \text{färre än } n \text{ instruktioner} \end{array}
 \end{array} \quad (25)$$

Vi antar (*induktionsantagande*) att satsen stämmer för loop-program byggda med färre än n instruktioner. Då kan P_1 , P_2 och P' i (25) beräknas med primitivt rekursiva funktioner. Och då kan det vänstra P -programmet i (25) beräknas inom klassen av primitivt rekursiva funktioner med hjälp av *sammansättning*.

Det högra P -programmet i (25) använder dels variabeln y som inte förändras alls under programmets gång (se kommentaren vid 3. på sidan 174), dels eventuella ytterligare variabler x_1, \dots, x_m som P' i slingans block nyttjar. Enligt induktionsantagandet finns det primitivt rekursiva funktioner, säg G_1, \dots, G_m , som kan beräkna de förändringar av P' 's variabler som *en* körning med P' orsakar. Den to-

[†]. Loop-program med *en* enda instruktion. Lägg förresten märke till att det tredje programmet har en tom slinga.

tala förändringen som variablerna x_i genomgår under slingans y genomlöpningar beräknas därmed av nedanstående funktioner F_i som för enkelhetens skull är presenterade för det fall att P' innehåller blott *två* variabler:

$$F_1(x_1, x_2, 0) = x_1$$

$$F_1(x_1, x_2, \text{Öka}(y)) = G_1(F_1(x_1, x_2, y), F_2(x_1, x_2, y))$$

$$F_2(x_1, x_2, 0) = x_2$$

$$F_2(x_1, x_2, \text{Öka}(y)) = G_2(F_1(x_1, x_2, y), F_2(x_1, x_2, y))$$

Som synes har F_i endast *en* rekursionsvariabel y precis som det ska vara vid primitiv rekursion, och y rekurerar *ett steg bakåt* precis som det ska vara vid primitiv rekursion.

Visserligen är funktionerna F_i *sammanflätade* på ett sätt som det *inte* ska vara vid primitiv rekursion, men som tur är kan sådana *flätor* göras om till oflätade primitivt rekursiva funktioner. Se sid 157.

Sammantaget visar detta att P kan beräknas inom klassen av primitivt rekursiva funktioner. \square

F_1 beräknas
med hjälp av
 F_2 som
beräknas med
hjälp av F_1 .

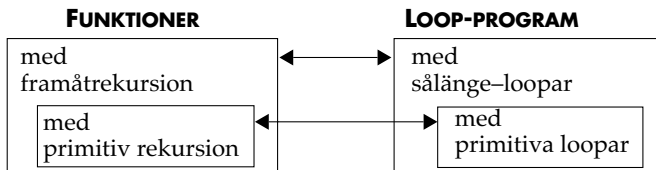
6.4 Program med sålänge-loopar

Det som begränsar *primitiva* loop-programs beräkningar till *primitivt* rekursiva beräkningar är det faktum att man i ett primitivt loop-program måste specificera antalet repetitioner på förhand. Genom att låta repetitionsslingornas egna beräkningar avgöra hur många gånger de skall repeteras, kan man visa att beräkningskraften utvidgas till vad framåtrekurerande beräkningar klarar av, dvs till (allmänt) rekursiva beräkningar. Nedanstående tre instruktioner bildar en sådan utvidgning.

1. $x \leftarrow 0$ "Nollställ x "
2. $x++$ "Öka x en enhet"
3. Sålänge $x \neq 0 \{ \dots \}$ "Repetera det som står inuti slingan så länge $x \neq 0$ "

Sammanfattningsvis råder således en perfekt korrespondens mellan de två typerna av rekursiva funktioner å ena sidan och de två typerna av loop-program å andra sidan:

Två ekvivalenta algoritmiska koncept.



Stopp- problemet och oavgörbarhet

Stopp-problemet	180
Två typer av TM-igenkännbara språk	182
Ett exempel på ett icke-algoritmiskt språk	183
Rices sats	184
Konsekvenser av Rices sats	185
Problemet om den flitiga bävern	187
Övningar	188

Betrakta följande enkla program där x rekurserar enligt två alternativ: *Jämnt* x halveras, medan *udda* x tredubblas och ökas med en enhet.

$$\begin{cases} F(1) = 1 \\ F(x) = Om(Jämn(x), F(Kvot(x, 2)), F(Öka(Mult(3, x)))) \end{cases}$$

Som Du ser kommer programmet om det t ex startas med $x = 7$ att ge upphov till följande beräkning:

$$\begin{aligned} F(7) &= F(22) = F(11) = F(34) = F(17) = F(52) = F(26) = F(13) = F(40) \\ &= F(20) = F(10) = F(5) = F(16) = F(8) = F(4) = F(2) = F(1) = 1 \end{aligned}$$

Trots programmet enkla form är det *ingen som vet* (ännu) huruvida det stannar för varje naturligt tal x , eller om det finns något x som orsakar oändlig rekursion. Retfullt, eller hur! Även om problemet kanske är fullständigt ointressant för övrigt, så illustrerar det en intressant punkt – att även enkla program kan vara svåra att förstå.

Exemplet ovan handlar uppenbarligen om ett öppet matematiskt problem inom området talteori. Men att problemet är öppet idag behöver förstås inte innebära att det är öppet i morgon. Det kan ju hända att någon (Du?) reder ut problemet. Problemet kan också förbli olöst. T ex av det skälet att de som ger sig i kast med problemet inte är tillräckligt motiverade eller "klipska". Men t ex också av det "enkla" skälet att problemet inte går att reda ut. Under sommaren 1930 bevisade nämligen Kurt Gödel (24 år gammal) att matematik har en så stark uttryckskraft att den möjliggör problemformuleringar som är alltför invecklade för att kunna redas ut – med matematik. Lagg märke till att innebörden i Gödels s.k. ofullständighetssats[†] inte är att det *kanske* går att formulera sådana problem, utan att det faktiskt går. Gödels upptäckt förändrade i ett slag synen på vetenskapen matematik. Från att ha uppfattats som en "vattentät" vetenskap vars tillkortakommanden enbart antogs hänföra sig till utövarnas bristande för-

måga – inte till verktygen – förvandlades matematik helt plötsligt till en vetenskap behäftad med fundamentala logiska "brister".

Men vänta nu ... eftersom problem rörande program "bara" utgör en del av den matematiska problemsfären, kanske problem om program skulle kunna gå att reda ut inom matematiken.

Förresten ..., problem som handlar *om* program kan kanske redas ut *med* program.

7.1. Stopp-problemet

Med tanke på det inledande exemplet reser sig följande fråga som går under namnet *stopp-problemet*.

*Kan man med något program P avgöra
ifall igångkörda program stannar?* (1)

Jag vill understryka att vi här menar att programmet P skulle kunna undersöka vilken programkod som helst (även sin egen) och efter en ändlig tidsrymd leverera ett svar av typen *ja* eller *nej*.

Vi ska se att *stopp-problemet* (formulerat för $\text{program} = \text{Turingmaskiner}$) har ett negativt svar. Dvs det finns *inte* någon TM som för godtycklig TM M och godtycklig inputsträng w kan avgöra – genom att svara "ja" eller "nej" – om M skulle stanna eller ej vid start på w .

Stopp-problemets oavgörbarhet bevisades först av Alan Turing (1936).

✧ SATS 7.1 (*Stopp-problemets oavgörbarhet*)

*Ingen TM kan för en godtycklig TM avgöra om den
senare stannar för en godtycklig sträng.*

BEVIS: Antag att det finns en TM *SUPER* som för godtycklig TM M och godtycklig sträng w kan avgöra om M skulle stanna eller ej efter att ha körts igång på strängen w , dvs så att *SUPER* skulle stanna och lämna svaret "ja" eller "nej" på tapen efter att ha körts igång på tapen $\#M\#w\#$. Se FIGUR 7.1. M skall här uppfattas som en sträng (M :s kod) beskrivande Turingmaskinen M , en sträng över t ex det binära alfabetet. w antas likaså vara en sträng över samma alfabet.[‡]

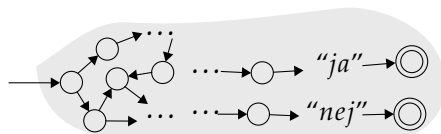
†. Gödels ofullständighetssats (sats sex av elva satser publicerade 1931 i Monatshefte für Mathematik und Physik 38, 173-198) kan i en fri översättning formuleras på följande sätt):

För varje tillräckligt stark matematisk teori så finns det ett påstående som kan formuleras i teorin, men varken bevisas eller motbevisas i den.

‡. Kom ihåg att den universella Turingmaskinen (se tidigare avsnitt) är en TM som precis likt *SUPER* matas med par (M, w) .

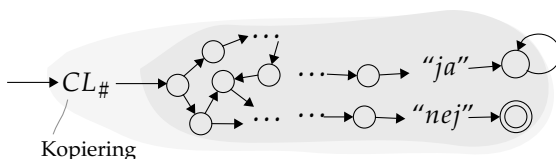
Givet *SUPER*, kan man bygga en ny TM *STUPER* som nedan.

Ändra lite
i *SUPER*
...



FIGUR 7.1
SUPER som
matas med
 $\#M\#w\#...$, avgör
om M skulle
stanna efter att ha
startats på w .

... så får
Du
STUPER



FIGUR 7.2
Detta är *STUPER*
som matas med
 $\#x\#...$ och som
är sammanfogad
av $CL\#$ och en
korrigerad
SUPER.

Vad gör
STUPER?

STUPER börjar med att kopiera sitt input. Dvs $\#input\#...$ omvandlas till $\#input\#input\#...$.

Sedan ställer *STUPER* tillbaka sitt läshuvud till blanktecknet så att tapekonfigurationen blir $\#input\#input\#...$, varefter *STUPER* kör igång en något modifierad *SUPER*. Modifieringen består i att där *SUPER* tidigare hade ett stopptillstånd och stannade efter "ja"-svar, har nu placerats en oändlig slinga. Det är allt.

Eftersom inget hindrar en TM från att ta sin egen kod som inputs-träng, (det värsta som kan inträffa är ju att körningen hänger sig), kan man fråga sig vad som skulle hända om *STUPER* startades på sin egen kod.

Svaret är följande.

- Om *STUPER* är en TM som stannar på sin egen kod så drivs den modifierade *SUPER* (vid körning på sin egen kod) via sitt "ja"-tillstånd in i den oändliga slingan. Se FIGUR 7.2. Och då stannar aldrig *STUPER*.

- Om *STUPER* inte stannar på sin egen kod så drivs den modifierade *SUPER* (vid körning på sin egen kod) via sitt "nej"-tillstånd till stopptillståndet. Där stannar *STUPER*.

Med andra ord,

*stannar STUPER, så stannar inte STUPER,
och stannar inte STUPER, så stannar STUPER.*

Detta är
orimligt.

Därmed måste vi förkasta antagandet som ledde oss fram hit – antagandet om *SUPER*:s existens. \square

⌘ Anm.7.1 Ur stopp-problemets oavgörbarhet – som bevisades av Alan Turing (1936) – följer (med det förbehållet att Turingmaskiner verkligen fångar det algoritmiska) att Hilberts *Entscheidungsproblem*

(se sidorna 3, 117) har ett negativt svar. Stopp-problemet är ju nämligen ett matematiskt problem som ingen TM-algoritm kan lösa.

■ Två typer av TM-igenkännbara språk

Fortsättningsvis säger vi att ett språk är TM-accepterbart (eller Turingaccepterbart) om det finns någon Turingmaskin som accepterar språket ifråga, och TM-avgörbart (eller Turingavgörbart) om det finns någon Turingmaskin som avgör språket. I kapitel 4 visade vi (på sid 129) hur en TM som accepterar språket $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ kunde korrigeras till en TM som avgör samma språk. Vi ska nu se att det finns Turingmaskiner som *inte* kan korrigeras på sådant sätt. Närmare bestämt att det finns (åtminstone) *ett* TM-accepterbart språk som inte är TM-avgörbart.

Notera först att varje par (M, w) – där M är en TM och w är en inputsträng till samma TM – kan kodas som *en* sträng över det binära alfabetet. (Se fotnot på sid 135.) En *mängd* av par (M, w) kan sålunda kodas som ett *språk* (över det binära alfabetet).

Låt $\mathcal{L}_{\text{stopp}}$ beteckna språket som på nämnda sätt kodar mängden av par (M, w) som uppfyller att M stannar för w . Något vårdslöst uttryckt är således $\mathcal{L}_{\text{stopp}} = \{(M, w) \mid M \text{ stannar för } w\}$.

Å ena sidan säger SATS 7.1 att $\mathcal{L}_{\text{stopp}}$ inte är TM-avgörbart. Å andra sidan är $\mathcal{L}_{\text{stopp}}$ TM-accepterbart. En välkänd TM – den universella – *accepterar* nämligen $\mathcal{L}_{\text{stopp}}$. Ty den universella Turingmaskinen tar par (M, w) som input och stannar om och endast om M stannar för w . Detta bevisar att de TM-accepter-

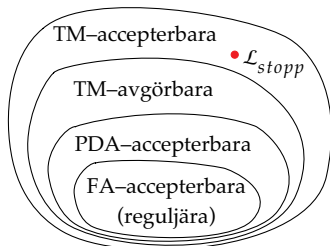
bara språken bildar en strikt större klass än de TM-avgörbara. Sammantaget har vi nu en hierarki av fyra olika *algoritmiska* språk.

Här följer en sats som ger en viktig karakteristik av de Turingavgörbara språken.

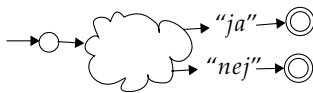
SATS 7.2 Ett språk L är Turingavgörbart
om och endast om både L och \bar{L} är Turingaccepterbara.

BEVIS: Antag först att L är Turingavgörbart. Dvs att det finns en TM M som stannar och svarar "ja" för varje sträng i L och "nej" för varje sträng i \bar{L} . I figuren nedanför visas hur M kan korrigeras dels till en TM som accepterar L dels till en TM som accepterar \bar{L} . Detta bevisar första halvan av satsen.

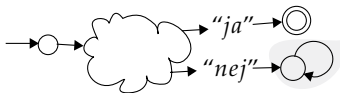
En hierarki av
algoritmiska
språk.



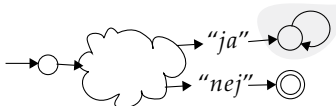
Om en TM av-
gör L , så kan
den korrigeras
så att man får ...



en TM
som accepterar
 L



och en
som accepterar
 \bar{L} .



Omvänt, antag att L är ett språk sådant att L och \bar{L} accepteras av varsin TM. Om Du hade två sådana TM:er till ditt förfogande, hur skulle Du med deras hjälp reda ut huruvida en viss sträng ligger i L eller ej? Säkert skulle Du köra igång de två TM:erna på strängen ifråga och sedan vänta tills en av dem stannar – för det vet Du ju att en av dem gör. (En av dem stannar och den andra stannar inte.)

Och när detta inträffar (att en av dem stannar), så behöver man bara se efter *vilken* av de två som stannade. Med ledning därav kan man avgöra om strängen ligger i L eller ej.

En TM som utför sådan provkörning kan alltså avgöra L . □

■ Ett exempel på ett icke-algoritmiskt språk

Med hjälp av ett enkelt s.k. *kardinalitetsresonemang* kan man visa att det måste finnas ickealgoritmiska språk, dvs språk som är så komplicerade att inga TM:er kan känna igen dem ens genom acceptans: Det finns överuppräknligt många språk (detta var en av våra första upptäckter, se sid 18), men "bara" uppräknligt många Turingmaskiner och därmed uppräknligt många Turingaccepterbara språk. Därför måste det finnas språk som inte är Turingaccepterbara. Vi kan till och med ge ett exempel:

EXEMPEL 7.1 Eftersom \mathcal{L}_{stopp} inte är Turingavgörbart (se sid 182), så följer av SATS 7.2 att \mathcal{L}_{stopp} eller dess komplementspråk inte är Turingaccepterbart. \mathcal{L}_{stopp} är Turingaccepterbart (se sid 182). Alltså är komplementspråket $\{(M, w) \mid M \text{ stannar inte för } w\}$ inte Turingaccepterbart.

⌘ Anm.7.2 Exemplet ovanför visar att komplementmängden till en mängd kan vara avsevärt mer komplicerad än mängden själv. Det tål att fundera över!

✂ **TEST 7.1** Sant eller falskt?

- a) "Om L är TM-avgörbart så är \bar{L} TM-avgörbart."
 b) "Om L är Turingaccepterbart är \bar{L} Turingaccepterbart."

7.2. Rices sats

En egenskap som varje Turing-accepterbart språk har eller som inget sådant språk har (Ge exempel!) kallar vi för en *trivial* egenskap. (Om Du har hittat några exempel, förstår Du varför vi kallar dessa egenskaper triviala.)

En *icketrivial* egenskap är – som Du kanske har gissat vid det här laget – en egenskap som äges av något Turing-accepterbart språk, men inte av alla sådana språk.

Satsen nedan brukar uttryckas som att ingen Turingmaskin kan avgöra en icke-trivial egenskap. Notera att vi i satsen har formaliserat begreppet "icketrivial egenskap" i mängd-termer. Syftet med detta är enbart att satsens formulering och dess bevis på så sätt skall få en strikt form.

✧ **SATS 7.3 (Rices sats)** Om Ω är en mängd av Turing-accepterbara språk som innehåller något men inte alla sådana språk, så kan ingen TM avgöra för ett godtyckligt Turing-accepterbart språk L om L tillhör Ω eller ej.

BEVIS: Att Ω innehåller något men inte alla Turing-accepterbara språk garanterar existensen av två Turing-accepterbara språk L_0 och L_1 :

$$L_0 \notin \Omega \quad (2)$$

$$L_1 \in \Omega \quad (3)$$

Vi kan anta att L_0 är det reguljära (och därmed Turing-accepterbara) språket \emptyset , dvs vi kan antaga att $\emptyset \notin \Omega$ ty om $\emptyset \in \Omega$ skulle vi kunna föra det fortsatta resonemanget med Ω och Ω 's komplement i ombyttan roller.

Vidare är $L_1 = L(M_1)$ för någon TM M_1 .

Härav,

$$\emptyset \notin \Omega \quad (4)$$

$$L(M_1) \in \Omega \quad (5)$$

För varje TM M och varje sträng w kan vi nu konstruera en TM $T_{M,w}$ som använder M_1 , M samt en w -skrivare som hjälpmaskiner:

Låt $T_{M,w}$ börja med att skriva w på tapen en bit till höger om sitt

input x . Därefter får $T_{M,w}$ starta M på w . För att hindra M från att skriva över x under sitt arbete kan man korrigera M till att shifta hela x till vänster varje gång som M "råkar kliva in på" x . T ex kan man inledningsvis låta $T_{M,w}$ sätta ut ett särskilt "skiljemärke" mellan x och w . Ett tecken som skall driva (den korrigerade) M i varje tillstånd till att utföra nämnda vänstershiftning. Om och när M stannar under sin körning på w , så ska $T_{M,w}$ starta M_1 på x , $T_{M,w}$:s egentliga input. Om M stannar för w , så kommer $T_{M,w}$ att acceptera exakt de strängar x som M_1 accepterar. Men om M aldrig stannar för w , så kommer ej heller $T_{M,w}$ att stanna oavsett input x .

Utan att vara alltför detaljerad kan vi beskriva $T_{M,w}$ som

$$T_{M,w} = R_{\#} \$ R \{ \text{Skriv } w \} M \{ \text{Sudda } M\text{:s output} \} L_{\$} \# L_{\#} M_1 \quad (6)$$

Av $T_{M,w}$:s konstruktion följer att

$$L(T_{M,w}) = L(M_1) \text{ om } M \text{ stannar för } w, L(T_{M,w}) = \emptyset \text{ annars.} \quad (7)$$

(7) tillsammans med (3) och (2) ger att

$$L(T_{M,w}) \in \Omega \text{ om } M \text{ stannar för } w, L(T_{M,w}) \notin \Omega \text{ annars.} \quad (8)$$

Dvs,

$$L(T_{M,w}) \in \Omega \text{ om och endast om } M \text{ stannar för } w. \quad (9)$$

Antag nu att det finns en TM som kan avgöra huruvida ett *godtyckligt* Turing-accepterbart språk tillhör Ω eller ej. Då kan denna TM avgöra (för varje $T_{M,w}$ som vi konstruerar) huruvida $L(T_{M,w}) \in \Omega$ eller ej. Och av (9) framgår att den i så fall skulle avgöra stopp-problemet, vilket är orimligt. Av motsägelsen följer att antagandet var falskt. Detta bevisar satsen. \square

■ Konsekvenser av Rices sats

Rices sats ger oss en hop av oavgörbara problem som rör vad Turingmaskiner accepterar, dvs vad Turingmaskiner stannar för.

T ex följer av Rices sats att ingen TM kan avgöra för godtycklig TM M om dess språk

- är reguljärt,
- är sammanhangsfritt,
- innehåller en viss sträng (t ex tomma strängen),
- innehåller en viss sträng och inget annat,
- är ändligt (innehåller enbart ändligt antal strängar),
- är oändligt,
- innehåller alla strängar som slutar på tre 1:or,

...

Vad gäller t ex den första punkten i denna lista, så följer oavgörbarheten av att $\Omega = \{L \mid L \text{ är reguljärt}\}$ innehåller enbart Turingaccepterbara språk men inte alla sådana språk.

⌘ Anm.7.3 Rices sats uttalar sig "bara" om egenskaper rörande Turingmaskiners *språk*. Ty medelst Ω i Rices sats kan man "bara" specificera just sådana egenskaper som Turingmaskiners språk kan ha.

Vad gäller *andra* egenskaper om Turingmaskiner, t ex sådana egenskaper som rör tillstånden, tillståndsövergångarna eller "det inre arbetet" (t ex huruvida ett visst tecken någonsin skrivs på tapen under en beräkning), så har man ingen hjälp av Rices sats.

EXEMPEL 7.2 Finns det någon TM T som för godtycklig TM M kan avgöra om M någonsin skriver något ickeblankt tecken efter att ha startats på blank tape i konfigurationen $\# \# \# \# \dots ?$

Detta problem rör inte de Turingaccepterbara språken (utan Turingmaskinernas "inre arbete"). Därför måste vi här föra ett resonemang *utan* hjälp av Rices sats.

Klart är att det finns en TM som för godtycklig TM M kan undersöka om M har någon tillståndsövergång under vilken M skriver något ickeblankt tecken. (Detta är ju bara en *ändlig* fallundersökning för varje TM M .) Men det givna problemet är inte säkert avklarat i och med detta. Om M *saknar* tillståndsövergång med ickeblankt skrivande, så är det visserligen helt avgjort att M aldrig kan skriva något ickeblankt tecken. Om å andra sidan *det finns* en sådan övergång i M 's repertoar, så är det inte säkert att M använder den. Och hur skall en TM T kunna reda ut huruvida M nyttjar den eller ej?

Få se ..., om man tänker sig in i T 's situation skulle man kanske kunna provköra M och därvid kolla om M nyttjar sådan övergång?

Men ..., skulle inte detta kunna bli ett evighetsarbete, på motsvarande sätt som det skulle kunna bli om man genom provkörning av M försökte upptäcka om M någonsin stannar eller ej ...

Nej, inte alls. Man behöver bara göra N inledande tillståndsövergångar där N är antalet tillstånd hos M . Sedan vet man. Ty *endera* hänger sig M under dessa N övergångar, eller så tvingas M *besöka* något tillstånd *två gånger*. I båda fallen gäller att om M inte har skrivit något ickeblankt tecken hittills, så kommer M aldrig att göra det. Om M har hängt sig så är detta helt uppenbart. Och om M har besökt ett tillstånd q två gånger utan att ha skrivit något ickeblankt tecken, så är tapen fortfarande blank när M för andra gången står i tillståndet q , varför M p.g.a. sin determinism måste agera på samma sätt som vid första besöket i q . Dvs M har kommit in i en loop under vilken M aldrig skriver något ickeblankt tecken.

Sammanfattningsvis: *det finns* en TM T som kan avgöra problemet genom att provköra M (Låt T härma M !) under N inledande tillståndsövergångar, där N är antalet tillstånd hos M .

■ Problemet om den flitiga bävern

Vid sidan av stopp-problemet finns det många andra mer eller mindre fantasifulla oavgörbarhetsproblem. Problemet om *den flitiga bävern* av Tibor Rado^{††} är ett sådant och handlar om hur *flitig* en Turingmaskin kan vara att skriva ickeblanka tecken på sin tape, dvs hur många sådana tecken den kan skriva innan den stannar.

För att renodla problemet, betraktar vi bara Turingmaskiner med *ett* ickeblankt tecken i sitt alfabet – säg tecknet “|”.

Och eftersom en Turingmaskin med flera tillstånd än en annan skulle ha en fördel, så låter vi bara Turingmaskiner med *lika antal tillstånd* “tävla” mot varandra. Slutligen låter vi alla Turingmaskiner starta på blank tape med läshuvudet längst till vänster. Givet dessa antaganden kan problemet om den flitiga bävern formuleras sålunda:

Vilket är det maximala antalet |:or som en Turingmaskin med n tillstånd, stopptillståndet oräknat, kan skriva på tapen innan den stannar?

⌘ Anm.7.4 Att det för varje $n \in \mathbb{N}$ finns ett maximum som ovan är uppenbart. Ty det finns ändligt många Turingmaskiner med n tillstånd. Och av ändligt många Turingmaskiner måste ju någon eller några vara flitigast.

Lägg märke till att det är helt trivialt att beräkna det största talet av ändligt många givna tal, men att problemet om den flitiga bävern är av en helt annan svårighetsgrad. De ändligt många Turingmaskinerna med n tillstånd som stannar (förr eller senare) efter att ha startats på blank tape är ju inte givna! Det är just häri som problemets svårighet ligger. För tillräckligt små värden på n är det emellertid inte svårt att finna en flitigaste bäver. Försök får Du se!

Problemet om den flitiga bävern är oavgörbart.

Man kan bevisa (vilket Rado gjorde) att *ingen* TM löser problemet om den flitiga bävern. Dvs det *finns inte* någon TM som kan beräkna funktionen $Max(n)$, det maximala antal |:or beskrivna i problemet om den flitiga bävern. Här nedan följer en skiss av beviset.

Antag att det finns en TM T som (för varje n) kan beräkna $Max(n)$, dvs som omvandlar $\# \mid^n \# \dots$ till $\# \mid^{Max(n)} \# \dots$.

1. Med T 's hjälp kan man då bygga en TM som beräknar $Max(2n)$. (Visa hur!)
2. $Max(n)$ är en (strikt) växande funktion av n . (Visa detta.)
Härav följer att $Max(n) < Max(2n)$.

††. T. Rado, “On Non-Computable Functions.” *Bell System Tech. J.* **41**, 3 (1962).

3. Om en funktion $F(n)$ kan beräknas av en TM med k_0 stycken tillstånd, så finns det ett tal k_1 så att

$$F(n) \leq \text{Max}(n + k_0 + k_1) \text{ för varje } n.$$

Dvs då finns det en med $n + k_0 + k_1$ stycken tillstånd utrustad TM som lägger ut åtminstone $F(n)$ stycken 1:or på tapen efter att ha startats på tom tape. (Visa hur en sådan TM kan konstrueras med hjälp av en TM som beräknar F .)

4. Härav (sätt $F(n) = \text{Max}(2n)$) följer att

$$\text{Max}(2n) \leq \text{Max}(n + k_0 + k_1) \text{ för varje } n.$$

Och av egenskapen om växande följer att för varje n

$$2n \leq n + k_0 + k_1$$

$$\text{Dvs } n \leq k_0 + k_1$$

Men eftersom k_0 och k_1 är fixa tal är detta orimligt. Antagandet om att det finns en TM som kan beräkna $\text{Max}(n)$ måste därmed förkastas. \square

7.3. Övningar

7.1 Diskutera för vart och ett av följande problem om det finns någon TM som kan avgöra problemet ifråga.

- Givet en godtycklig TM M , accepterar M någon sträng innehållande tecknet 1?
- Givet en godtycklig DFA M , accepterar M någon sträng innehållande tecknet 1?
- Givet en godtycklig TM M , accepterar M någon sträng med exakt tre tecken?
- Givet en godtycklig TM M , accepterar M någon sträng med flera tecken än antalet tillstånd i M ?
- Givet en godtycklig TM M med minst ett tillstånd q utöver stopptillstånd och givet en godtycklig sträng w , kommer M någonsin att uppsöka tillståndet q efter start på w ?
- Givet en godtycklig DFA M med minst ett ickeaccepterande tillstånd q och givet en godtycklig sträng w , kommer M någonsin att uppsöka tillståndet q efter start på w ?
- Givet en godtycklig TM M , skriver M någonsin tecknet 1 efter start på blank tape?
- Givet en godtycklig TM M och en godtycklig sträng w , besöker M – under körning på w – några andra rutor på tapen än de som w ligger på (vid start)?
- Givet en godtycklig TM M , har M flera tillstånd än tio?
- Givet en godtycklig TM M och en godtycklig sträng w , besöker M flera tillstånd än tio under körning på w ?

7.2 Finns det någon TM som kan avgöra för en godtycklig

- a) DFA ifall den accepterar enbart ändligt många strängar?
- b) PDA ifall den accepterar enbart ändligt många strängar?

7.3 Betrakta följande problem där M_1 är en godtycklig DFA och q_1 ett godtyckligt tillstånd i M_1 . Din uppgift är att reda ut huruvida problemen ifråga är avgörbara.

- a) Om M_2 är en godtycklig DFA, finns det någon sträng $w_2 \in L(M_2)$ som driver M_1 från q_1 till acceptans?
- b) Om M_2 är en godtycklig TM, finns det någon sträng $w_2 \in L(M_2)$ som driver M_1 från q_1 till acceptans?

7.4 Presentera de flitigaste bävrarna i de tre "tävlingsklasserna"
 $n = 1$, $n = 2$, $n = 3$.

7.5 Betrakta följande variant på "flitiga bävern"-problemet, där startvillkoret ser annorlunda ut än i "originalproblemet".

För varje $n \in \mathbb{N}$, låt $Max(n)$ vara det maximala antalet $|$:or som en TM med n tillstånd, stopptillståndet oräknat, kan ha på sin tape när den stannar givet att den startar med n $|$:or på tapen:

$$\# \mid^n \# \dots$$

Visa att ingen TM kan beräkna funktionen $Max(n)$.

Logik

Satslogik	191
Satslogikens semantik	192
Semantisk analys med sanningstabell	193
Några nya konnektiv skapade med hjälp av gamla	193
Tolkning, validitet och satisfierbarhet	195
Ekvivalens	195
Normalformerna	198
Validitetsproblemet och resolution	199
Två topdown-algoritmer	200
Första ordningens predikatlogik	205
Syntax	206
Predikatlogikens semantik	207
Allkvantorn	209
Några ekvivalensregler	210
Validitet och satisfierbarhet i predikatlogik	211
Validitetsproblemet och resolution i predikatlogik	212
Hornklausuler och Hornformler	215
Rekursiva funktioner och predikatlogik	216
Till varje rekursiv funktion hör en Hornformel	217
Övningar	219

V arför uppfattar Du följande resonemang som korrekt, trots att Du inte förstår vad det handlar om?

*Om jag jigolerar Dig, och Du är tydd för alla som jigolerar Dig,
så är Du tydd för mig.*

Svaret på den frågan är att resonemangets *form* har en sådan kraft att resonemangets innehåll blir oväsentligt. Resonemanget är korrekt oavsett vad det handlar om!

Även nästa uttalande – vars form sammanfaller med formen hos det första – uppfattar Du som korrekt (eller hur!) trots att dess slutsats och ena villkor (att *alla primtal är udda*) är falska.

*Om 2 är primtal, och alla primtal är udda,
så är 2 ett udda tal.*

Innan vi ger oss i kast med en matematisk modell inom vilken vi kan studera resonemang (utsagor) av de slag som vi nyss har sett exempel på, skall vi behandla en enklare modell, en som enbart fångar resonemangens "ytstruktur".

Ingen kan klandra sättet på vilket slutsatsen formas, fastän invändningar kan resas mot sanningshalten i slutsatsen och mot ett av villkoren.

8.1 Satslogik

■ **Symboler och syntax** "Ytstrukturen" i utsagor som t ex

Kalle spelar inte fiol. (1)

Om Kalle spelar han fiol så får han applåder. (2)

skall vi beskriva symboliskt med strängarna

$\neg p$ (i)

$p \rightarrow q$ (ii)

(i), (ii) använder två typer av symboler:

- atomer p, q för att representera de enklaste utsagorna
- konnektiv[†] \neg, \rightarrow med vars hjälp man kan bygga strängar som representerar mer komplexa utsagor.

Strängar som i ändligt många steg är byggda från atomer (som t ex (i) och (ii) eller t ex $(p \rightarrow (q \rightarrow \neg p))$), benämner vi *satslogiska strängar* eller *formler*, eller helt enkelt *satser*. (Du kan använda vilket som helst av dessa namn.) Här är en formell definition:

DEFINITION

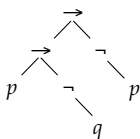
- 1 Atomer p, q, r, \dots är de enklaste satslogiska formerna.
- 2 Om u och v är satslogiska formler, så är $\neg u$ och $(u \rightarrow v)$ (sammansatta) satslogiska formler.

EXEMPEL 8.1 $(p \rightarrow \neg p), (p \rightarrow (p \rightarrow \neg p))$ är satslogiska formler, men *inte* $(p \rightarrow \neg), (p \rightarrow p \rightarrow \neg p)$.

EXEMPEL 8.2 Några enkla utsagor samt formler som härmar (representerar) dem.

utsaga	formel
Om Kalle spelar fiol så får han applåder	$p \rightarrow q$
Om Kalle spelar fiol så får han inte applåder	$p \rightarrow \neg q$
Om Kalle inte får applåder så spelar han inte fiol	$\neg q \rightarrow \neg p$
(Om Kalle inte får applåder om han spelar fiol) så spelar han inte fiol	$(p \rightarrow \neg q) \rightarrow \neg p$

†. connect = klistra



De satslogiska formlerna som binära träd En *satslogisk* formel har en *hierarkisk* struktur som bestäms av dess parenteser. Ett vanligt sätt att presentera den hierarkiska strukturen är att använda binära träd med formelns atomer i löven och konnektiven i de övriga noderna. Därvid behöver man förstås inte rita ut formelns parenteser, eftersom trädet tillhandahåller just den hierarkiska struktur som parenteserna annars behövs för.

■ Satslogikens semantik

Betrakta åter de sammansatta utsagorna (1), (2) ovanför.

Sanningshalten i dessa utsagor beror av sanningshalten i de enklare utsagorna som (1) och (2) är byggda av.

- Utsagan (1): "Inte spelar Kalle fiol" är sann om det inte är sant att "Kalle spelar fiol", och vice versa.
- Utsagan (2): "Om Kalle spelar fiol så får han applåder" skall uppfattas som att Kalles fiolspelande är ett tillräckligt villkor för (publikens) applåderande. Därför skall utsagan ifråga betraktas som falsk ifall Kalle fastän han spelar fiol inte får applåder, dvs då det är sant att "Kalle spelar fiol" men falskt att "Kalle får applåder". I ingen annan situation skall vi betrakta utsagan som falsk. T ex vore det fel att uppfatta den som falsk då Kalle får applåder fastän han inte spelar fiol, dvs då det är falskt att "Kalle spelar fiol" men sant att "Kalle får applåder". Ty utsagan säger inte att Kalles fiolspelande är ett nödvändigt villkor för applåderande – bara ett tillräckligt.

De s.k. sanningsvärdena (1=sant, 0=falsk) för formlerna $\neg u$ och $u \rightarrow v$ definieras med hjälp av sanningsvärdena för u, v på följande sätt.

DEFINITION

sanningstabeller

$$\neg u \text{ är } \begin{cases} \text{falsk} & \text{om } u \text{ är sann} \\ \text{sann} & \text{annars} \end{cases}$$

u	$\neg u$
0	1
1	0

$$u \rightarrow v \text{ är } \begin{cases} \text{falsk} & \text{om } u \text{ är sann och } v \text{ är falsk} \\ \text{sann} & \text{annars} \end{cases}$$

u	v	$u \rightarrow v$
0	0	1
0	1	1
1	1	1
1	0	0

†. Om semantik, se sid 10.

Eftersom *varje* sammansatt formel konstrueras med hjälp av konnektiven *negation* och *implikation* blir semantiken för *varje* sammansatt formel bestämd av ovanstående definition. Hur detta kan gå till i praktiken visas i nästa avsnitt.

■ Semantisk analys med sanningstabell

Beräkningen av en *sammansatt* formels sanningsvärden presenteras ofta i en s.k. sanningstabell.

EXEMPEL 8.3 Semantiken för $((p \rightarrow \neg q) \rightarrow p)$ beräknas steg för steg i en ordning som följer formelns hierarkiska struktur:

Notera att vi placerar sanningsvärdet för en sammansatt formel under det konnektiv som ligger *ytterst* i formelns hierarki (dvs i formelträdet rot).

	$((p \rightarrow \neg q) \rightarrow p)$					
	1			1		1
	1			0		1
	0			1		0
	0			0		0
steg 1	1		0	1		1
	1		1	0		1
	0		0	1		0
	0		1	0		0
steg 2	1	0	0	1		1
	1	1	1	0		1
	0	1	0	1		0
	0	1	1	0		0
steg 3	1	0	0	1	1	1
	1	1	1	0	1	1
	0	1	0	1	0	0
	0	1	1	0	0	0

■ Några nya konnektiv skapade med hjälp av gamla

De sammansatta formlerna $(\neg u \rightarrow v)$ och $\neg(u \rightarrow \neg v)$ beskriver att någon av u, v är sann respektive att båda är sanna (se TEST 8.1). Detta faktum kvalificerar formlerna ifråga att definiera nya konnektiv "*eller*" respektive "*och*". Även andra sammansatta formler har visat sig vara lämpliga som definitioner av nya konnektiv. Se tabellen nedanför. Observera att dessa nya konnektiv icke tillför satslogikens större

uttrycks kraft, blott ökar dess läsbarhet.

<i>konnektiv</i>	<i>benämning</i>	<i>skrivs</i>	<i>uttalas</i>	<i>definieras av</i>
\vee	disjunktion	$(p \vee q)$	"p eller q"	$(\neg p \rightarrow q)$
NOR	negerad "eller"	$NOR(p, q)$	"varken p eller q"	$\neg(p \vee q)$
\wedge	konjunktion	$(p \wedge q)$	"p och q"	$\neg(p \rightarrow \neg q)$
NAND	negerad "och"	$NAND(p, q)$	"inte både p och q"	$\neg(p \wedge q)$
\leftarrow	omvänd implikation	$(p \leftarrow q)$	"p om q"	$(q \rightarrow p)$
\leftrightarrow	ekvivalens	$(p \leftrightarrow q)$	"p omm q"	$(p \rightarrow q) \wedge (p \leftarrow q)$
XOR	exklusiv disjunktion	$XOR(p, q)$	"p eller q men inte båda"	$(p \vee q) \wedge \neg(p \wedge q)$

■ Sexton olika binära satslogiska formler

En satslogisk formel med två argument – en *binär* formel – har fyra inputkombinationer (0, 0), (0, 1), (1, 1), (1, 0), som var och en skall tillordnas ett av två möjliga sanningsvärden. Härav följer att det finns en potential av $2^4 = 16$ olika binära satslogiska formler (semantiskt sett), varken fler eller färre.

p	q	$\neg p$	$\neg q$	$p \rightarrow q$	$p \leftarrow q$	$p \vee q$	$p \wedge q$	$p \leftrightarrow q$	$(p \wedge \neg p)$	$(p \vee \neg p)$	$XOR(p, q)$	$NAND(p, q)$	$NOR(p, q)$	$\neg(p \leftarrow q)$	$\neg(p \rightarrow q)$
0	0	1	1	1	1	0	0	1	0	1	0	1	1	0	0
0	1	1	0	1	0	1	0	0	0	1	1	1	0	1	0
1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0
1	0	0	1	0	1	1	0	0	0	1	1	1	0	0	1

EXEMPEL 8.4 Här följer några språkliga utsagor härmade med en del av ovanstående satslogiska formler.

<i>utsaga</i>	<i>satslogisk formel</i>
Du är rik och du är vacker	$p \wedge q$
Du är rik men inte vacker	$p \wedge \neg q$
Du är rik eller vacker	$p \vee q$
Du är varken rik eller vacker	$NOR(p, q)$
Du är inte både rik och vacker	$NAND(p, q)$
Du kan eller så vill du inte	$p \vee \neg q$
Du kan om du vill	$p \leftarrow q$

■ Tolkning, validitet och satisfierbarhet

Att tolka en formel är att förstå dess betydelse.

En *tolkning* av en formel (*evaluering* är ett alternativt namn) avser en beräkning av formelns sanningsvärde för ett val av sanningsvärden på dess atomer. Man kan därför säga att en *tolkning* av en satslogisk formel representeras av en *rad* i formelns sanningstabell. En formel byggd av n stycken atomer har därför 2^n stycken tolkningar.

EXEMPEL 8.5 $(p \rightarrow q)$ har *fyra* tolkningar varav *en* är falsk:

	tolkningar	sanningsvärde
Jfr med definitionen på sid 192.	$(falsk \rightarrow falsk)$	<i>sann</i>
	$(falsk \rightarrow sann)$	<i>sann</i>
	$(sann \rightarrow falsk)$	<i>falsk</i>
	$(sann \rightarrow sann)$	<i>sann</i>

EXEMPEL 8.6 $(p \rightarrow p)$ har *två* tolkningar, båda är sanna:

tolkningar	sanningsvärde
$(falsk \rightarrow falsk)$	<i>sann</i>
$(sann \rightarrow sann)$	<i>sann</i>

DEFINITION En formel sägs vara

- ▶ *valid* (eller *tautologisk*) om den är sann i alla tolkningar,
 - ▶ *falsifierbar* (eller *ickevalid*) om den är falsk i någon tolkning,
 - ▶ *satisfierbar* om den är sann i någon tolkning,
 - ▶ *osatisfierbar* om den är falsk i alla tolkningar.
-

■ Ekvivalens

Två formler u och v sägs vara *ekvivalenta* om de har lika sanningsvärde i varje tolkning som inbegriper båda formlernas argument. Vi skriver $u \equiv v$.

TABELL 1 Några viktiga ekvivalenser

			<i>illustrationer</i>
Idempotens regler	$(u \vee u) = u$	(3)	Kalle ljuger eller Kalle ljuger = Kalle ljuger
	$(u \wedge u) = u$	(4)	Kalle ljuger och Kalle ljuger = Kalle ljuger
Dubbla negationers regel	$\neg \neg u = u$	(5)	det är inte så att Kalle inte ljuger = Kalle ljuger
Kommutativa regler	$(u_1 \vee u_2) = (u_2 \vee u_1)$	(6)	
	$(u_1 \wedge u_2) = (u_2 \wedge u_1)$	(7)	
Associativa regler	$u_1 \vee (u_2 \vee u_3)$	(8)	
	=		
	$((u_1 \vee u_2) \vee u_3)$		
	$u_1 \wedge (u_2 \wedge u_3)$	(9)	
	=		
	$((u_1 \wedge u_2) \wedge u_3)$		
Distributiva regler	$u \wedge (v_1 \vee v_2)$	(10)	x är kvadratisk och delbart med 3 eller med 5
	=		=
	$(u \wedge v_1) \vee (u \wedge v_2)$		x är kvadratisk och delbart med 3 eller kvadratisk och delbart med 5
	$u \vee (v_1 \wedge v_2)$	(11)	x är prima eller delbart med 2 och med 3
	=		=
	$(u \vee v_1) \wedge (u \vee v_2)$		x är prima eller delbart med 2 och x är prima eller delbart med 3
allmänna:			
	$(u_1 \wedge \dots \wedge u_m) \vee (v_1 \wedge \dots \wedge v_n)$	(12)	
	= $(u_1 \vee v_1) \wedge \dots \wedge (u_1 \vee v_n)$		
	$\wedge (u_2 \vee v_1) \wedge \dots \wedge (u_2 \vee v_n)$		
	$\wedge \dots$		
	$\wedge (u_m \vee v_1) \wedge \dots \wedge (u_m \vee v_n)$		

TABELL 1 Några viktiga ekvivalenser

		<i>illustrationer</i>
	$(u_1 \vee \dots \vee u_m) \wedge (v_1 \vee \dots \vee v_n) \quad (13)$ $\equiv (u_1 \wedge v_1) \vee \dots \vee (u_1 \wedge v_n)$ $\vee (u_2 \wedge v_1) \vee \dots \vee (u_2 \wedge v_n)$ $\vee \dots$ $\vee (u_m \wedge v_1) \vee \dots \vee (u_m \wedge v_n)$	
De Morgans regler	$\neg(u \vee v) \equiv (\neg u \wedge \neg v) \quad (14)$	“inte någon sann” = “båda falska” “inte båda sanna” = “någon falsk”
	$\neg(u \wedge v) \equiv (\neg u \vee \neg v) \quad (15)$	
	allmänna:	
	$\neg(u_1 \vee \dots \vee u_n) \equiv (\neg u_1 \wedge \dots \wedge \neg u_n) \quad (16)$	
	$\neg(u_1 \wedge \dots \wedge u_n) \equiv (\neg u_1 \vee \dots \vee \neg u_n) \quad (17)$	
Absorberings- regler	$u \wedge (u \vee v) \equiv u \quad (18)$	
	$u \vee (u \wedge v) \equiv u \quad (19)$	

⌘ Anm.8.1 Att tex (18) stämmer inses av att om högerledet är en *sann* formel, dvs om u är *sann* så är både u och $(u \vee v)$ *sanna*, och därmed är vänsterledet en *sann* formel – oavsett hur det är med v .

Och om u är *falsk* så är ena delen i vänsterledets konjunktion *falsk*, och därmed gäller detsamma för hela vänsterledet – oavsett hur det är med v . \square

⌘ Anm.8.2 De associativa reglerna (8), (9) rättfärdigar den friare notationen $(u_1 \vee u_2 \vee \dots \vee u_n)$ resp. $(u_1 \wedge u_2 \wedge \dots \wedge u_n)$.

EXEMPEL 8.7 Av NAND-konnektivets innebörd (sid 193) och idempotensregeln följer att

$$\neg v \equiv \neg(v \wedge v) \equiv \text{NAND}(v, v) \quad (20)$$

Vidare följer av disjunktionskonnektivets beskrivning (sid 193) att $(\neg u \vee v) \equiv (\neg \neg u \rightarrow v)$ och därmed att $(\neg u \vee v) \equiv (u \rightarrow v)$ (enligt regeln om dubbla negationer), varför

$$\begin{aligned}
 (u \rightarrow v) &\equiv (\neg u \vee v) \equiv (\neg u \vee \neg \neg v) \equiv \neg(u \wedge \neg v) && \text{Dubbla nega-} && \text{De Morgan} \\
 & && \text{tioners regel} && \\
 &\equiv \text{NAND}(u, \neg v) \equiv \text{NAND}(u, \text{NAND}(v, v)) && && \\
 &\text{av (20)} && &&
 \end{aligned} \quad (21)$$

NAND är ett (20) och (21) säger oss att NAND-konnektivet är så uttrycksfullt att
 universalklister! varje formel kan beskrivas enbart i termer av detta konnektiv.

Normalformerna

En satslogisk formel sägs vara skriven på *konjunktiv normalform* om den är en *konjunktion* av disjunktioner av s.k. *literals* L_{ik} .

$$\text{konjunktiv normalform} \quad (L_{11} \vee \dots \vee L_{1n_1}) \wedge \dots \wedge (L_{m1} \vee \dots \vee L_{mn_m}) \quad (22)$$

Med *literal* avses en *atom* eller en *negerad atom*.

På motsvarande sätt är en formel given på *disjunktiv normalform* om den är en *disjunktion* av konjunktioner av *literals*:

$$\text{disjunktiv normalform} \quad ((L_{11} \wedge \dots \wedge L_{1n_1}) \vee \dots \vee (L_{m1} \wedge \dots \wedge L_{mn_m})) \quad (23)$$

Normalformerna uppstår naturligt om man försöker formulera en formel utgående ifrån dess semantik, vilket illustreras nedanför.

EXEMPEL 8.8 Vi skall här konstruera en satslogisk formel w för utsagan

$$\text{"Om jag vill och du vill, så leker vi. Och om en av oss vill men inte den andre, så leker vi inte."} \quad (24)$$

Lägg märke till att utsagan inte uttalar sig om fallet att "ingen av oss vill".

Om j , d , v betecknar i tur och ordning "jag vill", "du vill", "vi leker", så ger "direkt" härledning av (24):

$$w = (((j \wedge d) \rightarrow v) \wedge (((j \wedge \neg d) \vee (\neg j \wedge d)) \rightarrow \neg v)) \quad (25)$$

Men då uppenbarar sig *inte* normalformerna.

Vi gör en annan ansats – med hjälp av semantiken hos formel (25).

Sanningsvärdena för nämnda formel kan visas vara:

Visa detta!

j	d	v	w
0	0	0	1
0	0	1	1
0	1	1	0
0	1	0	1
1	1	0	0
1	1	1	1
1	0	1	0
1	0	0	1

Dvs w är falsk då

j är falsk och d är sann och v är sann
 eller j är sann och d är sann och v är falsk
 eller j är sann och d är falsk och v är sann

Och w är sann då

j är falsk och d är falsk (oberoende av v 's sanningsvärde)
 eller j är falsk och d är sann och v är falsk
 eller j är sann och d är sann och v är sann
 eller j är sann och d är falsk och v är falsk

Beskrivningen av "sann w " ger oss w på *disjunktiv* normalform:

$$w = ((\neg j \wedge \neg d) \vee (\neg j \wedge d \wedge \neg v) \vee (j \wedge d \wedge v) \vee (j \wedge \neg d \wedge \neg v)) \quad (26)$$

Och beskrivningen av "falsk w " ger oss $\neg w$ på *disjunktiv* normalform:

$$\neg w = ((\neg j \wedge d \wedge v) \vee (j \wedge d \wedge \neg v) \vee (j \wedge \neg d \wedge v)) \quad (27)$$

från vilken det är lätt att komma till w :s *konjunktiva* normalform:

$$\begin{aligned} w &= \neg(\neg w) & w &= ((j \vee \neg d \vee \neg v) \wedge (\neg j \vee \neg d \vee v) \wedge (\neg j \vee d \vee \neg v)) \\ &= \neg((\neg j \wedge d \wedge v) \vee \dots) & & \\ &= (j \wedge \neg d \wedge \neg v) \wedge \dots & & \end{aligned} \quad (28)$$

✂ TEST 8.1 Visa

- a) att $(\neg u \rightarrow v)$ är sann om och endast om u eller v (någon av dem) är sann.
- b) att $\neg(u \rightarrow \neg v)$ är sann om och endast om u och v båda är sanna.
- c) att $(p \rightarrow p), (\neg p \rightarrow (p \rightarrow q))$ är valida, och att $(p \wedge \neg p)$ är osatisfierbar.
- d) att NOR-konnektivet har samma starka uttryckskraft som NAND, dvs att *varje* formel kan uttryckas (enbart) i termer av NAND.

■ 8.2 Validitetsproblemet och resolution

Givet en formel w , är den *valid*?

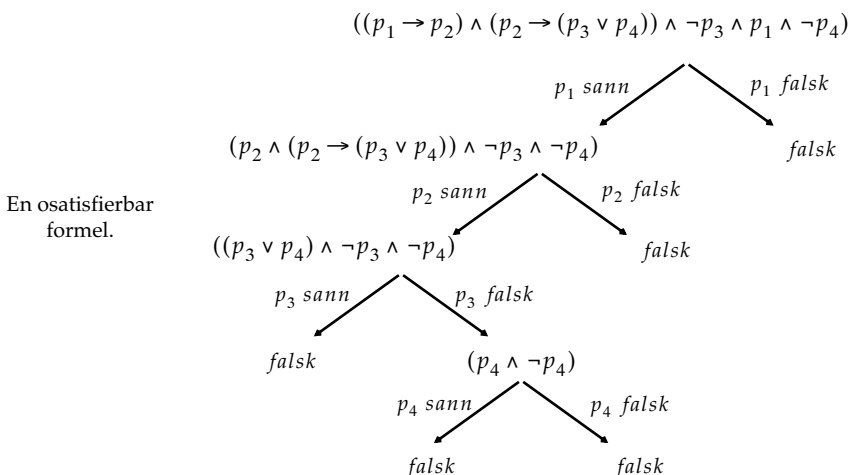
Eller, vilket är ekvivalent, är $\neg w$ *osatisfierbar*?

Frågan kan besvaras med hjälp av olika sorters algoritmer. T ex kan man teckna ned w :s sanningsstabell, och se efter om w är *sann* i alla tabellens rader. Detta är en sorts *bottom-up*-algoritm. Ty den utgår från sanningsvärdena i w :s hierarkiska botten – atomerna. En annan *bottom-up* algoritm (Quine:s trädalgoritm) illustreras i nästa exempel. Och därefter följer två *top-down*-algoritmer som visserligen inte är lika elementära som sanningsstabellsmetoden eller Quine:s trädalgoritm, men som har fördelen av att kunna generaliseras till användbara algoritmer för *predikatlogiken* (se sid 212).

■ Quine:s algoritm, en bottom-up-algoritm

EXEMPEL 8.9 Se på formeln $((p_1 \rightarrow p_2) \wedge (p_2 \rightarrow (p_3 \vee p_4)) \wedge \neg p_3 \wedge p_1 \wedge \neg p_4)$.

Genom att undersöka vad formeln betyder när p_1 är *sann* respektive *falsk*, får man två vägar att undersöka vidare. Vad betyder dessa när p_2 är *sann* respektive *falsk* osv ...? Det hela resulterar i ett *träd* vars löv ger besked om formelns semantik:



Finns sanningsvärdet *falsk* i samtliga löv har vi en *osatisfierbar* formel, och finns värdet *sann* i samtliga löv råder *validitet*. Annars är formeln varken *valid* eller *osatisfierbar*. Som synes är den här formeln *osatisfierbar*.

■ Två topdown-algoritmer

Vi skall nu bekanta oss med två *topdown*-algoritmer – semantisk tablå resp. klausulresolution – kompetenta att utföra samma syssla som sanningstabellmetoden och *Quine*s trädalgoritm ovan. De är *topdown*-algoritmer i bemärkelsen att man utgående från ett visst antagande om en formels sanningsvärde drar successiva slutsatser om sanningsvärdena för *enklare delar* av formeln. Under denna process "upplöses" formeln i sina enklaste beståndsdelar, atomerna. Båda algoritmerna har karaktär av motsägelsebevis, och fungerar naturligtast i sammanhang där man vill bevisa *osatisfierbarhet* eller *validitet* för en formel. *Validitet* bevisas enklast genom att betrakta den negerade formeln och bevisa att den är *osatisfierbar*. Normalformerna spelar viktiga roller i dessa två algoritmer. I den första transformeras den givna formeln till *disjunktiv* normalform, och i den andra till *konjunktiv* normalform.

■ Semantisk tablå. (Beth, Hintikka, Smullyan)

EXEMPEL 8.10 Låt $w = (((p \vee q) \rightarrow q) \rightarrow \neg p)$.

Genom att som nedan dra slutsatser av antagandet att w är sann, får man ett "slutledningsträd" (en semantisk tablå) vars noder innehåller dessa slutsatser. En slutsats av typen "det *och* det gäller" sträcker ut trädet (så det blir djupare). Och en slutsats av typen "det *eller* det gäl-

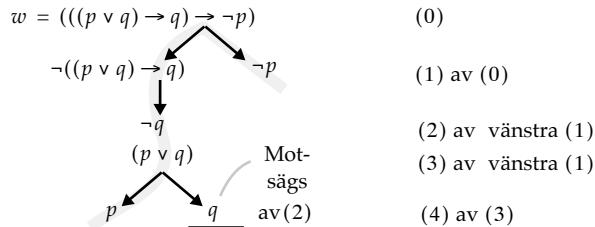
Med *väg* avses
en väg mellan
rotnoden och
ett löv

ler" gör att trädet förgrenas.

Om en *väg* av noder från trädets rot och nedåt innehåller en nod som motsäger en annan nod högre upp längs *samma* väg, så "*stänger*" algoritmen vägen ifråga. (Ty slutsatser i ett motsägelsefullt resonemang har inget värde.) Och om *varje* väg innehåller någon motsägelse så är formeln w falsk i varje tolkning – dvs då är den *osatisfierbar*. Omvänt, om någon väg blir motsägelsefri (när alla slutsatser är dragna), så är ursprungsformeln sann längs denna väg. Närmare bestämt utgör sanningsvärdena hos vägens atomära noder *en sann tolkning* av formeln.

För att återgå till detta exemplars slutledningsträd så finner man att *två* av trädets tre vägar är motsägelsefria:

Demarkerade
vägar
är
motsägelse-
fria

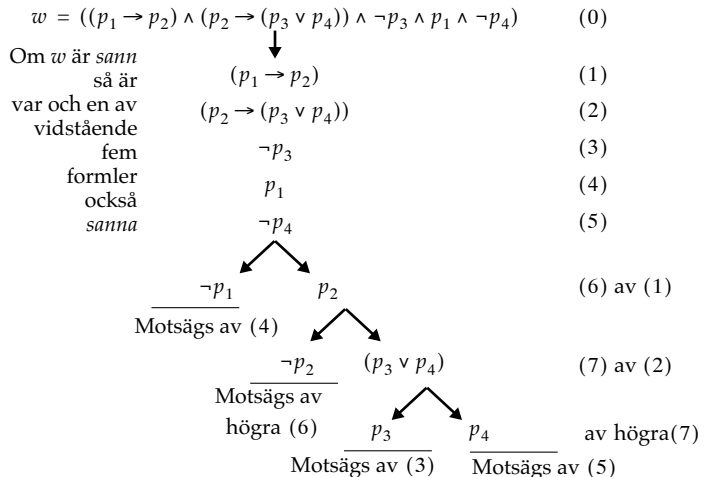


Detta visar att w är satisfierbar, närmare bestämt är w sann då q är falsk och p är sann eller då p är falsk (oavsett vilket sanningsvärde q har).

EXEMPEL 8.11 Vi tar $w = ((p_1 \rightarrow p_2) \wedge (p_2 \rightarrow (p_3 \vee p_4)) \wedge \neg p_3 \wedge p_1 \wedge \neg p_4)$ igen, samma formel som i EXEMPEL 8.9.

Här får man en semantisk tablå där *ingen* väg är motsägelsefri, vilket visar att denna formel är *osatisfierbar*.

Ingen väg är
motsägelsefri.



⌘ Anm.8.3 LÄGG MÄRKE till att slutsatsen i en nod inte säkert är en följd av det som finns i noden omedelbart ovanför, men däremot av *någon* nod ovanför. Notera också att om man drar en slutsats från en nod, så måste varje väg under noden ifråga – dock inte en redan avstängd väg – ta emot slutsatsen.

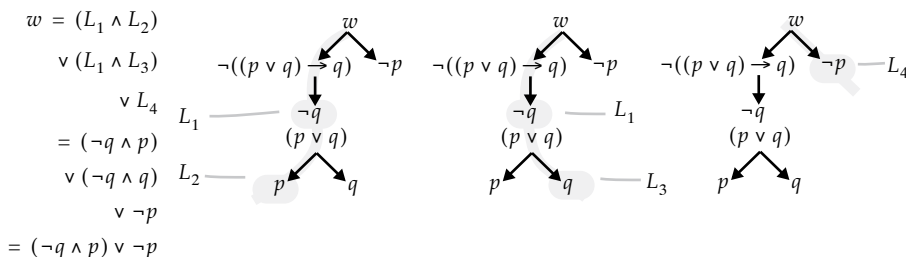
⌘ Anm.8.4 Ett fullständigt utdraget slutledningsträd utgör en transformation till *disjunktiv* (se sid 198) normalform:

$$(L_{11} \wedge \dots \wedge L_{1n_1}) \vee \dots \vee (L_{m1} \wedge \dots \wedge L_{mn_m}) \quad (29)$$

av rotnodsformeln.

Varje konjunktion $L_1 \wedge \dots \wedge L_n$ i (29) motsvarar här en väg i slutledningsträdet, och omvänt. Närmare bestämt utgörs literalerna L_1, \dots, L_n av de atomära noderna i den aktuella vägen. Allt detta är en konsekvens av att rotnodsformeln är sann om och endast om någon väg innehåller enbart sanna formler i sina noder.

Som illustration betraktar vi åter $w = (((p \vee q) \rightarrow q) \rightarrow \neg p)$ från EXEMPEL 8.10. Av dess träd med tre vägar (se nedan) framgår det att w kan skrivas på *disjunktiv normalform* som $(\neg q \wedge p) \vee (\neg q \wedge q) \vee \neg p$, dvs (efter förenkling) $(\neg q \wedge p) \vee \neg p$.



■ Klausul-resolution (Davis, Putnam)

Klausul och klausulmängd I detta sammanhang kallas $L_1 \vee \dots \vee L_n$ för *klausul*, och $((L_{11} \vee \dots \vee L_{1n_1}) \wedge \dots \wedge (L_{m1} \vee \dots \vee L_{mn_m}))$ för *klausulmängd*. I klausulresolutionsalgoritmen som vi strax skall beskriva börjar man med att transformera den givna formeln till konjunktiv normalform.

Mängdnotation används fortsättningsvis för att beskriva såväl klausuler som klausulmängder.

	<i>klausul</i>	<i>klausulmängd</i>
<i>mängdnotation</i>	$\{L_{11}, \dots, L_{1n_1}\}$	$\{\{L_{11}, \dots, L_{1n_1}\}, \dots, \{L_{m1}, \dots, L_{mn_m}\}\}$
<i>logiknotation</i>	$(L_{11} \vee \dots \vee L_{1n_1})$	$((L_{11} \vee \dots \vee L_{1n_1}) \wedge \dots \wedge (L_{m1} \vee \dots \vee L_{mn_m}))$

En mängd är bestämd av sina element – oavsett i vilken ordning de räknas upp.

En poäng med mängdnotationen är att den understryker att ordningsföljden mellan literalerna är ovidkommande, precis som ordningsföljden mellan klausulerna.

I exemplet som följer presenteras klausulresolutionen.

EXEMPEL 8.12 Betrakta (än en gång)

$$w = ((p_1 \rightarrow p_2) \wedge (p_2 \rightarrow (p_3 \vee p_4)) \wedge \neg p_3 \wedge p_1 \wedge \neg p_4)$$

Transformation till konjunktiv normalform ger

$$w = ((\neg p_1 \vee p_2) \wedge (\neg p_2 \vee p_3 \vee p_4) \wedge \neg p_3 \wedge p_1 \wedge \neg p_4)$$

en klausulmängd bestående av fem klausuler. Med mängdnotation:

$$w = \{\{\neg p_1, p_2\}, \{\neg p_2, p_3, p_4\}, \{\neg p_3\}, \{p_1\}, \{\neg p_4\}\} \quad (30)$$

Med två motstridiga literaler avses en atom och samma atom negerad.

Klausulresolutionen går ut på, att upprepade gånger leta (i klausulmängden) efter två klausuler med varsina motstridiga literaler.

$$\{\neg p_1, p_2\}, \{\neg p_2, p_3, p_4\} \quad (31)$$

är ett sådant par (med p_2 och $\neg p_2$ som motstridiga literaler).

Antag nu att (30) är sann (i någon tolkning).

Förklara!

$$\text{Då är de två utvalda klausulerna i (31) sanna.} \quad (32)$$

Av detta följer att den s k *resolventen*[†]

$$\{\neg p_1, p_3, p_4\} \quad (33)$$

till de två utvalda klausulerna är sann. Något som kräver en förklaring:

Varför?

Om (33) vore falsk, skulle *alla dess literaler vara falska*. Och därmed skulle de två utvalda klausulerna innehålla enbart falska literaler förutom de två motstridiga literalerna. Men de senare kan inte vara falska båda (eftersom de är motstridiga), och inte heller sanna båda. Härav följer att *en* av de två utvalda klausulerna skulle vara tvungen att innehålla enbart falska literaler, vilket skulle göra den falsk – i strid mot (32).

†. Resolventen till två klausuler är den klausul som bildas av disjunktionen mellan de literaler i de två klausulerna som återstår efter att de två motstridiga literalerna har avlägsnats.

Alltså är (33) sann under antagandet att (30) är det.
 Detta ges ofta en grafisk framställning

$$\begin{array}{c} \{\neg p_1, p_2\}, \{\neg p_2, p_3, p_4\}, \{\neg p_3\}, \{p_1\}, \{\neg p_4\} \\ \swarrow \quad \searrow \\ \{\neg p_1, p_3, p_4\} \end{array} \quad (34)$$

Observera att vi
inte hävdar att
 $\{\neg p_1, p_3, p_4\}$ är
 sann, bara att
 den är sann *om*
 övre raden i (34)
 är sann.

som utläses på följande sätt:

Om mängden av klausuler på övre raden i (34) är sann i någon tolkning, så är resolventen $\{\neg p_1, p_3, p_4\}$ sann i samma tolkning. Och det samma gäller nedanstående utvidgade klausulmängd.

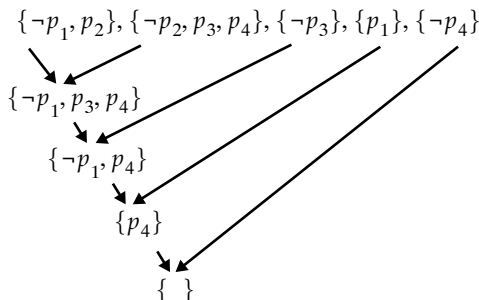
$$\{\{\neg p_1, p_2\}, \{\neg p_2, p_3, p_4\}, \{\neg p_3\}, \{p_1\}, \{\neg p_4\}, \{\neg p_1, p_3, p_4\}\} \quad (35)$$

Vid varje resolventbildning sker en motsvarande utvidgning.

Allt eftersom resolventbildningen fortskrider får man på detta sätt en allt större mängd av sanna klausuler – under förutsättning att den givna klausulmängden är sann.

Detta illustreras i grafen nedan, där fyra nya sanna klausuler (resolventer) har tillkommit. Den undre av dessa är en *tom* klausul.

Resolventen
 till paret $\{p_4\}$,
 $\{\neg p_4\}$ är tom.



Varför kan en
 tom klausul
 inte vara
 sann?

Men en tom klausul kan *inte* vara sann.

Således tvingas vi förkasta antagandet om att w är sann i någon tolkning, och istället konstatera att w är *osatisfierbar*. \square

✂ TEST 8.2

a) Vad visar de semantiska tablåerna för $\neg w$ och w om
 $w = (((p_1 \wedge p_2) \rightarrow p_3) \wedge (\neg p_1 \rightarrow p_4)) \rightarrow (p_3 \vee p_4)$?

b) Visa att varje klausul på formen $\neg p_1 \vee \dots \vee \neg p_k \vee p_{k+1} \vee \dots \vee p_m$
 kan skrivas som $(p_1 \wedge \dots \wedge p_k) \rightarrow (p_{k+1} \vee \dots \vee p_m)$.

- c) Bilda alla resolventer till $\{p_1, p_2\}, \{\neg p_1, \neg p_2\}$.
- d) Visa med klausulresolution att
 $\{\{\neg p_1, p_2\}, \{\neg p_2, p_3\}, \{\neg p_3, p_1\}, \{p_1, p_2, p_3\}, \{\neg p_1, \neg p_2, \neg p_3\}\}$
 är osatisfierbar.
- e) Visa med klausulresolution att $((p_1 \rightarrow p_2) \rightarrow \neg p_2) \rightarrow \neg p_2$ är valid.

8.3 Första ordningens predikatlogik

Med satslogiska formler kan man visserligen härma vissa enkla resonemang förda i naturligt språk. Men mer subtila resonemang kan inte fångas mer än på ett väldigt grovt sätt. Om Du försöker formulera satslogiska formler för nedanstående resonemang kan Du få en aning om vad satslogiken saknar.

Allt är förgängligt. (1)

Eftersom ett primtal endast är delbar med 1 och sig själv, men 6 är delbar med 2, så är 6 inget primtal. (2)

Det finns en Turingmaskin T sådan att $T(M, w) = M(w)$ för varje Turingmaskin M och varje input w . (3)

Ett försök att formalisera resonemangen (1), (2), (3) kräver uppenbarligen att man kan

- (i) uttrycka att "individer" har *egenskaper*, t ex
 – att *individen x* har egenskapen att *vara förgänglig*, eller att *vara prima* eller att *vara en Turingmaskin*,
 – att ett par av individer x, y har egenskapen att x är delbar med y ,
 – att individerna T, M, w har egenskapen att individerna $T(M, w), M(w)$ är lika.
- (ii) uttrycka att en egenskap ägs av *någon* eller *alla* individer, t ex
 – att x är förgänglig *för alla x* ,
 – att för *någon* T , individerna $T(M, w), M(w)$ är lika *för alla M och alla w* .
 Detta kallas för att *kvantifiera*.

Den utvidgning av satslogiken som (i) och (ii) tvingar oss till kallas *predikatlogik*.

Vi skall i denna framställning enbart ägna oss åt vad som kallas för *första ordningens predikatlogik*, vilket innebär att vi begränsar oss till att kvantifiera *enkla* individer, dvs individer som *inte är funktioner* av individer. I första ordningens predikatlogik kan man därför inte "ut-

an vidare" formalisera resonemanget (3) vilket (i den givna presentationen) involverar kvantifieringar av individer T och M som är *funktioner* av individer.

Syntax

Symboler När vi nu skall definiera *syntaxen* för de *predikatlogiska strängarna* behöver vi *symboler* för de allra enklaste resonemangen som vi vill fånga, samt *symboler* för de individer som resonemangen skall handla om. Här är vårt val av sådana symboler

- predikatsymboler (iii) Predikatsymboler $\{p, q, \dots\}$ för individers *egenskaper* och *förehavanden*.
- tre typer av individ-symboler (iv) Konstantsymboler $\{a, b, \dots\}$, för enkla konstanta individer.
- (v) Variabelsymboler $\{x, y, \dots\}$, för enkla variabla individer.
- (vi) Funktionssymboler $\{F, G, \dots\}$, för sammansatta individer, dvs funktioner av individer.

Vi behöver också symboler för att klistra ihop enkla resonemang till mer komplexa. Här tar vi samma *konnektivsymboler* som i satslogiken:

- konnektivsymboler (vii) \neg, \rightarrow

Slutligen behöver vi *kvantifierings-symboler*:

- kvantifierings-symboler (viii) Existenskvantorn \exists för att uttrycka att *någon* individ har en egenskap.
- (ix) Allkvantorn \forall för att uttrycka att *alla* individer har en egenskap.

Faktum är att det vi vill uttrycka med *en* av dessa två kvantorer kan uttryckas hjälp av den andra i kombination med negationskonnektivet. Av sparsamhetsskäl tar vi därför i definitionen (se nedan) endast med *en* av de två kvantorerna, *allkvantorn*, och introducerar i ett senare skede den andra med hjälp av den förra (se (4) på sidan 209).

Syntax-definition

DEFINITION Välformade strängar (predikat eller predikatlogiska form-
ler) i första ordningens predikatlogik definieras som nedan

- atomära formler ▶ Om p är en predikatsymbol med n argument och t_1, t_2, \dots, t_n är individsymboler så är $p(t_1, t_2, \dots, t_n)$ en predikatlogisk formel.
- sammansatta formler ▶ Om u och v är predikatlogiska formler, så är $\neg u$ och $(u \rightarrow v)$ predikatlogiska formler.
- ▶ Om u är en predikatlogisk formel och x är en enkel individsymbol, så är $(\exists x(u))$ en predikatlogisk formel.

- ⌘ Anm.8.5 $\exists x(u)$ utläses på något av följande två sätt
”det finns x så att u ”
” u för något x ”

EXEMPEL 8.13 Atomära och sammansatta formler ...

atomära formler	sammansatta formler
$p(x, y)$	$\neg p(F(x), x)$
$p(F(x), G(F(x)))$	$\exists x(p(F(x), F(G(x))))$

■ Fria och bundna variabler, öppna och slutna formler

Kvantorer sägs *binda* variabler. En *icke* bunden variabel kallas *fri*.

öppen resp. En formel utan fria variabler kallas *sluten*, och en formel med någon
 sluten formel fri variabel kallas *öppen*.

EXEMPEL 8.14 Öppna och slutna formler ...

öppna	slutna
$p(x, y), \exists y(p(x, y))$	$\exists x(\exists y(p(x, y)))$
$p(F(x), F(G(x)))$	$\exists x(p(F(x), F(G(x))))$

■ 8.4 Predikatlogikens semantik

För att ge mening åt predikatlogikens formler skall vi komplettera den semantik vi redan har för satslogikens formler. Allt som behövs är en regel för *existenskvantorn*, en regel som knappast kommer att överraska Dig (DEFINITION nedan). Men först några ord om vad som menas med att *tolka* en formel ...

att tolka är ... Att *tolka* en formel u är att beräkna dess sanningsvärde för

- ett val av *individuniversum* (icketom mängd) åt u :s enkla individ-symboler (se EXEMPLEN nedan),
- ett val av *element* ur *individuniversum* åt u :s konstantsymboler,
- ett val av *funktioner* åt u :s funktionssymboler,
- ett val av *predikat* åt u :s predikatsymboler.

EXEMPEL 8.15 En formel *tolkad* på två sätt över de naturliga talen:

två tolkningar av $p(a, b)$	sanningsvärde
$större(1, 0)$	<i>sann</i>
$mindre(1, 0)$	<i>falsk</i>

- ⌘ Anm.8.6 En *öppen* formel kan *inte* tolkas med mindre än att man väl-

jer element ur individuniversum åt de fria variablerna.

EXEMPEL 8.16 En öppen formel *tolkad* på två sätt över de naturliga talen:

<i>två tolkningar av</i>	<i>sanningsvärde</i>
$p(x, y) \vee p(y, x)$	
$\text{större}(1, 0) \vee \text{större}(0, 1)$	<i>sann</i>
$\text{större}(1, 1) \vee \text{större}(1, 1)$	<i>falsk</i>

Semantiken
för exis-
tenskvantorn.

DEFINITION $\exists x(u)$ är sann i en tolkning omm det finns något x i tolkningens individuniversum som gör u sann.

EXEMPEL 8.17 Några fler exempel på tolkade formler över individuniversum \mathbb{N} :

<i>formel</i>	<i>tolkad formel</i>	<i>utsaga</i>	<i>sanningsvärde</i>
$\neg \exists x(p(x))$	$\neg \exists x(\text{negativ}(x))$	“inget naturligt tal är negativt”	<i>sann</i>
$\neg \exists x(p(x))$	$\neg \exists x(\text{positiv}(x))$	“inget naturligt tal är positivt”	<i>falsk</i>
$\exists x(\neg p(x))$	$\exists x(\neg \text{positiv}(x))$		<i>sann</i> [†] .
$\exists x(\neg p(x))$	$\exists x(\neg \text{större} \vee \text{Lika} \vee \text{Noll}(x))$	“det finns ett naturligt tal som inte är större eller lika med noll”	<i>falsk</i>
$\exists x(p(x) \wedge q(x))$	$\exists x(\text{positiv}(x) \wedge \text{negativ}(x))$		<i>falsk</i>
$\exists x(p(x) \wedge q(x))$	$\exists x(\text{prima}(x) \wedge \text{jämn}(x))$		<i>sann</i> [†] .
$\exists x(\exists y(p(x, y)))$	$\exists x(\exists y(\text{större}(x, y)))$	“det finns naturliga tal sådana att det ena är större än det andra”	<i>sann</i>
$\exists x(\exists y(p(x, y)))$	$\exists x(\exists y(\text{negativSumma}(x, y)))$	“det finns naturliga tal sådana att deras summa är negativ”	<i>falsk</i>
$\neg(\exists x(\exists y(p(x, y))))$	$\neg(\exists x(\exists y(\text{negativSumma}(x, y))))$	“inga naturliga tal har negativ summa”	<i>sann</i>
$\neg(\exists x(\exists y(p(x, y))))$	$\neg(\exists x(\exists y(\text{större}(x, y))))$	“det finns inte naturliga tal sådana att det ena är större än det andra”	<i>falsk</i>

formel	tolkad formel	utsaga	sanningsvärde
$\exists x(\neg \exists y(p(y, x)))$	$\exists x(\neg \exists y(större(y, x)))$	”något naturligt tal är sådant att inget naturligt tal är större”	falsk
$\exists x(\neg \exists y(p(y, x)))$	$\exists x(\neg \exists y(mindre(y, x)))$	”något naturligt tal är sådant att inget naturligt tal är mindre”	sann [†] .

†. tag $x = 0$.

‡. tag $x = 2$.

EXEMPEL 8.18 Tolkade formler över ett människouniversum:

formel	tolkad formel	utsaga	sanningsvärde
$\exists x(\exists y(p(x, y)))$	$\exists x(\exists y(gillar(x, y)))$	”någon gillar någon”	ofta sann [†] .
$\neg(\exists x(\exists y(p(x, y))))$	$\neg(\exists x(\exists y(gillar(x, y))))$	”ingen gillar någon” [‡] .	ofta falsk [†] .

†. Det beror på vilka människor som är valda att ingå i individuniversum.

‡. = ”alla ogillar alla”

■ Allkvantorn

Låt $\forall x(u)$ beteckna $\neg \exists x(\neg u)$. (4)

Av semantikdefinitionen för $\exists x(u)$ (se sid 208) följer att formeln $\forall x(u) = \neg \exists x(\neg u)$ är sann i en tolkning omm det *inte* finns något x i tolkningens individuniversum som gör u falsk, vilket är detsamma som att u är sann för alla x i tolkningens individuniversum. Dvs hur än x väljs (i tolkningens individuniversum) gäller det att u är sann.

EXEMPEL 8.19 Utsagan

För godtyckligt tal y gäller det att om x är delbar med y , så måste y vara ...

x är endast delbar med 1 och sig själv

beskrivs av formeln

$$\forall y(delbar(x, y) \rightarrow (lika(y, 1) \vee lika(y, x)))$$

EXEMPEL 8.20 Utsagan

Med ett primtal menas här ett godtyckligt primtal.

Ett primtal är endast delbar med 1 och sig själv

beskrivs av

$$\forall x(prima(x) \rightarrow \forall y(delbar(x, y) \rightarrow (lika(y, 1) \vee lika(y, x))))$$

EXEMPEL 8.21 Utsagan

*Eftersom ett primtal endast är delbar med 1 och sig själv,
men 6 är delbar med 2,
så är 6 inget primtal.*

beskrivs av (notera att *men* uttrycks med \wedge):

$$\begin{aligned}
 &(\forall x(\text{prima}(x) \rightarrow \forall y(\text{delbar}(x, y) \rightarrow (\text{lika}(y, 1) \vee \text{lika}(y, x)))) \\
 &\quad \wedge \text{delbar}(6, 2)) \\
 &\rightarrow \neg \text{prima}(6)
 \end{aligned}$$

⌘ Anm.8.7 Jämför \forall -kvantorns semantik med \wedge -konnektivets:

	<i>sann omm</i>
$\forall x(u)$	<i>u är sann för alla x</i>
$(u_1 \wedge u_2 \wedge \dots \wedge u_n)$	<i>u_i är sann för alla i</i>

\forall generaliserar \wedge och \exists generaliserar \vee Som synes är \forall -kvantorn är en sorts generalisering av \wedge -konnektivet. På motsvarande sätt (se själv!) är \exists är en generalisering av \vee -konnektivet.

Några ekvivalensregler Ekvivalensregler som vi har presenterat i satslogiken (se sid 195) är givetvis giltiga även i predikatlogiken. Vidare kan en del av dem generaliseras till nya ekvivalensregler i predikatlogiken (eftersom \wedge , \vee är generaliseringar av \forall , \exists):

Driva in eller ut \neg	$\neg \exists x(u) \equiv \forall x(\neg u)^{\dagger}$	(5)
	$\neg \forall x(u) \equiv \exists x(\neg u)$	(6)
Driva in eller ut \forall	$\forall x(u \wedge v) \equiv (\forall x(u) \wedge \forall x(v))^{\ddagger}$	(7)
	$\forall x(u \wedge v) \equiv (u \wedge \forall x(v))$ <i>om u saknar fritt x</i>	(8)
	$(u \vee \forall x(v)) \equiv \forall x(u \vee v)$ <i>om u saknar fritt x^{††}</i>	(9)
Driva in eller ut \exists	$\exists x(u \vee v) \equiv (\exists x(u) \vee \exists x(v))^{\ddagger\ddagger}$	(10)
	$\exists x(u \vee v) \equiv (u \vee \exists x(v))$ <i>om u saknar fritt x</i>	(11)
	$(u \wedge \exists x(v)) \equiv \exists x(u \wedge v)$ <i>om u saknar fritt x^{†††}</i>	(12)

[†]. Jfr med de Morgans regler i satslogiken.

[‡]. Jfr med satslogikens

$$((u_1 \wedge v_1) \wedge \dots \wedge (u_n \wedge v_n)) \equiv ((u_1 \wedge \dots \wedge u_n) \wedge (v_1 \wedge \dots \wedge v_n)).$$

^{††}. Jfr med satslogikens $(u \vee (v_1 \wedge \dots \wedge v_n)) \equiv (u \vee v_1) \wedge \dots \wedge (u \vee v_n)$.

##. Jfr med satslogikens

$$((u_1 \vee v_1) \vee \dots \vee (u_n \vee v_n)) = ((u_1 \vee \dots \vee u_n) \vee (v_1 \vee \dots \vee v_n)).$$

+++ Jfr med satslogikens $(u \wedge (v_1 \vee \dots \vee v_n)) = (u \wedge v_1) \vee \dots \vee (u \wedge v_n)$.

EXEMPEL 8.22 I $\forall x(p(x) \vee q(y))$ gäller det att $q(y)$ saknar fritt x , varför $\forall x$ kan drivas ut – enligt (8). Man får $\forall x(p(x) \vee q(y)) = \forall x(p(x) \vee q(y))$.

EXEMPEL 8.23 Eftersom namnet på en kvantifierad variabel är en "intern" angelägenhet för den kvantifierade delen av formeln och inte har någon relevans utanför denna del, kan man byta namn på en kvantifierad variabel, bara man ser till att verkligen göra namnbytet för varje förekomst av variabeln i den kvantifierade delen av formeln. Nedan är ett sådant namnbyte ett nödvändigt moment för att möjliggöra utdrivning av allkvantorn:

Efter namnbyte på den kvantifierade variabeln y kan $\forall x$ drivas ut.

$$((p(y) \vee \forall y(q(y))) = (p(y) \vee \forall x(q(x)))) = \forall x(p(y) \vee q(x))$$

■ Validitet och satisfierbarhet i predikatlogik

Begreppen *validitet* och *satisfierbarhet* kan tas över ordagrant från satslogiken:

DEFINITION En formel är

- ▶ *valid* om den är sann i alla tolkningar,
- ▶ *ickevalid* (eller *falsifierbar*) om den är falsk i någon tolkning,
- ▶ *satisfierbar* om den är sann i någon tolkning,
- ▶ *osatisfierbar* om den är falsk i alla tolkningar.

EXEMPEL 8.24 $\exists x(\neg \exists y(p(y, x)))$ är *satisfierbar* men inte *valid*. (Se EXEMPEL 8.17.)

EXEMPEL 8.25 $(\forall x(p(x)) \wedge \exists y(\neg p(y)))$ är *osatisfierbar*. Ty i annat fall (om formeln vore sann i någon tolkning) skulle det finnas ett predikat p och ett individuniversum U sådana att både $\forall x(p(x))$ och $\exists y(\neg p(y))$ är *sanna*, dvs så att $p(x)$ är sann för alla $x \in U$ och $p(a)$ är falsk för något $a \in U$, vilket är orimligt.

EXEMPEL 8.26 $(\forall x(p(x)) \vee \exists y(\neg p(y)))$ är *valid*. Ty om formeln vore falsk i någon tolkning skulle (efter negering)[†] $(\exists x(\neg p(x)) \wedge \forall y(p(y)))$ vara sann i samma tolkning, vilket motsäger slutsatsen i EXEMPEL 8.25.

EXEMPEL 8.27 $\forall x \forall y \forall z (add(x, 0, x) \wedge (add(x, Öka(y), Öka(z)) \leftarrow add(x, y, z)))$ med $x, y, z \in \mathbb{N}$ är en tolkning av

$$\forall x \forall y \forall z (p(x, a, x) \wedge (p(x, f(y), f(z)) \leftarrow p(x, y, z))) \quad (13)$$

†. $\neg(\forall x(p(x)) \vee \exists y(\neg p(y))) = (\neg \forall x(p(x)) \wedge \neg \exists y(\neg p(y))) = (\exists x(\neg p(x)) \wedge \forall y(p(y)))$

med predikatet $\text{add}(x, y, z)$ uttryckande att $x + y = z$.

Uppenbarligen är (13) *sann* i denna tolkning, eftersom tolkningen helt enkelt uttrycker att $x + 0 = x$ och att $x + (y + 1) = z + 1$ om $x + y = z$, något som vi inte kan invända mot. Detta visar att (13) är *satisfierbar*.

8.5 Validitetsproblemet och resolution i predikatlogik

Som vi påtalade tidigare (se 8.2) är validitetsproblemet i satslogiken algoritmiskt avgörbart, och vi har presenterat flera algoritmer för detta.

Annorlunda är det i predikatlogiken.

I första ordningens predikatlogik är de *valida* formlerna *inte* algoritmiskt *avgörbara*: Att *stanna* för *varje* formel och *meddela* huruvida den är *valid* eller icke kan ingen algoritm klara av.[†]

Men däremot finns det algoritmer som på ett mindre upplysande sätt kan skilja de *valida* formlerna från de *icke-valida*, algoritmer som *stannar* blott för de *valida* formlerna och därvid meddelar att *validitet* råder, men som går in i oändliga slingor för de *icke-valida* formlerna.[‡] Vi skall se nedan hur de två resolutionsmetoderna från satslogiken kan generaliseras till sådana algoritmer. (Noga räknat så visar vi att en formel är valid genom att visa att negationen av densamma är osatisfierbar.)

I *högre* ordningars predikatlogik (där man får kvantifiera inte bara enkla individer utan även individer som är funktioner av individer) är det värre. Här är det *omöjligt* att tillverka någon algoritm ens med kompetens att skilja de *valida* formlerna från de *icke-valida* genom att stanna blott för de första.^{††}

EXEMPEL 8.28 Semantisk tablå. Vi visar att

$$\forall x(p(x) \rightarrow q(x)) \rightarrow (\forall x(p(x)) \rightarrow \forall x(q(x)))$$

är valid, genom att visa att dess negation är osatisfierbar:

†. Bevisades av Church (1936) och av Turing (1936-1937) (Hilberts ENTSCHEIDUNGSPROBLEM).

‡. I Gödels doktorsavhandling (1929) bevisades detta första gången, men i en något annorlunda tappning (1:a ordningens predikatlogik är *fullständig*.)

††. Gödel 1931.

Semantisk
tablå där
ingen väg
är motsä-
gelsefri.

$$\begin{array}{ll}
 \neg(\forall x(p(x) \rightarrow q(x)) \rightarrow (\forall x(p(x)) \rightarrow \forall x(q(x)))) & (0) \\
 \downarrow & \\
 \forall x(p(x) \rightarrow q(x)) & (1) \text{ av } (0) \\
 \neg(\forall x(p(x)) \rightarrow \forall x(q(x))) & (2) \text{ av } (0) \\
 \downarrow & \\
 \forall x(p(x)) & (3) \text{ av } (2) \\
 \neg \forall x(q(x)) & (4) \text{ av } (2) \\
 \downarrow & \\
 \neg q(a) & (5) \text{ av } (4) \\
 p(a) \rightarrow q(a) & (6) \text{ av } (1) \\
 \swarrow \quad \searrow & \\
 \neg p(a) \quad q(a) & (7) \text{ av } (6) \\
 \hline
 \text{Motsägs av (3)} \quad \text{Motsägs av (5)} &
 \end{array}$$

EXEMPEL 8.29 Klausul-resolution.

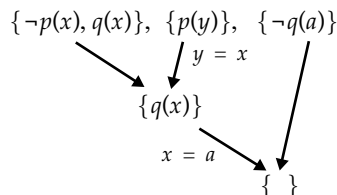
Vi tar samma formel som nyss och visar att dess negation är osatisfierbar. Men innan vi kan börja med själva resolutionen måste vi transformera (den negerade) formeln till klausulform:

$$\begin{array}{ll}
 \neg(\forall x(p(x) \rightarrow q(x)) \rightarrow (\forall x(p(x)) \rightarrow \forall x(q(x)))) & \\
 = \forall x(p(x) \rightarrow q(x)) \wedge \neg(\forall x(p(x)) \rightarrow \forall x(q(x))) & \text{transformerat bort ett "}\rightarrow\text{"} \\
 = \forall x(\neg p(x) \vee q(x)) \wedge (\forall x(p(x)) \wedge \neg \forall x(\neg q(x))) & \text{transformerat bort alla "}\rightarrow\text{"} \\
 = \forall x(\neg p(x) \vee q(x)) \wedge (\forall x(p(x)) \wedge \exists x(\neg q(x))) & \text{drivit in "}\neg\text{"} \\
 = \forall x(\neg p(x) \vee q(x)) \wedge (\forall x(p(x)) \wedge \neg q(a)) & \text{eliminerat "}\exists\text{"} \\
 = \forall x \forall y(\neg p(x) \vee q(x)) \wedge p(y) \wedge \neg q(a) & \text{drivit ut "}\forall\text{"} \\
 = \{\{\neg p(x), q(x)\}, \{p(y)\}, \{\neg q(a)\}\} & \text{klausulform}
 \end{array}$$

Lägg märke till att vi, av bekvämlighetsskäl, *inte* skriver ut \forall -kvanifieringen i klausulformen, och att klausulformen för övrigt inte är annat än den *konjunktiva normalformen* presenterad med mängdnotation. Notera också hur en *neutral* konstant uppenbarar sig på den \exists -kvanifierade variabelns plats när \exists -kvantorn elimineras.

Klausulresolutionen görs nu på motsvarande sätt som i satslogiken (jämför med klausulresolutionen i EXEMPEL 8.12 på sidan 203):

Klausulresolu-
tion med två
unifieringar
 $y = x$ och
 $x = a$.



Skillnaden är blott att man under resolventbildningen nu ibland

tvingas utföra ytterligare ett moment, s.k. unifiering. Detta tillgår så att man genom att tilldela lämpliga värden på variablerna söker matcha ett element i en klausul mot negationen av samma element i en annan klausul i syfte att ett motsatspar skall uppenbara sig (t ex sätts x till a när $q(x)$ matchas mot $\neg q(a)$ ovan). Att sådan substitution är tillåten beror på att en klausulmängds individvariabler är \forall -kvantifierade.

Den tomma klausulen – som i exemplet uppnås redan efter två resolventbildningar – visar att (den negerade) formeln är osatisfierbar. Dvs ursprungsformeln är valid.

Man skulle i det här exemplet också kunna göra en *annan* följd av två resolventbildningar med unifiering i var och en, eller hur.

I nästa exempel illustreras hur man med resolutionens hjälp kan utföra *beräkningar* med hjälp av formler.

EXEMPEL 8.30 Följande (tolkade) formel beskriver hur naturliga tal adderas:

jfr
EXEMPEL 8.27 $(add(x, 0, x) \wedge (add(x, \ddot{O}ka(y), \ddot{O}ka(z)) \leftarrow add(x, y, z)))$
Omformning till klausulform ger

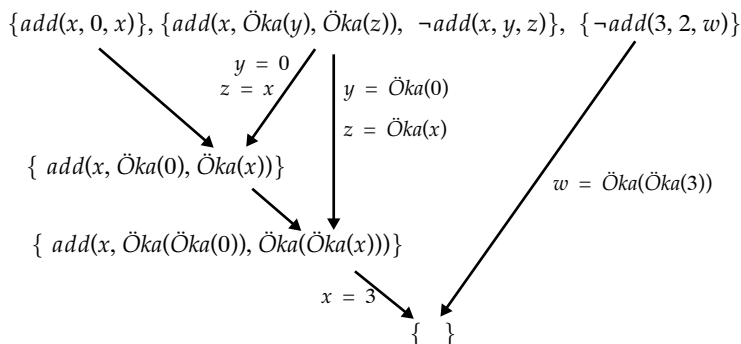
$$\{\{add(x, 0, x)\}, \{add(x, \ddot{O}ka(y), \ddot{O}ka(z)), \neg add(x, y, z)\}\} \quad (14)$$

För att beräkna t ex $3 + 2$ antar vi att det inte finns något naturligt tal w som har värdet av $3 + 2$:

$$\neg add(3, 2, w) \quad (15)$$

Sedan fogas (15) till (14) och på resultatet görs klausulresolution:

Klausulresolution som visar att antagandet (15) var felaktigt ...



Poängen är nu *inte* huvudresultatet av själva resolutionen – att $\{add(x, 0, x)\}, \{add(x, \ddot{O}ka(y), \ddot{O}ka(z)), \neg add(x, y, z)\}, \{\neg add(3, 2, w)\}$ är osatisfierbar.

Nej, en *sidoeffekt* av resolutionen är vad som intresserar oss. Nämligen att i samma stund som unifieringen driver fram den tomma klausulen, så blir w "beräknad":

$$w = \ddot{O}ka(\ddot{O}ka(3)) = 5.$$

...
och
att
 $w = 5$
är
ett värde
på summan.

✂ TEST 8.3

- a) Formulera följande utsagor som predikatlogiska formler.
 A: Alla som tror att logik är viktigt lär sig logik,
 B: Ingen som lär sig logik ljuger.
 C: Ingen som ljuger tror att logik är viktigt.
- b) Avgör medelst både semantisk tablå och klausulresolution huruvida
 $(A \wedge B) \rightarrow C$ är valid.

■ 8.6 Hornklausuler och Hornformler

Positiv atom =
atom som är
onegerad.

■ **Hornklausul** En klausul med *högst en* positiv atom kallas *Hornklausul* efter Alfred Horn som var den förste att intressera sig för sådana klausuler (1951).

Hornklausul
=
klausul med
högst en
positiv atom

$$\begin{aligned}\{p, \neg q_1, \neg q_2, \dots, \neg q_n\} &= (p \vee \neg q_1 \vee \neg q_2 \vee \dots \vee \neg q_n) \\ &= (p \vee \neg(q_1 \wedge q_2 \wedge \dots \wedge q_n)) \\ &= (p \leftarrow (q_1 \wedge q_2 \wedge \dots \wedge q_n))\end{aligned}$$

Notera ovan hur en *Hornklausul* kan skrivas med hjälp av "om"-konjunktivet.

EXEMPEL 8.31 Några (från EXEMPEL 8.30) välkända *Hornklausuler* ...

<i>Hornklausuler</i>	<i>atomer</i>
$\{\neg \text{add}(3, 2, w)\}$	<i>en negativ</i>
$\{\text{add}(x, 0, x)\}$	<i>en positiv</i>
$\{\text{add}(x, \text{Öka}(y), \text{Öka}(z)), \neg \text{add}(x, y, z)\}$	<i>en negativ</i> <i>och en positiv</i>

■ **Hornformel** En klausulmängd vars alla klausuler är *Hornklausuler* kallas för en *Hornformel*.

En *Hornformel*

$$\begin{aligned}&\{\{p_1, \neg q_{11}, \dots, \neg q_{1n_1}\}, \dots, \{p_m, \neg q_{m1}, \dots, \neg q_{mn_m}\}\} \\ &= (p_1 \vee \neg q_{11} \vee \dots \vee \neg q_{1n_1}) \wedge \dots \wedge (p_m \vee \neg q_{m1} \vee \dots \vee \neg q_{mn_m}) \\ &= ((p_1 \leftarrow (q_{11} \wedge \dots \wedge q_{1n_1}))) \wedge \dots \wedge (p_m \leftarrow (q_{m1} \wedge \dots \wedge q_{mn_m}))\end{aligned}$$

EXEMPEL 8.32 Med Hornklausulerna i exemplet ovan kan vi bygga ...

<i>Hornformler</i>	<i>antal</i> <i>Hornklausuler</i>
$\{\{\text{add}(x, 0, x)\}\}$	<i>en</i>

Hornformler	antal Hornklausuler
$\{\{add(x, 0, x)\},$ $\{add(x, Öka(y), Öka(z)), \neg add(x, y, z)\}\}$	<i>två</i>
$\{\{\neg add(3, 2, w)\},$ $\{add(x, 0, x)\},$ $\{add(x, Öka(y), Öka(z)), \neg add(x, y, z)\}\}$	<i>tre</i>

Ett intressant faktum är att resolution i 1:a ordningens predikatlogik kan göras *mer effektiv* med Hornformler än med godtyckliga form-
ler.

Ett annat intressant faktum är att Hornformler har tillräckligt stark uttryckskraft för att kunna simulera en *universell Turingmaskin* (Sten-Åke Tärnlund, *Horn Clause Computability*[†], BIT 1977). Eller, vilket är liktydigt, att varje *rekursiv funktion* (se nedan) kan beskrivas av en Hornformel.

8.7 Rekursiva funktioner och predikatlogik

■ **Till varje funktion hör ett predikat** Till varje funktion $F(\bar{x})$ med argument $\bar{x} = x_1, \dots, x_n$ kan man som nedan bilda ett *predikat* $f(\bar{x}, z)$ som (jämfört med funktionen) har ett extra argument:

$$f(\bar{x}, z) = \begin{cases} \text{sant} & \text{om } F(\bar{x}) = z \\ \text{falskt} & \text{annars} \end{cases} \quad (16)$$

Predikatets argument z fungerar som "behållare" för funktionens output. Notera att funktionen F och predikatet f båda *beskriver en viss egenskap* som $n + 1$ objekt x_1, \dots, x_n, z kan ha. Predikatet f gör detta på ett *direkt* och *naturligt* sätt genom att helt enkelt returnera *sant* eller *falskt* som svar på frågan huruvida x_1, \dots, x_n, z har den aktuella egenskapen. Funktionen F å andra sidan (vars input är x_1, \dots, x_n) returnerar z endast om x_1, \dots, x_n, z har den aktuella egenskapen. Dvs funktionen måste, för att reda ut huruvida $n + 1$ objekt har egenskapen ifråga, behandla de n första objekten som input och det sista som output.

†. Tärnlund bevisar dessutom att det räcker att nyttja *Hornklausuler* med högst en negativ atom. Vidare att högst en funktionssymbol behöver användas.

EXEMPEL 8.33 Som en illustration, se på den rekursiva funktionen $Add(x_1, x_2)$, vars tillhörande predikat är

$$add(x_1, x_2, z) = \begin{cases} \text{sant} & \text{om } Add(x_1, x_2) = z \\ \text{falskt} & \text{annars} \end{cases} \quad (17)$$

Predikatet $add(x_1, x_2, z)$ har alltså värdet *sant* om och endast om x_1, x_2, z har egenskapen

$$x_1 + x_2 = z \quad (18)$$

Lägg märke till att (18) är ekvivalent med $x_1 = z - x_2$ men även med $x_2 = z - x_1$. Dvs predikatet $add(x_1, x_2, z)$ kan svara på frågor både om addition och subtraktion.

Notera också att (17) gäller för *alla* x_1, x_2, z , dvs predikatet är *naturligt allkvantifierat* över \mathbb{N} (fastän vi inte har brytt oss om att sätta ut någon allkvantor).

■ Till varje rekursiv funktion hör en Hornformel

Av nedanstående punkter framgår det att *Hornformler* tillsammans med basfunktionen *Öka* bildar en tillräckligt kraftfull delmängd av 1:a ordningens predikatlogik för att beskriva klassen av rekursiva funktioner.

Sammansättning När man i de rekursiva funktionernas värld definierar en *sammansatt funktion*

$$F(\vec{x}) = G(H_1(\vec{x}), \dots, H_m(\vec{x})) \quad (19)$$

menar man att

Funktionen F returnerar z för givet input om H -funktionerna returnerar y :na för samma input, och G returnerar z för y :na som input.

$$\begin{aligned} F(\vec{x}) = z \quad &\text{om} \\ &H_1(\vec{x}) = y_1 \\ &\text{och} \dots \\ &\text{och } H_m(\vec{x}) = y_m \\ &\text{och } G(y_1, \dots, y_m) = z \end{aligned}$$

något som beskrivs av Hornformeln

$$f(\vec{x}, z) \leftarrow (h_1(\vec{x}, y_1) \wedge \dots \wedge h_m(\vec{x}, y_m) \wedge g(y_1, \dots, y_m, z)) \quad (20)$$

där f, g, h_1, \dots, h_m är predikaten hörande till F, G, H_1, \dots, H_m som i (16).

Notera att (20) är en Hornformel (se sid 215) innehållande *en* enda Hornklausul.

EXEMPEL 8.34 Till de två sammansatta funktionerna

Två
sammansatta
funktioner

$$F(x_1, x_2) = \text{TriangelTal}(\text{Add}(x_1, x_2))$$

$$\text{Noll?}(x) = \text{Sub}(\text{Ett}(x), x)^{\dagger}$$

hör Hornformlerna

beskrivna av
varsin
Hornformel.

$$f(x_1, x_2, z) \leftarrow (\text{add}(x_1, x_2, y) \wedge \text{triangelTal}(y, z))$$

$$\text{noll}(x, z) \leftarrow (\text{ett}(x, y) \wedge \text{sub}(y, x, z))$$

om $\text{add}(x_1, x_2, y)$, $\text{triangelTal}(y, z)$, $\text{ett}(x, y)$ och $\text{sub}(y, x, z)$ är predikat som beskriver att

$$\text{Add}(x_1, x_2) = y, \text{TriangelTal}(y) = z, \text{Ett}(x) = y, \text{Sub}(y, x) = z.$$

⌘ Anm.8.8 Vår konstruktion i ovanstående exempel, av en Hornformel till funktionen $\text{Noll?}(x)$, syftade till att illustrera formel (20). När det gäller att Hornformelbeskriva en funktion som enbart returnerar 0 eller 1 är det dock i allmänhet enklare (och naturligare) att göra en "direkt" konstruktion (utan att använda ett extra argument för funktionens output). Dvs att konstruera en Hornformel som har lika många argument som funktionen (i detta fall *ett* argument) och som är sann om och endast om funktionen returnerar 1. Text kan funktionen

$$\text{Noll?}(x)$$

representeras av Hornformeln

$$\text{noll}(0)$$

som lämpligen uttalas "det är sant att 0 är noll".

Primitiv rekursion Varje funktion tillverkad medelst den *primitivt rekursiva mallen*

$$\begin{cases} F(0) = b \\ F(\text{Öka}(x)) = G(x, F(x)) \end{cases}$$

kan beskrivas av en Hornformel innehållande två Hornklausuler

$$f(0, b) \wedge (f(\text{Öka}(x), z) \leftarrow (g(x, y, z) \wedge f(x, y))) \quad (21)$$

EXEMPEL 8.35 De primitivt rekursiva funktionerna *Add* och *Mult*

$$\begin{cases} \text{Add}(x, 0) = x \\ \text{Add}(\text{Öka}(x), y) = \text{Öka}(\text{Add}(x, y)) \end{cases}$$

[†]. Noll? är en funktion som returnerar 0 eller 1.

$$\begin{cases} Mult(x, 0) = 0 \\ Mult(\ddot{O}ka(x), y) = Add(y, Mult(x, y)) \end{cases}$$

beskrivs som i (21) av Hornformlerna

$$add(x, 0, x) \wedge (add(\ddot{O}ka(x), y, \ddot{O}ka(z)) \leftarrow add(x, y, z))$$

$$mult(x, 0, 0) \wedge (mult(\ddot{O}ka(x), y, u) \leftarrow (mult(x, y, z) \wedge add(y, z, u)))$$

Framåtrekursion Till varje beskrivning av en *framåtrekursiv funktion*

$$\begin{cases} F(\bar{x}) = H(\bar{x}, 0) \\ H(\bar{x}, y) = Om(P(\bar{x}, y), G(\bar{x}, y), H(\bar{x}, \ddot{O}ka(y))) \end{cases}$$

hör en Hornformel med tre Hornklausuler:

$$\begin{aligned} f(\bar{x}, z) &\leftarrow h(\bar{x}, 0, z) \\ &\wedge (h(\bar{x}, y, z) \leftarrow (p(\bar{x}, y, \ddot{O}ka(u)) \wedge g(\bar{x}, y, z))) \\ &\wedge (h(\bar{x}, y, z) \leftarrow (p(\bar{x}, y, 0) \wedge h(\bar{x}, \ddot{O}ka(y), z))) \end{aligned} \quad (22)$$

Lägg märke till att i (22) härmas *Om*-funktionens beräkningsförfarande med hjälp av uppdelningen av predikatet *h* i två klausuler, varav den övre tar hand om fallet med nollskilt första argument hos *Om*-funktionen.

8.8 Övningar

8.1 Avgör såväl med semantisk tablå som med klausulresolution huruvida följande satslogiska formel är satisfierbar eller ej.

$$\neg((((p \rightarrow q) \rightarrow r) \wedge (p \rightarrow (\neg q \rightarrow q))) \rightarrow r)$$

8.2 Konstruera en Hornformel

a) till var och en av de primitivt rekursiva funktionerna

Noll, *Ett*, *Sub*

b) till den framåtrekursiva funktionen *Kvot*

c) som beskriver att $x + y + z = w$, och visa sedan med

klausulresolution hur $2 + 3 + 4$ kan beräknas med Hornformeln
hjälp.

8.3 Den *ickerekursiva* Hornformel

$$rest(x, y, r) \leftarrow (mindre(r, y) \wedge mult(y, z, u) \wedge add(u, r, x)) \quad (23)$$

uttrycker att

för (alla) naturliga tal x, y, q, r gäller att

$$\text{om } r < y \text{ och } x = y \cdot q + r$$

så är r resten vid divisionen x med y .

Visa med resolution hur Hornformel (23) kan användas för att beräkna resten vid divisionen 3 med 2.

8.4

- a) Formulera följande utsagor som predikatlogiska formler över ett individuniversum bestående av alla människor.

A : Frisörer klipper alla människor som inte klipper sig själva.

B : Ingen frisör klipper någon människa som klipper sig själv.

C : Det finns ingen frisör.

- b) Visa medelst semantisk tablå eller klausulresolution att $(A \wedge B) \rightarrow C$ är valid om A, B och C är som i a).

8.5

- a) Presentera en Hornformel för följande meningar.

En drake är glad om alla dess barn kan flyga.

Gröna drakar kan flyga.

En drake är grön om den är barn till en grön drake.

- b) Visa medelst semantisk tablå eller klausulresolution att *Gröna drakar är glada.*

8.6 Visa med hjälp av resolution att varje predikat p med egenskaperna A, B, C nedan har egenskapen D . Dvs visa att $(A \wedge B \wedge C) \rightarrow D$ är valid.

$$A: \forall x \forall y (p(x, y) \rightarrow p(y, x))$$

$$B: \forall x \forall y \forall z ((p(x, y) \wedge p(y, z)) \rightarrow p(x, z))$$

$$C: \forall x \exists y p(x, y)$$

$$D: \forall x p(x, x)$$

Lösningsförslag

1

TEST 1.1 a) $|\varepsilon| = 0, |wa| = |w| + 1$

b) Induktionsbevis av *associativa* lagen

$$(uv)w = u(vw) \quad (1)$$

BASFALLET: (1) stämmer om $w = \varepsilon$, ty då reduceras (1) till $(uv) = u(v)$ dvs $uv = uv$

INDUKTIONSSTEGET: (1) stämmer för wa om (1) stämmer för w , ty

$$\begin{aligned} (uv)(wa) &= ((uv)w)a && \text{rekursiva def av strängsammanfogning} \\ &= (u(vw))a && \text{om (1) stämmer för } w \text{ (induktionsantagandet)} \\ &= u((vw)a) && \text{rekursiva def av strängsammanfogning} \\ &= u(v(wa)) && \text{rekursiva def av strängsammanfogning} \end{aligned}$$

c) $\{\varepsilon\}$ **d)** Strängarna som ser lika ut oavsett om de läses framifrån eller bakifrån (palindromer).

e) Palindromerna av jämn längd.

f) $\{xxx|x \in \Sigma^*\}$. Detta inses t ex på följande sätt:

Av $ww = uuu$ följer (mät längden av strängarna i vänster- och högerled) att $2|w| = 3|u|$. Av detta följer i sin tur att $|u|$ är delbart med 2 eftersom 3 inte är det. Och om $|u|$ är delbar med 2, måste u vara en sammanfogning $u = xy$ av två lika långa strängar x och y . Insätts nu $u = xy$ i $ww = uuu$ fås $ww = xyxyxy$, och eftersom x och y är lika långa måste även xyx och yxy vara lika långa vilket visar att

$$w = xyx \text{ och } w = yxy. \quad (1)$$

Av (1) i kombination med att x och y är lika långa följer att $x = y$, och därmed att $w = xxx$.

Omvänt, om $w = xxx$ så är $ww = xxxxxx$, dvs (sätt $u = xx$) $ww = uuu$.

ÖVN 1.1 En rekursiv definition:

BASFALLET ε, a, b är palindromer över Σ

REKURSIONSTEGET Om w är en palindrom över Σ så är awa och bwb palindromer över Σ . (Inga andra strängar är palindromsträngar.)

ÖVN 1.2 a) Det stämmer onekligen för $n = 1$. Och för $n > 1$ stämmer det också. Ty har man inga strängar att sammanfoga, så får man inga sammanfogade strängar.

b) Följer av $\emptyset^* = \emptyset^0 \cup \emptyset^1 \cup \emptyset^2 \cup \dots$ i kombination med a) och att $\emptyset^0 = \{\varepsilon\}$.

ÖVN 1.3 $Prefix(L) = \{\varepsilon, i, in, u, ut\}$, $Suffix(L) = \{\varepsilon, n, in, t, ut\}$ för
 $L = \{in, ut\}$. Och för $L = \{1\}\{0\}^*$ är $Prefix(L) = L \cup \{\varepsilon\}$,
 $Suffix(L) = L \cup \{0\}^*$

ÖVN 1.4 Tex $\{ab, aab, aaab, \dots\}$ eller $\{ab, aabb, aaabbb, \dots\}$.

ÖVN 1.5 (7) följer av att en sträng i $L \cup M$ ligger i L eller i M , och att därmed ett suffix till en sådan sträng är ett suffix till en sträng i L eller till en i M .

(8) följer av att en sträng i L^* endera är tom eller (om L är icke-tom) av typ $w_1 \dots w_n$ där varje $w_i \in L$. Ett suffix y till $w_1 \dots w_n$ har i sin högra ände noll eller flera hela strängar $w_p \dots w_n$, och i sin vänstra ände ett suffix till w_{p-1} .

2

TEST 2.1 a) $abab, abba, baab, baba$ respektive $aaaa, aabb, bbaa, bbbb$.

b) En enstaka sträng w är en sammanfogning av noll eller flera (ändligt många) tecken:

$$w = \varepsilon \text{ (noll tecken) eller}$$

$$w = a_1 \dots a_k, \text{ där } k > 0 \text{ och } a_i \in \Sigma$$

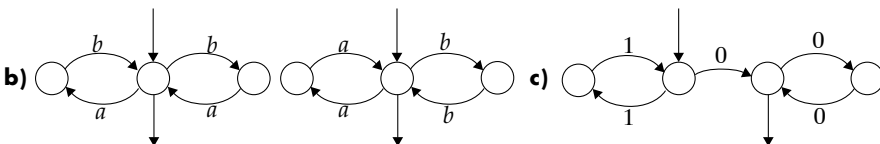
vilket innebär att $\{w\} = \{\varepsilon\}$ eller $\{w\} = \{a_1\} \dots \{a_k\}$, där $a_i \in \Sigma$,
 $\{\varepsilon\}$ är definitivt reguljärt (EXEMPEL 2.1).

$\{a_1\} \dots \{a_k\}$ då? Här har vi en sammanfogning av k st reguljära språk (ty för varje tecken $\sigma \in \Sigma$ är ju $\{\sigma\}$ ett reguljärt språk). Och då språksammanfogning är en reguljär operation, följer (efter $k-1$ sådana operationer) att $\{a_1\} \dots \{a_k\}$ är reguljärt.

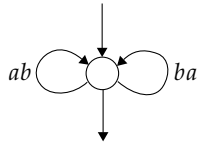
c) Ett ändligt språk är endera tomt (och då är det reguljärt), eller en ändlig union
 $L = \{w_1, w_2, \dots, w_k\} = \{w_1\} \cup \{w_2\} \cup \dots \cup \{w_k\}$ av k st ensträngsspråk vilka
 vart och ett är reguljärt enligt TEST 2.1 b). Då unionsbildning är en reguljär operation följer (efter $k-1$ unionsoperationer) att L är reguljärt.

d) $(11)^*0(00)^*$ **e)** $(0 \cup 1 \cup 2)^*0$ (eller $0 \cup (1 \cup 2)(0 \cup 1 \cup 2)^*0$ om vi inte tillåter nollskilda tal att börja på 0). **f)** $(0 \cup 1)^*00$ (eller $0 \cup 1(0 \cup 1)^*00$).

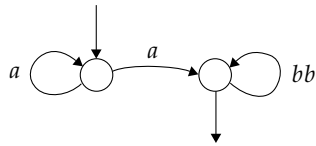
TEST 2.2 a) $m \cdot n$



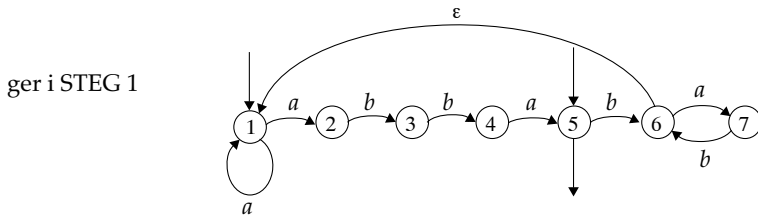
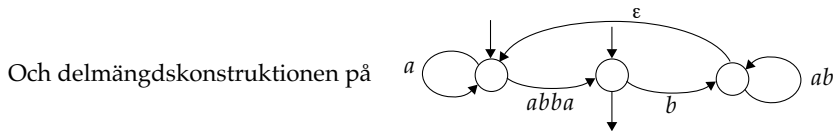
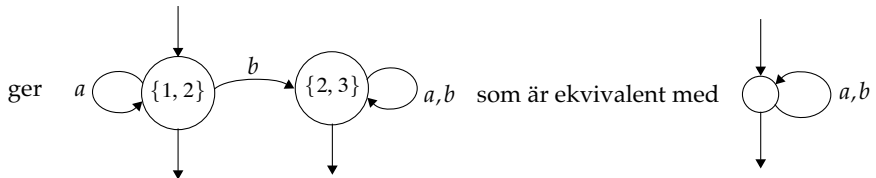
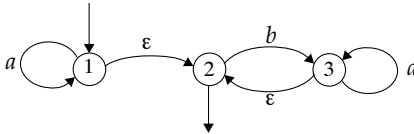
TEST 2.3 a)



b)

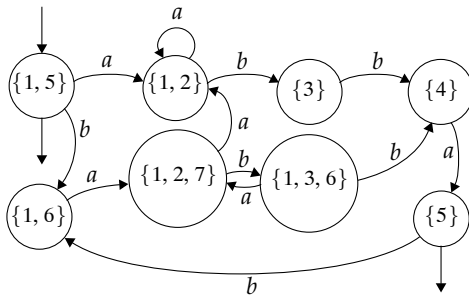


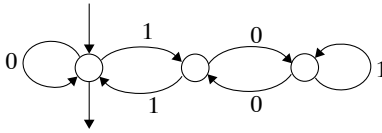
TEST 2.4 a) Delmängdskonstruktionen på



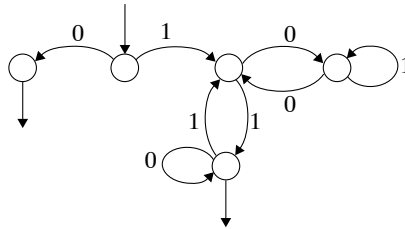
STEG 2

skräptillstånd
ej utsatt



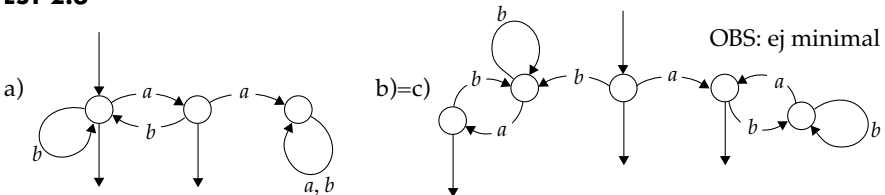
TEST 2.4 b)

eller (om inledande nollor inte tillåts i nollskilda tal)

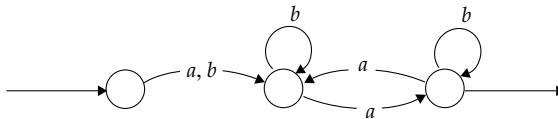
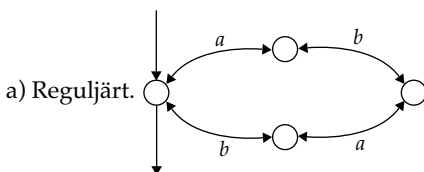


TEST 2.5 a) $0^*1(00^+1 \cup 1)^*$ b) $(0 \cup 1^+00)^*(\varepsilon \cup 1^+0 \cup 1^+01(0 \cup 1)^*)^*$

c) $(0 \cup 11 \cup 10(1 \cup 00)^*01)^*$. (Detta uttryck tillåter inledande 0:or.)

TEST 2.6

TEST 2.7 Den minimala DFA:n ser ut så här:

**TEST 2.8**

b) Inte reguljärt. Detta bevisas t ex av att $a^n b^n$ inte kan pumpas någonstans bland de n inledande tecknen utan att falla ur språket.

c) Reguljärt. $(ab \cup ba)^*(a \cup b \cup \varepsilon)$. KOMMENTAR: Eftersom *varje* prefix av jämn längd skall tillhöra L_2 , gäller det att bekanta sig med strängarna i L_2 . Här är några

smakprov: $\epsilon, ab, ba, abba, abab, baba, baab, abbaab, \dots$ Som du förstår måste nu strängarna i L_3 inledas på dessa sätt, något som leder till att deras uppbyggnad från vänster måste göras med strängarna ab, ba . Och som avslutning efter ett antal sammanfogningar av ab, ba skall strängarna ha a eller b om de har udda längd.

d) Reguljärt. $L_4 = a\Sigma^+a \cup b\Sigma^+b$, ty varje sträng i $a\Sigma^+a \cup b\Sigma^+b$ kan skrivas xyx^{rev} med $x, y \in \Sigma^+$. (Låt x vara a respektive b !). Och omvänt gäller det att varje sträng i L_4 kan skrivas som aza eller bzb med $z \in \Sigma^+$.

e) Inte reguljärt. Ty betrakta $aba^Kaa^Kbba^Kaa^Kb$ som ligger i L_5 (med $x = ba^Kaa^Kb$). Pumpning av ett eller flera a :n i det vänstra stora a -blocket leder ut ur L_5 . (Notera att $aba^Kaa^Kbba^Kaa^Kb$, som också ligger i L_5 kan pumpas i det vänstra stora a -blocket utan att falla ur L_5 , eftersom den alltid – även efter sådan pumpning – har ett icke tomt suffix av typ xx^{rev} , nämligen ba^Kaa^Kb .)

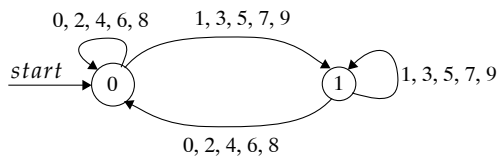
f) Reguljärt. Ty L_6 ges av

$$(1^2 \cup 1^3 \cup 1^5 \cup 1^7 \cup \dots)^* = \epsilon \cup (1^2 \cup 1^3 \cup 1^5 \cup 1^7 \cup \dots)^+ \\ = \epsilon \cup 1^2(\epsilon \cup 1 \cup 1^3 \cup 1^5 \cup \dots)^* = \epsilon \cup 1^2 1^*$$

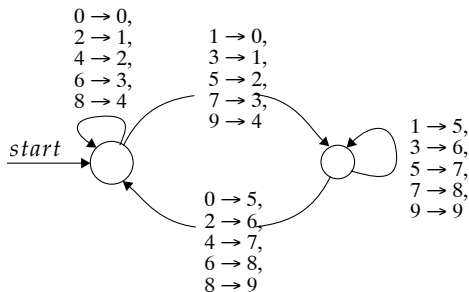
där det sista uttrycket onekligen är ett reguljärt uttryck.

(Den sista likheten motiveras t ex så här: $(\epsilon \cup 1 \cup 1^3 \cup 1^5 \cup \dots)^* \supseteq 1^*$ eftersom $\epsilon \cup 1 \cup 1^3 \cup 1^5 \cup \dots \supseteq 1$, och $(\epsilon \cup 1 \cup 1^3 \cup 1^5 \cup \dots)^* \supseteq 1^*$ eftersom 1^* beskriver det största språket över $\{1\}$.)

TEST 2.9 a)



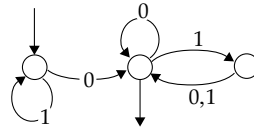
b)



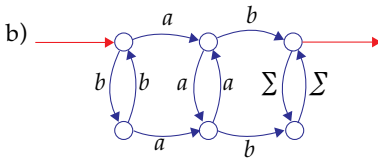
ÖVN 2.1 $dd^* \cup ((dd^* \cup \epsilon).dd^*)$ **ÖVN 2.2 a)** $(0 \cup 1)^*0(11)^*$ resp.

$(0 \cup 1)^*0((0 \cup 1)0 \cup 11)^*$

ÖVN 2.2 b) M_1 och M_2 är båda ekvivalenta med

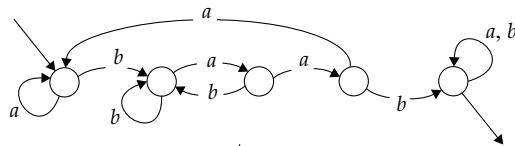


ÖVN 2.3 a) $(\Sigma\Sigma)^*ab(\Sigma\Sigma)^* \cup (\Sigma\Sigma)^*\Sigma ab(\Sigma\Sigma)^*\Sigma$



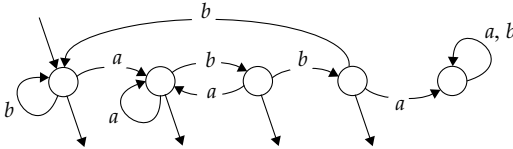
ÖVN 2.4 16.

ÖVN 2.5 a) Att DFA:n bredvid (med fem tillstånd) är minimal följer av tabellen nedan, som visar att det aktuella språket särskiljer fem strängar. (För varje par $\{x, y\}$ av de fem strängarna $\epsilon, b, ba, baa, baab$ visas i tabellen ett z sådant att exakt en av xz och yz innehåller $baab$.)



		a a b			
	b	a b	a b		
y	ba				
	baa	b	b	b	
	baab	ϵ	ϵ	ϵ	ϵ
		ϵ	b	ba	baa
		x			

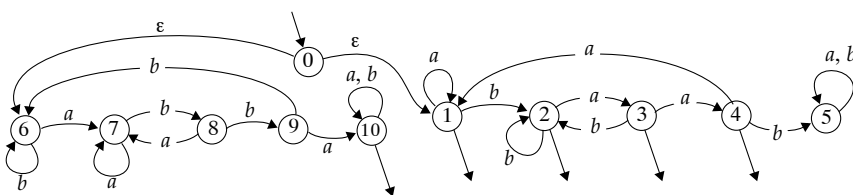
b) Minimal DFA för strängarna som *inte* innehåller $abba$:



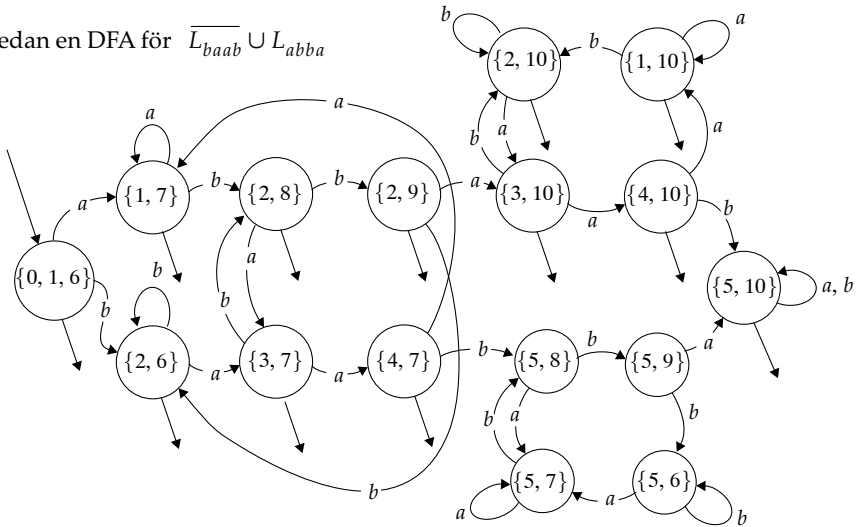
Minimaliteten följer av att DFA:n i a) är minimal.

c) Minimal DFA för strängarna som innehåller $baab$ men (och) inte $abba$:

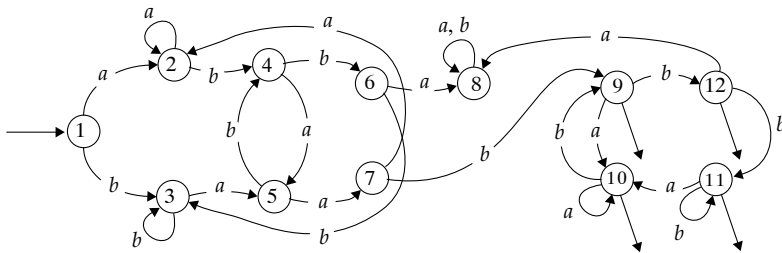
Först en NFA för $\overline{L_{baab}} \cup L_{abba}$



Sedan en DFA för $\overline{L_{baab}} \cup L_{abba}$



Slutligen en DFA för $\overline{\overline{L_{baab}} \cup L_{abba}} = L_{baab} \cap \overline{L_{abba}}$. Med sina tolv tillstånd är den minimal, vilket framgår av särskiljandetabellen nedan, som visar att tolv strängar $\epsilon, a, b, ab, ba, abb, baa, abba, baab, baabb, baabb, baabbb$ särskiljs av språket ifråga.

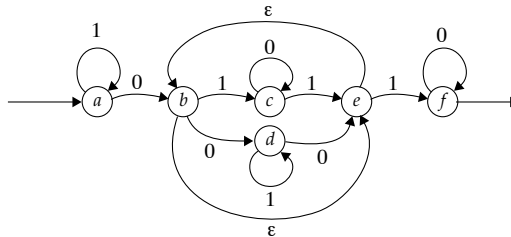


a	bbaab												
b	aab	aab											
ab	aab	aab	aab										
ba	ab	ab	ab	ab									
abb	abaab	abaab	aab	aab	ab								
baa	b	b	b	b	b	b							
abba	baab	baab	aab	aab	ab	baab	b						
baab	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ					
baabb	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	a				
baaba	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ba	a			
baabbb	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ba	a	bba		
	ϵ	a	b	ab	ba	abb	baa	abba	baab	baabb	baaba		

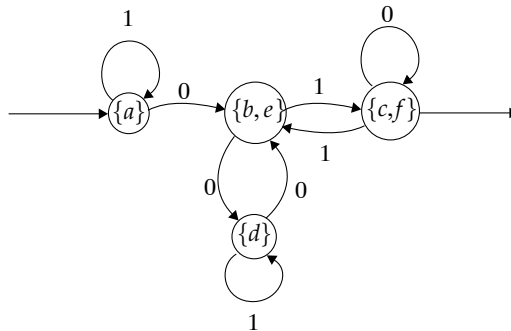
d) Den minimala DFA:n har 7 tillstånd.

ÖVN 2.6 a)  b) $(a \cup b)^*a$ eller $(b^*a)^+$ eller

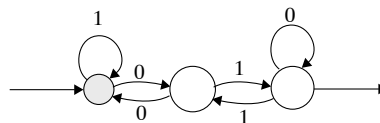
ÖVN 2.7 Först en NFA:



som kan omformas till en DFA (delmängdskonstruktionen!):



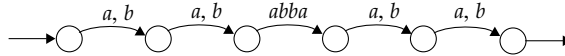
Ovanstående DFA är emellertid inte minimal. Man ser att två tillstånd kan slås ihop (eller hur!).



ÖVN 2.8 a) Palindromspråket $L_1 = \{w \mid w \text{ är en palindrom}\}$ är icke reguljärt. Ty om N är som i pumpsatsen, skulle t ex palindromen $w = a^N b a^N$ kunna pumpas med pumpdelen bland de N första tecknen och fortfarande vara en palindrom. Men å andra sidan leder ju sådan pumpning till att w får flera eller färre a :n i början än i slutet, dvs w faller ur palindromspråket.

b) $L_2 = \{w \mid w \text{ är inte en palindrom}\}$ är inte heller reguljärt. Ty då L_1 är komplementspråket till L_2 och komplementbildning bevarar reguljäritet, skulle i annat fall även L_1 vara reguljärt.

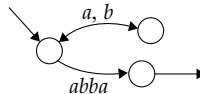
- c) $L_3 = \{uabbbav \mid u, v \in \{a, b\}^* \text{ och } |u| = |v| = 2\}$ är reguljärt eftersom språket innehåller ändligt många strängar. Här är f.ö. en NFA för språket



- d) $L_4 = \{uabbbav \mid u, v \in \{a, b\}^* \text{ och } |u| = |v|\}$ är *icke* reguljärt.

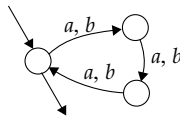
Ty om N är som i pumpsatsen skulle t ex $w = a^N abba a^N$ kunna pumpas med pumpdelen bland de N första tecknen utan att falla ur L_4 . Men sådan pumpning leder ju till att pumpade w får en längre eller (vid urpumpning) kortare sträng före $abba$ än efter. Dvs w skulle falla ur L_4 .

- e) $L_5 = \{uvabba \mid u, v \in \{a, b\}^* \text{ och } |u| = |v|\}$ är reguljärt, ty
 $L_5 = \{xabba \mid x \in \{a, b\}^* \text{ och } |x| \text{ är jämn}\}$ och accepteras av



- f) $L_6 = \{uvvw \mid u, v, w \in \{a, b\}^* \text{ och } |u| = |v| = |w|\}$ är reguljärt, ty

$L_6 = \{x \in \{a, b\}^* \mid |x| \text{ är delbar med } 3\}$ och accepteras av



ÖVN 2.9 a) *ej* reguljärt b) reguljärt c) reguljärt

ÖVN 2.10 LEDNING: Se på ett reguljärt uttryck med N stycken vänsterparenteser.

ÖVN 2.11 a) $L_1 = \{a \cup b\}^*$, $L_2 = \{a^m b^n \mid m > n\}$ Här är L_2 är icke-reguljär och

$L_1 \cup L_2 = L_1 = \{a \cup b\}^*$. b) $L_1 = \{a^m \mid m > 0\}$, $L_2 = \{a^m b^n \mid m > n\}$. Här är

$L_1 \cap L_2 = L_1 = a^+$ (dvs oändliga) och L_2 är icke-reguljär.

ÖVN 2.12 $x = (ac)^m$ och $y = (ac)^n$ särskiljs av $z = (bc)^m$.

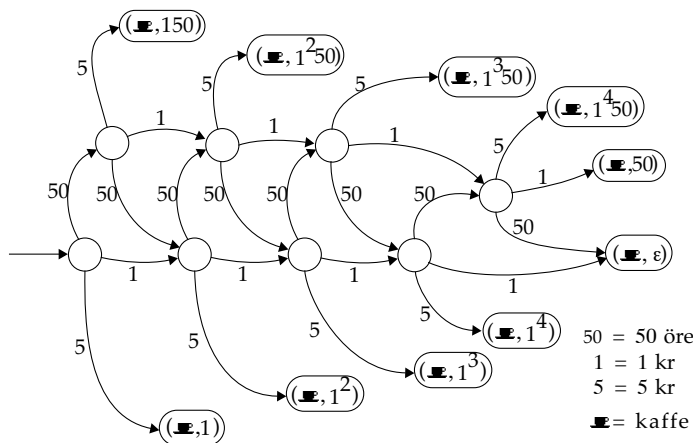
ÖVN 2.13 L beskrivs av $\Sigma^* aa \Sigma^* \cup \Sigma^* bb \Sigma^* \cup (ab)^* \cup (ba)^*$.

ÖVN 2.14 a) Falskt. Se t ex på $\{11, 111, 11111, 1^7, \dots\} \subset \{1\}^*$, där den vänstra mängden är ett välkänt ickereguljärt språk. (Se sid 62.)

b) Falskt. Se t ex på $\{11\} \cup \{111\} \cup \{11111\} \cup \{1^7\} \cup \dots$.

c) Falskt. Se t ex på $\{\epsilon\} \cup \{1\} \cup \{11\} \cup \{111\} \cup \dots$, dvs $\{1\}^*$.

ÖVN 2.16 b)



ÖVN 2.17 Här är ett program skrivet i pseudokod

```
Sätt resten till 0
Sätt kvoten till tom sträng
Så länge teckenströmmen pågår
{
    sätt resten till tillstånd(resten, tecken)
    foga kvottecken(rest, tecken) till kvoten
}
```

Programmet använder två arrayer:

		tecken									
rest	tillstånd	0	1	2	3	4	5	6	7	8	9
	0	0	1	2	0	1	2	0	1	2	0
	1	1	1	2	0	1	2	0	1	2	0
	2	2	2	0	1	2	0	1	2	0	1

		tecken									
rest	kvottecken	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	1	1	1	2	2	3
	1	3	3	4	4	4	5	5	5	6	6
	2	6	7	7	7	8	8	8	9	9	9

3

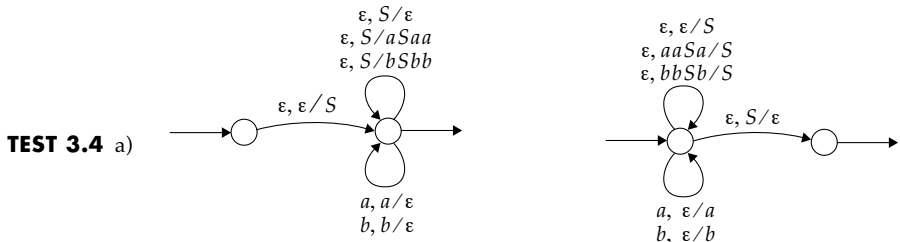
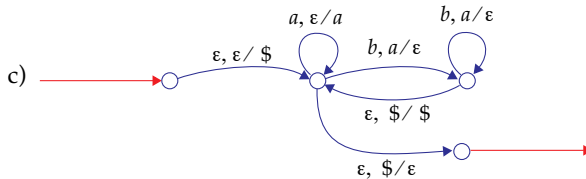
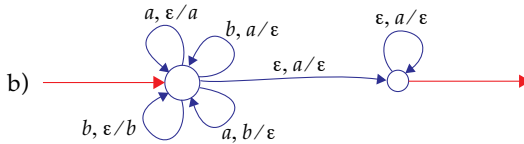
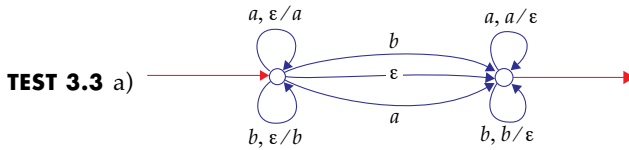
TEST 3.1 $S \rightarrow aS|bbB$
 $B \rightarrow bB|\epsilon$

TEST 3.2

a) $\begin{cases} S \rightarrow \varepsilon | SA \\ A \rightarrow \varepsilon | aAb \end{cases}$ b) $\begin{cases} S \rightarrow aS | aS' | bSS \\ S' \rightarrow \varepsilon | aS'b | bS'a | S'S' \end{cases}$ eller $S \rightarrow a | aS | SSb | SbS | bSS$

c) $\begin{cases} S \rightarrow A | B \\ A \rightarrow a | aA | AAb | AbA | bAA \\ B \rightarrow b | bB | BBa | BaB | aBB \end{cases}$ A representerar strängar med a -överskott, och B strängar med b -överskott.

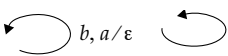
d) $S \rightarrow \varepsilon | aSa | bSb$ e) $S \rightarrow \varepsilon | a | b | aSa | bSb$ f) $S \rightarrow \varepsilon | -1S + 1 | +1S - 1 | SS$



TEST 3.5 Om det inledande målet "att driva M från s till s med oförändrad stack" betecknas med M^ε , blir vår första produktionsregel $S \rightarrow M^\varepsilon$.

De pushande övergångarna $\begin{matrix} a, \varepsilon/a \\ \curvearrowright \end{matrix}$, $\begin{matrix} \curvearrowright \\ b, \varepsilon/b \end{matrix}$ ger $M^\varepsilon \rightarrow aM^a$ och $M^a \rightarrow aM^aM^a$

respektive $M^\varepsilon \rightarrow bM^b$ och $M^b \rightarrow bM^bM^b$. Och de poppande övergångarna



ger $M^a \rightarrow bM^\varepsilon$ respektive $M^b \rightarrow aM^\varepsilon$.

Slutligen, det kravlösa målet M^ε kan uppfyllas utan att något tecken konsumeras: $M^\varepsilon \rightarrow \varepsilon$

FÖRENKLINGAR: Eftersom S inte producerar något annat än M^ε , kan vi sätta M^ε till S . Om vi dessutom betecknar M^a med B och M^b med A får vi

$$S \rightarrow \varepsilon \mid aB \mid bA$$

$$A \rightarrow bAA \mid aS$$

$$B \rightarrow aBB \mid bS$$

TEST 3.6 a) $a^{K+2}b^{K+1}c^K$ kan *inte* pumpas någonstans med ett pumpblock av längd mindre eller lika med K (utan att man faller ur språket). Ty ett sådant (kort) pumpblock kan inte innehålla både a och c . Och om pumpblocket inte innehåller något a , så leder *upp-pumpning* till en sträng med underskott av $a:n$, medan pumpblocket som inte innehåller något c , vid *urpumpning* leder till en sträng med underskott av $a:n$ eller $b:n$.

b) $v^Ku^Kh^Kn^K$ kan *inte* pumpas någonstans med ett pumpblock av längd mindre eller lika med K . (Något som den sammanhangsfria pumpsatsen utlovar ifall L vore sammanhangsfritt.) Pumpning i ett så kort pumpblock kan nämligen inte inbegripa både u och n , inte heller både v och h . Därmed ger sådan pumpning upphov till "obalans". Dvs till strängar utanför L .

ÖVN 3.1 a) En sträng i språket skall tydligen börja med en delsträng (kallad A) som är vilken binär sträng som helst (ty detta är vad A -produktionsreglerna ger oss). Därefter skall det komma 01. Och till sist en eller flera 1:or eller 01:or (detta är B). Här är en reguljär grammatik för strängarna:

$$S \rightarrow 0S \mid 1S \mid 01B, \quad B \rightarrow 1 \mid 01 \mid 01B \mid 1B$$

b) Reglerna $S \rightarrow 01 \mid 001 \mid AAS$, $A \rightarrow \varepsilon \mid 01 \mid 10$ kan endast producera enligt följande enkla mönster innan basfallen används:

$$S \Rightarrow AAS \Rightarrow AAAAS \Rightarrow AAAAAAS \Rightarrow \dots \Rightarrow \overbrace{A \dots A}^{2n \text{ st}} S \Rightarrow \dots$$

Eftersom upprepad användning av basfallet $A \rightarrow \varepsilon$ låter oss sudda ut hur många som helst av A :na, kan vi dock från S producera A^kS med k godtyckligt naturligt tal. Och när sedan basfallen används får vi som slutresultat en sträng som i sin vänstra ände sammanfogar noll eller flera strängar av typ 01 eller 10 (detta är vad A kan vara), och som avslutas i höger ände med 01 eller 001. Alltså kan reglerna ersättas med följande *reguljära regler*: $S \rightarrow 01 \mid 001 \mid 01S \mid 10S$.

ÖVN 3.2 a) $\begin{cases} S \rightarrow aSa \mid bSb \mid bAa \mid aAb \\ A \rightarrow \varepsilon \mid bA \mid aA \end{cases}$ b) $S \rightarrow aaSb \mid aSbb \mid \varepsilon$ c) $\begin{cases} S \rightarrow \varepsilon \mid aSa \mid B \\ B \rightarrow \varepsilon \mid bBa \end{cases}$

ÖVN 3.3

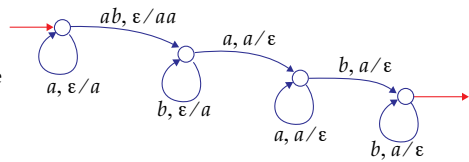
$$S \rightarrow aSb \mid aBb \mid aAb$$

$$B \rightarrow bBb \mid D$$

$$A \rightarrow aAa \mid D$$

$$D \rightarrow bDa \mid ba$$

respektive

**ÖVN 3.4** a) $S \rightarrow \epsilon \mid (S) \mid \{S\}$ b) $S \rightarrow (\mid (S) \mid 1S1$ **ÖVN 3.5** a) $S \rightarrow a \mid B' \mid aSa$, $B' \rightarrow b \mid B'b$

$$b) S \rightarrow a \mid b \mid B'B \mid AD, A \rightarrow a, D \rightarrow SA, B' \rightarrow b \mid B'B, B \rightarrow b$$

ÖVN 3.6 I den undre av de två givna syntaxdiagrammen beskrivs U som en sträng av *en* eller flera *d:n*. (Dvs U representerar datatypen *Int* om d representerar en godtycklig siffra.) I det övre syntaxdiagrammet (det som beskriver S) finns en inledande del



som beskriver strängar av typen $U.U$ (dvs datatypen *Float*). I samma diagram finns en avslutande del som beskriver strängar av typ $e+U$ eller $e-U$. Följande CFG är en mer eller mindre direkt översättning av de två diagrammen:

$$S \rightarrow \text{Int} \mid \text{Float} \mid \text{Int } e \text{ Sgn Int} \mid \text{Float } e \text{ Sgn Int}$$

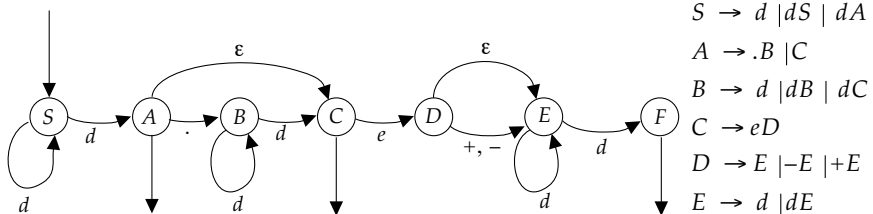
$$\text{Float} \rightarrow \text{Int}.\text{Int}$$

$$\text{Int} \rightarrow d \mid d \text{ Int}$$

$$\text{Sgn} \rightarrow \epsilon \mid + \mid -$$

Utgående från denna CFG kan man förstås tillverka en PDA i form av en top-down eller bottom-up parser.

Men ... i själva verket kan man beskriva strängarna med en *reguljär grammatik* och med en *finit automat* som nedan. Eftersom en finit automat är en PDA utan stackagerande, kan Du se den finita automaten som svaret på uppgiften att konstruera en PDA för strängarna ifråga:



$$S \rightarrow d \mid dS \mid dA$$

$$A \rightarrow \cdot B \mid C$$

$$B \rightarrow d \mid dB \mid dC$$

$$C \rightarrow eD$$

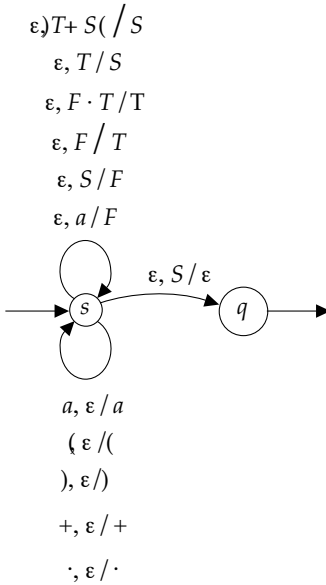
$$D \rightarrow E \mid -E \mid +E$$

$$E \rightarrow d \mid dE$$

ÖVN 3.7 a) $(S \rightarrow \epsilon \mid aS_1)$ och $S_1 \rightarrow \epsilon \mid aS_1 \mid aS_1b \mid S_1S_1$

ÖVN 3.8

a) bottom-up parser



b) provkörning

tillstånd	input	stacken
s	$(a + a \cdot a) \cdot a$	ϵ
\vdots	$a + a \cdot a) \cdot a$	$($
	$+ a \cdot a) \cdot a$	$a($
	$+ a \cdot a) \cdot a$	$F($
	$+ a \cdot a) \cdot a$	$T($
	$+ a \cdot a) \cdot a$	$S($
	$a \cdot a) \cdot a$	$+ S($
	$\cdot a) \cdot a$	$a + S($
	$\cdot a) \cdot a$	$F + S($
	$\cdot a) \cdot a$	$T + S($
	$a) \cdot a$	$\cdot T + S($
	$) \cdot a$	$a \cdot T + S($
	$) \cdot a$	$F \cdot T + S($
	$) \cdot a$	$T + S($
	$\cdot a$	$)T + S($
	$\cdot a$	S
	$\cdot a$	F
	$\cdot a$	T
	$\cdot a$	T
	a	$\cdot T$
	ϵ	$a \cdot T$
	ϵ	$F \cdot T$
\vdots	ϵ	T
s	ϵ	S
q	ϵ	ϵ

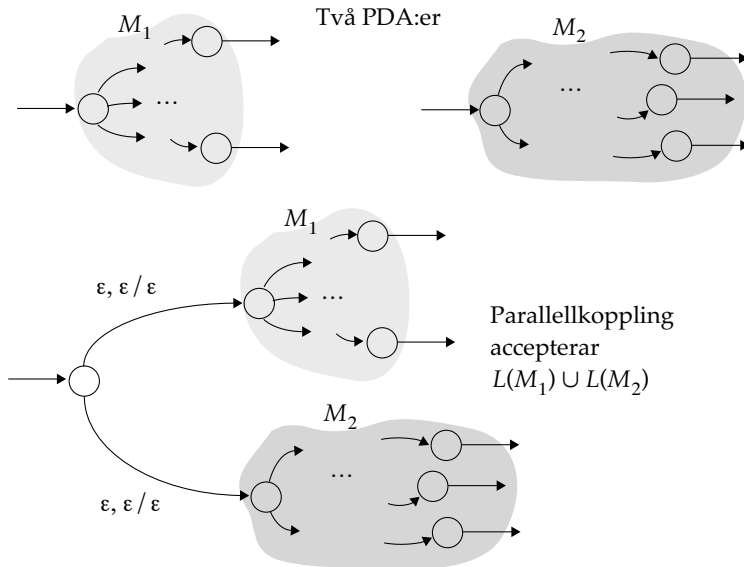
ÖVN 3.9

a) Bilda unionen av G_1 :s och G_2 :s regelmängder och tillfoga en ny start-symbol S samt en regel $S \rightarrow S_1 \mid S_2$, där S_1, S_2 antas vara startsymboler i G_1, G_2 . På detta sätt kan man sätta igång att producera strängar ur L_1 eller strängar ur L_2 , vilket är precis vad vi vill.

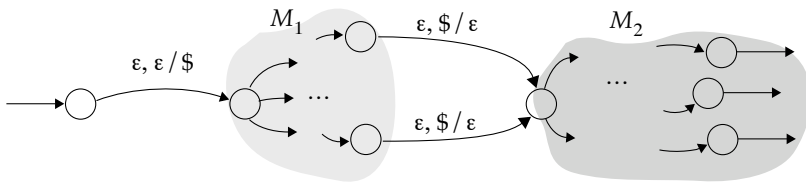
Observera dock att om G_1, G_2 råkar använda samma namn på vissa icketerminerande symboler måste man först korrigera för detta genom namnbyte, så att ingen olycklig sammanblandning uppstår. (Vi vill undvika möjligheten att hoppa mellan de två grammatikernas regler.)

b) Likadant som i a) så när som att den tillfogade regeln blir $S \rightarrow S_1 S_2$.

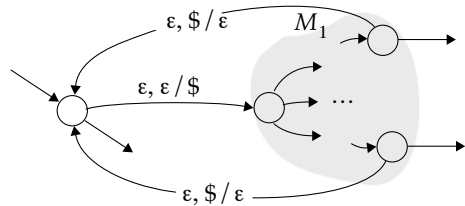
c) Tillfoga ny startsymbol S samt reglerna $S \rightarrow S S_1 \mid \epsilon$.

ÖVN 3.10

Seriekoppling accepterar $L(M_1)L(M_2)$, om man kan ordna så att inget skräp ligger på stacken vid själva seriekopplingen. Genom att inledningsvis placera en ändmarkering på stacken som tas bort i samband med seriekopplingen, garanteras skräpfri stack vid själva seriekopplingen. (Utan sådan garanti skulle vi kunna råka ut för att felaktiga strängar accepteras.):



Återkoppling med motsvarande ändmarkeringsarrangemang som det ovanför, ger oss en PDA för $(L(M_1))^*$



ÖVN 3.11 a) LEMMA Om ett språk L innehåller två strängar x och y med egenskapen att exakt en av xz , yz tillhör L för något z , så måste en DPDA för L drivas av x och y med tömd

stack till olika tillstånd.

BEVIS: *Antag* (tvärtom) att en DPDA för L driver x och y med tömd stack till ett gemensamt tillstånd (som då måste vara accepterande eftersom $x, y \in L$).

Men om DPDA:n har tömt sin stack när den står i det gemensamma tillståndet, så kan den inte minnas om den innan har konsumerat x eller y . DPDA:ns determinism gör då att fortsatt konsumtion av z (från det gemensamma tillståndet) driver DPDA:n på ett bestämt sätt oavsett om den har konsumerat x eller y innan, vilket medför att DPDA:n inte kan acceptera exakt en av xz och yz . Dvs DPDA:n kan inte vara en DPDA för L .

- b) SATS: Om ett språk L har egenskapen att det finns en oändlig mängd av strängar som parvis har egenskapen som x och y har i lemmat, så finns det ingen DPDA för L .

BEVIS: Först konstaterar vi att en DPDA inte kan särskilja flera strängar än den har tillstånd. Med andra ord, om den har N tillstånd måste av $N + 1$ strängar minst två stycken driva DPDA:n till ett gemensamt tillstånd. Alltså måste minst två av de oändligt många strängarna i satsen driva DPDA:n till ett gemensamt tillstånd. Men detta motsäger lemmat.

Alltså ...

- ÖVN 3.12** w innehåller lika många förekomster av $+1$ som av -1 , och lika många förekomster av $+x$ som av $-x$. Om språket vore sammanhangsfritt skulle det finnas något K så att varje sträng i språket av längd minst K skulle kunna pumpas någonstans med ett pumpblock av längd högst K . Dvs då skulle nedanstående sträng kunna pumpas.

$$\begin{aligned}
 & (+1)^K (+x)^K (-1)^K (-x)^K \\
 &= \underbrace{+1 + 1 + \dots + 1}_{K \text{ st}} \underbrace{+x + x + \dots + x}_{K \text{ st}} \underbrace{-1 - 1 - \dots - 1}_{K \text{ st}} \underbrace{-x - x - \dots - x}_{K \text{ st}}
 \end{aligned}$$

Men eftersom pumpning med ett pumpblock av längd högst K inte förmår påverka både ett $+x$ -block och ett $-x$ -block i ovanstående sträng, kan pumpdelarna inte innehålla $+x$ eller $-x$ utan att pumpningen leder ut ur språket. Detsamma gäller för $+1$ och -1 . Detta bevisar att språket *inte* är sammanhangsfritt.

OBSERVERA att $(+1)^K (-1)^K (+x)^K (-x)^K$ och $(+1)^K (+x)^K (-x)^K (-1)^K$ också ligger i språket och är av längd större än K . Men båda dessa kan pumpas med ett pumpblock av längd högst K utan att falla ur språket. Dvs dessa strängar kan *inte* användas för att bevisa brist på sammanhangsfrihet.

- ÖVN 3.13** a) Enligt pumpsatsen skall man (vid sammanhangsfrihet) kunna pumpa upp K eller färre 1:or i strängen 1^{K^2} , och få en ny sträng i språket. Men en sådan pumpning ger *inte* tillräckligt många nya 1:or. Nästa kvadrattal (efter K^2) är nämligen $(K+1)^2$. Och $1^{(K+1)^2}$ har $2K+1$ flera 1:or än 1^{K^2} .

- b) *Antag* att L är sammanhangsfritt. Enligt pumpsatsen för sammanhangsfria språk gäller då för varje $w \in L$ som är tillräckligt lång att w kan skrivas som $uvxyz$ där

$vy \neq \varepsilon$. Dvs $w = 1^i 1^j 1^k 1^l 1^m$, med $j+l \geq 1$. Och vidare:

$$1^i 1^{nj} 1^k 1^{nl} 1^m \in L \text{ för varje } n \in \mathbb{N},$$

dvs $i + nj + k + nl + m = i + k + m + n(j+l)$ är prima för varje $n \in \mathbb{N}$.

Att $i + k + m$ är prima innebär att $i + k + m \geq 2$.

Med andra ord, om p sättes till $i + k + m$ och q till $j + l$, har vi

$$p \geq 2$$

$$p + nq \text{ är prima för varje } n \in \mathbb{N}$$

Men $p + nq$ är inte prima för t ex $n = p$. Ty då är

$$p + nq = p + pq = p(1 + q) = \underbrace{(i + k + m)}_{\geq 2} \cdot \underbrace{(1 + (j + l))}_{\geq 1 + 1}$$

Av denna motsägelse följer att L inte är sammanhangsfritt.

ÖVN 3.14 Se t ex på den kodade parentessträngen $1^{K+1}(1^K)^{K+1}$ som faller ur språket vid sammanhangsfri pumpning. (Endera går villkoret "större parenteser utanför mindre" förlorat. Eller så blir det obalans.)

ÖVN 3.16 Se på $w = \alpha a^K b^K \beta a^K b^K \gamma$. Vi visar att varhelst ett pumpblock av längd $\leq K$ placeras i w så leder pumpning till att w faller ur L :

$$\text{Om } w = \alpha \overbrace{a^K b^K}^u \beta \overbrace{a^K b^K}^u \gamma = \dots \overbrace{vxy}^{\text{pumpblock}} \dots$$

med pumpdelar vxy så har vi följande fall att diskutera

- (i) Antag att pumpblocket ligger till vänster om β och att tecken *inuti* α pumpas: $(\alpha)a^K b^K \beta a^K b^K \gamma$

Vid urpumpning förlorar α ett eller flera tecken, säg $p \leq K$ tecken. Enda möjligheten för den pumpade w att ligga kvar i L (dvs ha α som prefix) är att de förlorade tecknen är a :n så att dessa kan ersättas med p stycken a :n från a -blocket bredvid. Men då blir vänstra u :et p tecken kortare än högra ofördärvade u :et. Enda möjligheten att reparera denna skada vore att β kunde "flytta sig" $p/2$ till höger genom att lämna över $p/2$ stycken tecken till b -blocket i vänstra skadade u :et och samtidigt förvärva lika många a :n från a -blocket i högra ofördärvade u :et. Men för att β inte skall ändras måste de bortlämnade tecknen överensstämma med de förvärvade tecknen – dvs vara a :n. Därmed kommer de två nya u :na att sluta på olika sorts tecken!

- (ii) Antag att pumpblocket ligger till vänster om β , och att tecken *inuti* vänstra u :et pumpas: $\alpha(a^K b^K)\beta a^K b^K \gamma$

Vid urpumpning blir vänstra u :et säg $p \leq K$ tecken kortare. På motsvarande sätt som under i) följer att en sådan skada *inte* går att reparera genom att "flytta" på β .

- (iii) Antag att pumpblocket ligger till höger om a -blocket i vänstra u :et men till

vänster om b -blocket i högra u :et, så att b :n till vänster om β och/eller a :n till höger därom pumpas: $\alpha a^K (b^K \beta a^K) b^K \gamma$

Efter urpumpning kommer högra u och vänstra u att sluta på *olika* antal b :n, eller inledas med *olika* antal a :n.

(iv) Antag att pumpblocket ligger *inuti* β : $\alpha a^K b^K (\beta) a^K b^K \gamma$

Då måste ett urpumpat β ersättas med lika många tecken från vänstra u :et som från högra u :et – om pumpade w skall ligga kvar i L . Men därvid kommer de nya u :na att bli *olika*, eftersom högra u :et skänker bort inledande a :n medan vänstra u :et skänker bort avslutande b :n.

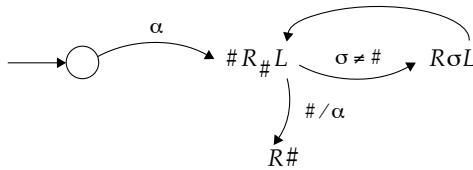
Övriga fall är symmetriska till de redan diskuterade.

4

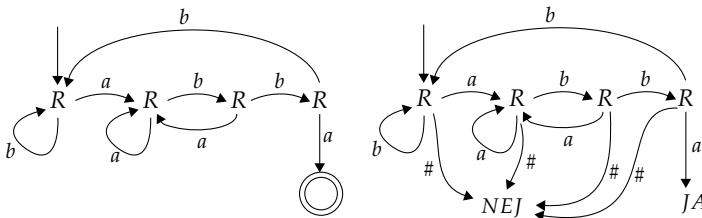
TEST 4.1.a) Nej.

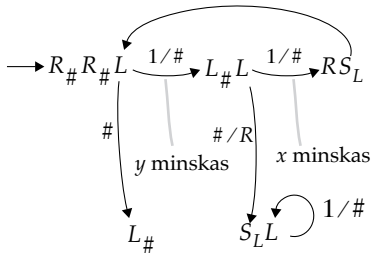
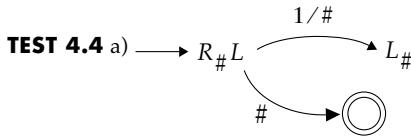
$$\begin{aligned} b) \quad & S \rightarrow aAS | bBS | \$ \\ & A\$ \rightarrow \$a \\ & B\$ \rightarrow \$b \\ & Aa \rightarrow aA \\ & Ba \rightarrow aB \\ & Ab \rightarrow bA \\ & Bb \rightarrow bB \\ & \$ \rightarrow \varepsilon \end{aligned}$$

TEST 4.2 a)



TEST 4.3





b) "TM:en går först till höger förbi det argumentåtskiljande blanktecknet ut till blankteckensvansen och tar sedan ett steg till vänster för att undersöka om högra strängen representerar *nollskilt* y (något som avslöjas av 1:forekomst). Om så är fallet minskas y med en enhet (genom att *högra strängens* avslutande 1:a suddas) varefter (ifall x är nollskild) även *vänstra strängens* avslutande 1:a ersätts med blanktecken som sedan får tjäna

som nytt argumentåtskiljande blanktecken sedan det gamla shiftats bort. Detta arbete med att minska både y och x fortgår tills endera y eller x blir noll. Om y är noll, så är man färdig så fort läshuvudet flyttats till tapens början. Om istället x är noll (något som innebär att $F(x, y) = 0$), så suddas alla eventuella återstående 1:or i y .

ÖVN 4.1 De första två reglerna bildar en rekursiv fläta som producerar $a^n S a^n$ eller $a^{n+1} A a^n$ där $n \geq 0$. Och den sista regeln är en basfallsregel, med vars hjälp man avslutar en rekursivt tillverkad sträng som innehåller $a A a$. Man får således strängar av typ $a^n b a^{n-1}$ där $n \geq 1$. Och sådana strängar kan produceras med en sammanhangsfri grammatik: $S \rightarrow ab | a S a$.

ÖVN 4.2 $S \rightarrow \epsilon | VHS | UNS$

$VH \rightarrow HV$
 $VU \rightarrow UV$
 $VN \rightarrow NV$
 $HU \rightarrow UH$
 $HN \rightarrow NH$
 $UN \rightarrow NU$

} permutationsregler

$V \rightarrow v$
 $H \rightarrow h$
 $U \rightarrow u$
 $N \rightarrow n$


ÖVN 4.4 a) En lyckosam körning av $M = RR_1OR_01L_1L_1$ kan beskrivas som nedan:


$\begin{smallmatrix} \#w \\ \Delta \end{smallmatrix}$	
$\begin{smallmatrix} \#0^n1\dots \\ \Delta \end{smallmatrix}$	R_1 : M söker upp första 1:an till höger (... betecknar återstoden av w)
$\begin{smallmatrix} \#0^n0\dots \\ \Delta \end{smallmatrix}$	0 skriver en 0:a
$\begin{smallmatrix} \#0^n01^k0\dots \\ \Delta \end{smallmatrix}$	R_0 söker upp första 0:an till höger
$\begin{smallmatrix} \#0^n01^k1\dots \\ \Delta \end{smallmatrix}$	1 skriver en 1:a
$\begin{smallmatrix} \#0^n01^j11\dots \\ \Delta \end{smallmatrix}$	L_1 söker upp första 1:an till vänster
$\begin{smallmatrix} \#0^n01^i111\dots \\ \Delta \end{smallmatrix}$	L_1 söker upp första 1:an till vänster


vilket innebär att w ursprungligen måste ha formen $w = 0^n11^i110x$, där n och i är godtyckliga naturliga tal och x är godtycklig i $(0 \cup 1)^*$. Härav följer att w beskrivs av $0^*1^*1^*0(0 \cup 1)^*$.

b) $\{0^n1(0 \cup 1)^m1^{n+1} \mid m, n \in \mathbb{N}\}$, c) "Lika antal av de tre tecknen."

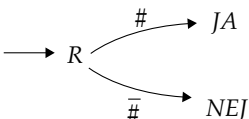
ÖVN 4.5

a) \emptyset accepteras av  vilken om och om igen skriver det tecken som läshuvudet står på vid start.

$\{\epsilon\}$ accepteras av  som stannar om blanktecken påträffas i cellen till höger om den första.

Σ^* accepteras av  som stannar direkt – oavsett vilken sträng som ligger på tapen vid start.

b) \emptyset avgörs av NEJ

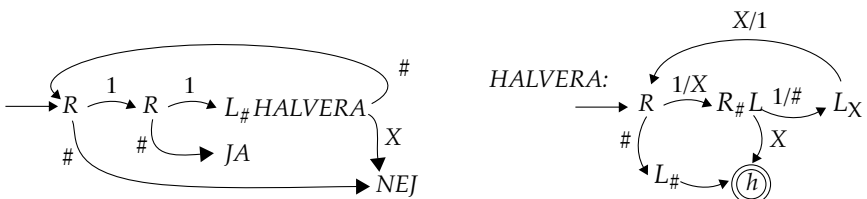
$\{\epsilon\}$ avgörs av 

Σ^* avgörs av JA

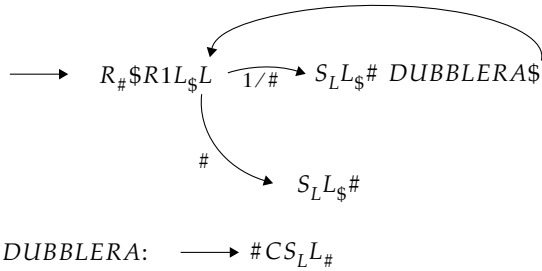
ÖVN 4.6

TM som beräknar F	Kommentar
<p>a)</p>	<p>”Minskar x (i höger ände) med <i>två</i> 1:or om och om igen så länge x är större eller lika med 2. Om det inte finns några flera 1:or kvar efter att x har minskats med den första av de två ettorna, så sätts den suddade 1:an tillbaka igen. Och om inte ens den första av de två 1:orna finns, så är beräkningen färdig.”</p>
<p>b)</p>	<p>”Eftersom beräkning av kvoten vid division med 2 väsentligen går ut på att halvera input, så stryker vi helt sonika ut (dvs ersätter med X) den ena av två konsekutiva 1:or i x under en genomsökning från höger till vänster av x. När hela x på detta sätt är genomsökt och till hälften struken, vidtar en städning som går ut på att genom shiftning avlägsna alla X-tecknen.”</p>

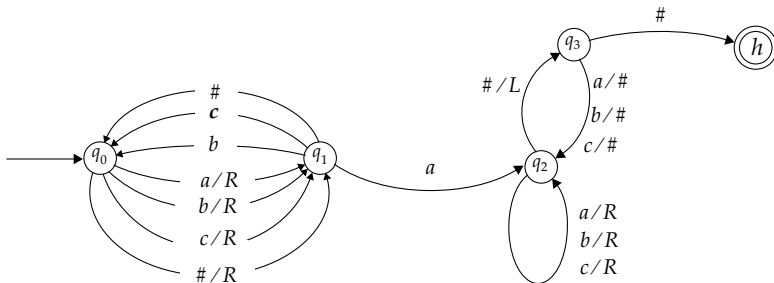
ÖVN 4.7 a) Hjälpmaskinen *HALVERA* (se figur) reducerar antalet 1:or till hälften och ställer tillbaka läshuvudet på tapens första cell ifall den undersökta strängen har ett jämnt antal ettor, men ställer läshuvudet på en cell med tecknet X (varefter huvudmaskinen drivs till *NEJ*) om strängen har ett udda antal ettor. Eftersom huvudmaskinen (den vänstra) använder *HALVERA* om och om igen, kommer tapekonfigurationen för en sträng i språket (antal ettor = tvåpotens) att reduceras till $\#1\#$ (om den inte redan är denna sträng från början) och därvid driva maskinen till *JA*. Och en icke-tom sträng *utanför* språket kommer att reduceras till en sträng med ett udda antal ettor och sedan drivas till *NEJ*. Tom sträng driver också huvudmaskinen till *NEJ*. (*JA*- och *NEJ*-maskinerna antas fungera på gängse sätt.)



- b) En TM som, efter start på x stycken 1:or, stannar med tapen innehållande 2^x stycken 1:or:



ÖVN 4.8 För att konstruera en grammatik som *härmar* den givna TM:en M konstruerar vi först en ny TM M' (se figuren nedan)



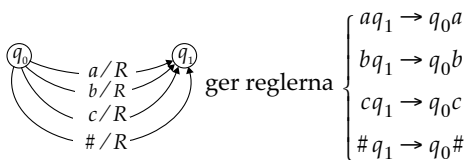
som accepterar samma strängar över $\{a, b, c\}$ som den förra. Skillnaden mellan de två Turingmaskinerna är bara att M' gör hela tapen blank och samtidigt ställer läshuvudet på tapens första cell. En grammatik som härmar M' och därmed M , kommer nu att producera precis de strängar som M accepterar. På gängse sätt skall maskinens övergångar härmas med regler enligt:

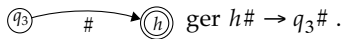
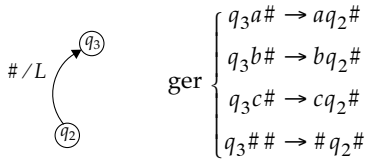
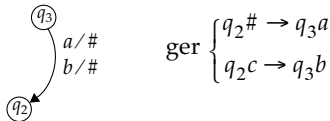
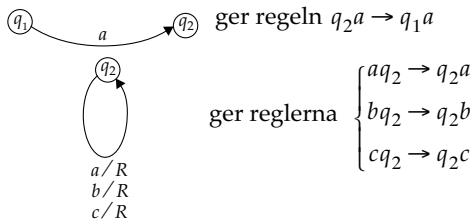
$$\begin{array}{ll} \text{tillståndsövergång} & \text{produktionsregel} \\ \delta(q_i, \sigma) = (q_j, \sigma'') & q_j \sigma'' \rightarrow q_i \sigma \end{array} \quad (1)$$

$$\delta(q_i, \sigma) = (q_j, R) \quad \sigma q_j \rightarrow q_i \sigma \quad (2)$$

$$\delta(q_i, \sigma) = (q_j, L) \quad q_j \sigma' \sigma \rightarrow \sigma' q_i \sigma \text{ för varje tecken } \sigma' \quad (3)$$

Härav,





För att på ett korrekt sätt kunna starta och avsluta härminingen behövs även reglerna:

$$S \rightarrow [h\#]$$

$$\#] \rightarrow \#\#]$$

$$[q_0\# \rightarrow \varepsilon$$

$$\#\#] \rightarrow \#]$$

$$\#] \rightarrow \varepsilon$$

ÖVN 4.9

Antag att M_F och $M_{G_1}, M_{G_2}, \dots, M_{G_m}$ är TM:er som beräknar F, G_1, \dots, G_m . Då kan $H(x) = F(G_1(x), \dots, G_m(x))$ beräknas på följande sätt där understruket tecken markerar läshuvudets placering:

$\#x\#\#\# \dots$

$\#x\#x\#\#\# \dots$

$\#x\$G_1(x)\#\#\# \dots$

$\#x\$G_1(x)\#x\#\#\# \dots$

$\#x\$G_1(x)\$G_2(x)\#\#\# \dots$

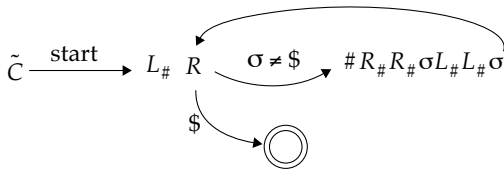
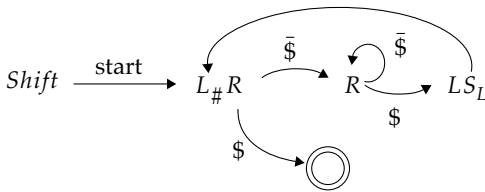
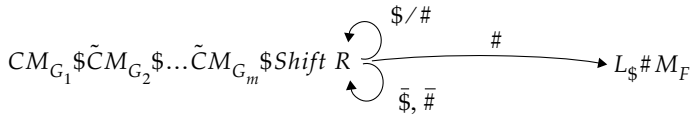
Kopierar x

och kör M_{G_1} på kopian. Sedan skrivs \$.

Kopierar (med en korrigerad kopierare \tilde{C}),

kör M_{G_2} på kopian och skriver sedan \$.

...
 $\#x\$G_1(x)\$G_2(x)\$...\#x### \dots$ Kopierar (med \tilde{C} igen),
 $\#x\$G_1(x)\$G_2(x)\$...\$G_m(x)### \dots$ kör M_{G_m} på kopian och skriver sedan \$.
 $\$G_1(x)\$G_2(x)\$...\$G_m(x)### \dots$ Shiftar bort \$.
 $\$G_1(x)\#G_2(x)\#...\#G_m(x)### \dots$ Sätter tillbaka argumentsåtskiljare (#).
 $\#G_1(x)\#G_2(x)\#...\#G_m(x)### \dots$ Skriver # i första cellen (där det står \$).
 $\#F(G_1(x), \dots, G_m(x))### \dots$ Kör M_F .



5

TEST 5.1 a) $Add(x, Add(y, z))$

b) Funktionen $Udda$ ges av $\begin{cases} Udda(0) = 0 \\ Udda(\ddot{O}ka(x)) = Noll?(Udda(x)) \end{cases}$

c) $Min(x, y) = Om(Mindre(x, y), x, y)$

e) $\begin{cases} Fak(0) = 1 \\ Fak(\ddot{O}ka(x)) = Mult(\ddot{O}ka(x), Fak(x)) \end{cases}$

f) $F(x) = H(x, x)$ med $\begin{cases} H(x, 0) = G(x, 0) \\ H(x, \ddot{O}ka(y)) = Mult(G(x, \ddot{O}ka(y)), H(x, y)) \end{cases}$

$$g) \begin{cases} F(0, y) = y \\ F(\ddot{O}ka(x), y) = G(F(x, y)) \end{cases}$$

TEST 5.2
$$\begin{cases} F(x) = H(x, 0) \\ H(x, y) = Om(Mindre(Add(x, y), Mult(x, y)), y, H(x, \ddot{O}ka(y))) \end{cases}$$

F är odefinierad enbart för $x = 0$ och $x = 1$.

ÖVN 5.1 $F(x) = Om(\ddot{J}amn(x), \ddot{O}ka(x), Minska(x))$

eller
$$\begin{cases} F(0) = 1 \\ F(\ddot{O}ka(x)) = Om(\ddot{J}amn(\ddot{O}ka(x)), Add(3, F(x)), Minska(F(x))) \end{cases}$$

ÖVN 5.2
$$\begin{aligned} f(0, y) &= 0 \\ f((\ddot{O}ka(x), y)) &= Add(Pot((\ddot{O}ka(x), y)), f(x, y)) \end{aligned}$$

ÖVN 5.3
$$\begin{aligned} Avrunda(x) &= Mult(5, Kvot(x, 5)) \quad \text{eller} \\ Avrunda(0) &= 0 \\ Avrunda(\ddot{O}ka(x)) &= Om(Delbar(\ddot{O}ka(x), 5), \\ &\quad \ddot{O}ka(x), \\ &\quad Avrunda(x)) \end{aligned}$$

ÖVN 5.4 En periodisk funktion är av följande slag

x	0	1	...	$p-2$	$p-1$	p	$p+1$...
$F(x)$	b_0	b_1	...	b_{p-2}	b_{p-1}	b_0	b_1	...

och beräknas nog enklast med en sammansättning av *primitiv rekursiva funktioner*:

$$\begin{aligned} F(x) &= Om(Lika(Rest(x, p), 0), b_0 \\ &\quad Om(Lika(Rest(x, p), 1), b_1 \\ &\quad \dots \\ &\quad Om(Lika(Rest(x, p), p-2), b_{p-2}, b_{p-1})) \end{aligned}$$

T ex beräknas

x	0	1	2	3	4	5	6	7	...
$F(x)$	5	2	73	5	2	73	5	2	...

av

$$\begin{aligned} F(x) &= Om(Lika(Rest(x, 3), 0), 5 \\ &\quad Om(Lika(Rest(x, 3), 1), 2, 73)) \end{aligned}$$

ÖVN 5.5 a) Funktionen $TvåLog(x)$ beräknad med hjälp av *primitiv rekursion*:

$$TvåLog(0) = 0$$

$$TvåLog(Öka(x)) = Om(Lika(Öka(x), Pot(2, Öka(TvåLog(x)))), \\ Öka(TvåLog(x)), \\ TvåLog(x))$$

och med *framåtrekursion*:

$$TvåLog(x) = H(x, 0) \\ H(x, y) = Om(Mindre(x, Pot(2, y)), \\ Minska(y), \\ H(x, Öka(y)))$$

c) *TriangelTal?* kan konstrueras på följande sätt:

$$TriangelTal?(x) = LikaMedNågotTriangelTal(x, x) \\ LikaMedNågotTriangelTal(x, 0) = Lika(x, TriangelTal(0)) \\ LikaMedNågotTriangelTal(x, Öka(y)) = Eller(Lika(x, TriangelTal(Öka(y))), \\ LikaMedNågotTriangelTal(x, y))$$

ÖVN 5.7 Outputtabellen nedan, som förresten visar att $F(x)$ är heltalsdelen av

$Öka(10 \log(x))$, ger uppslag till en lösning som tar fasta på att F 's värden uppgraderas när input är en 10-potens:

x	0	1	2	3	4	5	6	7	8	9	10	...	99	100	...	999	1000	...
$F(x)$?	1	1	1	1	1	1	1	1	1	2	...	2	3	...	3	4	...

a) *framåtrekursion* $F(x) = H(x, 0)$
 $H(x, y) = Om(Mindre(x, Pot(10, y)),$
 $y,$
 $H(x, Öka(y)))$

b) *primitiv rekursion* $F(0) = 0$
 $F(Öka(x)) = Om(Mindre(Öka(x), Pot(10, F(x))),$
 $F(x),$
 $Öka(F(x)))$

OBS! I bägge lösningarna hävdas att talet 0 har 0 siffror. Till de läsare som menar att detta är onaturligt (och kanske t.o.m felaktigt) vill jag säga följande. Det här "nonchalanta" sättet att räkna nollor i talet 0 är inget annat än en tillämpning av principen att nonchalera inledande nollor i ett tal. (T ex att strunta i de inledande nollorna i 00078 och säga 78 istället, eller – för att ta ett aktuellt exempel – att strunta i den inledande nollan i 0 och istället säga ϵ som är 0 siffror långt.)

ÖVN 5.8

$DFA(w) = H(1, w)$ 1 representerar starttillståndet

$H(q, w) = Om(Tom(w),$
 $Accept(q),$
 $H(NästaTillstånd(q, Första(w)), UtomFörsta(w)))$

$Tom(w) = Noll?(w)$

$Första(w) = Rest(w, k)$ Konsumerat tecken

$UtomFörsta(w) = Kvot(w, k)$ Resten av strängen

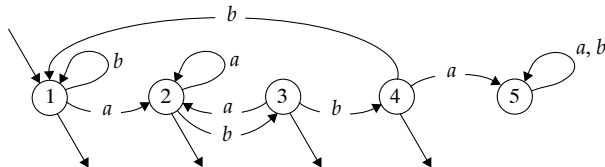
Kommentar: w antas vara ett naturligt tal representerande en sträng enligt den modell som illustreras med följande exempelsträng:

sträng med n tecken naturligt tal

$$bbab \dots a \Leftrightarrow \underbrace{2}_{Rest(w, k)} + \underbrace{2k^1 + 1k^2 + 2k^3 + \dots + 1k^{n-1}}_{Kvot(w, k)},$$

där alfabetets enskilda tecken representeras av talen 1, 2, ..., och basen k är en enhet högre än antalet tecken i alfabetet (för den DFA som härmas).

Funktionerna *Accept* och *NästaTillstånd* är relaterade till den DFA som härmas, och ges av falldefinierade funktioner som i exemplet nedan.



$Accept(q) = Eller(Eller(Eller(Ett?(q), Två?(q)), Tre?(q)), Fyra?(q))$

$NästaTillstånd(q, \sigma) = Om(Ett?(q),$
 $Om(Ett?(\sigma), 2, 1),$
 $Om(Två?(q),$
 $Om(Ett?(\sigma), 2, 3),$
 $Om(Tre?(q),$
 $Om(Ett?(\sigma), 2, 4),$
 $Om(Fyra?(q),$
 $Om(Ett?(\sigma), 5, 1),$
 $5)))$

Ovanstående DFA-härmande funktion $DFA(w)$ är uppenbarligen en rekursiv funktion där dess hjälpfunktion $H(q, w)$ har båda argumenten som rekurerande

argument. Rekursionen liknar framåtrekursion, men är icke äkta sådana.

Faktum är att det går att tillverka en DFA-härmande funktion inom klassen av *primitivt* rekursiva funktioner:

$$DFA(w) = Accept(Tillstånd(Längd(w), w))$$

$$Tillstånd(0, w) = 0$$

$$Tillstånd(Öka(n), w) = NästaTillstånd(Tillstånd(n, w), Symbol(w, Öka(n)))$$

$Tillstånd(n, w)$ är konstruerad så att den returnerar det tillstånd som DFA:n drivs till efter konsumtion av de n första tecknen i w .

$Längd(w)$ (som är tänkt att returnera antalet tecken i strängen w) överlåtes till läsaren att konstruera. Och $Symbol(w, n)$ som returnerar den n :te symbolen i w finns beskriven på sid 166.

6

TEST 6.1 Ett tvåraders **primitivt** loop-program:

```
r ← z
x ggr{r ← y}
```

7

TEST 7.1 a) Sant! Om L är Turingavgörbart är även \bar{L} Turingavgörbart, ty om Turingmaskinen T avgör L , så kan man korrigera T till en Turingmaskin som avgör \bar{L} . Man behöver nämligen bara låta JA - och NEJ -skrivarna byta plats i T .

b) Falskt! Se på \mathcal{L}_{stopp} och dess komplementspråk i EXEMPEL 7.1 på sidan 183.

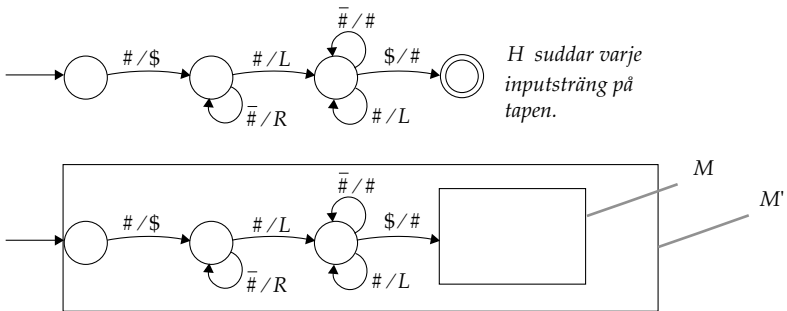
ÖVN 7.1 a) OAVGÖRBART. Denna deluppgift handlar om att avgöra för godtycklig TM M om $L(M) \in \Omega$, där Ω är mängden av alla Turingaccepterbara språk som innehåller någon sträng med en eller flera 1:or.

Men Ω är *icketrivial*, vilket inses t ex av att det reguljära språket $\{1\}$ tillhör Ω medan det reguljära \emptyset inte gör det. Av Rices sats följer därför att problemet är *oavgörbart*.

b) Här är Rices sats *inte* tillämplig, ty Rices sats handlar om Turingaccepterbara språk, medan denna deluppgift handlar om DFA-accepterbara språk. I själva verket är problemet AVGÖRBART. Man behöver ju bara undersöka om DFA:n M har någon 1-övergång på väg från starttillståndet till något accepterande tillstånd, vilket är en ändlig fallundersökning i M :s kod.

c) OAVGÖRBART.

- d) Rices sats är *inte* tillämpbar, ty detta problem handlar om en egenskap som berör TM:ernas *tillstånd*. Ändå kan vi visa att problemet är OAVGÖRBART med motiveringen att en TM som kunde avgöra problemet ifråga också skulle kunna användas för att avgöra stoppproblemet för blank tape, ett välkänt oavgörbart problem. Antag nämligen att M är en godtycklig TM och låt $M' = HM$, där H är en suddande TM som nedan:



Antagandet att en TM kan avgöra problemet i texten innebär att samma TM kan avgöra om ovanstående M' stannar för någon sträng med lika många tecken som antalet tillstånd i M' . Men M' 's inledande suddande utföres lyckosamt oavsett vilken inputsträng man placerar på tapen. Och detta vare sig strängen har lika många tecken som M' har tillstånd eller ej (bara den är skriven i M :s dvs i M' 's inputalfabet). Därför beror M' 's eventuella stannande enbart av huruvida M stannar för blank tape eller ej.

Alltså, skulle en TM med kompetens att avgöra textens problem för en godtycklig TM M kunna avgöra stoppproblemet för blank tape. Men det senare är bevisligen fel. Alltså ...

- e) Inte heller här är Rices sats tillämpbar, ty Rices sats handlar om icke-triviala egenskaper hos de språk som Turingmaskiner accepterar – *inte* om Turingmaskinens "inre arbete" i form av vilka tillstånd som de besöker eller inte besöker under körning. Därför måste denna fråga attackeras annorlunda. Vi kan visa att problemet är OAVGÖRBART med ett motsägelsebevis liknande det förra.

Antag att någon TM T kan avgöra problemet. Då skulle man kunna använda T till att avgöra stopp-problemet. Hur? Jo, om man för en TM M och en sträng w vill veta huruvida M stannar eller ej vid start på w , ändrar man först M på följande sätt.

Ersätt M 's stopptillstånd med två tillstånd: q (som är ett för M helt nytt tillstånd) samt ett stopptillstånd. Byt närmare bestämt varje övergång i M som leder till det gamla stopptillståndet mot en övergång som leder till q följt av en övergång till det nya stopptillståndet. (Den senare övergången från q till stopptillståndet görs *ovillkorlig*, dvs oberoende av vilket tecken läshuvudet står på.)

Genom att nu låta T avgöra huruvida *korrigerade* M någonsin uppsöker q eller ej, vid start på w (detta var ju vad T antogs kunna), skulle T i själva verket avgöra huruvida ursprungs- M någonsin stannade vid start på w .

Dvs T skulle avgöra stopp-problemet, något som vi vet är omöjligt. Alltså ...

f) AVGÖRBART. En TM som avgör frågan behöver "bara" kunna provköra DFA:n M på w , och returnera "ja" eller "nej" beroende på om q påträffas eller ej under provkörningen. Varje sådan provkörning involverar här (till skillnad från förra delfrågan) endast ändligt många tillståndsövergångar (eftersom DFA:n M har konsumerat w efter lika många övergångar som w har tecken). En universell Turingmaskin korrigerad på lämpligt sätt (för att hålla uppsikt på tillståndet q) skulle klara denna avgörbarhetsuppgift.

g) Här är Rices sats *inte* (direkt) tillämpbar, eftersom den inte handlar om Turingmaskinens "skrivande" utan om icke-triviala egenskaper hos Turingaccepterbara språk. Ändå är problemet OAVGÖRBART. Ty om det finnes en TM T som kunde avgöra problemet, skulle man kunna använda T till att avgöra för *godtycklig* TM om denna *någonsin stannar* efter start på blank tape. Se här:

Om man vill veta huruvida en TM stannar eller ej (vid start på blank tape) ändrar man först en eventuell 1:a i dess alfabet till något tecken a som inte redan ingår där. Sedan ändrar man varje stopptillståndsövergång så att 1 skrivs omedelbart innan: $\sigma/1 \rightarrow \odot$. Dvs så att 1 skrivs just innan stoppet. Genom de två korrigeringarna kommer den korrigerade TM:en att skriva 1 innan den stannar – och aldrig annars. Att avgöra ifall den *ursprungliga* TM:en stannar blir då liktydigt med att avgöra ifall den *korregerade* skriver 1. Om T för *godtycklig* TM M kunde avgöra huruvida M någonsin skriver 1 (efter start på blank tape), skulle T kunna avgöra detta för vår korrigerade TM, och därmed skulle T för godtycklig TM kunna avgöra ifall den stannar (på blank tape). Men detta senare problem är ett icke-trivialt problem om Turingaccepterbara språk (eller hur!) och därmed oavgörbart enligt Rices sats.

h) AVGÖRBART. Problemet kan avgöras genom provkörning i ändligt många steg, säg N , där N är antalet (olika) konfigurationer som M kan befinna sig i under körning på det begränsade område av tapen som w ligger på (vid start).

Ty om det under sådan provkörning visar sig att M söker upp stopptillståndet innan M lämnar området eller att M lämnar området, så är problemet avgjort.

Och om M under sådan provkörning inte uppför sig på ovanstående sätt, dvs inte stannar och inte heller lämnar området, så är också problemet avgjort. Ty i detta fall kommer inte M heller under någon längre provkörning att lämna nämnda område eller att stoppa. Se nedan!

Efter N steg har nämligen M befunnit sig i $N + 1$ konfigurationer. Och när M håller sig till det begränsade området av tapen måste minst två av konfigurationerna sammanfalla (eftersom M inte kan befinna sig i fler än N olika konfigurationer när M håller sig till det begränsade området!). Och från och med att en viss konfiguration har uppträtt för andra gången, så har M kommit in i en slinga som innebär att uppförandet mellan den första och andra gången upprepas. (Lika konfigurationer ger upphov till lika ageranden p.g.av determinismen!)

i) AVGÖRBART. Att räkna antalet tillstånd i en beskrivning (kodning) av en TM är en enkel sak om kodningen är gjord t ex som för den universella Turingmaskinen.

j) OAVGÖRBART. Om en TM T kunde avgöra det här problemet, skulle man med dess hjälp kunna avgöra stopp-problemet. Låt nämligen U vara en universell TM

som *inte* har fler tillstånd än tio. Sådan universell TM finns (Shannon 1956). Korrigera U genom att efter U :s stopptillstånd (som berövas stoppstatus) tillfoga en följd av nya tillstånd där det sista får stoppstatus och där de extra tillståndsövergångarna görs ovillkorliga. Se till att de nya tillstånden blir så många att U får *flera* tillstånd än tio. Då gäller: Ursprungs- U stannar för ett givet input omm den utökade U besöker flera än tio tillstånd för samma input. Vidare gäller ju som bekant att en universell TM (U) kan härma en godtycklig TM M' på en godtycklig inputsträng w' vid körning på (en kodning av) M' , w' . Så om en TM T kunde avgöra problemet i denna uppgift, skulle T kunna avgöra om korrigerade U besöker flera än tio (skilda) tillstånd vid körning på M' , w' . Och då skulle T avgöra stopp-problemet för M' , w' .

ÖVN 7.2 För att tackla denna uppgift kan man använda sig av följande två egenskaper för oändliga DFA- resp. PDA-språk.

- Låt M vara en DFA med N tillstånd. Då accepterar M oändligt många strängar omm M accepterar någon sträng av längd $N \leq |w| < 2N$.
- Låt M vara en PDA vars språk kan beskrivas av en CFG med

regler som använder färre än N_1 stycken icketerminerande symboler och som inte har något högerled längre än l . (1)

Sätt $N = l^{N_1}$.

Då accepterar M oändligt många strängar omm M accepterar någon sträng av längd $N \leq |w| < 2N$.

- a) Ja, detta är avgörbart. Efter provkörning av DFA:n på de ändligt många strängarna (över DFA:ns alfabet) av längd $N \leq |w| < 2N$, vet vi svaret. Sådan provkörning kan t ex den universella TM:en utföra.
- b) Även detta är avgörbart. Men här kan man *inte* genom *provkörning* som i DFA-fallet få direkt visshet i frågan. Ty en PDA kan ju pyssla med stackarbete utan att konsumera något från inputtapen, varför antalet övergångar i en PDA:s strängkonsumtion kan överstiga strängens längd. Dessutom kan det ju hända (p.g.av ickedeterminismen) att vissa körningar av en sträng inte leder till acceptans, trots att andra körningar av samma sträng leder till acceptans. Trots dessa svårigheter kan vi visa – med hjälp av en CFG för PDA:ns språk sådan att alla *icketomma* strängar i PDA:ns språk kan produceras med regler som nedan – att problemet är avgörbart

ingen terminerande regel har tom sträng i högerledet (2)

alla icketerminerande regler är förlängningsregler (3)

Låt oss först se varför man med regler av denna typ kan avgöra problemet.

Om G är en CFG med egenskaperna (2) och (3) finns det i G bara ändligt många produktionsträd av viss bredd. Ty, om G :s regler är som ovan, så ökar bredden i ett produktionsträd för varje ny nivå som inte är en avslutande nivå. Därmed finns det för ett sådant G bara *ändligt* många (eventuellt noll stycken) produktionsträd vars *bredd* uppfyller $N \leq \text{bredd} < 2N$. Genom att tillverka dessa ändligt många träd med terminerande symboler i löven, får man visshet i det givna problemet. Finns det

noll stycken är språket ändligt. Annars oändligt.

Frågan är nu, kan man för en given PDA på ett algoritmiskt sätt finna en CFG för PDA:ns språk så att (2) och (3) gäller?

Javisst. Här följer en informell beskrivning av en algoritm som åstadkommer (2) för en given regeluppsättning:

För varje regelförekomst $A \rightarrow \epsilon$ och $B \rightarrow \alpha A \beta$, där α, β är strängar av terminerande eller icketerminerande symboler, tillfoga den nya regeln $B \rightarrow \alpha \beta$. Upprepa detta tills inga nya regler kan tillfogas. (Inse att processen tar slut efter ändligt många steg, och att den nya grammatiken beskriver samma språk som den gamla.) Tag sedan bort varje $A \rightarrow \epsilon$. Om Du därvid har tagit bort $S \rightarrow \epsilon$ måste Du lägga till en ny startsymbol S' samt reglerna $S' \rightarrow \epsilon | S$.

Nu återstår det att fixa (3). Här gäller det att göra sig av med eventuella regler av typ $A \rightarrow B$. Detta kan göras på följande sätt. För varje par av skilda icketerminerande symboler A, B där den förra kan producera den senare i ett eller flera steg (dvs sådana att $A \Rightarrow^* B$) och där det finns en regelförekomst av typ $B \rightarrow \alpha$ lägg till regeln $A \rightarrow \alpha$. Inse att varje symbolpar A, B som ovan kan hittas algoritmiskt! Tag slutligen bort alla regler av typ $A \rightarrow B$. Inse att hela detta förfarande har ett slut, och att den nya grammatiken producerar samma strängar som den gamla!

Sammanfattning:

Konstruera först en CFG för PDA:ns språk. (Detta kan åstadkommas algoritmiskt i den anda som vi i något enstaka exempel har genomfört en sådan konstruktion.)

Transformera sedan (vid behov) CFG:n till en ny CFG med egenskaperna (2) och (3). Kalla den senare för G .

Låt N_1 vara ett tal större än antalet icketerminerande symboler i G ,
 l maximala längden hos G :reglernas högerled och $N = l^{N_1}$.

Producera till sist med hjälp av reglerna i G de strängar av längd $N \leq |w| < 2N$ som det går att producera. (Det finns bara ändligt många.)

Om Du finner att det finns noll stycken sådana strängar, så vet Du att den givna PDA:n accepterar ändligt många strängar. Och om Du finner att det finns en eller flera, vet Du att PDA:n accepterar oändligt många strängar.

ÖVN 7.3 a) AVGÖRBART. Motivering: Mängden av strängar som driver M_1 från q_1 till acceptans bildar ett reguljärt språk L . (Om M_1 korrigeras så att q_1 blir dess starttillstånd får man en DFA M_1' för L .) Det givna problemet kan sedan formuleras på följande sätt.

"Finns det någon sträng $w_2 \in L(M_2) \cap L(M_1')$?"

Eller – om M är en DFA för $L(M_2) \cap L(M_1')$ –

"Är $L(M) \neq \emptyset$?"

Detta problem kan avgöras genom att först konstruera M (detta har en välkänd algoritmisk lösning) och sedan undersöka om det i M finns någon följd av över-

gångar från starttillståndet till något accepterande tillstånd. (Ett avgörbart problem i form av en ändlig fallundersökning.)

- b) OAVGÖRBART. Om nämligen problemet *vore* avgörbart för godtycklig M_1 och godtyckligt q_1 , skulle det vara avgörbart för en DFA M_1 vars språk är $\{\varepsilon\}$ och för q_1 vald som M_1 's starttillstånd. Dvs då skulle följande problem vara avgörbart.

"Accepteras ε av Turingmaskinen M_2 ?" (4)

Men detta är ett oavgörbart problem (se nedan) varför detsamma måste gälla för det givna problemet.

Oavgörbarheten hos (4) följer av Rices sats genom att betrakta den icke-triviala

$$\Omega = \{L \mid L \text{ Turingaccepterbart och } \varepsilon \in L\}$$

Att Ω är icke-trivial bevisas t ex av att $\emptyset \notin \Omega$ och $\{\varepsilon\} \in \Omega$.

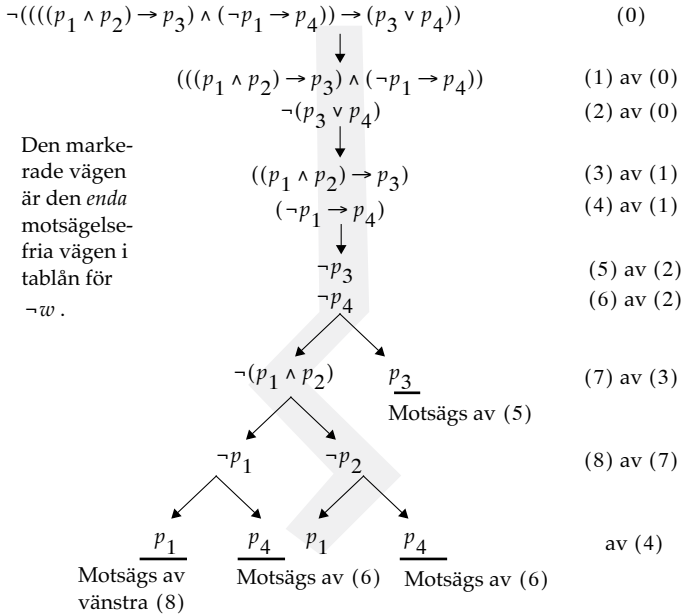
8

TEST 8.2 a) Tablån för $\neg w$ har *en* motsägelsefri väg (se figuren nedanför) längs vilken $\neg p_3 \wedge \neg p_4 \wedge \neg p_2 \wedge p_1$ är sann. Dvs w är falsk om p_3, p_4, p_2 är falska och p_1 är sann. Tablån för w har inga motsägelser, vilket innebär att w är sann längs *varje* väg i dess tablå. Betraktas vägarna ifråga finner man att w är sann i följande fall:

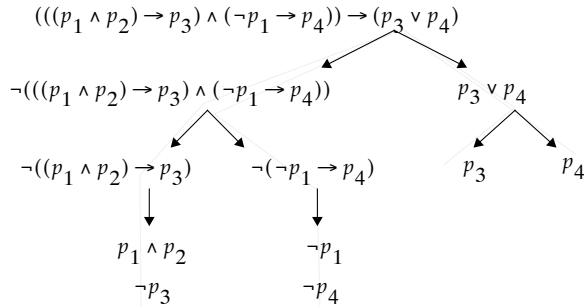
- någon av p_3, p_4 är sann
- p_1, p_2 är sanna och p_3 är falsk
- p_1, p_4 är båda falska.

(Inse att dessa fall sammanfaller med komplementet till fallet med p_3, p_4, p_2 falska

och p_1 sann.)



Alla (fyra) vägar är motsägelsefria i tablå för w .



- c) Resolventen $\{p_1, \neg p_1\}$ (som alltid är sann) erhålles om de motstridiga literalerna $p_2, \neg p_2$ avlägsnas, och resolventen $\{p_2, \neg p_2\}$ erhålles på motsvarande sätt om de motstridiga $p_1, \neg p_1$ avlägsnas.

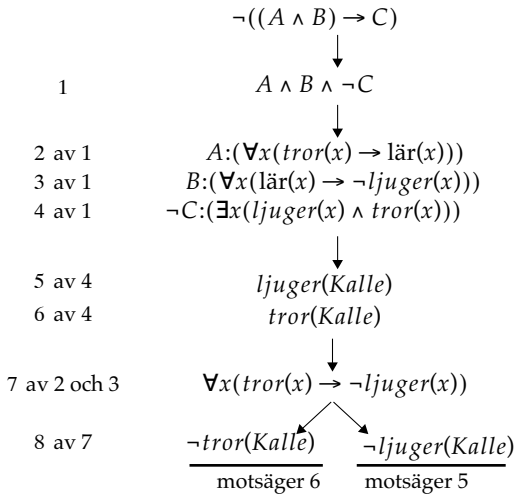
TEST 8.3

a) A: $\forall x(\text{tror}(x) \rightarrow \text{lär}(x))$

B: $\neg \exists x(\text{lär}(x) \wedge \text{ljuger}(x)) \equiv \forall x(\neg \text{lär}(x) \vee \neg \text{ljuger}(x)) \equiv \forall x(\text{lär}(x) \rightarrow \neg \text{ljuger}(x))$

C: $\neg \exists x(\text{ljuger}(x) \wedge \text{tror}(x)) \equiv \forall x(\neg \text{ljuger}(x) \vee \neg \text{tror}(x)) \equiv \forall x(\text{ljuger}(x) \rightarrow \neg \text{tror}(x))$

b) Först en semantisk tablå som visar att formeln är valid:



Sedan en klausulresolution som visar samma sak (men först transformeras formeln till konjunktiv normalform och skrivs som en klausulmängd):

$$\neg((A \wedge B) \rightarrow C) \equiv A \wedge B \wedge \neg C \equiv$$

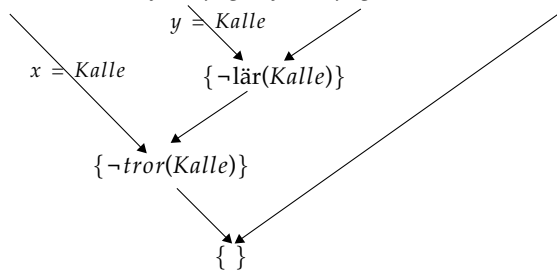
$$\forall x (\neg tror(x) \vee lär(x)) \wedge \forall x (\neg lär(x) \vee \neg ljuger(x)) \wedge \exists x (ljuger(x) \wedge tror(x)) \equiv$$

$$\forall x (\neg tror(x) \vee lär(x)) \wedge \forall y (\neg lär(y) \vee \neg ljuger(y)) \wedge (ljuger(Kalle) \wedge tror(Kalle)) \equiv$$

$$\forall x \forall y (\neg tror(x) \vee lär(x)) \wedge (\neg lär(y) \vee \neg ljuger(y)) \wedge ljuger(Kalle) \wedge tror(Kalle) \equiv$$

$$\forall x \forall y \{ \{ \neg tror(x), lär(x) \}, \{ \neg lär(y), \neg ljuger(y) \}, \{ ljuger(Kalle) \}, \{ tror(Kalle) \} \}$$

$$\{ \{ \neg tror(x), lär(x) \}, \{ \neg lär(y), \neg ljuger(y) \}, \{ ljuger(Kalle) \}, \{ tror(Kalle) \} \}$$



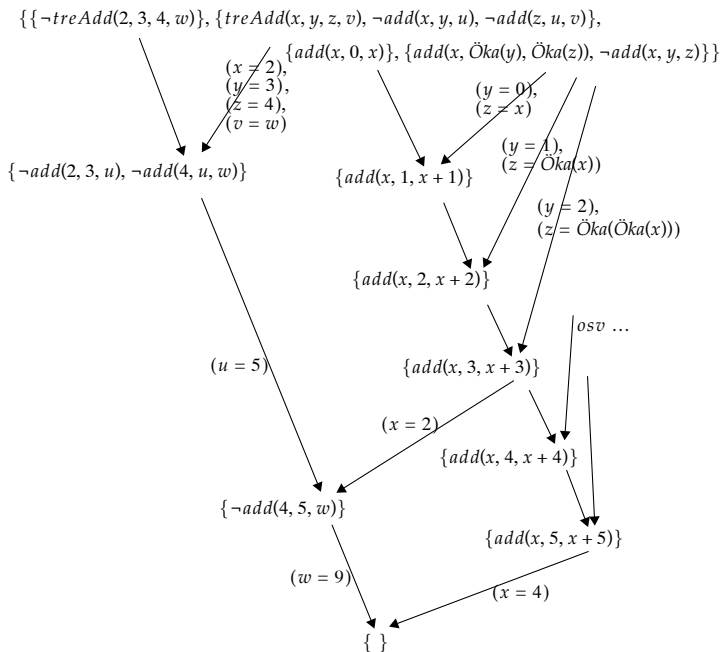
ÖVN 8.1 Formeln är osatisfierbar.

ÖVN 8.2 a) $returnerarNoll(0, 0) \wedge$
 $(returnerarNoll(\ddot{O}ka(x), y) \leftarrow returnerarNoll(x, y))$

$$\begin{aligned} & \text{returnerarEtt}(0, 1) \wedge \\ & (\text{returnerarEtt}(\ddot{\text{O}}\text{ka}(x), y) \leftarrow \text{returnerarEtt}(x, y)) \\ & \text{sub}(x, 0, x) \wedge \\ & (\text{sub}(x, \ddot{\text{O}}\text{ka}(y), \text{minska}(z)) \leftarrow \text{sub}(x, y, z)) \end{aligned}$$

c) $((treAdd(x, y, z, u) \leftarrow add(x, y, v) \wedge add(v, z, u)) \wedge$
 $add(x, 0, x) \wedge (add(x, \ddot{O}ka(y), \ddot{O}ka(z)) \leftarrow add(x, y, z))) \equiv$
 $((treAdd(x, y, z, u) \vee \neg add(x, y, v) \vee \neg add(v, z, u)) \wedge$
 $add(x, 0, x) \wedge (add(x, \ddot{O}ka(y), \ddot{O}ka(z)) \vee \neg add(x, y, z))) \equiv$
 $\{\{treAdd(x, y, z, u), \neg add(x, y, v), \neg add(v, z, u)\}, \{add(x, 0, x)\},$
 $\{add(x, \ddot{O}ka(y), \ddot{O}ka(z)), \neg add(x, y, z)\}$

Till sist klausulresolutionen ...



ÖVN 8.3 Predikaten *mindre*, *mult*, *add* beskrivs av följande tre Hornformler.

$$\begin{aligned} & add(x, 0, x) \wedge (add(x, \ddot{O}ka(y), \ddot{O}ka(z)) \leftarrow add(x, y, z)) \\ & mult(x, 0, 0) \wedge (mult(x, \ddot{O}ka(y), u) \leftarrow (mult(x, y, z) \wedge add(x, z, u))) \\ & mindre(x, \ddot{O}ka(x)) \wedge (mindre(x, \ddot{O}ka(y)) \leftarrow mindre(x, y)) \end{aligned}$$

Om man nu \wedge -klistrar $\neg rest(3, 2, w)$ med den givna Hornformeln samt med de tre Hornformlerna för *add*, *mult* och *mindre* så får man *en* Hornformel (innehållande 8

stycken klausuler) som med mängdnotation får följande utseende (notera hur de enskilda klausulerna har numrerats med hjälp av index):

$$\begin{aligned} & \{\{\neg \text{rest}(3, 2, w)\}\}_1, \\ & \{\text{rest}(x, y, r), \neg \text{mindre}(r, y), \neg \text{mult}(y, z, u), \neg \text{add}(u, r, x)\}_2, \\ & \{\text{add}(x, 0, x)\}_3, \{\text{add}(x, \text{Öka}(y), \text{Öka}(z)), \neg \text{add}(x, y, z)\}_4, \\ & \{\text{mult}(x, 0, 0)\}_5, \{\text{mult}(x, \text{Öka}(y), u), \neg \text{mult}(x, y, z), \neg \text{add}(x, z, u)\}_6, \\ & \{\text{mindre}(x, \text{Öka}(x))\}_7, \{\text{mindre}(x, \text{Öka}(y)), \neg \text{mindre}(x, y)\}_8 \end{aligned}$$

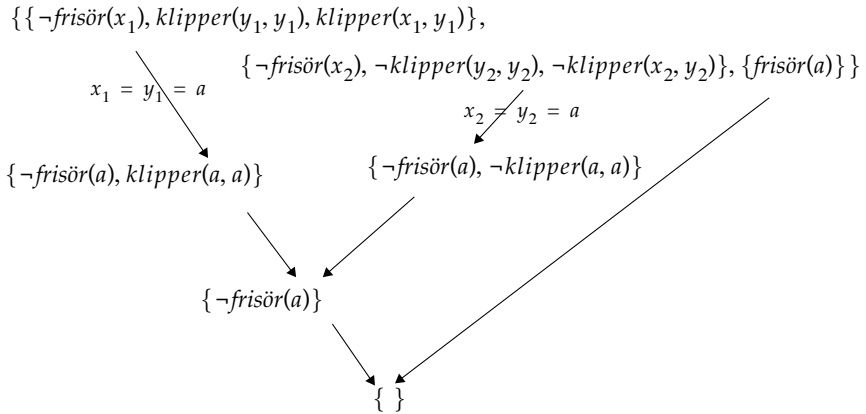
Här följer nu en klausulresolution (vars biprodukt är en beräkning av *resten* vid divisionen 3 med 2):

- 1 och 2: $(x = 3, y = 2, r = w)$ ger $\{\neg \text{mindre}(w, 2), \neg \text{mult}(2, z, u), \neg \text{add}(u, w, 3)\}_9$
- 3 och 4: $(y = 0, z = x)$ ger $\{\text{add}(x, 1, \text{Öka}(x))\}_{10}$
- 9 och 10: $(x = 2, u = 2, w = 1)$ ger $\{\neg \text{mindre}(1, 2), \neg \text{mult}(2, z, 2)\}_{11}$
- 5 och 6: $(y = 0, z = 0)$ ger $\{\text{mult}(x, 1, u), \neg \text{add}(x, 0, u)\}_{12}$
- 3 och 12: $(u = x)$ ger $\{\text{mult}(x, 1, x)\}_{13}$
- 11 och 13: $(x = 2)$ ger $\{\neg \text{mindre}(1, 2)\}_{14}$
- 7 och 14: $(x = 1)$ ger $\{\}$

ÖVN 8.4

- a) A: $\forall x \forall y (\text{frisör}(x) \rightarrow (\neg \text{klipper}(y, y) \rightarrow \text{klipper}(x, y)))$
 B: $\neg \exists x \exists y (\text{frisör}(x) \wedge \text{klipper}(y, y) \wedge \text{klipper}(x, y))$
 C: $\forall x (\neg \text{frisör}(x))$
- b) Vi förbereder för en klausulresolution ...
 A: $\forall x \forall y (\neg \text{frisör}(x) \vee \text{klipper}(y, y) \vee \text{klipper}(x, y))$
 B: $\forall x \forall y (\neg \text{frisör}(x) \vee \neg \text{klipper}(y, y) \vee \neg \text{klipper}(x, y))$
 C: $\forall x (\neg \text{frisör}(x))$
 $(A \wedge B) \rightarrow C$ är valid omm $A \wedge B \wedge \neg C$ är osatisfierbar.
 $A \wedge B \wedge \neg C = \forall x_1 \forall y_1 \forall x_2 \forall y_2 ((\neg \text{frisör}(x_1) \vee \text{klipper}(y_1, y_1) \vee \text{klipper}(x_1, y_1)) \wedge$
 $(\neg \text{frisör}(x_2) \vee \neg \text{klipper}(y_2, y_2) \vee \neg \text{klipper}(x_2, y_2)) \wedge$
 $\text{frisör}(a))$

Här kommer nu klausulresolutionen:



ÖVN 8.5 Om individuniversum är mängden av alla drakar, så kommer följande formel att beskriva de tre meningarna.

- a) $(\forall x(\forall z(\text{barn}(z, x) \rightarrow \text{kanFlyga}(z)) \rightarrow \text{glad}(x))) \wedge$
 $(\forall x(\text{grön}(x) \rightarrow \text{kanFlyga}(x))) \wedge$
 $(\forall x(\exists y(\text{barn}(x, y) \wedge \text{grön}(y)) \rightarrow \text{grön}(x)))$

vilken även kan skrivas som

$$(\forall x(\exists z(\text{barn}(z, x) \wedge \neg \text{kanFlyga}(z)) \vee \text{glad}(x))) \wedge$$

$$(\forall x(\neg \text{grön}(x) \vee \text{kanFlyga}(x))) \wedge$$

$$(\forall x(\forall y(\neg \text{barn}(x, y) \vee \neg \text{grön}(y)) \vee \text{grön}(x)))$$

eller (om $b(x)$ är ett icke flygkunnigt barn till x)

$$(\forall x((\text{barn}(b(x), x) \wedge \neg \text{kanFlyga}(b(x))) \vee \text{glad}(x))) \wedge$$

$$(\forall x_1(\neg \text{grön}(x_1) \vee \text{kanFlyga}(x_1))) \wedge$$

$$(\forall x_2(\forall y(\neg \text{barn}(x_2, y) \vee \neg \text{grön}(y)) \vee \text{grön}(x_2)))$$

eller

$$(\forall x((\text{barn}(b(x), x) \vee \text{glad}(x)) \wedge (\neg \text{kanFlyga}(b(x)) \vee \text{glad}(x))))$$

$$(\forall x_1(\neg \text{grön}(x_1) \vee \text{kanFlyga}(x_1))) \wedge$$

$$(\forall x_2(\forall y(\neg \text{barn}(x_2, y) \vee \neg \text{grön}(y)) \vee \text{grön}(x_2)))$$

eller (driv ut all-kvantorererna och låt *ledsen* vara motsatsen till *glad*)

$$u = \forall x \forall x_1 \forall x_2 \forall y((\text{barn}(b(x), x) \vee \neg \text{ledsen}(x)) \wedge$$

$$(\neg \text{kanFlyga}(b(x)) \vee \neg \text{ledsen}(x)) \wedge$$

$$(\neg \text{grön}(x_1) \vee \text{kanFlyga}(x_1)) \wedge$$

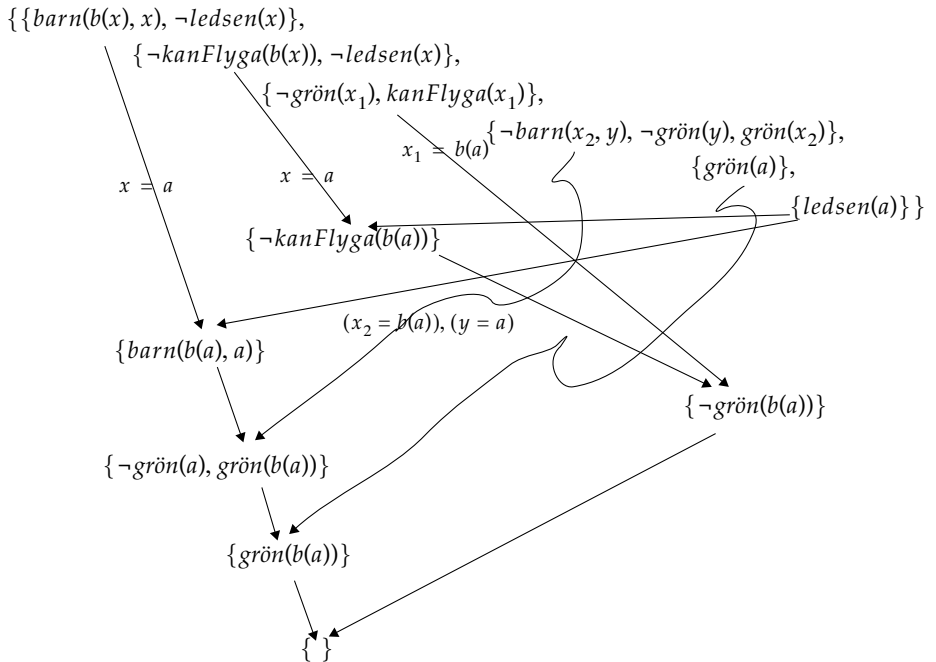
$$(\neg \text{barn}(x_2, y) \vee \neg \text{grön}(y) \vee \text{grön}(x_2)))$$

vilket är en Hornformel bestående av fyra Hornklausuler.

Nu gör vi klausulresolution på u kompletterad med förnekandet att "gröna drakar är glada": $u \wedge \neg \forall z(\text{grön}(z) \rightarrow \neg \text{ledsen}(z)) \equiv (u \wedge \text{grön}(a) \wedge \text{ledsen}(a))$,

dvs på följande formel (som skall uppfattas som allkvantifierad)

$$\begin{aligned} &(\text{barn}(b(x), x) \vee \neg \text{ledsen}(x)) \wedge \\ &(\neg \text{kanFlyga}(b(x)) \vee \neg \text{ledsen}(x)) \wedge \\ &(\neg \text{grön}(x_1) \vee \text{kanFlyga}(x_1)) \wedge \\ &(\neg \text{barn}(x_2, y) \vee \neg \text{grön}(y) \vee \text{grön}(x_2)) \wedge \\ &\text{grön}(a) \wedge \text{ledsen}(a) \end{aligned}$$



Litteraturförteckning

Aho–Sethi–Ullman, Compilers, Principles, Techniques, and Tools, ADDISON-WESLEY 1986

Aho–Ullman, The Theory of Parsing, Translation and Compiling Vol. 1 & 2, PRENTICE-HALL 1972

Beckman, Mathematical Foundations of Programming, ADDISON-WESLEY 1981

Hopcroft–Ullman, Introduction to Automata Theory, Languages, and Computation, ADDISON-WESLEY 1979

Lewis–Papadimitriou, Elements of the Theory of Computation, PRENTICE-HALL 1981

Manna, Mathematical Theory of Computation, McGRAW-HILL 1974

Martin, Introduction to Languages and the Theory of Computation, McGRAW-HILL 1991

Minsky, Computation, Finite and infinite Machines, PRENTICE-HALL 1967

Odifreddi, Classical Recursion Theory, NORTH-HOLLAND 1989

Salomaa, Computation and Automata, CAMBRIDGE UNIVERSITY PRESS 1985

Index

A-C

acceptera 123
Ack Ackermanns funktion 154
Ackermann 142, 153
addera 126
alfabet 15
algoritmiska koncept 163
aritmetiska uttryck 104
associativa lagen 17, 221
Automater
 finita, se Finita automater
 pushdown, se Pushdownautomater
 Turingmaskiner, se Turingmaskiner
avgöra 123
balanserade parentessträngar 76, 84, 88, 89, 90, 100
basfallet 16
bitsträng 9
blanktecken 118
blankteckensvans 118
bottom-up parser 88
CFG 106
Chomsky 82, 106
Church 142
Churchs tes 163
computer 110
CSG 106

D-F

de Morgans regel 49, 197, 267
delbarhet 151
delmängdskonstruktionen, Se Finita automater
deterministisk 30, 98
DFA, Se Finita automater
DPDA 98
Driva

 driva en DFA, Se Finita automater
 driva en NFA, Se Finita automater
 driva en PDA, Se Pushdownautomater
 driva en TM, Se Turingmaskiner
Entscheidungsproblem 110, 181
Fib, Fibonaccifunktionen 156
Fib, se Primitivt rekursiva funktioner
Finita automater 26–71
 delmängdskonstruktionen 38
 deterministisk 30
 DFA 30
 driva en DFA 34
 driva en NFA 37
 GFA 45
 glupsk 35
 hänga sig 34
 Mealymaskin 66
 minimering av 50
 minne 63
 Mooremaskin 65
 NFA 36
 parallellkoppling 43
 seriekoppling 43
 skräptillstånd 32
 utvidgad övergångsfunktion 33
flitiga bävern 187
framåtrekursion 156, 160
funktionella språk 173
förlängningsregler 80

G-J

GFA 45
Grammatiker
 av typ 0 – 3 106
 sammanhangsfria 75
 sammanhangskänsliga 106
Gödel 109, 142, 179

heltalsfunktion 10
 Hilbert 11, 110, 142
 Hilberts hotell 12
 hänga sig 34, 108
 härledningsträd 80
 ickedeterminism 138
 icketerminerande noder 81
 icketerminerande symboler 74
 icketrivial egenskap 184
 if-then-else 149
 induktionssteget 16
Jämn 157

K-N
 Kleene 142
 Kleenestjärna 74
 Kleenestjärnatillslutning 17, 20
 konfiguration 118
 konsumtion 73
KvadratRot, se Primitivt rekursiva funktioner
 kvintupeldefinition
 av en DFA 31
 av en NFA 36
Kvot, definierad med framåtrekursion 162
Lika, se Primitivt rekursiva funktioner
 logikspråk 173
 loop-program 174
 McCarthy, John 149
 Mealymaskin 66
 minimering, Se Finita automater
 minimeringsalgoritm
 genom dubbel reversering 57
 minne 63
 minne, oändligt 109
 Mooremaskin 65
Mult, se Primitivt rekursiva funktioner
 mängdbegreppet
 delmängd 266
 mängdoperationer
 differens 266
 kartesisk produkt 267
 komplement 267
 snitt 266

union 266
 naturligt tal 9
 NFA, Se Finita automater
Nummer, se Primitivt rekursiva funktioner

O-Q

Oettinger 72, 82
Olika, se Primitivt rekursiva funktioner
Om, se Primitivt rekursiva funktioner
 oändlig sträng 11
 palindrom 21, 69
 parallellkoppling 43
 parentessträng 124
 partiellt definierad funktion 11
 partiellt definierade funktioner 161
 PDA 82
 periodisk funktion 171
 periodiska mönster 62
 pop 83
Pot, se Primitivt rekursiva funktioner
 predikat 148
 prefix 16
 prefixegenskapen 21
 äkt prefix 16
Prefix(L), Se *Språk*
 primitiva loop-program 174
 Primitivt rekursiva funktioner 143–152
 Basfunktionen *Öka* 143
 den primitivt rekursiva mallen 145
 Fib 156
 Lika 149
 Mult 147
 Nummer 150
 Olika 149
 Om 149
 Pot 147
 predikat 148
 Rest 151
 sammansättning 143
 TriangelTal? 150
 X 150
 Y 150
Primtal 159, 161

primtal 62, 105
 procedurella språk 173
 produktion 73
 produktionsträd 80
 programmeringsspråk 173
 pumpsats
 för reguljära språk 60
 för sammanhangsfria språk 94
 Pushdownautomater 82– 102
 bottom-up parser 88
 definition 85
 deterministiska 98
 DPDA 98
 driva en PDA 85
 härma en PDA med en CFG 90
 sextupeldefinition 85
 tillståndsovergång 83
 top-down parser 87
 övergångsmängden 85
 Péter 142

R

Rado 187
 Reguljära språk 23, 106
 periodiska mönster 62
 prefixspråk av reguljära språk 50
 pumpsatsen 60
 pumpsatsen i termer av produktion 93
 reverserade reguljära språk 49
 slutenhetsegenskaper 48
 suffixspråk av reguljära språk 49
 reguljära uttryck 23
 rekursion 16, 73
 rekursiva flätor 157
 Rekursiva funktioner 160– 171
 kan härma Turingmaskiner 169
 kan härmas av Turingmaskiner 170
 klassen av 161
 repetera 16
 Rest 140
Rest, se Primitivt rekursiva funktioner
 Restriktionsfria språk 106
 en grammatik som härmar en TM 130

reversera 16
 Rices sats 184

S

sammanfoga 15
 sammanhang 78, 107
 Sammanhangsfria grammatiker 75
 härma en PDA med en CFG 90
 Sammanhangsfria språk
 gränser 93
 pumpsatsen 94
 sammanhangsfria grammatiker 75
 sammanhangskänslig 106
 semantik 10
 seriekoppling 43
 shiftning 166
 skräptillstånd, Se Finita automater
 Språk 17– 21
 differens 18
 Kleenestjärnatillslutning 19, 20
 komplement 18
 Operationer på språk 18
 Prefix(L) 20
 sammanfogning 19
 snitt 18
 största språket över ett alfabet 17
 Suffix(L) 20
 tomma språket 18
 union 18
 stack 83
 stopp-problemet 180
 sträng 15
 strängproduktion 73
 suffix 16
Suffix(L), Se *Språk*
 Syntax 10
 syntaxdiagram 103
 sålänge-loopar 178
 Särskilja
 strängar 50, 52
 Särskiljandesatsen 60
 tillstånd 55

T

tape 117

teckenletare 122
 terminerande noder 81
 terminerande symboler 74
 TM, Se Turingmaskiner
 tomma strängen 15
 top-down parser 87
 triangeltal 126
TriangelTal? predikat 150
 trivial egenskap 184
 Turing 109, 110, 180, 181
 Turingmaskiner 109– 139
 acceptera 123
 avgöra 123
 beräkna funktioner 125
 blanktecken 118
 blankteckensvansen 118
 den universella 133
 driva en TM 120
 ekvivalens med restriktionsfria
 grammatiker 129
 hängning 120
 kan härma rekursiva funktioner 170
 kan härmas med hjälp av rekursiva
 funktioner 169
 konfiguration 118
 sextupeldefinition 119
 stopptillstånd 117
 tape 117
 teckenletare 122
 varianter 135
 övergångsfunktionen 119
 Turings tes 110, 163

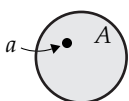
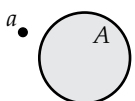
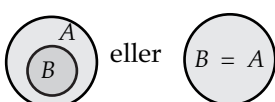
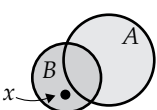
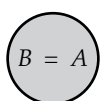
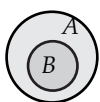
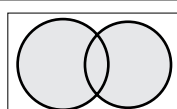
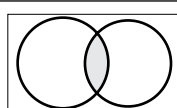
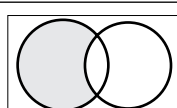
U-Ö

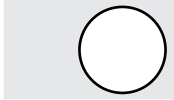
Udda 157
 universella Turingmaskinen 133
 unär 25
 uppräknelig 14
 variabel 174
 X, se Primitivt rekursiva funktioner
 Y, se Primitivt rekursiva funktioner
 övergångsfunktion 119
 överuppräknelig 14

Beteckningar

<i>Notation</i>	<i>Uttalas</i>	<i>Representerar</i>	<i>Sid</i>
\mathbb{N}		mängden av naturliga tal	9
ε	epsilon	tomma strängen	15
\emptyset		tomma mängden	18
σ	(lilla) sigma	tecken	15
Σ	(stora) sigma	ett alfabet (mängd av tecken)	15
Γ	(stora) gamma	ett (tape)alfabet	15, 85, 119
L		ett språk, eller vänstergående TM	17, 121
$L_1 L_2$		sammanfogning av två språk	19
L^n	L repeterad n gånger	repetition av ett språk	19
L^*	L -stjärna	Kleenestjärnatillslutning	20
L^+	L -plus	L^* förutom tom sträng	20
M		automat	31, 85, 119
$L(M)$		det språk som en automat M accepterar	34, 86, 121
Q		tillståndsmängd	31, 85, 119
R		högergående TM	121
R_a, L_a		teckenletande TM	122
S_R		högershiftande TM	122
δ	delta	övergångsfunktion	31, 119
δ^*	delta-stjärna	utvidgad övergångsfunktion	33
Δ	stora delta	övergångsrelation	36, 85
\Rightarrow^*		strängproduktion i noll eller flera steg	74
G		grammatik	74
$L(G)$		det språk som en grammatik G beskriver	74
CFG		sammanhangsfri grammatik	78
CFL		sammanhangsfritt språk	78
PDA		pushdownautomat	85
$DPDA$		deterministisk pushdownautomat	98
DFA		deterministisk finit automat	31
NFA		(ickedeterministisk) finit automat	36
GFA		generaliserad finit automat	45
TM		Turingmaskin	119
h		stopptillstånd	119
$\#$		blanktecken	118

Elementära mängdbegrepp

<i>Notation</i>	<i>Uttalas</i>	<i>Betyder</i>	<i>Illustration</i> (med s.k. Vennndiagram)
$a \in A$	a tillhör A	a är ett element i A	
$a \notin A$	a tillhör inte A	a är inte ett element i A	
$B \subseteq A$	B är en delmängd av A	Om $x \in B$, så $x \in A$	
$B \not\subseteq A$	B är inte en delmängd av A	Det finns något x så att $x \in B$ och $x \notin A$	
$A = B$	A är lika med B	$A \subseteq B$ och $B \subseteq A$	
$B \subset A$	B är en äkta delmängd av A	$B \subseteq A$ men $B \neq A$	
$A \cup B$	A union B	$\{x \mid (x \in A) \text{ eller } (x \in B)\}$	
$A \cap B$	A snitt B	$\{x \mid (x \in A) \text{ och } (x \in B)\}$	
$A - B$	A förutom B (mängddifferensen mellan A och B)	$\{x \mid x \in A \text{ och } x \notin B\}$	

<i>Notation</i>	<i>Uttalas</i>	<i>Betyder</i>	<i>Illustration</i> (med s.k. Venndiagram)
\bar{B} eller C_B	komplementet av B med avseende på något s.k. mängd- univers [†] .	Mängduniverset $- B$	
$A \times B$	den kartesiska pro- dukten av A och B	$\{(a, b) a \in A \text{ och } b \in B\}$ dvs elementen i $A \times B$ är s.k. ordnade par (a, b) av element, av vilka det första kommer från A , och det andra från B .	

†. Ett mängdunivers är en mängd som innehåller alla element som man för tillfället sysslar med.

De Morgans regler

$$\overline{A \cup B} = \bar{A} \cap \bar{B}$$

$$\overline{A \cap B} = \bar{A} \cup \bar{B}$$