Diskreta strukturer

1 Introduktion

När vetenskapsmän och ingenjörer gör modeller av verkligheten använder de matematik. För fysiker, kemister och maskintekniker är modellerna ofta *kontinuerliga* och beskrivs med differentialekvationer, integraler, vektorfält, analytiska funktioner etc. Datatekniker använder i stor utsträckning *diskreta* modeller med mängder, träd, grafer, formella språk, ändliga tillståndsmaskiner, logik etc. Den linjära algebran ligger däremellan med vektorer och matriser som har ett ändligt antal element som ibland representerar diskreta och ibland kontinuerliga storheter.

Denna kurs handlar primärt om att göra programmässiga modeller och abstraktioner. Eftersom datorer är digitala måste alla modeller i datorn vara diskreta. Modeller är i grunden matematiska abstraktioner. Det gäller både i diskreta och kontinuerliga sammanhang.

I utbildningsmålen för civilingenjörsexamen anges bland annat att studenten skall ha kunskap om det valda teknikområdets vetenskapliga grund och kunskaper i matematik och naturvetenskap. För en datateknikingenjör utgör den diskreta matematiken en lika viktig del av den vetenskapliga grunden som den kontinuerliga matematiken.

1.1 Konventioner

Symbolen = används på flera olika sätt i matematiken. En programmerare skulle säga att symbolen är *överlastad* (eng. overloaded). När man skriver $x = y$ betyder det ofta att x och y har samma värde. När vi skriver $1 + 1 = 2$ så är det en likhet som gäller; uttrycken på vänster och höger sida har samma värde. I programspråk måste vi själva definiera en metod som `boolean equals(Object)` för att avgöra om två objekt är "lika". För fördefinierade typer som `int` och `double` används olika maskininstruktioner för att bestämma om likhet råder eller ej. När vi skriver $x^2 = -1$ är det en ekvation som kan ha noll eller flera lösningar. I det här fallet är antalet noll om x är ett reellt tal och två det är ett komplext tal. I $e = \sum_{i=0}^{\infty} \frac{1}{i!}$ är det en definition av talet e. Med $f(x) = x + 1$ definierar vi en funktion med namnet f. Namnet på variabeln x är ej signifikant.

Det kan vara förvillande att använda samma symbol för helt olika ändamål. I denna text kommer vi att därför att använda symbolen $\triangleq$ när vi definierar någonting:

$$\begin{aligned} e &\triangleq \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \\ f(x) &\triangleq x + 1 \end{aligned}$$

I den andra definition är det funktionen f som definieras. Det skulle vara tydligare om man kunde skriva $f \triangleq \ldots$. I avsnittet om funktioner kommer vi att introducera sådana möjligheter.

När definitionen väl är gjord gäller att namnet och det definierande uttrycket är lika, till och med i omvänd ordning.

$$\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e$$

Vi sätter apostrofer kring ett ord när texten handlar om ordet och inte det som ordet står för. Vi kan alltså skriva att 'sats' har fyra bokstäver. Med formuleringen 'ordet sats har fyra bokstäver' fyller 'ordet' samma funktion. När vi skriver citationstecken kring ett ord innebär detta att vi använder ordet på ett vårdslöst eller ironiskt sätt. När vi ovan skriver "lika" beror detta på att vi inte definierat vad som menas med att två objekt är lika.

2 Satslogik

2.1 Motivering

Boolesk algebra handlar om hur man räknar med de logiska operatorerna 'icke', 'och' och 'eller'. Vi hoppas att läsaren redan kan det, men om någon känner sig osäker så återfinns reglerna nedan.

Satslogiken handlar också om logiska uttryck, men tillämpningsområdet är mer att modellera logiska resonemang. Därmed får implikationsoperatorn en framträdande plats. I nästa kapitel kommer vi att generalisera till något som kallas *predikatlogik*. Med predikatlogiken kan vi formalisera matematiska satser och bevis. Denna abstraktion är lätt att översätta till objektorienterade modeller och illustrerar hur symbolmanipulerande program som Maple konstrueras. De ger också en ide om hur man kan bygga in "intelligens" i datorprogram. På köpet får vi en definition av vad som menas med ett formellt bevis och bättre förståelse för vad ett vanligt matematiskt bevis egentligen är. Vi inför regler som formaliserar vårt sätt att resonera om logiska uttryck som ansluter till det sätt som man bevisar satser i matematiken.

I texten förekommer en del logiska uttryck som är mer komplicerade än de som bör användas i datorprogram. Motivet är att diskutera hur uttryck skall tolkas i frånvaro av parenteser. Vi föredrar att göra detta i ett sammanhang där konventionerna inte är lika etablerade som för aritmetiska uttryck. Detta är relevant att förstå när man skriver eller använder program som kommunicerar med användaren med hjälp av någon form av uttryck. Det förekommer till exempel i vanliga kalkylprogram och i frågespråk för databassystem.

2.2 Mål

Efter att ha studerat detta kapitel och arbetat med förelagda problem och programmeringsuppgifter skall du kunna

1. översätta påståenden i naturligt språk till satslogisk notation.
2. konstruera sanningstabeller för satslogiska uttryck.
3. avgöra om ett bevisträd är korrekt konstruerat.
4. analysera ett uttryck när regler för precedens och associativitet är givna.

2.3 Implikation i matematiken

I matematiken bevisar man satser. En sats har ofta formen "Om P så Q" där P är förutsättningar eller antaganden och Q är slutsatsen. Ibland skriver man $P \Rightarrow Q$ och säger att P implicerar Q. När man har bevisat en sats på denna form så kan man använda den i ett sammanhang där man vet att P sann och kan då dra slutsatsen att Q är sann utan att titta på beviset.

Vi tar ett exempel som handlar om primtal.

Två tal är *primtalstvillingar* om båda är primtal och skillnaden mellan dem är 2. De första primtalstvillingarna är alltså $(3, 5), (5, 7), (11, 13)$ och $(17, 19)$. Vi sätter namn på två påståenden som handlar om primtal.

- $P_0 \triangleq$ det finns oändligt många primtalstvillingar
- $Q_0 \triangleq$ det finns oändligt många primtal

Sats.

$$P_0 \Rightarrow Q_0$$

■

Det är lätt att bevisa satsen. Eftersom vi har antagit att det finns oändligt många primtalstvillingar så kan vi plocka ut den minsta tvillingen ur varje par och vi står där med oändligt många primtal.

Är antagandet sant? Det är ingen som vet! De största kända primtalstvillingarna har drygt 100 000 siffror, men ingen har ännu lyckats bevisa att det finns oändligt många.

Är slutsatsen sann? Ja, det finns ett annat bevis för detta påstående. Det var Euklides som konstruerade det för 2400 år sedan.

Vi ser alltså att man mycket väl kan bevisa satsen $P \Rightarrow Q$ även om P inte kan vara sann. Det betyder förstås att man inte kan använda satsen för att dra slutsatsen Q.

En sats till på samma tema:

- $P_1 \triangleq$ det finns ändligt många primtalstvillingar
- $Q_1 \triangleq$ det finns ett största primtalstvillingpar

Sats.

$$P_1 \Rightarrow Q_1$$

■

Denna sats är nästan självklar; den bygger på egenskapen att ändlig mängd tal har ett största element. Förmodligen är det så att P_1 är falsk och då är också Q_1 falsk. Den kan alltså vara så att både premiss och slutsats kan vara falska i en sats som vi kan bevisa. Det enda som inte kan inträffa är vi kan bevisa en sats $P \Rightarrow Q$ där det kan vara så att P är sann medan Q är falsk.

När man använder $\Rightarrow$ i matematiken, $P \Rightarrow Q$, så är $\Rightarrow$ en *relation*. Vi återkommer till relationer i kapitel 8. I satslogiken har vi en operator, $\rightarrow$, som också kallas implikation. I matematiken är också $<$ en *relation* och vi skriver bara $x < y$ om x är mindre än y. I Java och andra programspråk är $<$ en operator och $x < y$ är ett logiskt uttryck vars värde kan vara sant eller falskt.

När man använder $\Rightarrow$ i matematiken, $P \Rightarrow Q$, så finns det alltid en orsaksrelation mellan P och Q. I satslogiken skriver vi $P \rightarrow Q$ och det behöver inte finnas någon relation alls mellan P

och Q. P och Q innehåller variabler som kan anta sanningsvärden, men vi bortser helt från vad variablerna står för.

Eftersom vi vill använda satslogiken och senare predikatlogiken för att modellera logiska resonemang och matematiska bevis så vill naturligtvis definiera den satslogiska implikationsoperatorn, $\rightarrow$, så att den fungerar på samma sätt som den matematiska implikationssymbolen, $\Rightarrow$. Vi kommer sålunda att definiera $P \rightarrow Q$ så att det är falskt bara när P är sann och Q är falsk.

2.4 Tillämpning

Inför kvartsfinalerna i Europamästerskapen i fotboll 2008 gällde att

> Om Sverige vinner över eller spelar oavgjort mot Ryssland så går Sverige till kvartsfinal.

Detta är en sammansättning av tre stycken påståenden. Det blir tydligare med formuleringen

> Om Sverige vinner över Ryssland eller om Sverige spelar oavgjort mot Ryssland så går Sverige till kvartsfinal.

Satslogiken handlar om sådana påståenden där man abstraherat bort innebörden i påståendena och bara intresserar sig för strukturen. I vårt exempel har vi

$$(p \vee q) \rightarrow r$$

där p, q och r är påståenden som kan vara falska eller sanna och $\vee$ står för 'eller' och $(p \vee q) \rightarrow r$ utläses 'p eller q implicerar r'. På svenska formulerar vi det ofta som 'om p eller q så r'.

Aktivitet 1.
Konstruera ett uttryck som avspeglar strukturen i

> Om Sverige vinner mot Ryssland så får Sverige möta Holland i kvartsfinalen och om Sverige förlorar så får Ryssland möta Holland.

2.5 Grundläggande begrepp

Satslogiken (eng. propsitional logic) handlar om hur man hur drar *slutsatser* (eng. conclusion) från givna förutsättningar, *premisser*. Premisser och slutsatser formuleras som *satslogiska uttryck* med variabler och operatorer. Operatorerna, särskilt de med två operander, kallas *konnektiv* (eng. connective).

Variablerna antar *sanningsvärden* varav det finns två stycken, *sanning* (eng. truth) och *falskhet* (eng. falsity). Vi använder T och F som beteckningar för sanningsvärdena.

Det enklaste satslogiska uttrycket består av en ensam variabel, en *satsvariabel* (eng. statement letter). Vi använder bokstäverna från p och framåt för sådana variabler. När vi tillämpar satslogiken står sådana variabler för påståenden som är sanna eller falska. I ett resonemang om en triangel så är 'triangeln är rätvinklig', 'triangeln är liksidig' och 'triangeln är likbent' sådana påståenden.

De viktigaste operatorerna är $\neg$, $\wedge$, $\vee$, $\rightarrow$ och $\leftrightarrow$ som utläses 'icke', 'och', 'eller', 'implicerar' resp. 'är ekvivalent med'.

När vi bildar satslogiska uttryck använder vi parenteser på vanligt sätt för att klargöra vad som är operander till ett konnektiv. Vi använder versaler från P och framåt för att beteckna satslogiska uttryck.

Om P och Q är satslogiska uttryck så är $\neg P$, $(P \wedge Q)$, $(P \vee Q)$, $(P \rightarrow Q)$ och $(P \leftrightarrow Q)$ också satslogiska uttryck. Detta är en *induktiv* eller rekursiv beskrivning av hur man konstruerar satslogiska uttryck.

De olika sorternas uttryck har namn: $\neg P$ kallas för en *negation*, $(P \wedge Q)$ *konjunktion*, $(P \vee Q)$ *disjunktion*, $(P \rightarrow Q)$ *implikation* och $(P \leftrightarrow Q)$ *ekvivalens*.

Vi kan konstatera att

p	q	$\neg p$	$(p \wedge q)$	$(p \vee q)$
$(p \rightarrow q)$	$(p \leftrightarrow q)$	$(\neg p \vee p)$	$\neg(p \wedge q)$	$\neg\neg p$
$(p \wedge (p \vee q))$	$(\neg p \wedge \neg(p \vee q))$			

är satslogiska uttryck. Enligt definitionen är följande strängar **inte** satslogiska uttryck, även om ingen skulle missförstå vad som menas:

(p) $\quad \neg(\neg p)$ $\quad ((p \vee \neg p) \leftrightarrow \mathsf{T})$

Enligt den formella definitionen kommer komplicerade uttryck att innehålla många parenteser. Vi kommer senare att introducera regler för hur uttryck skall tolkas i frånvaro av de föreskrivna parenteserna. Tills dess nöjer vi oss med att tillåta att man utesluter det yttersta parentesparet i ett sammansatt uttryck.

För givna värden på variablerna i ett satslogiskt uttryck har hela uttrycket också ett sanningsvärde. Reglerna för hur man räknar ut detta värde är utformade så att de ansluter till normal användning av orden 'inte', 'och' etc.

Den logiska icke-operatorn, $\neg$, fungerar på samma sätt som *inte* i vanligt språk. Om p har värdet sanning så har $\neg p$ värdet falskhet och omvänt. Vi kan beskriva reglerna för alla konnektiven i en *sanningstabell*.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
F	F	T	F	F	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
T	T	F	T	T	T	T

Om ett uttryck har värdet sanning för givna värden på variablerna så säger vi att det är *sant*. Omvänt, om det har värdet falskhet så säger vi att det är *falskt*.

Den första raden ger värdena för sammansatta uttryck under förutsättning att både P och Q har värdet falskhet.

Och-operatorn $\wedge$ fungerar ofta på samma sätt som 'och' i naturligt språk; $P \wedge Q$ är sant precis då både P och Q är sanna. I naturligt språk kan 'och' även ha andra innebörder. Påståendet 'han kom in i rummet och han satte sig på stolen' betyder inte samma sak som 'han satte sig på stolen och han kom in i rummet'. Man måste förstå meningen för att inse att 'och' i detta sammanhang handlar om händelser som sker efter varandra. Sådant kan inte modelleras med den logiska operatorn.

Aktivitet 2.
Komplettera sanningstabellen

P	Q	$\neg P$	$\neg P \vee Q$
F	F		
T	F		
F	T		
T	T		

Överensstämmer den sista kolonnen med någon kolonn i föregående tabell?

I vanligt språkbruk är *eller* inte lika entydigt. Ibland betyder 'A eller B' att exakt en av A och B är sanna och ibland att minst en av dem är sanna. I satslogiken är det den senare betydelsen som gäller för eller-operatorn $\vee$.

Aktivitet 3.
Är $(p \wedge q) \vee r = p \wedge (q \vee r)$ för alla värden på p, q och r? Komplettera hela tabellen eller fyll i en rad som visar att uttrycken kan ha olika värden.

p	q	r	$(p \wedge q) \vee r$	$p \wedge (q \vee r)$
F	F	F		
F	F	T		
T	F	F		
T	F	T		

p	q	r	$(p \wedge q) \vee r$	$p \wedge (q \vee r)$
F	T	F		
F	T	T		
T	T	F		
T	T	T		

Aktivitet 4.
Sanningstabell för $(P \wedge Q) \rightarrow P$. Använd den för att motivera de två återstående raderna i sanningstabellen för $\rightarrow$.

P	Q	$P \wedge Q$	$(P \wedge Q) \rightarrow P$
F	F		T
T	F		T
F	T		T
T	T		T

Den sista operatorn är *ekvivalens*, $\leftrightarrow$. $P \leftrightarrow Q$ är sant precis då P och Q har samma värde.

2.6 Ställighet

Alla är bekanta med de fyra vanliga aritmetiska operatorerna, $+, -, \cdot$ och $/$. Ofta har en operator två operander och lämnar ett resultat som är av samma typ som operanderna, men det är inget krav. En operator med två operander kallas för en *binär* operator.

Det finns också operatorer med en operand, *unära* operatorer. Den logiska operatorn $\neg$ är en sådan. Det finns också en aritmetisk unär operator, $-$, som har samma symbol som den binära subtraktionsoperatorn. I uttrycket $-x$ har vi använt den unära operatorn medan $0 - x$ har den

binära operatorn även om uttryckens värden är desamma. Att det är en fundamental skillnad på unära och binära operatorer framgår tydligt av det meningslösa $P\neg Q$, som inte är ett uttryck.

Det är inget som hindrar att man definierar operatorer med fler än två operander. På svenska talar man om n-ställiga (eng. n-ary) operatorer som har n operander. Antalet operander kallas för *ställigheten* (eng. arity). Flerställiga operatorer kan inte anges med en ensam symbol. I matematiken är det ovanligt med flerställiga operatorer, men i Java finns det en treställig villkorsoperator, `e1?e2:e3`.

Aktivitet 5.
Antag att P, Q och R är logiska uttryck. Gör en sanningstabell för $P\,?\,Q : R$.

P	Q	R	$P\,?\,Q : R$
F	F	F	
T	F	F	
F	T	F	
T	T	F	

P	Q	R	$P\,?\,Q : R$
F	F	T	
T	F	T	
F	T	T	
T	T	T	

Definiera $P\,?\,Q : R$ med hjälp av de vanliga operatorerna,

$$P\,?\,Q : R \quad \triangleq$$

2.7 Precedens

Om man skriver $p \wedge q \rightarrow p$ är det oklart om q är en operand till $\wedge$ eller $\rightarrow$. Detta var anledningen till att vi använde parenteser när vi angav reglerna för hur satslogiska uttryck bildas. $(p \wedge q) \rightarrow p$ och $p \wedge (q \rightarrow p)$ har olika betydelse.

För att slippa en massa parenteser i större uttryck inför man ofta konventioner för hur starkt en operator binder i förhållande till andra operatorer. Man talar om operatorns *precedens* och att en operator har högre precedens en annan eller *precederar över* den.

I aritmetiken har multiplikationsoperatorn, $\cdot$, högre precedens än additionsoperatorn, $+$. Uttrycket $x + y \cdot z$ tolkas alltid som $x + (y \cdot z)$ och $x \cdot y + z$ betyder $(x \cdot y) + z$. Binärt $-$ och $+$ har samma precedens. När det förekommer flera operatorer med samma precedens i ett uttryck behövs regler för hur dessa binder. Vi diskuterar detta i avsnittet om associativitet (sid. 10).

I satslogiken finns ingen allmänt vedertagen konvention om precedens. Vissa författare använder inga precedensregler och sätter ut alla parenteser. En del inför några precedensnivåer medan andra ger en unik precedens för varje operator. Här sällar vi oss till den sista gruppen men tillåter oss att sätta ut parenteser när vi tror att det underlättar för läsaren. Med ordning från högst till lägst precedens har vi

$$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

Detta betyder att $p \rightarrow q \leftrightarrow \neg p \vee q$ skall tolkas som $((p \rightarrow q) \leftrightarrow (\neg p \vee q))$.

Likhetsoperatorn, $=$, har nästan alltid lägst precedens av alla operatorer och man brukar inte ens ange att så är fallet när man definierar likhet. Man behöver nästan aldrig sätta ut parenteser kring de två leden i en ekvation.

Aktivitet 6.
Skriv om med alla parenteser:

$$p \wedge q \vee r \rightarrow \neg s \vee t \leftrightarrow u \quad =$$
$$\neg p \rightarrow q \leftrightarrow r \vee \neg s \quad =$$

Tag bort alla onödiga parenteser:

$$(p \vee (\neg q \wedge ((r \rightarrow s) \leftrightarrow (t \rightarrow u)))) \quad = \quad p \vee \neg q \wedge (r \rightarrow s \leftrightarrow t \rightarrow u)$$

2.8 Tautologier

Ett påstående som är sant för all värden på de ingående variablerna kallas för *logiskt giltigt* (eng. valid) eller en *tautologi*. Ett exempel på en sådan är $p \vee \neg p$. Om P är en tautologi skriver vi

$$\models P$$

Det är enkelt att avgöra om ett påstående är en tautologi med hjälp av en sanningstabell.

p	$p \vee \neg p$
F	T
T	T

Aktivitet 7.
Det är inte uppenbart att $R \triangleq (\neg p \rightarrow \neg q) \rightarrow ((\neg p \rightarrow q) \rightarrow p)$ är en tautologi, men en sanningstabell verifierar att så är fallet.

p	q	$\neg p$	$\neg q$	$\neg p \rightarrow \neg q$	$\neg p \rightarrow q$	$(\neg p \rightarrow q) \rightarrow p$	R
F	F						
T	F						
F	T						
T	T						

Ett påstående som alltid är falskt kallas för en *motsägelse* (eng. contradiction). Den enklaste motsägelsen är $p \wedge \neg p$. Negationen till en tautologi är en motsägelse.

Ett påstående som är sant för någon uppsättning av värden på variablerna kallas *satisfierbart* (eng. satisfiable).

2.9 Räknelagar

En del tautologier kan användas för att omforma eller förenkla satslogiska uttryck utan att deras värde förändras. Det är lätt att kontrollera att $\neg\neg p$ och p alltid har samma värde. De är alltså ekvivalenta och $\neg\neg p \leftrightarrow p$ är en tautologi. Det är lätt att inse att $\neg\neg P \leftrightarrow P$ blir en tautologi när P ersätts med ett godtyckligt satslogiskt uttryck. När vi formulerar en räknelag så gör vi det i form av ett sådant schema. I stället för att tala om ekvivalenser och tautologier formulerar vi lagarna som ekvationer och en regel säger att vänster och höger led alltid är lika.

Sats.

$$\begin{array}{cc} P \wedge Q = Q \wedge P & P \vee Q = Q \vee P \\ (P \wedge Q) \wedge R = P \wedge (Q \wedge R) & (P \vee Q) \vee R = P \vee (Q \vee R) \\ \neg\neg P = P & P \to Q = \neg P \vee Q \end{array}$$

■

Enligt den första raden kan man kasta om ordningen mellan operanderna i en konjunktion och en disjunktion utan att uttryckets värde förändras. En binär operator med den egenskapen säges vara *kommutativ*. Generellt gäller alltså för en kommutativ operator, $\star$, att $a \star b = b \star a$ för alla tillåtna värden för a och b.

Man säger att en binär operator, $\star$, är *associativ* om $a \star (b \star c) = (a \star b) \star c$ för alla värden på a, b och c. Räknelagarna i den andra raden visar att $\wedge$ och $\vee$ är associativa.

När man använder räknelagarna är det viktigt att man inte utan eftertanke utelämnar parenteser när man ersätter P och Q med konkreta uttryck. Om man ersätter P med $p \to p$ och Q med q i den första räknelagen måste man skriva $(p \to p) \wedge q = q \wedge (p \to p)$. Utan parenteser är de två leden inte ekvivalenta.

Aktivitet 8.
Visa att $p \to p \wedge q$ och $q \wedge p \to p$ inte är ekvivalenta.

Varje räknelag i föregående sats handlar bara om en operator. Det finns många lagar som handlar om uttryck där två operatorer är inblandade. Här följer de viktigaste.

Sats.

$$\begin{array}{cc} P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R) & P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R) \\ \neg(P \wedge Q) = \neg P \vee \neg Q & \neg(P \vee Q) = \neg P \wedge \neg Q \end{array}$$

■

Lagarna i den första raden kallas för *distributiva* lagar, $\wedge$ *distribuerar över* $\vee$ och omvänt. Man kan ”multiplicera in” på samma sätt som i $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$. I aritmetiken distribuerar $\cdot$ över $+$, men inte vice versa.

Lagarna i den andra raden är uppkallade efter *de Morgan*. Alla räknelagarna går att bevisa mekaniskt med hjälp av sanningstabeller. När det är tre variabler inblandade behövs 8 rader i tabellen.

Aktivitet 9.
Vilka uttryck är tautologier?

$$\begin{array}{lcl} p \to (q \wedge r) & \leftrightarrow & (p \to q) \wedge (p \to r) \\ (p \wedge q) \to r & \leftrightarrow & (p \to r) \wedge (p \to q) \end{array}$$

2.10 Associativitet

I det aritmetiska uttrycket $3+2+1$ spelar det ingen roll om man beräknar det som $(3+2)+1$ eller $3+(2+1)$; resultatet blir i båda fallen detsamma. Motsvarande gäller inte för $3-2-1$. $(3-2)-1=0$ medan $3-(2-1)=2$. Konventionen föreskriver att det är den första tolkningen som gäller. Man säger att den binära operatorn $-$ är *vänsterassociativ* eller att den *associerar åt vänster*. Om två operatorer har samma precedens kan man inte föreskriva att den ena är vänsterassociativ och den andra högerassociativ. Konflikten skulle bli tydlig i uttrycket $1-2+3$. Plusoperatorn är alltså också vänsterassociativ.

Det finns också operatorer som är *högerassociativa*. Exponentieringsoperatorn är en sådan. Själva operatorn har normalt ingen egen symbol i aritmetiska uttryck; man skriver den andra operanden snett ovanför den första, x^y. I tryckt text brukar exponenten vara i en något mindre font, men när man skriver för hand blir detta inte lika tydligt. Konventionen säger att 2^{3^2} skall tolkas som $2^{(3^2)}=512$ och inte som $(2^3)^2=64$.

Huruvida en operator är vänster- eller högerassociativ är något man bestämmer för att kunna utelämna en del parenteser. Om en operator är associativ eller ej är något som man bevisar med hjälp operatorns definition.

Om en operator är associativ och ensam på sin precedensnivå spelar det ingen roll om man definierar den som vänster- eller högerassociativ och man utelämnar gärna parenteserna som i $p \wedge q \wedge r$.

Hur är det då med operatorn $\rightarrow$? Associerar den åt vänster eller höger? Svaret är att det inte finns en vedertagen konvention. Man måste alltså sätta ut parenteser $(p \rightarrow q) \rightarrow r$ eller $p \rightarrow (q \rightarrow r)$ beroende på vilket man menar eller ange vilken konvention man använder.

Aktivitet 10.
När gäller det att $(p \rightarrow q) \rightarrow r$ och $p \rightarrow (q \rightarrow r)$ är olika?

I matematiska satser och bevis använder man ofta symbolen $\Rightarrow$, som också utläses 'implicerar' eller formuleras med 'om ... så'. Eftersom 'implicerar' används på olika sätt och har olika egenskaper i satslogiken och i matematiska bevis har vi valt att använda olika symboler för dem.

När en sats har formen $A \Rightarrow B$ så är den bara intressant och användbar när A är sann medan $p \rightarrow q$ är meningsfullt för alla värden på variablerna. Satsen är sann även om A är falsk och då spelar det ingen roll om B är sann eller falsk. Det är på det viset vi valt att definiera $\rightarrow$.

I bevis skriver man ibland $A \Rightarrow B \Rightarrow C$ och drar sedan slutsatsen $A \Rightarrow C$. I vår satslogik är inte $p \rightarrow q \rightarrow r$ meningsfullt och även med utsatta parenteser är uttrycket inte ekvivalent med $p \rightarrow r$. $A \Rightarrow B \Rightarrow C$ betyder egentligen att $A \Rightarrow B$ och $B \Rightarrow C$. Vi återkommer till detta i avsnittet om relationer.

2.11 Naturlig härledning

När man bevisar någonting använder man *härledningsregler* eller *inferensregler* när man kombinerar påståenden som man tidigare bevisat eller antagit vara sanna. En vanlig härledningsregel är *avskiljningsregeln*. Om man vet att $P \rightarrow Q$ och P är sanna får man dra slutsatsen att Q är sann. Man 'skiljer av' Q i $P \rightarrow Q$. Ofta används den latinska benämningen på regeln, *modus ponens*.

En inferensregel beskrivs ibland grafiskt. Avskiljningsregeln ser då ut på följande sätt.

$$\frac{P \qquad P \to Q}{Q}\,[MP]$$

Ovanför linjen finns förutsättningarna, de påståenden som måste vara sanna för att man skall få dra slutsatsen Q. Man kan indikera namnet på regeln i anslutning till linjen. Ordningen mellan förutsättningarna är inte signifikant.

När man använder härledningsregeln skall man ersätta P och Q med satslogiska uttryck. Man säger att regeln är ett *schema* och att användningarna är *instanser*. Följande visar två instanser av modus ponens.

$$\frac{p \qquad p \to q}{q}\,[MP] \qquad \frac{(p \wedge q) \qquad (p \wedge q) \to \neg p}{\neg p}\,[MP]$$

När man använder härledningsregler är det tillrådligt att alltid sätta ut parenteser kring sammansatta uttryck och sedan ta bort dem om de inte behövs. I det här fallet kan vi undvara dem.

När man gör en matematisk *teori* måste man ange vilka härledningsregler som man får använda när man bevisar satser. I regel finns det också *axiom*, påstående som antas vara sanna utan bevis. Axiomen beskriver de grundläggande egenskaperna hos det som teorin handlar om. I nästan alla teorier ingår logiken som en integrerad del och man använder härledningsregler som formaliserar innebörden av de logiska operatorerna.

Vi skall här ange en del av de härledningsregler som man använder i matematiska bevis. Formaliseringen brukar kallas *naturlig härledning* (eng. natural deduction). Med hjälp av härledningsreglerna är det möjligt att härleda alla tautologier. Syftet med detta avsnitt är att visa hur man gör härledningar så att det blir möjligt att kontrollera dem mekaniskt, t ex med ett datorprogram. Om man bara är ute efter att bevisa att ett uttryck är en tautologi så är det enklare att mekaniskt fylla i en sanningstabell. Om det finns många variabler kan man ibland hitta genvägar genom att utnyttja strukturen på uttrycket. Det är dock inte alltid så; problemet tillhör en klass av problem som snabbt blir ohanterliga när antalet variabler ökar.

Till varje operator finns regler för att *introducera* och *eliminera* den i satslogiska uttryck. Namnen på reglerna indikerar huruvida en operator elimineras (E) eller introduceras (I).

Reglerna för $\wedge$.

$$\frac{P \wedge Q}{P}\,[\wedge_{E_1}] \qquad \frac{P \wedge Q}{Q}\,[\wedge_{E_2}] \qquad \frac{P \qquad Q}{P \wedge Q}\,[\wedge_I]$$

Reglerna är naturliga. Den första säger att om vi vet att $P \wedge Q$ är sant så får vi dra slutsatsen att P är sann. Den andra regeln säger motsvarande om Q. Den tredje regeln betyder att om vi vet att P är sann och att Q är sann så är $P \wedge Q$ sann.

Man kan kombinera härledningsregler så att slutsatserna i en eller flera regler blir förutsättningar i andra. En sådan konstruktion kallas för en *härledning* (eng. derivation). En härledning är alltså ett träd. En härledning breskriver ett logiskt resonemang där påståendena i löven är premisser som är tillräckliga för att man skall kunna dra slutsatsen i roten.

Med hjälp av de ovan presenterade reglerna går det att göra en härledning som visar att om $p \wedge q$ är sant så är $q \wedge p$ sant. Eftersom det är lätt att se vilken regel som använts i varje steg kan man välja att presentera härledningen utan indikation om detta. Eftersom ordningen mellan premisserna inte är signifikant duger den tredje härledningen lika bra.

$$\dfrac{\dfrac{p \wedge q}{q}\,[\wedge_{E_2}] \qquad \dfrac{p \wedge q}{p}\,[\wedge_{E_1}]}{q \wedge p}\,[\wedge_I] \qquad \dfrac{\dfrac{p \wedge q}{q} \qquad \dfrac{p \wedge q}{p}}{q \wedge p} \qquad \dfrac{\dfrac{p \wedge q}{p} \qquad \dfrac{p \wedge q}{q}}{q \wedge p}$$

När det finns en härledning som utifrån en mängd premisser A leder till slutsatsen Q skriver man

$$A \vdash Q$$

Detta kallas för en *sekvent* (eng. sequent). Med vårt exempel har vi visat att $\{p \wedge q\} \vdash q \wedge p$.

Aktivitet 11.
Gör en härledning som visar att $\{(p \wedge q) \wedge r\} \vdash p \wedge (q \wedge r)$.

Det finns några inferensregler som tillåter att man använt ett *hypotetiskt antagande* tidigare i härledningen. De hypotetiska antagandet skrivs inom hakparenteser när man introducerar det och strykes över när man tillämpar regeln. Man säger att man *upphäver* (eng. discharge) antagandet.

Vi möter ett hypotetiskt antagande i introduktionsregeln för $\rightarrow$ som säger att om vi kan härleda Q utifrån ett hypotetiskt antagande, P, så får vi dra slutsatsen $P \rightarrow Q$ och samtidigt stryka över hypotesen i härledningen.

$$\dfrac{\begin{array}{c}[\cancel{P}] \\ \vdots \\ Q\end{array}}{P \rightarrow Q}\,[\rightarrow_I]$$

Beteckningen $\begin{array}{c}[P] \\ \vdots \\ Q\end{array}$ symboliserar ett bevisträd där det finns ett antal löv som består av P och en rot som består av Q. Det kan finnas andra löv i bevisträdet. Hakparenteserna är till för att markera de förekomster av P som skall strykas när man tillämpar regeln. Ett exempel på ett sådant träd är

$$\dfrac{\dfrac{[p \wedge q]}{q} \qquad \dfrac{p \wedge r}{r}}{q \wedge r} \quad \text{som symboliseras av} \quad \begin{array}{c}[p \wedge q] \\ \vdots \\ q \wedge r\end{array}$$

När vi tillämpar härledningsregeln stryker vi $p \wedge q$ och får

$$\dfrac{\dfrac{\dfrac{[\cancel{p \wedge q}]}{q} \qquad \dfrac{p \wedge r}{r}}{q \wedge r}}{p \wedge q \rightarrow q \wedge r}$$

Den enda kvarstående förutsättningen är $p \wedge r$. Hela härledningen sammanfattas av sekventen $\{q \wedge r\} \vdash p \wedge q \to q \wedge r$.

I det generella fallet kan $\begin{array}{c}[P]\\ \vdots \\ Q\end{array}$ innehålla noll, en eller flera löv som är P. Härledningen av Q är ett träd och det är tillåtet att introducera det hypotetiska antagandet på noll, ett eller flera ställen. Några enkla exempel på användning av regeln:

$$\frac{\cancel{[p]}}{p \to p}\,[\to_I] \qquad \frac{\dfrac{\cancel{[p \wedge q]}}{p}\,[\wedge_{E_1}]}{p \wedge q \to p}\,[\to_I] \qquad \frac{\dfrac{\dfrac{\cancel{[p \wedge q]}}{q}\,[\wedge_{E_2}] \quad \dfrac{\cancel{[p \wedge q]}}{p}\,[\wedge_{E_1}]}{q \wedge p}\,[\wedge_I]}{p \wedge q \to q \wedge p}\,[\to_I] \qquad \frac{\dfrac{\cancel{[p]}}{p \to p}\,[\to_I]}{q \to (p \to p)}\,[\to_I]$$

I det sista exemplet finns inget löv som är q.

Vi har därmed visat att $\emptyset \vdash p \to p$, $\emptyset \vdash p \wedge q \to p$, $\emptyset \vdash p \wedge q \to q \wedge p$ och $\emptyset \vdash q \to (p \to p)$. När premissmängden är tom utelämnar man den ofta: $\vdash p \to p$, $\vdash p \wedge q \to p$, $\vdash p \wedge q \to q \wedge p$ och $\vdash q \to (p \to p)$.

Eliminationsregeln för $\to$ är precis det som vi kallade för modus ponens i inledningen.

$$\frac{P \qquad P \to Q}{Q}\,[\to_E]$$

Aktivitet 12.
Låt oss använda reglerna för att visa att $\{p \to q, q \to r\} \vdash p \to r$. Härledningen är påbörjad med ett hypotetiskt antagande. Komplettera härledningen och indikera vilka regler som använts.

$$\frac{\underline{[p] \qquad p \to q}}{\dfrac{\dfrac{\qquad}{\qquad}}{p \to r}}$$

Reglerna för $\vee$:

$$\frac{P \vee Q \qquad \begin{array}{c}\cancel{[P]}\\ \vdots \\ R\end{array} \qquad \begin{array}{c}\cancel{[Q]}\\ \vdots \\ R\end{array}}{R}\,[\vee_E] \qquad \frac{P}{P \vee Q}\,[\vee_{I_1}] \qquad \frac{Q}{P \vee Q}\,[\vee_{I_2}]$$

Den första regeln beskriver ett bevis med uppdelning i olika fall. Om man vet att P eller Q är sann och var och en av dem medför att R är sann så får man dra slutsatsen att R är sann utan att behöva tala om vem av P och Q som är sann. Introduktionsreglerna är lätta att förstå.

Aktivitet 13.
Gör en härledning för $\{p \vee (q \wedge r)\} \vdash p \vee q$.

$$\dfrac{p \vee (q \wedge r) \qquad \dfrac{[p]}{p \vee q} \qquad \dfrac{[q \wedge r]}{p \vee q}}{p \vee q}$$

Introduktionsregeln för $\neg$ handlar om *motsägelsebevis*. I ett motsägelsebevis antar man motsatsen till det man vill bevisa och genomför ett resonemang som visar att detta leder till en *motsägelse*, dvs två påståenden där det ena är negationen till det andra.

$$\dfrac{\begin{array}{cc}\cancel{[P]} & \cancel{[P]} \\ \vdots & \vdots \\ R & \neg R\end{array}}{\neg P}\,[\neg_I]$$

Precis som tidigare är det tillåtet att introducera det hypotetiska antagande noll, en eller flera gånger. Om man härlett både R och $\neg R$ utan några hypotetiska antaganden så är det tillåtet att dra vilken slutsats som helst.

Ett enkelt exempel på användningen visar att $\vdash \neg(p \wedge \neg p)$.

$$\dfrac{\dfrac{\cancel{[p \wedge \neg p]}}{p} \qquad \dfrac{\cancel{[p \wedge \neg p]}}{\neg p}}{\neg(p \wedge \neg p)}$$

Eliminationsregeln för $\neg$ skiljer sig från de andra eliminationsreglerna så tillvida att den eliminerar två operatorer på en gång:

$$\dfrac{\neg\neg P}{P}\,[\neg_E]$$

I och med detta har vi introduktions- och eliminationsregler för operatorerna $\wedge, \vee, \rightarrow$ och $\neg$. Med hjälp av de givna inferensreglerna går det att härleda alla tautologier utan några kvarstående antaganden och tvärtom:

Sats. $\models P$ om och endast om $\vdash P$. ■

En del tautologier kräver förvånansvärt långa härledningar. En sådan är $p \vee \neg p$ där vi använder tre hypotetiska antaganden.

Aktivitet 14.
Indikera vilken regel som använts i varje nod.

$$\dfrac{\dfrac{\dfrac{[p]}{p \vee \neg p} \qquad \dfrac{[\neg(p \vee \neg p)]}{\neg p}}{\neg\neg(p \vee \neg p)} \qquad \dfrac{\dfrac{\dfrac{[\neg p]}{p \vee \neg p} \qquad [\neg(p \vee \neg p)]}{\neg\neg p}}{p}}{p \vee \neg p}$$

När härledningen är klar har vi upphävt alla hypotetiska antaganden. Vi har visat att $\vdash p \vee \neg p$.

Det är lätt att förstå att man i föregående härledning kan byta ut p mot ett godtyckligt satslogiskt uttryck. Vi skulle kunna lägga till en ny inferensregel

$$\frac{}{P \vee \neg P}$$

utan att därmed kunna härleda fler resultat, men det skulle förkorta en del härledningar. Sådana regler kallas för *härledda* (eng. derived). I vanliga matematiska bevis använder man ofta sådana härledda regler.

Den härledda regeln ovan skiljer sig från de övriga så tillvida att den saknar premisser. Ett axiom kan uppfattas som en härledningsregel utan premisser.

2.12 Teorier

I en matematisk *teori* (eng. theory) utgår man från *axiom* och härledningsregler och bevisar *satser* (eng. theorem). Ett axiom är ett påstående som föreskrives vara sant utan bevis. Man kan säga att axiomen beskriver egenskaperna hos de element som teorin handlar om. I Euklides geometriska teori är *punkt*, *linje* och *plan* några av de fundamentala elementen. En punkt har ingen utsträckning i rummet och en linje har ingen bredd och kan därför inte synas eller observeras; de är bara matematiska abstraktioner.

När man bevisar vanliga satser i matematiken genomförs dessa inte så detaljerat som ovan. Detta gör att ett matematiskt bevis sällan kan kontrolleras mekaniskt. Man betraktar ett sådant bevis som korrekt om det är så utförligt att det i princip går att översätta det till ett härledningsträd, men man måste tillfoga härledningsregler som tillhör predikatlogiken och axiom som beskriver egenskaperna hos de fundamentala begrepp som teorin handlar om.

2.13 Historik

Redan de gamla grekerna intresserade sig för logik med Sokrates, Platon och Aristoteles i spetsen.

På 1800-talet formaliserade Boole logiken. Hans namn återkommer i datatypen `boolean` och i Boolesk algebra. Frege använde logiken som bas för en formalisering av den grundläggande matematiken. Russell och Whitehead fullbordade arbetet i sin Principia Mathematica [4]. Det var i samband med detta arbete som Russell upptäckte det fanns motsägelser i Freges beskrivning av mängdläran, det som nu kallas *Russells paradox*. Bevisen i Principia Mathematica är utomordentligt detaljerade och noggrant genomförda, ungefär som härledningarna i vårt avsnitt om naturlig härledning. På sidan 379 i volym 1 av 3 bevisar författarna (anpassad till vår notation)

Sats. $\vdash n \in 1 \wedge m \in 1 \rightarrow (n \cap m = \emptyset \leftrightarrow n \cup m \in 2)$. ■

Satsen säger väsentligen att om n och m är mängder med ett element så har unionen av mängderna två element precis då skärningen mellan mängderna är tom. Beviset omfattar 6 rader och hänvisar till lika många tidigare bevisade satser. I en kommentar till satsen skriver författarna: "From this proposition it will follow, when arithmetic addition has been defined, that $1 + 1 = 2$." Boken är ett av de mest kända verken i den matematiska litteraturen, men det lär inte vara många som tagit del av alla detaljer.

Naturlig härledning introducerades av Gentzen och Jaśkowski oberoende av varandra år 1934. En svensk logiker, Dag Prawitz, skrev sin doktorsavhandling om naturlig härledning [2] och den refereras ofta. Den gavs ut som en bok när den skrevs för drygt 40 år sedan, vilket var de normala på den tiden. Det som är unikt är att det för några år sedan trycktes en ny upplaga [3].

En annan svensk logiker/matematiker, Per Martin-Löf, har vidareutvecklat området. Hans typteori är relevant för datavetenskapens grunder.

Även om vi kan tycka att härledningsreglerna i avsnittet om naturlig härledning är självklara så finns det matematiker som är skeptiska till härledningsregeln som tillåter motsägelsebevis:

$$\dfrac{\begin{array}{cc}\overset{[\neg P]}{\vdots} & \overset{[\neg P]}{\vdots} \\ R & \neg R\end{array}}{\neg P}\,[\neg_I]$$

De kräver ett "konstruktivt" bevis för att ett påstående är sant; det är inte tillräckligt att ett påståendet leder till en motsägelse för att man skall kunna sluta att negationen till påståendet är sant.

Aktivitet 15.
Betrakta

Sats. Denna sats är falsk. ■

Är satsen sann eller falsk?

2.14 Alternativa beteckningar

Symbolerna i satslogiken är inte lika standardiserade som i aritmetiken. För sanningsvärdena används bl a

F	f	false	$\bot$	0
T	t	true	$\top$	1

och naturligtvis också orden falskt och sant. Anledningen till att vi använder 'sanning' och 'falskhet' är att ett värde bör vara ett substantiv eller ett räkneord snarare än ett adverb eller ett adjektiv.

För de logiska operatorena finns också alternativ:

$\neg$	$\sim$		
$\wedge$	$\&$		
$\vee$	$\mid$		
$\rightarrow$	$\Rightarrow$	$\supset$	
$\leftrightarrow$	$\Leftrightarrow$	$\equiv$	$\sim$

När man använder motsvarande ord (inte, och, etc.) i löpande text för att sätta samman påståenden betyder dessa samma sak som symbolerna.

Vissa författare skriver 'välbildade uttryck' (eng. well formed formula, wff) i stället för satslogiska uttryck som om det fanns en meningsfull användning för 'missbildade uttryck'.

Ibland utelämnar man mängdklamrarna i $A \vdash P$, och skriver $p, q \vdash q \wedge p$ i stället för $\{p, q\} \vdash q \wedge p$.

3 Predikatlogik

3.1 Motivering

I satslogiken är de minsta beståndsdelarna satslogiska variabler som kan anta värdena F och T. När man använder satslogiken har man abstraherat bort allt som variablerna står för utom deras sanningsvärden. Vi behöver något mer uttrycksfullt för att kunna formulera påstående som handlar om egenskaper hos objekt eller element. Pythagoras sats är ett sådant påstående som handlar om trianglar.

> **Sats.** Om t är en rätvinklig triangel så är summan av kvadraterna på kateterna i t lika med kvadraten på hypotenusan. ■

Här finns två påståenden:

1. t är en rätvinklig triangel.
2. Summan av kvadraterna på kateterna i t är lika med kvadraten på hypotenusan.

Om vi bara intresserar oss för strukturen på påståendena så handlar båda om en triangel, t, som vi inte vet vilken den är. Om vi låter $p(t)$ vara det första påståendet och $q(t)$ det andra så har satsen formen

$$p(t) \rightarrow q(t)$$

Pythagoras sats handlar inte bara om en speciell triangel utan om alla. Med predikatlogiken gör vi detta explicit

$$\forall t.p(t) \rightarrow q(t)$$

Operatorn $\forall$ utläses 'för alla'.

3.2 Mål

Efter att ha studerat detta kapitel och arbetat med förelagda problem och programmeringsuppgifter skall du kunna

1. tolka predikatlogiska uttryck.
2. översätta påståenden i naturligt språk till predikatlogik.

3.3 Grundläggande begrepp

I *predikatlogiken* (eng. predicate logic) kan de minsta beståndsdelarna som bär sanningsvärden ha en inre struktur. I stället för satsvariabler har man *predikat*. Ett predikat är en funktion av noll eller flera variabler som lämnar ett sanningsvärde som värde. I den rena predikatlogiken spelar det ingen roll vad funktionerna betyder eller hur man räknar ut deras värden; man ger bara funktionerna och variablerna namn: $p(x)$ och $q(x, y)$. Som vanligt är det också tillåtet att sätta in konstanter i stället för variabler. När det är ointressant vilket värde konstanten har använder vi symboler från början av alfabetet, $p(a)$ och $q(a, y)$. När predikatet inte har några variabler, p, fungerar det som en satslogisk variabel; det kan ha värdet T eller F, men vi talar inte om vilket. När vi tillämpar predikatlogiken definierar vi predikaten och ger dem meningsfulla namn: $\mathit{positive}(x) \triangleq x > 0$ och $\mathit{greater}(x, y) \triangleq x > y$ och lika gärna använder vi det definierande uttrycket som predikat, dvs $x > 0$ och $4 > y$. Värdena av predikaten $\mathit{positive}(x)$ och $4 > y$ beror naturligtvis på vad x och y är.

Man får kombinera predikat med de satslogiska operatorerna på samma sätt som i satslogiken till *predikatlogiska uttryck*.

Slutligen tillkommer två stycken operatorer som kallas *kvantorer* (eng. quantifier), universalkvantorn eller allkvantorn $\forall$ och existenskvantorn $\exists$. Om P är ett predikatlogiskt uttryck och x är en variabel, som kan förekomma i uttrycket, kan man skriva

$$(\forall x \,.\, P) \qquad \text{och} \qquad (\exists x \,.\, P)$$

Det första uttrycket är sant om P är sant för alla tillåtna värden på x. Det andra uttrycket är sant om det finns något värde på x så att P är sant. Om P beror på andra variabler än x så är naturligtvis värdet av $(\forall x \,.\, P)$ och $(\exists x \,.\, P)$ beroende av dessa.

Om det inte är klart av sammanhanget vilka värden som är tillåtna kan vi ange det explicit:

$$(\forall x \in S \,.\, P) \qquad \text{och} \qquad (\exists x \in S \,.\, P)$$

där S är mängden av de tillåtna värdena.

Om variabeln x inte förekommer i uttrycket P betyder $(\forall x \,.\, P)$ samma sak som P. Motsvarande gäller $(\exists x \,.\, P)$.

Som tidigare kommer vi att utelämna det yttersta parentesparet i ett helt uttryck.

Några exempel:

Låt *even(i)* vara en funktion som tar ett heltal som argument och som lämnar värdet T om i är jämnt och F om i är udda. Då är $\forall i \,.\, \mathit{even}(i)$ falskt eftersom 1 är udda medan $\forall i \,.\, \mathit{even}(6i)$ är sant. I dessa fall är det underförstått att i skall anta alla heltalsvärden. I dessa exempel skulle vi kunna skriva $\forall i \in \mathbb{Z} \,.\, \mathit{even}(i)$ och $\forall i \in \mathbb{Z} \,.\, \mathit{even}(6i)$, där $\mathbb{Z}$ betecknar mängden av de hela talen.

$\exists i \,.\, \mathit{even}(i)$ är sant eftersom t ex 0 är jämnt. Däremot är $\exists i \,.\, \mathit{even}(4i + 3)$ falskt eftersom $4i + 3$ är udda för alla heltal i.

Aktivitet 1.
Skriv ett uttryck som säger att

a. heltalet i är jämt. Använd $\exists$, $*$ och $=$ men inte divisions- eller restoperator.

b. för alla heltal så är kvadraten på talet är större än eller lika med talet.

c. det finns något heltal så att kvadraten på talet är lika med talet.

Låt *prime(n)* vara en funktion som tar ett positivt heltal som argument och som lämnar värdet T om n är ett primtal och F om så inte är fallet. Följande uttryck

$$\forall n \,.\, (\exists p \,.\, (prime(p) \wedge p > n))$$

säger att för alla n så finns det ett tal p så att p är ett primtal och större än n. Annorlunda uttryckt innebär detta att det finns hur stora primtal som helst. Påståendet är sant och det bevisades av Euklides för 2300 år sedan.

Om man kastar om ordningen mellan kvantorerna i Euklides påstående förändras betydelsen.

$$\exists p \,.\, (\forall n \,.\, (prime(p) \wedge p > n))$$

Detta betyder att det finns ett tal som är ett primtal och som är större än alla tal. Detta är uppenbarligen falskt.

Aktivitet 2.
Ibland måste man själv förstå att det behövs kvantorer. Skriv ett uttryck som säger att

a. summan av två heltal är alltid densamma oberoende av i vilken ordning operanderna tages.

b. Kvadraten på ett jämnt tal är jämn.

Aktivitet 3.
I kursen 'Analys i en variabel' finns följande definition.

> Låt f vara en funktion och antag att varje omgivning av punkten a innehåller punkter ur D_f. Då säges f ha gränsvärdet A då x går mot a om det till varje tal $\epsilon > 0$ finns ett tal $\delta = \delta(\epsilon) > 0$ sådant att
>
> $$\left.\begin{array}{l} |x-a| < \delta \\ x \in D_f \end{array}\right\} \Rightarrow |f(x) - A| < \epsilon \,.$$

Skriv ett predikatlogiskt uttryck som är sant precis när f har gränsvärdet A då x går mot a.

3.4 Syntax

Vi skall presentera den formella grammatiken för predikatlogiska uttryck. I detta sammanhang är en *term* antingen

1. en *variabel* eller en *konstant*. Vi använder ofta bokstäver i slutet av alfabetet som namn på variabler, $x, y, x, \ldots$. Namn på konstanter väljs från början av alfabetet, $a, b, c, \ldots$. Konkreta konstanter som 0 och T kan också förekomma.

eller

2. ett *funktionsuttryck*, $f(t_1, \ldots, t_n)$, där f är ett funktionsnamn med *ställigheten* (eng. arity) n och $t_1, \ldots, t_n$ är termer. n är ett naturligt tal. Ställigheten anger hur många argument funktionen har.

Exempel på funktionsuttryck är $f(x, a)$ och f(g(x,y,z), 0).

Ett *predikat* har antingen formen

1. $R(t_1, \ldots, t_n)$, där R är ett predikatnamn med ställigheten n och $t_1, \ldots, t_n$ är termer. n är ett naturligt tal.

eller

2. $\neg P$, $(P \wedge Q)$, $(P \vee Q)$ eller $(P \rightarrow Q)$, där P och Q är predikat.
3. $(\forall x.P)$ och $(\exists x.P)$ är predikat.

3.5 Semantik

Man ger mening, semantik, åt ett predikatlogiskt uttryck genom att tala om vad funktions- och predikatnamn står för. I en tillämpning skulle f och g kunna vara

$$\begin{aligned} f(x, y) &\triangleq x + y \\ g(x, y, z) &\triangleq x * y - z \end{aligned}$$

och

$$P(t_1, t_2) \triangleq t_1 > t_2$$

Predikatnamn skiljer sig från funktionsnamn genom att det står för något som ger ett sanningsvärde medan funktioner kan ge värden av vilken typ som helst.

3.6 Specifikation

En del av de predikatlogiska uttryck som finns i detta avsnitt handlar om elementen i vektorer. En vektor har ett givet antal element. I matematiken brukar man numrera dem från 1 och uppåt. I många programspråk börjar man från 0. För att inte behöva ange vilka värden en kvantifieringsvariabel kan anta använder vi i denna text konventionen att kvantifieringen skall omfatta alla värden som gör att elementindex blir giltiga. Om x är en vektor med elementen indicerade från 1 till n betyder sålunda $\forall i \,.\, x_i \neq x_{i+1}$ samma sak som $\forall i \in \{1, \ldots, n-1\} \,.\, x_i \neq x_{i+1}$.

Aktivitet 4.
Vad betyder

$\forall i \,.\, x_i \leq x_{i+1}$
$\forall i \,.\, \forall j \,.\, i = j \vee x_i \neq x_j$
$\exists i \,.\, \forall j \,.\, i \neq j \rightarrow x_i < x_j$
$\forall i \,.\, x_i = y_i$

Aktivitet 5.
Låt x och y vara vektorer med lika många element. Formulera med predikatlogik

a. Alla värden som finns i x finns också i y.

b. Varje element i x är mindre än motsvarande element i y.

c. Alla element i x är mindre än varje element i y.

d. Det minsta elementet i x är större än det största elementet i y.

3.7 Bundna och fria variabler

När vi definierar en funktion är namnet på variabeln inte signifikant. $f(x) \triangleq x + 1$ och $f(y) \triangleq y + 1$ definierar samma funktion. På samma sätt är det med x i $\forall x \,.\, P(x)$ och $\exists x \,.\, P(x)$. Det är tillåtet att byta namn på kvantorvariabeln, så länge det inte blir några *namnkollisioner*.

Om $g(x) \triangleq x + y$, där y får sitt värde av det sammanhang där definitionen göres, så får man inte ersätta x med y. Inte heller får man ersätta y med ett uttryck som innehåller x.

För att kunna definiera substitution i uttryck som innehåller kvantorer inför vi begreppen *fria* och *bundna* förekomster av variabler.

> **Definition.** En förekomst av en variabel är *bunden* om det finns en omgivande kvantifiering där variabeln introduceras.
>
> En förekomst av en variabel som inte är bunden säges vara *fri*. △

Om det inte finns några kvantorer i ett predikatlogiskt uttryck så är alla variabelförekomster fria. I $prime(p) \wedge p > n$ är båda förekomsterna av p och den enda av n fria.

I $\exists p \,.\, prime(p) \wedge p > n$ är alla förekomsterna av p bundna medan förekomsten av n är fri.

I $\forall n \,.\, \exists p \,.\, prime(p) \wedge p > n$ är alla variabelförekomster bundna.

Aktivitet 6.
Markera fria förekomster av variabler i $(\forall n \,.\, n > 0) \vee (n = 0)$

3.8 Substitution

Universalkvantorn är en generalisering av $\wedge$-operatorn. Om de tillåtna värdena är ändligt många går det att skriva $\forall x \,.\, P$ som en konjunktion av uttryck.

$$\forall n \in \{1, 2, 3, 4\} \,.\, prime(2^{2^n}+1) = prime(2^{2^1}+1) \wedge prime(2^{2^2}+1) \wedge prime(2^{2^3}+1) \wedge prime(2^{2^4}+1)$$

I högerledet har vi ersatt n med i tur och ordning talen 1, 2, 3 och 4. Det är praktiskt att ha en operator som beskriver sådana *substitutioner*.

Vi kommer generellt att använda beteckningen $e[x \backslash t]$ för det uttryck man får när man i uttrycket e ersätter variabeln x med uttrycket t. Det finns en komplikation som orsakas av *namnkollisioner* som kräver en omsorgsfull definition. Vi börjar med substitution i aritmetiska uttryck med heltal,

variabler, de fyra aritmetiska operatorerna och parenteser. Där uppträder ej komplikationen med namnkollisioner.

För att slippa problem med utelämnade parenteser förutsätter vi att alla sammansatta uttryck omges av parenteser. Det finns sex sorters uttryck, e, och vi definierar $e[x\backslash t]$ för vart och ett av dem. Vi börjar med numeriska literaler, som 1 och -14. Om n är en numerisk literal så har vi

$$n[x\backslash t] \quad \triangleq \quad n$$

Om ett uttryck bara består av en variabel, v, har vi två fall

$$\begin{aligned} v[x\backslash t] &\triangleq t \qquad \text{om } v \text{ och } x \text{ är samma variabel} \\ v[x\backslash t] &\triangleq v \qquad \text{om } v \text{ och } x \text{ är olika variabler} \end{aligned}$$

De återstående fallen handlar om sammansatta uttryck.

$$\begin{aligned} (e_1 + e_2)[x\backslash t] &\triangleq (e_1[x\backslash t] + e_2[x\backslash t]) \\ (e_1 - e_2)[x\backslash t] &\triangleq (e_1[x\backslash t] - e_2[x\backslash t]) \\ (e_1 \cdot e_2)[x\backslash t] &\triangleq (e_1[x\backslash t] \cdot e_2[x\backslash t]) \\ (e_1/e_2)[x\backslash t] &\triangleq (e_1[x\backslash t]/e_2[x\backslash t]) \end{aligned}$$

Dessa definitioner är rekursiva; vi definierar substitution i sammansatta uttryck med hjälp av substitution i deluttrycken.

Aktivitet 7.
Högerleden i de fyra sista definitionerna av substitution tillåter två tolkningar beträffande hur mycket som skall utsättas för den sista substitutionen. Sätt ut parenteser så att det inte råder någon tvekan i

$$(e_1[x\backslash t] + e_2[x\backslash t])$$

Vilken precedens skall substitutionsoperatorn ha relativt de aritmetiska operatorerna för att man skall kunna utelämna parenteserna i högerledet?

När vi gör substitutioner för hand är vi knappast medvetna om den bakomliggande definitionen. Vi kan identifiera de variabler som skall ersättas med ett snabbt ögonkast och vi brukar kunna hantera utelämnade parenteser utan större ansträngning. Om vi skall skriva ett program som utför substitution måste vi implementera operationen rekursivt på det sätt som definitionen föreskriver.

Aktivitet 8.
Utför substitutionerna

$$\begin{aligned} (x \cdot x)[x\backslash x + 1] &= \\ (y \cdot x)[x\backslash y + 1] &= \\ (x \cdot x)[x\backslash x + 1][x\backslash x + 1] &= \end{aligned}$$

3.8.1 Substitution i kvantifierade uttryck

Vad borde $(\forall x \in \mathbb{N} \,.\, x > 0)[x\backslash 2]$ betyda? Vi är vana vid att man kan ersätta ett uttryck med ett ekvivalent uttryck i de flesta sammanhang. Resultatet borde alltså bli detsamma om vi räknar ut värdet inom parentes först och substituerar sedan eller tvärtom. Konsekvensen blir att man inte skall gör något alls:

$$(\forall v \,.\, P)[x\backslash t] \quad \triangleq \quad (\forall v \,.\, P) \qquad \text{om } v \text{ och } x \text{ är samma variabel}$$

I $(\forall y \in \mathbb{N} \,.\, y \geq x)[x\backslash 2]$ finns inga problem. Uttrycket inom parentes har olika värden beroende på vilket värde x har. Substitutionen innebär bara att vi är intresserade av dess värde när $x = 2$.

I $(\forall y \in \mathbb{N} \,.\, y \geq x)[x\backslash y - 1]$ är det y som finns i $[x\backslash y - 1]$ inte samma variabel som introduceras parentesen; kvantifieringen tar slut vid högerparentesen. Om vi substituerar rent mekaniskt får vi $(\forall y \in \mathbb{N} \,.\, y \geq y - 1)$ där $\forall$ har "fångat" den tidigare fria förekomsten av y i $y - 1$. Om y inte förekommer i det uttryck som skall ersätta x så blir definitionen enkel

$$(\forall y \,.\, P)[x\backslash t] \quad \triangleq \quad (\forall y \,.\, P[x\backslash t]) \qquad \text{om } y \text{ och } x \text{ är olika variabler och } y \text{ inte är fri i } t$$

Lösningen på problemet är enkel; vi byter ut kvantifieringsvariabeln x mot en annan variabel som inte finns i det uttryck vi vill substituera in.

I det återstående fallet har vi en namnkollision som är enkel att hantera genom att byta namn på kvantifieringsvariabeln. Att byta kvantifieringsvariabeln y mot z i $(\forall y \in \mathbb{N} \,.\, y \geq x)$ förändrar inte uttryckets värde. Den enda restriktionen är att den nya variabeln inte får finnas i det som står efter punkten.

Den sista delen av definitionen blir

$$\begin{aligned}(\forall y \,.\, P)[x\backslash t] \quad \triangleq \quad & (\forall z \,.\, ((P[y\backslash z])[x\backslash t])) \qquad \text{om } y \text{ och } x \text{ olika variabler,} \\ & \qquad\qquad y \text{ förekommer i } t \text{ och } z \text{ är en "ny" variabel}\end{aligned}$$

3.8.2 Stora operatorer

Det finns "stora" $\wedge$- och $\vee$-operatorer som har samma relationer till de små som $\sum$ har till $+$.

$$\bigwedge_{i=1}^{n} P_i \quad \triangleq \quad P_1 \wedge \cdots \wedge P_n = \forall i \in \{1, \ldots, n\} \,.\, P_i$$

$$\bigvee_{i=1}^{n} P_i \quad \triangleq \quad P_1 \vee \cdots \vee P_n = \exists i \in \{1, \ldots, n\} \,.\, P_i$$

Aktivitet 9.
Det vore praktiskt om

$$\bigwedge_{i=1}^{n} P_i \quad = \quad (\bigwedge_{i=1}^{k} P_i) \wedge (\bigwedge_{i=k+1}^{n} P_i)$$

$$\bigvee_{i=1}^{n} P_i \quad = \quad (\bigvee_{i=1}^{k} P_i) \vee (\bigvee_{i=k+1}^{n} P_i)$$

för alla $0 \leq k \leq n$. Hur skall man definiera de stora operatorerna när antalet operander är 0.

3.9 Inferensregler

Eftersom kvantifieringar kan göras över mängder med oändligt många element så finns det ingen generell motsvarighet till satslogikens sanningstabeller. I regel är man hänvisad till att använda inferensregler.

Inferensreglerna för satslogiken gäller oförändrade för predikatlogiken. Det tillkommer regler för kvantorerna. Till reglerna hör bivillkor för att förhindra namnkollisioner.

$$\frac{P}{\forall x \,.\, P}\,[\forall_I] \qquad\qquad \frac{\forall x \,.\, P}{P[x\backslash t]}\,[\forall_E]$$

Den första regeln säger att om man har ett bevis för P som får innehålla variabeln x så får man dra slutsatsen att P gäller för alla x. I härledningen av P får det inte finnas några (oupphävda) antaganden om x.

Den andra regeln säger att om P är sant för alla x så är P sant om vi ersätter x med ett godtyckligt uttryck t. Detta betyder implicit att kvantifieringar alltid görs över icketomma mängder.

Följande härledning visar att $\forall x \,.\, \forall y \,.\, P(x,y) \vdash \forall y \,.\, \forall x \,.\, P(x,y)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x \,.\, \forall y \,.\, P(x,y)}{\forall y \,.\, P(x,y)}\,[\forall_E]}{P(x,y)}\,[\forall_E]}{\forall x \,.\, P(x,y)}\,[\forall_I]}{\forall y \,.\, \forall x \,.\, P(x,y)}\,[\forall_I]$$

Om man bryter mot bivillkoret för $[\forall_I]$ så kan det bli fel:

$$\dfrac{\dfrac{\cancel{[x = 0]}}{\forall x \,.\, x = 0}\,[\text{felaktig användning av } \forall_I]}{x = 0 \to \forall x \,.\, x = 0}\,[\to_I]$$

Inferensreglerna för $\exists$ är

$$\frac{P[x\backslash t]}{\exists x \,.\, P}\,[\exists_I] \qquad\qquad \frac{\exists x \,.\, P \qquad \cancel{[P[x\backslash y]]}Q}{Q}\,[\exists_E]$$

Den första regeln handlar om ett påstående P som beror på en variabel x. Om vi har ett bevis för att P är sant om vi har ersatt x med en konstant eller en mer komplicerad term t så får vi dra slutsatsen $\exists x \,.\, P$.

Den andra regeln säger att om vi bevisat att det finns något värde på x som gör att P är sant så kan vi kalla detta värde för y utan att veta vilket det är och använda $P[x\backslash y]$ i härledningen av Q. Q får inte inte innehålla y och i härledningen av Q får det naturligtvis inte finnas något annat antagande som handlar om y.

3.10 Kvantorernas räckvidd och parenteskonventioner

Om man utelämnar parenteserna kring konjunktionen i $\forall x \,.\, (P(x) \wedge Q(x))$ inträder tvetydighet. Den gängse konventionen säger att utelämnade parentespar i kvantifieringar omfattar så mycket som möjligt av det logiska uttrycket som eventuellt omslutande parenteser tillåter. Detta betyder t ex att $\forall x \,.\, \exists y \,.\, P(y) \vee Q(x)$ skall tolkas som $\forall x \,.\, (\exists y \,.\, (P(y) \vee Q(x)))$.

3.11 Räknelagar

Det finns alltför många räknelagar som handlar om kvantorerna och logiska operatorerna för att det skall vara meningsfull att räkna upp dem. Många är lätta att förstå

$$\begin{aligned} \forall x \,.\, P \wedge Q &= (\forall x \,.\, P) \wedge (\forall x \,.\, Q) \\ \forall x \,.\, \forall y \,.\, P &= \forall y \,.\, \forall x \,.\, P \end{aligned}$$

men man skall inte förledas att tro att alla snarlika formler gäller. Om man byter $\forall$ mot $\exists$ i ekvationerna ovan gäller de i regel inte.

Särskilt användbara är motsvarigheten till de Morgans lagar i satslogiken:

$$\begin{aligned} \neg(\forall x \,.\, P) &= \exists x \,.\, \neg P \\ \neg(\exists x \,.\, P) &= \forall x \,.\, \neg P \end{aligned}$$

Vi kan använda inferensreglerna för att bevisa räknelagarna. Vi genomför en del av detta bevis genom att bevisa $\{\forall x \,.\, \neg P(x)\} \vdash \neg(\exists x \,.\, P(x))$

$$\dfrac{\dfrac{\dfrac{[\cancel{\exists x \,.\, P(x)}]}{P(a)}\,[\exists_E] \qquad \dfrac{\forall x \,.\, \neg P(x)}{\neg P(a)}\,[\forall_E]}{\neg(\exists x \,.\, P(x))}\,[\neg_I]}$$

Vi kan använda $[\rightarrow_I]$ för att dra slutsatsen $\vdash (\forall x \,.\, \neg P(x)) \rightarrow \neg(\exists x \,.\, P(x))$.

4 Mängder

4.1 Motivering

Mängden är den mest grundläggande diskreta strukturen. Nästan alla matematiska begrepp går att återföra till mängdlärans begrepp och nästan alla matematiska värden går att representera med mängder. Anledningen till att vi skriver 'nästan' förklaras i avsnitt 4.13.

Mängdläran har tre grundläggande begrepp, *mängd* (eng. set), *element* och *tillhör* (eng. belongs to), som inte definieras med mer primitiva begrepp.

4.2 Grundläggande begrepp

En mängd består av element. Vi föreskriver inte närmare vad som får vara element, men vi kommer att vara generösa. I våra inledande exempel kommer elementen att vara tal. Om mängden är liten kan vi beskriva den genom att räkna upp elementen. Vi skriver $\{1, 3, 5\}$ För att beteckna mängden som innehåller de tre talen 1, 3 och 5.

Vi betraktar två mängder som lika om de har precis samma element. När vi beskriver en mängd på detta sätt spelar det därför ingen roll i vilken ordning vi räknar upp elementen; $\{1, 3, 5\}$ och $\{3, 1, 5\}$ beskriver samma mängd. I uppräkningen undviker man att ta med samma element flera gånger, men även $\{3, 1, 5, 1\}$ beskriver samma mängd.

Den mängd som inte innehåller några element kallas för den *tomma mängden* (eng. empty set), $\{\}$. Den betecknas också med $\emptyset$.

Om ett element x tillhör en mängd M skriver vi $x \in M$. Om så inte är fallet skriver vi $x \notin M$. Det gäller alltså att $3 \in \{1, 3, 5\}$ och att $2 \notin \{1, 3, 5\}$. När vi skriver $x, y \in M$ betyder detta att $x \in M$ och $y \in M$.

En del mängder har standardiserade namn:

- $\mathbb{N}$ betecknar mängden av *naturliga tal* (eng. natural number). Den innehåller talen $0, 1, 2, 3$ etc.
- $\mathbb{Z}$ är mängden av *hela tal* (eng. integer). Den innehåller talen $0, 1, -1, 2, -2, 3, -3$ etc.
- $\mathbb{Q}$ är mängden av *rationella tal* (eng. rational number). Den innehåller alla bråktal på formen $\frac{p}{q}$, där $p \in \mathbb{Z}$, $q \in N$, $q \neq 0$ och p och q inte har någon gemensam faktor större än 1.
- I kursen i endimensionell analys är mängden av *reella tal* (eng. real number), $\mathbb{R}$, fundamental.
- I logiken använder vi $\mathbb{B} \triangleq \{\mathsf{F}, \mathsf{T}\}$ med sanningsvärdena F(falskt) och T(sant).

4.3 Kardinalitet

För en ändlig mängd M använder vi beteckningen $|M|$ för antalet element i mängden. Vi har alltså att $|\{1, 3, 5\}| = 3$ och $|\emptyset| = 0$.

Vi kommer att använda samma notation för storleken av oändliga mängder och kalla det för mängdens *Kardinalitet* (eng. cardinality). Det visar sig att oändliga mängder kan ha olika kardinalitet. $|\mathbb{N}|$ och $|\mathbb{Z}|$ har samma kardinalitet och vi använder beteckningen $\aleph_0$ för den. Symbolen utläses alef-noll. $\aleph$ är den första bokstaven i det hebreiska alfabetet. $|\mathbb{R}|$ är "större".

Aktivitet 1.
Vad är

$|\{1, 3, 1, 5\}| =$
$|\{\emptyset\}| =$

4.4 Mängdbyggare

För stora eller oändliga mängder är det inte möjligt att beskriva en mängd genom att räkna upp elementen. Ibland kan vi vara informella och lita på läsarens intelligens och skriva $\{0, 1, 2, \dots\}$. Ett bättre sätt är att använda *mängdbyggare* (eng. set builder). Vi använder skrivsättet $\{x \in U \mid p(x)\}$ för att beteckna mängden av element $x \in U$ sådana att $p(x)$ är sant. $p(x)$ är ett logiskt uttryck som beror på x, ett *predikat* (eng. predicate). I detta sammanhang kallas U för *universalmängden* (eng. universal set).

Mängden av de positiva heltalen är sålunda $\{z \in \mathbb{Z} \mid z > 0\}$.

Ibland använder man $\mathbb{Z}^+$ som beteckning för denna mängd. Motsvarande notation används i $\mathbb{Q}^+$ och $\mathbb{R}^+$.

Definition. $\mathbb{Z}^+ \triangleq \{z \in \mathbb{Z} \mid z > 0\}$. △

Mängden av alla naturliga tal som är mindre än 10000 är sålunda $\{x \in \mathbb{N} \mid x < 10000\}$.

När det är uppenbart av sammanhanget vad som är universalmängd tillåter vi oss att utelämna den, $\{x \mid p(x)\}$.

Mängden av alla udda naturliga tal som är mindre än 10000 beskrivs av $\{x \in \mathbb{N} \mid x < 10000 \wedge x \bmod 2 = 1\}$. Ibland använder man kommatecken i stället för $\wedge$ och skriver $\{x \in \mathbb{N} \mid x < 10000,\ x \bmod 2 = 1\}$ för samma mängd.

Vi tillåter oss också att skriva ett uttryck till vänster om |. $\{2 * n + 1 \mid n \in \mathbb{N},\ n < 5000\}$ beskriver samma mängd som ovan.

Aktivitet 2.
Beskriv följande mängder med hjälp av mängdbyggare.

a. Mängden av alla naturliga tal som inte är delbara med 3

b. Mängden av alla primtal.

4.5 Induktivt definierade mängder

Vi kan definiera mängden av alla satslogiska uttryck, $\mathcal{P}$, som den minsta mängd med följande egenskaper:

- Om x är ett variabelnamn så $x \in \mathcal{P}$.
- Om $P \in \mathcal{P}$ så $\neg P \in \mathcal{P}$.
- Om $P \in \mathcal{P}$ och $Q \in \mathcal{P}$ så tillhör alla $(P \wedge Q)$, $(P \vee Q)$ $(P \rightarrow Q)$ och $(P \leftrightarrow Q)$ också mängden $\mathcal{P}$.

Vi känner igen kompositmönstret.

Detta kallas för en *induktivt definierad mängd*. En sådan innehåller ett eller flera basfall och ett eller flera sammansatta eller induktiva fall. Typer i programspråk är en sorts mängder och en rekursiv typ svarar mot en induktiv definition.

4.6 Delmängder

En mängd M_1 är en delmängd av en mängd M_2 om alla element i M_1 också är element i M_2. Vi skriver $M_1 \subseteq M_2$.

Definition.

$$M_1 \subseteq M_2 \triangleq \forall m.\ m \in M_1 \to m \in M_2$$

△

Vi har en god förståelse av vad en delmängd är så följande sats kommer inte som någon överraskning.

Sats. Om $A \subseteq B$ och $B \subseteq C$ så är $A \subseteq C$. ■

Vi genomför ett bevis av detta mer för att demonstrera hur ett bevis går till än att övertyga någon misstroende.

Bevis. Antag att $a \in A$. Detta betyder att a är ett godtyckligt element i A, men att a är samma element i hela beviset. Vi skall visa att $a \in C$.

Av definitionen på $A \subseteq B$ så följer att $a \in B$. Eftersom $B \subseteq C$ så gäller på samma sätt att $a \in C$. □

Detta bevis är typiskt för hur man bevisar satser i matematiken med en blandning av formler och naturligt språk och utan att ange vilka härledningsregler som används. Med hjälp av reglerna för naturlig härledning kan vi genomföra samma bevis helt formellt. När man säger att ett vanligt matematiskt bevis är korrekt menar man att det går att översätta till ett helt formellt bevis.

Bevis.

$$\{\forall m.\ m \in A \to m \in B,\ \forall m.\ m \in B \to m \in C\} \vdash \forall m.\ m \in A \to m \in C$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall m.\ m \in A \to m \in B}{m \in A \to m \in B} \qquad \dfrac{}{[\sout{m \in A}]}}{m \in B} \qquad \dfrac{\forall m.\ m \in B \to m \in C}{m \in B \to m \in C}}{m \in C}}{m \in A \to m \in C}}{\forall m.\ m \in A \to m \in C}$$

□

Ibland använder man $A \subset B$ för att ange att $A \subseteq B$ och $A \neq B$.

Vi har tidigare talat om att två mängder är lika om de har samma element. Vi kan nu definiera detta formellt:

Definition. $M_1 = M_2 \triangleq M_1 \subseteq M_2 \wedge M_2 \subseteq M_1$ △

Återigen har vi en god intuition för vad likhet mellan mängder betyder, men vi kan också bevisa följande räknelagar utan att använda intuitionen.

Sats. Antag att M_1, M_2 och M_3 godtyckliga mängder. Då gäller

$M_1 = M_1$.
Om $M_1 = M_2$ så är $M_2 = M_1$.
Om $M_1 = M_2$ och $M_2 = M_3$ så är $M_1 = M_3$.

■

Vi bevisar det andra påståendet för att illustrera hur det går till.

Bevis. Antag att $M_1 = M_2$. Enligt definitionen på likhet gäller då att $M_1 \subseteq M_2$ och $M_2 \subseteq M_1$. Vi använder dessa fakta i omvänd ordning i definitionen av likhet och finner att $M_2 = M_1$. □

Den tredje räknelagen bevisas enklast med hjälp av föregående sats.

4.7 Mängdoperationer

Unionen av två mängder innehåller alla element som tillhör minst en av mängderna.

Definition. $M_1 \cup M_2 \triangleq \{x \mid x \in M_1 \vee x \in M_2\}$. △

Snittet (eng. intersection) mellan två mängder innehåller alla element som tillhör båda mängderna.

Definition. $M_1 \cap M_2 \triangleq \{x \mid x \in M_1 \wedge x \in M_2\}$. △

Komplementet av mängden M_1 i mängden M_2 är mängden av alla element i M_1 som inte är element i M_2.

Definition. $M_1 - M_2 \triangleq \{x \in M_1 \mid x \notin M_2\}$. △

När universalmängden U är underförstådd så är definierar vi *komplementmängden.*

Definition. $\overline{M} \triangleq U - M$ △

4.8 Precedens och associativitet

I varje räknelag ovan finns två eller flera operatorer om vi räknar = som en operator. Läsaren hade förmodligen inga svårigheter att förstå vad som var operanderna till varje operator. Här finns inte heller något att tveka om; operanderna får fel typ om man grupperar fel. Den första lagen skall tolkas $(A \cup \emptyset) = A$. $\emptyset = A$ är ett korrekt uttryck, men $A \cup (\emptyset = A)$ är det inte eftersom båda operanderna till $\cup$ skall vara mängder.

I $A \cup B = B \cup A$ finns också bara en rimlig tolkning, $(A \cup B) = (B \cup A)$.

I mängdläran brukar man ge $\cup$ och $\cap$ samma precedens vilket gör att man ofta måste använda parenteser för att göra tolkningen entydig.

4.9 Räknelagar

Det finns många räknelagar för mängdoperationerna. De flesta är ”självklara” utifrån vår förståelse av operationerna och de formella bevisen är mycket korta.

Sats.

$A \cup \emptyset = A$	$A \cap \emptyset = \emptyset$
$A \cup A = A$	$A \cap A = A$
$A \cup B = B \cup A$	$A \cap B = B \cap A$
$A \subseteq A \cup B$	$A \cap B \subseteq A$

■

Vi ser att både $\cup$ och $\cap$ är kommutativa.

En operator, $\star$, sådan att $a \star a = a$ för alla a säges vara *idempotent*. Både $\cup$ och $\cap$ är idempotenta.

Följande räknelagar är direkta motsvarigheter till räknelagarna för $\wedge$, $\vee$ och $\neg$.

Sats.

$A \cup (B \cup C) = (A \cup B) \cup C$	$A \cap (B \cap C) = (A \cap B) \cap C$
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
$\overline{A \cup B} = \overline{A} \cap \overline{B}$	$\overline{A \cap B} = \overline{A} \cup \overline{B}$

■

Vi bevisar den första lagen i rad två.

Bevis.

$$
\begin{aligned}
x \in A \cup (B \cap C) &\Leftrightarrow x \in A \vee x \in B \cap C \\
&\Leftrightarrow x \in A \vee (x \in B \wedge x \in C) \\
&\Leftrightarrow (x \in A \vee x \in B) \wedge (x \in A \vee x \in C) \\
&\Leftrightarrow (x \in A \cup B) \wedge (x \in A \cup C) \\
&\Leftrightarrow (x \in (A \cup B) \cap (A \cup C)
\end{aligned}
$$

□

Vi ser att både $\cup$ och $\cap$ är associativa och kommutativa och att de distribuerar över varandra.

4.10 Potensmängd

Mängden av alla delmängder till en mängd kallas *potensmängden* (eng. power set).

Definition. $\mathcal{P}(M) \triangleq \{S \mid S \subseteq M\}$. △

Det finns fyra olika delmängder till $\{1, 2\}$ så att $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

Ibland används även beteckningen 2^M för potensmängden till M. Den är inspirerad av följande sats.

Sats. Om M är en ändlig mängd så är $|2^M| = 2^{|M|}$. ■

4.11 Partionering

Ibland delar man upp en mängd i disjunkta icketomma delmängder. En sådan uppdelning kallas för en *partionering* (eng. partition). Två exempel på partioneringar av $\{1, 3, 5\}$ är $\{\{1, 3\}, \{5\}\}$ och $\{\{1\}, \{3\}, \{5\}\}$.

Aktivitet 3.
Använd predikatlogik för att ge ett villkor för att $\{A_1, A_2, \ldots, A_n\}$ är en partionering av A.

4.12 Produktmängd

Ett *par* (eng. pair) har två komponenter. Om komponenterna är a och b betecknar vi paret med (a, b). I matematiken är ordningen mellan komponenterna i ett par i regel signifikant. $(1, 2)$ och $(2, 1)$ är olika par. När man vill poängtera detta säger man att paren är *ordnade* (eng. ordered).

Två stycken par är lika om de har samma komponenter.

Definition. $(a_1, b_1) = (a_2, b_2) \triangleq (a_1 = a_2) \wedge (b_1 = b_2)$ $\triangle$

När vi bildar par kommer komponenterna från mängder. Mängden av alla par där den första komponenten kommer från mängden M_1 och den andra komponenten från M_2 kallas *produkten* av M_1 och M_2.

Definition. $M_1 \times M_2 \triangleq \{(m_1, m_2) \mid m_1 \in M_1, m_2 \in M_2\}$ $\triangle$

En produktmängd kallas ofta för en *kartesisk produkt* (eng. Cartesian product). Namnet kartesisk kommer från den franske filosofen och matematikern Descartes med den latinska formen Cartesius. Descartes rekryterades till det svenska hovet av drottning Kristina. Descartes dog i lunginflammation på Stockholms slott år 1650.

Man kan naturligtvis definiera *tupler* (eng. tuple) med flera komponenter och motsvarande produktmängder. Eftersom vi bildar en produkt av flera mängder är det naturligt att använda produktoperatorn Π.

Definition. $M_1 \times M_2 \times M_3 \triangleq \{(m_1, m_2, m_3) \mid m_1 \in M_1, m_2 \in M_2, M_3\}$ etc.
$M_1 \times M_2 \times \cdots \times M_n \triangleq \{(m_1, m_2, \ldots, m_n) \mid m_1 \in M_1, m_2 \in M_2, \ldots, m_n \in M_n\}$
eller

$$\prod_{k=1}^{n} M_k \triangleq \{(m_1, m_2, \ldots, m_n) \mid \forall k \in \{1..k\} \,.\, m_k \in M_k\}$$

$\triangle$

4.13 Russells paradox*

Bertrand Russell upptäckte att det inte var möjligt att bilda mängder på det sätt som Cantor och Frege utgått ifrån utan att det ledde till motsägelser. Låt oss anta att det finns en mängd U som innehåller alla mängder. Bilda mängden

$$M \triangleq \{S \in U \mid S \notin S\}$$

Vi kan nu fråga oss om M tillhör M eller ej.

Om $M \in M$ skall enligt definitionen M inte tillhöra M, $M \notin M$. Detta kan inte vara fallet.

Om i stället $M \notin M$ så skall $M \in M$, vilket är lika omöjligt. Vi tvingas dra slutsatsen att vårt antagande om att U är en mängd inte kan vara riktigt.

5 Alfabeten och språk

I vanligt språkbruk betyder *alfabet* de tecken (bokstäver), som används för att bilda ord i ett språk. Det finns olika slags alfabet: svenskt, engelskt, grekiskt etc. Här skall vi behandla *formella språk* där ett alfabet är en ändlig icketom mängd vars element kallas *symboler*. Som exempel har vi det svenska alfabetet, $\{A, a, B, b, \ldots, \text{ö}\}$, och det binära, $\{0, 1\}$. Symbolerna i ett alfabet behöver inte vara vanliga tecken utan kan vara vad som helst. När vi studerar programspråk kommer vi att ibland att använda alfabet som innehåller symboler som `begin`, `end` och `<=`.

Vi kommer ofta att använda Σ som namn på det aktuella alfabetet och σ som namn på en godtycklig symbol i alfabetet, $\sigma \in \Sigma$.

Med en *sträng* på ett alfabet menas en ändlig följd av symboler ur alfabetet. Till exempel är 001 och 1 strängar på det binära alfabetet. En sträng som inte innehåller några symboler kallas för en *tom sträng*. Eftersom en tom sträng inte syns när den skrivs ut använder vi en särskild beteckning, ϵ, när vill göra förekomsten tydlig. Om vi formellt vill visa egenskaper hos de operationer vi inför måste vi definera både strängar och operationer formellt. Vi använder en induktiv definition:

Definition. Låt Σ vara ett alfabet.

ϵ är en sträng på Σ.

Om $\sigma \in \Sigma$ och α är en sträng på Σ så är $\sigma\alpha$ en sträng på Σ. △

Operationer på strängar definieras också induktivt. Vi börjar med längden av en sträng. För längden av strängen α använder vi beteckningen $|\alpha|$.

Definition. $|\epsilon| \triangleq 0$

$|\sigma\alpha| \triangleq 1 + |\alpha|$ △

Man kan sätta ihop, konkatenera, två strängar. När vi vill vara tydliga använder vi symbolen $\circ$ för konkateneringsoperationen.

Definition.

$\epsilon \circ \alpha \triangleq \alpha$

$(\sigma\alpha) \circ \beta \triangleq \sigma(\alpha \circ \beta)$ △

Man kan visa att strängkonkatenering är associativ, dvs att $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$ vilket gör att det inte är nödvändigt att tala om vilken av konkateneringarna utföras först; resultatet blir i båda fallen detsamma. Vi skriver därför $\alpha \circ \beta \circ \gamma$. Ibland underförstår vi konkateneringsoperationen och skriver $\alpha\beta\gamma$.

Man använder beteckningen α^R för att beteckna den sträng man får genom att reversera strängen α. $(abc)^R = cba$.

Om $\omega = \alpha \circ \beta \circ \gamma$, där α, β och γ är strängar kallar vi α för ett prefix, γ ett suffix och β en delsträng till ω. α kallas för ett äkta prefix om $\omega \neq \alpha$ och analogt för de andra begreppen.

Om α är en sträng definierar vi $\alpha^n, n \in N$ induktivt:

$$\begin{aligned} \alpha^0 &\triangleq \epsilon \\ \alpha^{n+1} &\triangleq \alpha \circ \alpha^n \end{aligned}$$

För induktivt definerade mängder har vi alltid en induktionsprincip, som definierar vad som menas att visa att ett påstående gäller för alla element i mängden. För strängar lyder den som följer:

> **Inferensregel.** Antag att $P(s)$ är ett predikatlogiskt uttryck som handlar om en sträng s på alphabetet Σ. Om vi kan visa att
>
> 1. $P(\epsilon)$ är sant och om
> 2. $P(\alpha) \rightarrow P(\sigma\alpha)$ för alla $\sigma \in \Sigma$ och alla strängar α på alfabetet,
>
> så får vi dra slutsatsen att $P(s)$ är sann för alla strängar s på alfabetet Σ. ▼

Detta är den inferensregel som beskriver ett *induktionsbevis*. Med vår vanliga notation har vi

$$\frac{P(\epsilon) \qquad \forall\sigma\,.\,\forall\alpha\,.\,P(\alpha) \rightarrow P(\sigma\alpha)}{\forall\alpha\,.\,P(\alpha)}$$

Med ett *språk* på ett alfabet menas en mängd strängar på alfabetet. Mängden $\{jag, du, ni, vi, de\}$ är ett språk på det svenska (och engelska) alfabetet. Mängden $\{\epsilon, 0, 1, 01\}$ är ett språk på det binära alfabetet. Mängden av alla strängar på $\{0, 1\}$ är ett språk.

Den mängd som inte innehåller några element betecknas $\emptyset$. Denna mängd är också ett språk, det *tomma* språket.

Eftersom språk är mängder kan vi använda de vanliga mängdoperationerna, $\cup$, $\cap$, $\setminus$ (mängddifferens) och $\bar{}$ (komplement) på språk.

Om L_1 och L_2 är språk så är $L_1 \setminus L_2 \triangleq \{\omega \in L_1 \mid \omega \notin L_2\}$

Om L är ett språk på Σ så är $\overline{L}$ mängden av alla strängar på Σ utom de som tillhör L.

Vi definierar konkateneringen av två språk som mängden av alla strängar som erhålles genom att konkatenera en sträng ur det första språket med en sträng ur det andra. Vi använder samma symbol för operationen som för strängar och tillåter oss att utelämna den.

$$L_1 \circ L_2 \triangleq L_1 L_2 \triangleq \{u \circ v \mid u \in L_1 \wedge v \in L_2\}$$

Vi definierar också L^k, där k är ett naturligt tal, som språket av alla strängar man får genom att konkatenera exakt k strängar ur L:

$$\begin{aligned} L^0 &\triangleq \{\epsilon\} \\ L^{k+1} &\triangleq L \circ L^k \end{aligned}$$

och

$$L^* \triangleq \bigcup_{k=0}^{\infty} L^k$$

L^* innehåller alla strängar man erhåller genom att konkatenera ett ändligt antal strängar ur L. L^* kallas för *höljet* (eng. *closure*) till L. Vi definierar också det *positiva höljet* till L med $L^+ \triangleq L \circ L^*$.

Exempel. Om $L = \{a, bb\}$ så är

$$L^* = \{\epsilon, a, bb, aa, abb, bba, aaa, \ldots\}$$

och

$$L^+ = \{a, bb, aa, abb, bba, aaa, \ldots\}$$

$\Diamond$

Man brukar inte skilja på symbolerna i ett alfabet och motsvarande strängar på alfabetet med längden ett. Med denna konvention blir Σ ett språk (på sig självt) och Σ^* betecknar mängden av alla strängar på Σ. Varje språk på Σ är en delmängd av Σ^*.

Låt $\Sigma_1 = \{a, \ldots, z\}$ och $\Sigma_2 = \{0, \ldots, 9\}$. Språket som består av alla tillåtna namn i Pascal kan skrivas $\Sigma_1(\Sigma_1 \cup \Sigma_2)^*$.

Vi kan relatera de införda begreppen till ett naturligt språk som till exempel det svenska. Mängden av alla svenska ord är ett formellt språk på det svenska alfabetet, $\{a, \ldots, \ö\}$. De svenska orden är i sin tur det alfabet som används för att bygga upp meningar i det svenska språket. I skriftspråket ingår också versaler, mellanslag och andra skiljetecken, som vi bortser från i detta sammanhang.

6 Reguljära uttryck

I unix-skal finns enkla mönster för filnamn med * och ?. En del program, t ex emacs, egrep och sed, erbjuder möjlighet att söka efter strängar som matchar mer komplicerade mönster som beskrivs med *reguljära uttryck*. I Java-paketet `java.util.regex` finns också klasser som tillhandahåller sådan funktionalitet.

Ett reguljärt uttryck beskriver ett språk och de nämnda programmen kan alltså avgöra om en given sträng tillhör språket. T ex så beskriver utrycket $0 \mid 1 \cdot (0 \mid 1)^\star$ det språk som innehåller alla binära tal utan onödiga inledande nollor, $\{0, 1, 10, 11, 100, \ldots\}$. I detta uttryck är 0 och 1 symbolerna i alfabetet medan $\mid$, $\cdot$ och * är operatorer som kan förekomma i reguljära uttryck. Parenteser används som vanligt för att gruppera.

Definition. Mängden av reguljära uttryck på alfabetet Σ definieras av

$\boldsymbol{\emptyset}$ är ett reguljärt uttryck

$\boldsymbol{\epsilon}$ är ett reguljärt uttryck

om $\sigma \in \Sigma$ så är $\boldsymbol{\sigma}$ ett reguljärt uttryck

om α och β är reguljära uttryck så är $(\alpha \cdot \beta)$ ett reguljärt uttryck

om α och β är reguljära uttryck så är $(\alpha \mid \beta)$ ett reguljärt uttryck

om α är ett reguljärt uttryck så är $\alpha^\star$ ett reguljärt uttryck

$\triangle$

Vi har använt ett tjockare typsnitt i $\boldsymbol{\emptyset}$ och $\boldsymbol{\epsilon}$ för att göra det tydligt att de är reguljära uttryck och inte den tomma mängden och den tomma strängen. På samma sätt gör vi inledningsvis när en symbol i alfabetet Σ används i ett reguljärt uttryck.

När vi skriver reguljära uttryck exakt enligt denna definition säger vi att de är på *kanonisk* form.

Exempel.

$$\boldsymbol{\emptyset} \quad \mathbf{a} \qquad (\mathbf{a} \cdot \mathbf{b}) \quad (((\mathbf{a} \mid \mathbf{b}) \cdot \mathbf{a}) \cdot (\mathbf{b} \cdot \mathbf{a})^\star)$$

är reguljära uttryck på alfabetet $\{a, b\}$. $\Diamond$

Den normala konventionen är att * har högst precedens följt av $\cdot$ och lägst $\mid$ när man utelämnar parenteser och man inte skriver ut $\cdot$. Det sista uttrycket kan alltså skrivas $((a \mid b)a)(ba)^\star$. När vi har definierat hur man räknar ut värdet av ett reguljärt uttryck kommer vi att se att både $\cdot$ och $\mid$ är associativa så att vi kan undvara ytterligare några parenteser, $(a \mid b)a(ba)^\star$.

6.1 Semantik

Värdet av ett reguljärt uttryck är ett språk. Ordet *semantik* betyder 'mening' eller 'betydelse'. För att tydligt skilja på ett reguljärt uttryck och uttryckets värde kommer vi i denna kurs skriva $\mathcal{L}(\alpha)$ när vi menar det språk som är värdet av uttrycket α. Definitionen av $\mathcal{L}$ är induktiv med ett fall för varje sorts reguljärt uttryck.

Definition.

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\emptyset}) &\triangleq \emptyset & \qquad \mathcal{L}(\alpha\,\beta) &\triangleq \mathcal{L}(\alpha)\mathcal{L}(\beta)\\
\mathcal{L}(\boldsymbol{\epsilon}) &\triangleq \{\epsilon\} & \qquad \mathcal{L}(\alpha \mid \beta) &\triangleq \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)\\
\mathcal{L}(\boldsymbol{\sigma}) &\triangleq \{\sigma\}, \quad \sigma \in \Sigma & \qquad \mathcal{L}(\alpha^\star) &\triangleq (\mathcal{L}(\alpha))^\star
\end{aligned}$$

$\triangle$

I den första fallet, $\mathcal{L}(\boldsymbol{\emptyset}) \triangleq \emptyset$ är $\boldsymbol{\emptyset}$ i vänsterledet ett reguljärt uttryck medan $\emptyset$ i högerledet är den vanliga tomma mängden.

I det andra fallet är det tydligare att bokstaven epsilon betyder helt olika saker när den används som ett reguljärt uttryck och när den är ett namn på en tomma strängen. Det språk som $\boldsymbol{\epsilon}$ betecknar, $\{\epsilon\}$ är inte särskilt intressant, men $\boldsymbol{\epsilon}$ är praktiskt att kunna användas som en del i mer komplexa reguljära uttryck.

Uttrycket $\boldsymbol{\sigma}$ betyder också ett språk med bara ett element, men är nödvändigt som bas när man skall beskriva större språk.

Med hjälp av | och (den osynliga) $\cdot$ kan man beskriva språk med flera strängar och längre strängar.

Exempel.

$$\mathcal{L}(ab) = \mathcal{L}(a)\mathcal{L}(b) = \{a\}\{b\} = \{ab\}$$
$$\mathcal{L}(a \mid b) = \mathcal{L}(a) \cup \mathcal{L}(b) = \{a\} \cup \{b\} = \{a, b\}$$
$$\mathcal{L}((a \mid b)(a \mid b)) = \mathcal{L}(a \mid b)\mathcal{L}(a \mid b) = \{a, b\}\{a, b\} = \{aa, ab, ba, bb\}$$

$\Diamond$

Stjärnoperatorn ger oss möjlighet att beskriva språk med oändligt många element.

Exempel.

$$\mathcal{L}(a^\star) = (\mathcal{L}(a))^\star = \{a\}^\star = \{\epsilon, a, aa, aaa, \dots\}$$
$$\mathcal{L}(a(a \mid b)^\star) = \cdots = \{a, aa, ab, aaa, aab, aba, abb, \dots\}$$

$\Diamond$

En del författare använder $\cup$ i stället för | i reguljära uttryck. Det är lätt att förstå varför. I vårt sammanhang är det en fördel att ha olika operatorer i reguljära uttryck och mängduttryck.

I matematiken har man sällan anledning att skilja på ett uttryck och uttryckets värde. När man skriver ett uttryck menar man nästan alltid dess värde. I programsammanhang är det annorlunda. Om man i ett program skriver uttrycket `x*x+1` så är det dess värde som skall beräknas med användning av det värde `x` har vid tillfället. Om man matar in $x * x + 1$ i Matlab eller Maple så måste uttrycket själv sparas för att kunna beräknas eller t ex deriveras vid ett senare tillfälle.

Exempel. Det reguljära uttrycket $(0 \mid 1)^\star$ betecknar språket av alla binära strängar.

$$\mathcal{L}((0 \mid 1)^\star) = (\mathcal{L}((0 \mid 1)))^\star = (\mathcal{L}(0) \cup \mathcal{L}(1))^\star = (\{0\} \cup \{1\})^\star = \{0, 1\}^\star =$$
$$\{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$$

$\Diamond$

Exempel. $(0 \mid (1(0 \mid 1)^\star))$ beskriver språket av alla binära strängar utan extra inledande nollor, $\{0, 1, 10, 11, 100, \dots\}$ $\Diamond$

6.2 Utvidgad notation

Reguljära uttryck används ofta för att beskriva hur man får skriva numeriska konstanter och variabelnamn i programspråk. I sådana sammanhang inför man ofta ytterligare några konstruktioner. I stället för att skriva 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 tillåter vi oss $[0-9]$.

Om man vill upprepa något en eller flera gånger skriver man ett plustecken i stället för stjärna, $[0-9]^+ = [0-9][0-9]^\star$.

Ett frågetecken efter ett reguljärt uttryck betyder upprepning ingen eller en gång, $\alpha? \triangleq \alpha \mid \epsilon$. Operatorn har samma precedens som $\star$ och $+$.

Det kan också vara bekvämt att sätta namn på reguljära uttryck och använda dessa namn i andra reguljära uttryck. Den utvidgade notationen innebär inte att man kan beskriva flera språk utan bara att en del beskrivningar blir kortare.

Exempel.

$$\text{DIGIT} \triangleq [0-9]$$
$$\text{NAT} \triangleq \text{DIGIT}^+$$
$$\text{INT} \triangleq (-)?\text{NAT}$$
$$\text{FLOAT} \triangleq \text{INT} \,.\, \text{NAT}$$

$\Diamond$

Det är emellertid inte tillåtet att använda sådana namn rekursivt. En sådan utvidgning behandlas i kapitlet om *grammatik*.

6.3 Reguljära uttryck med Java

I `java.util.regex` finns två klasser för effektiv matchning med reguljära uttryck. Om samma reguljära uttryck skall användas flera gånger ”kompileras” det först:

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

Kompileringen skapar en tillståndsmaskin som representeras av en tabell med tillståndsövergångar. ”Matcharen” innehåller ett tillstånd som beskriver hur långt matchningen kommit i strängen, som alltså kan utföras en bit i taget. Metoden `matches()` försöker matcha det reguljära uttrycket mot hela strängen.

Om det reguljära uttrycket bara skall användas en gång skriver man:

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

7 Grammatiker

De språk som kan beskrivas med reguljära uttryck är tämligen primitiva. Man har bara tre operationer: en för sammasättning, en för alternativ och en för upprepning.

Tidigare i kursen har vi haft anledning att diskutera den *externa* eller *konkreta* representationen av aritmetiska uttryck. Reguljära uttryck är för primitiva för att kunna beskriva mängden av aritmetiska uttryck.

> Ett *uttryck* består av en eller flera *termer* separerade av enkla plus- eller minustecken. En *term* består i sin tur av en eller flera *faktorer* separerade av enkla multiplikations- eller divisions- tecken. En *faktor* är ett *tal*, en *variabel* eller ett *uttryck* inom parenteser. Ett *tal* består av en eller flera siffror och får inledas med ett minustecken. En *variabel* består av en eller flera bokstäver bland a–z.

Vi använde en särskild formell notation, *Backus–Naur–form* (BNF), för detta:

```
expr ::= term (addop term)*
term ::= factor (mulop factor)*
factor ::= NUMBER | NAME | '(' expr ')'
addop ::= '+' | '-'
mulop ::= '*' | '/'
```

Detta kallas för en *grammatik* och den består av *produktioner*. Varje produktion har ett vänsterled som vi kallar *syntaxsymbol* (eng. nonterminal symbol). Syntaxsymbolen är ett namn. Eftersom grammatiken ibland skall översättas till ett program som kan analysera strängar skrivna enligt grammatiken använder vi gärna namn som duger som metodnamn i Java.

Högerledet i en produktion påminner om reguljära uttryck; vi använder samma operatorer, $|$, $*$ och sammansättningsoperatorn (som inte syns). Vi använder parenteser för gruppering. Det som skiljer är att man får använda syntaxsymboler i uttrycket och att symbolerna i alfabetet omges av citationstecken för att skilja dem från syntaxsymboler. Symbolerna i alfabetet kallas för *slutsymboler* (eng. terminal symbols). `NUMBER` och `NAME` är också slutsymboler och betecknar tal resp. variabelnamn. Normalt använder reguljära uttryck för att beskriva hur tal och variabelnman skall skrivas.

I grammatiken ovan är $N \triangleq \{\texttt{expr}, \texttt{term}, \texttt{factor}, \texttt{addop}, \texttt{mulop}\}$ mängden av syntaxsymboler och $\Sigma \triangleq \{+, -, *, /, \texttt{NUMBER}, \texttt{NAME}\}$ slutsymboler.

Ibland gör man tvärt om och dekorerar syntaxsymbolerna i stället för slutsymbolerna.

```
<expr> ::= <term> (<addop> <term>)*
<term> ::= <factor> (<mulop> <factor>)*
<factor> ::= NUMBER | NAME | ( <expr> )
<addop> ::= + | -
<mulop> ::= * | /
```

Varje rad namnger en *syntaxsymbol* och räknar upp beståndsdelarna. Syntaxsymbolen står till vänster om "::=" och beståndsdelarna till höger. Det som står inom parentes och följs av en asterisk får upprepas noll eller flera gånger. När det finns alternativa beståndsdelar separeras dessa av `|`-tecken som utläses *eller*. Det som omges av apostrofer står för sig själv. Vi ser alltså att en `addop` är ett plus- eller minus-tecken och att ett `expr` består av en eller flera `term`er separerade

av plus- eller minus-tecken. Symbolen för alternativ binder svagast så att `'(' expr ')'` är ett av tre alternativ för `factor`.

Vi har nu en *grammatik* för aritmetisk uttryck och kan göra en *härledning* för att visa hur en sträng kan konstrueras. En härledning börjar med grammatikens *startsymbol*, som enligt konvention brukar vara den symbol som finns i vänsterledet i den första raden. I varje härledningssteg ersätter man en syntaxsymbol med motsvarande högerled. Härledningen är klar när det inte finns några syntaxsymboler kvar.

Följande är en härledning av strängen `1+x` eller egentligen av `NUMBER + NAME`. När det inte kan bli några missförstånd i en härledning utelämnar vi citationstecken kring slutsymbolerna .

```
expr ⇒ term addop term ⇒ factor addop term ⇒ NUMBER addop term ⇒
NUMBER addop term ⇒ NUMBER + term ⇒ NUMBER + factor ⇒ NUMBER + NAME
```

Vi utläser ⇒som 'härleder i ett steg'. Ett steg i en härledning innebär att man ersätter en syntaxsymbol med motsvarande högerled från någon produktion.

Om vi vill sammanfatta en sådan härledning i noll eller flera steg skriver vi

`expr` $\Rightarrow^*$ `NUMBER + NAME`

Definition. Låt G vara en grammatik med syntaxsymboler i N, slutsymboler i Σ, produktioner som definierar ⇒och startsymbol S. Mängden av alla strängar i Σ^* som kan härledas från S kallas det språk som *genereras* av grammatiken och betecknas L(G).

$$\mathcal{L}(G) \triangleq \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

△

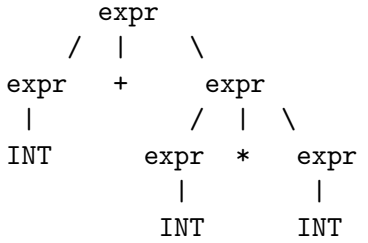
7.1 Härledningsträd

Följande grammatik beskriver enkla aritmetiska uttryck med addition och multiplikation.

$expr$ ::= $expr$'+'$expr$
$expr$ ::= $expr$'*'$expr$
$expr$ ::= INT

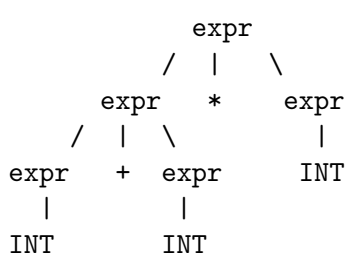
Härledning av `INT + INT * INT` tillsammans med ett träd som visar hur produktionerna har använts. Man kan notera likheten med härledningar i satslogiken.

```
expr  =>
expr + expr =>
expr + expr * expr =>
INT + expr * expr =>
INT + INT * expr =>
INT + INT * INT
```



Med samma grammatik går det att härleda samma sträng på ett annat vis. Härledningsträdet blir fundamentalt annorlunda.

```
expr  =>
expr * expr =>
expr + expr * expr =>
INT + expr * expr =>
INT + INT * expr =>
INT + INT * INT
```



Om vi bygger en abstrakt representation som svarar mot det senare härledningsträdet så blir den inte den avsedda.

En grammatik är *tvetydig* om det finns mer än ett härledningsträd för någon sträng i språket.

Om en grammatik är tvetydig måste man försöka hitta en grammatik som inte är det och som genererar samma språk. I det aktuella fallet vill man ha en grammatik som respekterar gängse precedens för operatorerna.

Hur man konstruerar grammatiker som gör det lätt att bygga rätt abstrakt representation studeras i kursen i Kompilatorteknik.

7.2 Inferensregler

Vi använder samma begrepp, härledning, i samband med grammatiker som vi gjorde i satslogiken. Egentligen är de samma sak fast vi formulerar reglerna olika och ritar träden upp och ner.

I naturlig härledning bevisar vi påståenden. Låt $Expr$ betyda mängden av alla strängar som går att härleda från startsymbolen *expr* i grammatiken

$expr ::= expr\text{'+'}expr$
$expr ::= expr\text{'*'}expr$
$expr ::= INT$

Vi kan då formulera inferensregler som handlar om påståenden av formen $e \in Expr$ med tolkningen att stränge e tillhör språket $Expr$.

$$\frac{e_1 \in Expr \qquad e_2 \in Expr}{e_1 + e_2 \in Expr} \qquad \frac{e_1 \in Expr \qquad e_2 \in Expr}{e_1 * e_2 \in Expr} \qquad \frac{}{INT \in Expr}$$

Den sista regeln är lite speciell; den har inga förutsättningar. Sådana regler brukar kallas *axiom*. I det aktuella fallet vet man alltså att $INT \in Expr$.

En härledning av $INT + INT * INT \in Expr$ är

$$\dfrac{\dfrac{\dfrac{}{INT \in Expr} \qquad \dfrac{}{INT \in Expr}}{INT + INT \in Expr} \qquad \dfrac{}{INT \in Expr}}{INT + INT * INT \in Expr}$$

7.3 Syntaxanlys

Det är enkelt att skriva en metod som avgör om en sträng innehåller ett variabelnamn eller en följd av siffror och returnerar motsvarande `Variable`- eller `Int`-objekt. I kompilatortekniken kallas detta för en *lexikalanalysator*. I `java.io` finns en klass, `StreamTokenizer`, som gör jobbet. I följande kodavsnitt skapas ett sådant objekt som kommer att läsa från en sträng via en `StringReader`.

```
public class ExprParser extends StreamTokenizer {
    private int token;
    public ExprParser(String string) throws IOException  {
        super(new StringReader(string));
        ordinaryChar('-');
        ordinaryChar('/');
        token = nextToken();
    }
```

`StreamTokenizer` initieras så att minus- och divisionstecken behandlas som vanliga tecken. De är annars initierade för att passa analys av programspråk där man har negativa tal och kommentarer som inleds med `'/'`. `nextToken` returnerar ett heltal som anger vad för slags tecken den hittat.

I klassen finns metoder med samma namn som syntaxsymbolerna i grammatiken för uttryck. Metoden `expr` skall analysera ett helt uttryck enligt grammatiken

```
expr ::= term (addop term)*
```

Metoden `term` skall göra sammalunda med

```
term ::= factor (mulop factor)*
```

och `factor` skall klara av

```
factor ::= number | name | '(' expr ')'
```

Vi börjar med `factor`. Om nästa tecken (`token`) är en vänsterparentes så bygger vi ett helt uttryck genom att anropa `expr` rekursivt. Om `token` anger att den hittat ett tal `StreamTT_NUMBER` så finns detta att hämta i attributet `nval`.

```
    private Expr factor() throws IOException {
        Expr e;
        switch (token) {
            case '(' :
                token = nextToken();
                e = expr();
                token = nextToken();
                return e;
            case TT_NUMBER :
                double x = nval;
                token = nextToken();
                return new Int((int) x);
```

```
            case TT_WORD :
                String s = sval;
                token = nextToken();
                return new Variable(s);
        }
    }
```

Om `token` i stället markerar att den hitta ett `StreamTT_WORD` finns strängen att hämta i `sval`.

Metoden `term` skall analysera en eller flera faktorer med hjälp av `factor`.

```
    private Expr term() throws IOException {
        Expr result, factor;
        result = factor();
        while (token == '*' || token == '/') {
            int op = token;
            token = nextToken();
            factor = factor();
            switch (op) {
                case '*' :
                    result = new Mul(result, factor);
                    break;
                case '/' :
                    result = new Div(result, factor);
                    break;
            }
        }
        return result;
    }
```

Slutligen skall `expr` ta hand om termerna. Detta är helt analogt med metoden `factor`. Detta lämnas som övning.

Slutligen definieras `build` som en synonym till `expr`.

```
    public Expr build() throws IOException {
        return expr();
    }
```

8 Relationer

Operatorn $<$ är ett exempel på en *relation*. Det gäller t.ex. att $1 < 2$ och $1 < 3$. Om ett värde inte är relaterat till ett annat med avseende på en relation skriver man ibland ett streck genom relationssymbolen, $2 \not< 1$. Symbolen $\in$ i mängdläran är också en relation. Om m är ett element i mängden M skriver $m \in M$ och om så inte är fallet $m \notin M$.

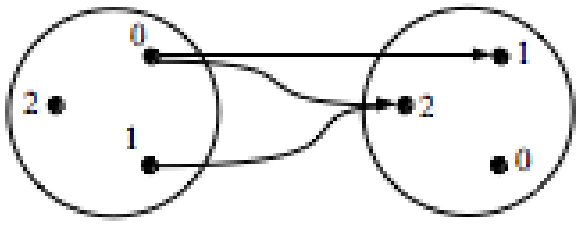
Relationer behöver naturligtvis inte handla om matematiska abstraktioner. Om Per och Erik är personer så kan Per eventuellt vara en vän till Erik. Om man är sociolog och särskilt intresserar sig för relationen "är vän till" kan man introducera en symbol för relationen, Per $\heartsuit$ Erik.

Om man är Prolog-programmerare sätter man ett namn på relationen och skriver `friend(per, erik)` för att ange att Per är vän till Erik. Ett enkelt sätt att beskriva vilka par av personer

som har relationen `friend` är att räkna upp dem. I praktiken är det väl omöjligt att göra det för just denna relation och relationen ändrar sig också med tiden. Idén ligger dock till grund för dem matematiska definition av begreppet *relation*.

> **Definition.** En *relation* från mängden A till mängden B är en delmängd av produktmängden $A \times B$. △

Om $A \triangleq \{a1, a2, a43\}$ och $B \triangleq \{b1, b2, b3\}$ och relationen $\rho \triangleq \{(a2, b1), (a2, b2), (a3, b1)\}$ så kan vi illustrera relation



Vi ser att $a1$ inte är relaterat till något element i B, $a2$ är relaterat till två element i B och $a3$ är relaterat till ett element.

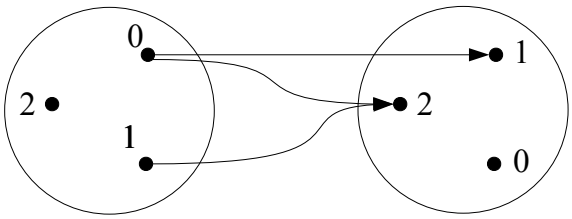
Vi kommer att använda symbolen ρ när vi talar om en relation i allmänhet och inte vill specificera vilken. Vi använder symbolen både när vi tänker på mängden av relaterade par och som relationsoperator. Vilken användning det handlar om framgår i regel av sammanhanget. För att vara tydliga använder vi ibland (ρ) för att markera att det är mängdaspekten som avses. Formellt sett definieras användningen av operatorn med hjälp av mängden:

> **Definition.** $x \, \rho \, y \triangleq (x, y) \in \rho$. △

Om S är en mängd studenter och C en mängd kurser så kan vi definiera en relation från S till C som beskriver vilka studenter som läser vilka kurser med en tabell med två kolonnner där varje rad innehåller en student och en kurs som studenten läser. Det är på detta sätt man gör i en relationsdatabas. Ordningen mellan raderna i tabellen är inte signifikant och den matematiska abstraktionen är återigen en mängd av par.

Våra exempel har handlat om relationer från en mängd med två operander. Sådana kallas *binära relationer*. När vi har en binär relation från en mängd A till samma mängd så säger vi att det är en *relation på* A.

Relationen $<$ på mängden $\mathbb{N}_3 \triangleq \{0, 1, 2\}$ ges av $\{(0, 1), (0, 2), (1, 2)\}$ och illustreras med följande figur



En relation som man så gott som alltid definierar på en mängd A är likhetsrelationen, $=$.

$$(=) \triangleq \{(x, x) \mid x \in A\}$$

Om vill vara noggranna och indikera att likhetsrelationen hör till mängden A dekorerar vi symbolen som i $=_A$. $=_{\mathbb{N}_3}$ definieras alltså som mängden $\{(0,0),(1,1),(2,2)\}$.

Aktivitet 1.
När vi skriver $n \in \{0, 2, 4, \dots\}$ så är $\in$ förmodligen en relation från $\mathbb{N}$ till en annan mängd. Vilken?

Formellt är relationen $<$ mängden av alla par (x, y) sådana att x är mindre än y. Vi kan inte gärna definiera $<$ genom att använda $<$. Vi får göra på följande sätt:

$$(<) \triangleq \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid \exists z \in \mathbb{N} \,.\, x + z + 1 = y\}$$

Eftersom relationer är mängder kan vi använda de vanliga mängdoperationerna på relationer.

Exempel. Ett enkelt sätt att definiera relationen $\leq$ är

$$(\leq) \triangleq (<) \cup (=)$$

Detta kan se kryptiskt ut för den oinvigde, men om man bara tittar på symbolerna så ser det vettigt ut; om vi lägger till (halva) likhetstecknet i $<$ så blir det $\leq$. $\Diamond$

9 Java-modell

En enkel Java-modell av en relation använder en lista av par för att representera relationen.

```
public class Relation<A, B> {
  private List<Pair<A, B>> relation = new ArrayList<Pair<A, B>>();
  public void add(A a, B b) {
    set.add(new Pair<A, B>(a,b));
  }
  public boolean related(A a, B b) {
    return relation.contains(new Pair<A, B>(a,b));
  }
}
```

Klassen `Pair` följer.

```
public class Pair<A, B> {
  private A a;
  private B b;
  public boolean equals(Object object) ...
  public int hashCode() ...
}
```

Beroende på vilka operationer som skall implementeras för relationer kan andra representationer vara lämpliga.

9.1 Operationer

Exempel. Om vi har en mängd av fyra personer $\{\texttt{per}, \texttt{anna}, \texttt{erik}, \texttt{eva}\}$ och `anna` och `per` är föräldrar till `erik` och `eva` så kan detta beskrivas med relationen $\texttt{parent} \triangleq \{(\texttt{per}, \texttt{erik}), (\texttt{per}, \texttt{eva}), (\texttt{anna}, \texttt{eva}), (\texttt{anna}, \texttt{erik})\}$. $\Diamond$

För en given relation definierar vi *domänen* (eng. domain) som mängden av alla förstakomponenter och kodomänen (eng. codomain eller range) som mängden av alla andrakomponenter.

Definition. Låt ρ vara en relation från A till B.

$$\mathrm{dom}\,(\rho) \triangleq \{x \in A \mid \exists y \,.\, (x, y) \in \rho\}$$
$$\mathrm{ran}\,(\rho) \triangleq \{y \in B \mid \exists x \,.\, (x, y) \in \rho\}$$

$\triangle$

Med $\texttt{parent} \triangleq \{(\texttt{per}, \texttt{erik}), (\texttt{per}, \texttt{eva}), (\texttt{anna}, \texttt{eva}), (\texttt{anna}, \texttt{erik})\}$ så är

$$\mathrm{dom}\,(\texttt{parent}) = \{\texttt{per}, \texttt{anna}\}$$
$$\mathrm{ran}\,(\texttt{parent}) = \{\texttt{erik}, \texttt{eva}\}$$

Man definierar *bilden* av x under relationen ρ, $x\,\rho$, samt *argumentet* till y under ρ, $\rho\,y$:

Definition. Låt ρ vara en relation från A till B med $x \in A$ och $y \in B$.

$$x\,\rho \triangleq \{y \in B \mid (x, y) \in \rho\}$$
$$\rho\;y \triangleq \{x \in A \mid (x, y) \in \rho\}$$

$\triangle$

Om `parent`-relation ovan betecknas med $\downarrow$ så har vi

$$\texttt{anna} \downarrow = \{\texttt{eva}, \texttt{erik}\}$$
$$\downarrow \texttt{erik} = \{\texttt{per}, \texttt{anna}\}$$

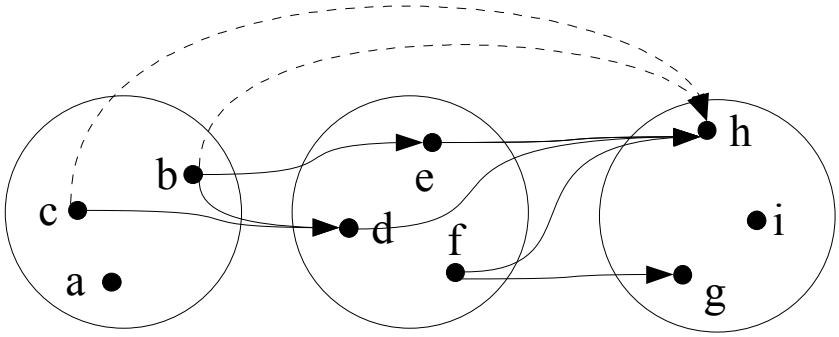
Aktivitet 1.
Finns det något enkelt uttryck som betecknar mängden av alla relationer från A till B?

Låt oss inkludera några barnbarn i familjen ovan
$\texttt{parent} \triangleq \{(\texttt{per}, \texttt{erik}), (\texttt{per}, \texttt{eva}), (\texttt{anna}, \texttt{eva}), (\texttt{anna}, \texttt{erik}), (\texttt{eva}, \texttt{moa}), (\texttt{erik}, \texttt{sune})\}$.
Vi kan då 'sätta samman' relationen `parent` med sig själv för att bilda relationen `grandparent`: Om a är förälder till b och b är förälder till c så är a mor/far-förälder till c. Jag skulle vilja använda operatorn $\circ$ för sammansättning av relationer, men det är upptagen för besläktad operation (sid. 49). I denna text använder symbolen ; som vi använder för att sätta samman satser i ett programspråk.

$$\texttt{grandparent} \triangleq (\downarrow)\,;\,(\downarrow)$$

`grandparent` blir mängden $\{(\texttt{per}, \texttt{moa}), (\texttt{per}, \texttt{sune}), (\texttt{anna}, \texttt{moa}), (\texttt{anna}, \texttt{sune})\}$.

Generellt kan vi sätta samman två relationer om "mängen i mitten" är gemensam. I figuren är den sammansatta relationen indikerad med streckade pilar.



I den sammansatta relationen finns två par: $\{(\mathrm{b},\mathrm{h}),(\mathrm{c},\mathrm{h})\}$.

Definition. Låt ρ_1 vara en relation från A till B och ρ_2 en relation från B till C. *Sammansättningen* av ρ_1 och ρ_2 är

$$\rho_1 \,;\, \rho_2 \triangleq \{(x,z) \in A \times C \mid \exists y \in B \,.\, (x,y) \in \rho_1 \ \wedge (y,z) \in \rho_2\}$$

△

Exempel. Låt

$$\begin{aligned} A &\triangleq \{0,1,2\} \\ \rho &\triangleq \{(0,1),(0,2),(1,2)\} \\ \rho_2 &\triangleq \{(0,1),(0,2),(1,2),(2,2)\} \end{aligned}$$

Då är

$$\rho_1 \,;\, \rho_2 \triangleq \{(0,2),(1,2)\}$$

◊

Definition. Låt ρ vara en relation från A till B. *Inversen* till ρ är

$$\rho^{-1} \triangleq \{(y,x) \in B \times A \mid (x,y) \in \rho\}$$

△

Exempel. Relationen $\geq$ är inversen till $\leq$. ◊

Definition. Låt ρ vara en relation från A till B. *Motsatsen* till ρ är

$$\overline{\rho} \triangleq A \times B - \rho$$

△

Exempel. Relationen $>$ är motsatsen till $\leq$. ◊

Definition. Låt ρ vara en relation på A. Vi definierar ρ^n där $n \in \mathbb{N}$ med

$$\begin{array}{rcll} \rho^0 & \triangleq & =_A & \\ \rho^{n+1} & \triangleq & \rho \,;\, \rho^n, & n > 0 \end{array}$$

Symbolen $=$ i den första raden är likhetsrelationen för en aktuella mängden, $= \triangleq \{(x, x) \mid x \in A\}$. △

Definition. Det *transitiva höljet* till ρ är

$$\rho^\star \triangleq \bigcup_{i=0}^{\infty} \rho^i$$

△

Vi sätter namn på en del egenskaper som en relation på en mängd kan ha.

Definition. Låt ρ vara en relation på A.

ρ är *reflexiv* om $(x, x) \in \rho$ för alla $x \in A$

ρ är *symmetrisk* om $(x, y) \in \rho \rightarrow (y, x) \in \rho$ för alla $x, y \in A$

ρ är *transitiv* om $(x, y) \in \rho \wedge (y, z) \in \rho \rightarrow (x, z) \in \rho$ för alla $x, y, z \in A$

ρ är *antisymmetrisk* om $(x, y) \in \rho \wedge (y, x) \in \rho \rightarrow x = y$ för alla $x, y \in A$

△

Flera av de relationer vi sett i detta kapitel är reflexiva: $=, \leq, \subseteq$. Av dessa tre är det bara $=$ som också är symmetrisk. Relationen 'vara syskon till' (eng. sibling) är en symmetrisk relation som inte är reflexiv; man är knappast syskon till sig själv.

$=, \leq, \subseteq$ är alla transitiva medan `sibling` inte är det. $=, \leq, \subseteq$ är också antisymmetriska.

Vi har tidigare noterat att relationen $=$ normalt är både reflexiv, symmetrisk och transitiv.

Aktivitet 2.
Låt ρ vara en relation på A. Kontrollera att följande påståenden är korrekta:

ρ är reflexiv precis då $(=_A) \subseteq (\rho)$

ρ är symmetrisk precis då $(\rho^{-1}) = (\rho)$

ρ är transitiv precis då $(\rho) \,;\, (\rho) = (\rho)$

ρ är antisymmetrisk precis då $(\rho) \cup (\rho^{-1}) = (=_A)$

9.2 Flerställiga relationer

Man kan också definiera relationer mellan tre eller flera mängder analogt. Ordet 'mellan' kan vara vilseledande eftersom ordningen mellan mängderna är signifikant. En relation mellan mängderna A, B och C är alltså en mängd trippler eller en delmängd av $A \times B \times C$. I en databas kan man ha en relation där varje trippel innehåller ett personnummer, ett namn och en adress.

> **Definition.** En n-ställig relation mellan n mängder $A_1, \ldots, A_n$ är en delmängd av den kartesiska produkten $A_1 \times \cdots \times A_n$. △

En 1-ställig relation kallas för en *egenskap* (property). $even \triangleq \{0, 2, 4, \ldots\}$ är en egenskap på $\mathbb{N}$.

9.3 Funktioner

I kursen i endimensionell analys definieras funktioner genom att man ger funktionen ett namn, talar om vad variabeln heter och anger ett uttryck som innehåller variabeln

> **Exempel.**
>
> $$f(x) \triangleq x + 1$$
> $$sqr(y) \triangleq y^2$$
> $$abs(x) \triangleq \begin{cases} x, & \text{om } x \geq 0 \\ -x, & \text{om } x < 0 \end{cases}$$
>
> ◊

Detta sätt att definiera funktioner förutsätter att vi kan konstruera ett matematiskt uttryck som ger funktionsvärdet för ett givet argument. Det är inte lämpligt att basera definitionen av begreppet *funktion* på detta sätt att beskriva funktioner.

I stället gör vi precis som med binära relationer.

Låt oss börja med den logiska funktionen *not* som tar ett sanningsvärde som argument och lämnar det motsatta sanningsvärdet som värde.

$$not(b) = \begin{cases} \mathsf{F}, & \text{om } b = \mathsf{T} \\ \mathsf{T}, & \text{om } b = \mathsf{F} \end{cases}$$

Denna funktion representeras av $\{(\mathsf{F}, \mathsf{T}), (\mathsf{T}, \mathsf{F})\}$ och vi kan skriva $not = \{(\mathsf{F}, \mathsf{T}), (\mathsf{T}, \mathsf{F})\}$.

Funktionen f ovan representeras av $\{(x, x + 1) \in \mathbb{R} \times \mathbb{R} \mid x \in \mathbb{R}\}$.

Eftersom en funktion är en mängd av par så är det en binär relation, men alla binära relationer är inte funktioner; det som särskiljer funktionerna är att två olika par får inte ha samma första komponent.

> **Definition.** f är en funktion från A till B om och endast om
>
> $$f \subseteq A \times B \text{ och } (a_1, b_1), (a_1, b_2) \in f \rightarrow (b_1 = b_2)$$
>
> $$A \rightarrow B \triangleq \{f \subseteq A \times B \mid (a_1, b_1), (a_1, b_2) \in f \rightarrow (b_1 = b_2)\}$$
>
> △

Den mängd som definierar funktionen kallas också för funktionens *graf*.

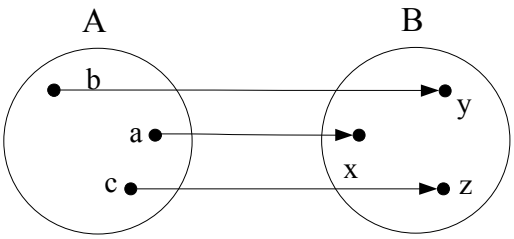
Vi har tidigare definierat sammansättning av relationer. Eftersom en funktion är en relation så innebär det att vi redan har en opertion för sammansättning av funktioner. Emellertid brukar man ange operanderna i omvänd ordning när man sätter samman funktioner. Den konventionella symbolen är $\circ$.

> **Definition.** Om $f \in B \to C$ och $g \in A \to B$ så definieras funktionen $f \circ g$ med $(f \circ g)(x) \triangleq f(g(x))$. $\triangle$

Tydligen gäller att $f \circ g = g \,;\, f$.

9.4 Oändliga mängder

Kan man avgöra om två ändliga mängder har samma kardinalitet utan att räkna elementen? Jo, det finns enklare sätt som dessutom går att använda när vi skall definiera kardinalitet för oändliga mängder. Vi säger att två mängder har samma kardinalitet om det går att para ihop varje element i en ena mängden med ett element i den andra mängden så att varje element finns i exakt ett par.



Man brukar använda begreppet *bijektion* för att beskriva en sådan hopparning.

> **Definition.** En funktion $f \in A \to B$ är en *bijektion* om varje element i A finns som förstakomponent i exakt ett par och varje element i B förekommer som andrakomponent i exakt ett par f. $\triangle$

Att f är en bijektion betyder att f avbildar A på hela B och att f har en invers som avbildar B på hela A.

Nu kan vi definiera vad som menas med att två mängder har samma kardinalitet.

> **Definition.** Mängderna A och B har samma kardinalitet om det finns en bijektion $f \in A \to B$. $\triangle$

> **Exempel.** Låt C vara mängden av all syntaktiskt korrekta Java-klasser. Vi skall visa att C har samma kardinalitet som $\mathbb{N}$. C är ett språk och en av de kortaste strängarna i språket är `class A{}`. Vi kan ordna strängarna efter hur långa de är med de kortaste först. Strängar som är lika långa ordnas lexikografiskt som i en ordlista. Detta gör att vi kan numrera elementen i C med talen $0, 1, \ldots$ och vi har en bijektion från $\mathbb{N}$ till C. $\Diamond$

En mängd som har samma kardinalitet som en delmängd till $\mathbb{N}$ sägs vara *uppräknelig*. Eftersom $\mathbb{N}$ är en delmängd till sig själv så innebär detta att en uppräknelig mängd kan vara ändlig eller oändlig.

Det är lätt se att $\mathbb{N}$ och mängden av de jämna naturliga, $\{2n|n \in \mathbb{N}\}$, talen har samma kardinalitet. Funktionen $f(n) \triangleq 2n$ är en bijektion. Det kan tyckas förvånande att en äkta delmängd av en mängd kan ha samma kardinalitet som mängden själv, men oändliga mängder har en del oväntade egenskaper. Just denna används i själva verket för att definiera begreppet *oändlig* mängd.

> **Definition.** En mängd A är *oändlig* om det finns en äkta delmängd $B \subset A$ så att $|A| = |B|$. $\triangle$

Om A är en ändlig mängd så är $|A|$ ett vanligt naturligt tal. Om A är oändlig så skulle man kunna tro att kardinaliteten är ∞. Vi skall snart se att det finns oändliga mängder med olika kardinalitet, så det finns tydligen flera olika "oändligheter".

> **Exempel.** Vi skall visa att mängden av de reella talen i intervallet $[0, 1)$ inte har samma kardinalitet som $\mathbb{N}$. Vi använder decimalbråksutvecklingar för att representera talen. Många tal som har oändliga decimalbråksutvecklingar, t ex $1/3, \pi$ och e. Det finns en liten komplikation med de tal som har ändliga utvecklingar; sådana tal (utom talet 0) har två decimalbråkstvecklingar. Till exempel så representerar $0.4999\ldots$ och 0.5 samma tal. Vi väljer att inte använda den representation där utvecklingen slutar med oändligt många nior, men fylla på med nollor.
>
> Antag nu att det finns en bijektion, f, från $\mathbb{N}$ till mängden av sådana utvecklingar.
>
> $$\begin{aligned} f(0) &= 0.d_{00}d_{01}d_{02}d_{03}\ldots \\ f(1) &= 0.d_{10}d_{11}d_{12}d_{13}\ldots \\ f(2) &= 0.d_{20}d_{21}d_{22}d_{23}\ldots \\ f(3) &= 0.d_{30}d_{31}d_{32}d_{33}\ldots \\ &\vdots \end{aligned}$$
>
> Varje d_{ij} är en av siffrorna 0 till 9. Vi konstruerar nu ett tal $0.d_0d_1d_2\ldots$, där
>
> $$d_i \triangleq \begin{cases} 0 & \text{om } d_{ii} \neq 0 \\ 1 & \text{om } d_{ii} = 0 \end{cases}$$
>
> Detta är ett tal i samma intervall. Vi konstaterar att detta tal inte finns med i uppräkningen eftersom åtminstone en siffra skiljer det från varje tal i uppräkningen. Detta motsäger antagandet att f är en bijektion. Vi drar sålunda slutsatsen att det inte finns någon bijektion. $\Diamond$

> **Definition.** Om A har samma kardinalitet som en delmängd av B skriver vi $|A| \leq |B|$. Om $|A| \leq |B|$ och A och B inte har samma kardinalitet så skriver $|A| < |B|$. $\triangle$

Vi har sålunda visat att $|\mathbb{N}| < |\mathbb{R}|$.

Man kan visa att om $|A| \leq |B|$ och $|B| \leq |A|$ så är $|A| = |B|$. Beviset är dock lite för långt för att vi skall ta med det här. Man kan också visa att för två godtyckliga mängder så gäller precis en av $|A| < |B|$, $|A| = |B|$ och $|B| < |A|$.

9.5 Lambda-notation

I så kallade *funktionella* programspråk är alla program funktioner och dessa kan funktioner både som parametrar och resultat. Det finns inga variabler och satser.

Det första språket av denna typ var Lisp som konstruerades i slutet av 1950-talet. Lisp är inte rent funktionellt eftersom det också innehåller variabler och möjlighet att ge dem värden.

Haskell är ett av de moderna funktionella programspråken. Det har ett rikt typsystem, förmåga att beräkna typen för (nästan) alla uttryck utan att programmeraren behöver göra några typdeklarationer. Språket har också *lat evaluering*. Det betyder att ett uttrycks värde inte beräknas förrän det behövs vilket tillåter att man till exempel kan programmera med oändliga listor så länge man inte använder så stor del av dem att de inte ryms i minnet.

Om man vill definiera funktionen $suc(x) \triangleq x + 1$ i Haskell skriver man

`suc(x) = x+1` eller bara `suc x = x+1`

Haskellsystemet räknar ut typen för `suc` och anger den som `Integer -> Integer`. Typen `Integer` är fördefinierad och kan användas för godtyckligt stora heltal.

Man beräknar $suc(3)$ med

`suc 3` och får resultatet med dess typ `4 :: Integer`

Man kan definiera samma funktion med så kallad *lambda-notation*.

`suc = \x -> x+1` eller använda den utan att ge den ett namn `(\x -> x+1) 3`

med samma resultat som tidigare.

När man inte är begränsad av vilka tecken som finns på tangentbordet använder man den grekiska bokstaven λ i stället för `\` och brukar ersätta pilen med en punkt

$$\lambda x \,.\, x + 1$$

Man kan definiera funktioner av flera variabler

$\lambda x \,.\, \lambda y \,.\, x + y$ eller i Haskell `plus = \x -> \y -> x+y`

och använda dem

$(\lambda x \,.\, \lambda y \,.\, x + y)\ 1\ 3$ resp. `plus 1 3` som båda ger resultatet 4.

Både i Haskell och med lambda-notation behöver man inte ge en funktion av flera variabler alla sina argument.

$(\lambda x \,.\, \lambda y \,.\, x + y)1 = (\lambda y \,.\, 1 + y)$ eller i Haskell `suc = (\x -> \y -> x+y) 1`

I båda fallen är resultatet en funktion av en variabel som adderar 1 till sitt argument.

10 Problem

Hur många olika program finns det? Svaret är oändligt många, åtminstone om vi inte sätter någon gräns för hur långa de får vara. Det finns dock mer att säga om detta.

Går det att skriva en metod för varje funktion från $\mathbb{N}$ till $\mathbb{N}$? I Java får man problem när talen blir så stora att de inte ryms i en `long`, men det gå att hantera genom att representera tal med `java.math.BigInteger` eller med vektorer med tillräckligt många `long`-element. Vi bortser i fortsättningen från sådana praktiska problem och antar att det finns en datatyp som kan användas för hur stora naturliga tal som helst med aritmetiska operationer på den. Det finns programspråk där man kan räkna med så stora helttal som det tillgänglig minnet tillåter. Haskell är ett sådant språk.

Vi skall visa, att även om vi bortser från att dagens datorer inte kan utrustas med hur mycket minne som helst så finns det funktioner i $\mathbb{N} \to \mathbb{N}$ som inte går att implementera. Vi kommer att göra det genom att jämföra kardinaliteterna för mängden P av alla program som beräknar sådana funktioner och $\mathbb{N} \to \mathbb{N}$.

Vi skall först konstatera att mängden P är *uppräknelig.*

> **Definition.** En mängd M är uppräknelig om det finns en funktion $f \in \mathbb{N} \to M$ sådan att $\text{ran}\,(f) = M$. $\triangle$

Det innebär ingen inskränkning att föreskriva att funktionen skall vara 1-till-1, dvs om $f(n_1) = f(n_2)$ så är $n_1 = n_2$.

Mängden $\mathbb{N}$ själv är uppenbarligen uppräknelig. Funktionen $f(n) \triangleq n$ uppfyller kravet i definitionen.

Aktivitet 1.
Visa att mängden av de hela talen, $\mathbb{Z}$, är uppräknelig genom att konstruera en funktion i $\mathbb{N} \to \mathbb{Z}$ som har hela $\mathbb{Z}$ som kodomän.

Åter till mängden P. Vi visar att den är uppräknelig genom att beskriva hur man kan "räkna upp elementen" vilket innebär samma sak som att beskriva funktionen f i definitionen av *uppräknelig.* Vi räknar upp metoderna i ordning efter hur många tecken de består av. Bland de kortaste metoderna kommer vi att hitta den som beräknar funktionen $g(x) \triangleq 0$.

Det kommer att finnas flera metoder i uppräkningen som är lika långa. Då använder vi "bokstavsordning" för att göra uppräkningen entydigt bestämd. När metoderna blir mer komplicerade med `while`-satser och rekursion kan det vara svårt att avgöra om exekveringen av metoden kommer att terminera för alla värden på parametern. Vi återkommer till denna "praktiska" svårighet.

Nu till frågan om uppräkningen kommer att innehålla alla funktioner i $\mathbb{N} \to \mathbb{N}$? Det gör den inte! Vi kan nämligen konstruera följande funktion med hjälp av uppräkningen.

Låt $f_i(x)$ vara den funktion som beräknas av metod nummer i i uppräkningen. Definiera

$$g(x) \triangleq f_x(x) + 1$$

Detta är uppenbarligen en funktion i $\mathbb{N} \to \mathbb{N}$. Ingen av metoderna i uppräkningen kommer att beräkna g eftersom metod nummer i ger fel värde åtminstone om argumentet är i.

Slutsatsen blir att det finns funktioner som inte går att implementera.

Vi kan använda samma sorts resonemang för att visa att mängden $\mathbb{N} \to \mathbb{N}$ inte är uppräknelig. Vi antar motsatsen, dvs att det finns en uppräkning,

$$f_0(n), f_1(n), f_2(n), \ldots$$

Funktionen $g(n) \triangleq f_n(n)+1$ tillhör $\mathbb{N} \to \mathbb{N}$ men saknas i uppräkningen. Detta är en motsägelse och vi konstaterar att det inte finns någon sådan uppräkning.

Denna bevisteknik brukar kallas Cantors diagonalbevis; man ändrar på något i "diagonalelementet" i uppräkningen. Man kan använda den för att visa att $\mathbb{R}$ inte är uppräknelig.

10.1 Haltproblemet

Att det finns funktioner som inte går att implementera gör inget om de ser ut som exemplet i föregående avsnitt. Finns det då funktioner som man skulle vilja implementera, men där det är omöjligt? Ja, det gör det.

Den viktigaste egenskapen som ett program skall ha är att det är korrekt, dvs att det gör det som är avsett. För ett program som skall beräkna något är viktigt att programmet till att börja med terminerar. Det är inte alltid enkelt att se. Tag till exempel följande program.

```
public int f(int x) {
  int count = 0;
  while (x>1) {
    count++;
    if (x%2==0) {
       x = x/2;
    } else {
       x = 3*x+1;
    }
  }
  return count;
}
```

Kommer det att terminera för alla värden på `x`?

Svaret är att ingen vet. Det är möjligt att någon kommer att bevisa att så är fallet. Det är också möjligt att någon hittar ett `x` där metoden inte terminerar och kan visa detta. I båda fallen kommer resultatet att orsaka stor uppmärksamhet bland matematiker.

Detta indikerar att man kanske inte kan skriva ett program som analyserar en metod och avgör om den kommer att terminera för givna värden på parametrarna. Redan på den första övningen i kursen gjorde vi en objektorienterad modell av program i ett programspråk. Språket innehöll inte metoder och metodanrop, men det är inte särskilt svårt att utvidga modellen så att den klarar detta också. Antag alltså att vi har klasser som modellerar metoder och metodanrop:

```
public class Method {
  private String name;
  private List<Parameter> parameters;
  private Type returnType;
  private Statement body;
  \\ omissions
}
```

och

```
public class Call {
  private String name;
  private List<Expr> arguments;
  \\ omissions
}
```

Antag att vi kan implementera en metod som avgör om ett metodanrop terminerar för givna värden på parametrarna. Då kan vi också implementera specialfallet där metodanropet bara har en strängparameter.

```
public boolean terminates(Method method, String data)
```

Låt `Method method` vara representationen av följande program.

```
public void loop(Method method) {
  if (terminates(method, method.toString())
       while(true) {};
  }
}
```

Vi ställer nu frågan: 'Kommer `loop(method)` att terminera?'.

Om så är fallet kommer `terminates(method, method.toString())` att returnera `true` och anropet går in i en loop som inte terminerar; en motsägelse.

Lika illa blir det om så inte är fallet för då kommer anropet att terminera genast.

Motsägelserna gör att det måste vara fel på antagandet om att `terminates` går att implementera.

Den fråga vi försökt besvara brukar på engelska kallas för 'the halt problem'. På svenska borde man säga 'termineringsproblemet'. De som känner till problemet kommer att förstå vad man menar men skulle själva kanske säga 'haltproblemet' eller 'stopproblemet'.

Om man ändrar frågan till 'Kommer metoden `m` att terminera på en dator med 4 Gb minne?' så blir problemet i teoretisk mening *avgörbart*. Genom att undersöka om innehållet i datorns minne och alla register blir detsamma två gånger så kommer metoden `m` inte att terminera. Om vi skall hålla reda på alla tillstånd som ett minne på 4 Gb kan ha behövs en hårddisk som kan lagra $2^{2^{4 \cdot 2^9}}$ bitar. Ett sådant minne kommer knappast att kunna konstrueras. Talet är mycket större än antalet elementarpartiklar i det universum som vi nu känner till.

10.2 Ohanterliga problem

Vi har nu sett att det finns program som vore utomordentligt bra att ha som är omöjliga att skriva. Vi skall i detta avsnitt visa problem som också är angelägna att lösa med dator, men där det sannolikt inte är möjligt att göra det. Att man inte riktigt vet är att av datavetenskapens stora olösta frågeställningar.

> **Exempel.** Antag att man vill planera resrutten för en handelsresande som skall besöka 30 företag så att resvägen blir så kort som möjligt. Som indata till problemet behöver man avstånden mellan alla par av företag, dvs en matris

```
int distance[][] = new int[30][30];
```

Det enklaste sättet att lösa problemet är att testa alla möjliga rutter och välja den som är kortast. Antalet rutter är 30! = 265252859812191058636308480000000. Så även om man kan testa en rutt på en nanosekund så kommer det att ta 8411113 miljarder år. Datorn kommer säkert att sluta fungera långt tidigare. ◊

Ett annat problem som snabbt blir ohanterligt när storleken växer handlar om satslogik. Frågan är helt enkelt huruvida ett givet satslogiskt uttryck är sant för någon uppsättning av värden på variablerna. Ett enkelt sätt att ta reda på svaret är testa alla uppsättningar. En enkel modifikation av `testAll` i den första laborationen kan göra detta. Haken är bara att om antalet variabler är 40 stycken så räcker inte all tid i världen till för att testa alla kombinationer. Idag känner man inte till någon metod för att besvara frågan som är väsentligt effektivare.

Många kombinatoriska problem har samma karaktär; det kan gälla schemaläggning, packningsproblem, "färgläggning" av grafer, etc. Ett annat exempel är följande.

Exempel. Givet en mängd av hela tal. Finns det någon icketom delmängd så att summan av alla element är 0.

För mängden $\{-7, -3, -2, 5, 8\}$ är svaret ja. ◊

Det finns naturligtvis också kombinatoriska problem som är hanterliga även de växer. Ett sådant är att bestämma kortaste vägen mellan två orter givet en tabell med längden av alla vägar som förbinder två orter utan att passera någon annan ort.

10.3 P och NP

Ett problem kallas för ett *beslutsproblem* (eng. decision problem) om lösningen eller svaret, givet indata, är 'ja' eller 'nej'. Exemplet med summan av elementen i en delmängd i föregående avsnitt är typiskt.

Exemplet med handelsresanden är inte ett beslutsproblem utan ett *optimeringsproblem*. Följande variant av problemet är ett beslutsproblem. 'Finns det en rutt som är kortare än 300 km?'

En del beslutsproblem är lätta eller hanterliga medan andra är ohanterliga. Man har definierat några *problemklasser* för att karaktärisera svårigheten hos olika beslutsproblem. I det här sammanhanget betyder 'klass' precis samma sak som 'mängd'. Klassen P omfattar alla beslutsproblem som lösas på *polynomtid*. För att definiera detta begrepp måste man vara överens om hur man skall mäta storleken på indata till ett problem och vilken dator som man använder. Med tanke på den utveckling på datorsidan som varit sedan de första elektroniska datorerna konstruerades för mer än 60 år sedan vill man inte knyta definitionen till vad dagens datorer kan. Man använder i stället en matematisk modell av en primitiv dator, en *Turingmaskin*. Man konstruerar en ny Turingmaskin för varje "program" man vill exekvera. Kvalitativt innebär detta inget om vi använder en modern dator. De problem som kan lösas på polynomtid med moderna datorer är precis de som kan lösas på polynomtid med en Turingmaskin. Polynomen kommer naturligtvis att vara olika, både beträffande gradtal och koefficienter.

Storleken på indata mäts med längden av den sträng som behövs för att representera informationen. För summationsproblemet i exemplet är det alltså längden av strängen `{-7,-3,-2,5,8}` som gäller. Det är inte väsentligt exakt hur man representerar t ex tal så länge alternativen är "polynomiskt relaterade". Om använder binär representation så blir strängarna ca 3 gånger så långa och relationen kan beskrivas med ett polynom av grad 1. Däremot är det inte tillåtet att representera talet n med n stycken ettor. Då blir relationen mellan längderna exponentiell.

Definition. Ett beslutsproblem tillhör P om det finns ett polynom $p(n)$ sådant att en Turingmaskin kan lösa problemet för givna indata på en tid som är mindre än $p(n)$ där n är längden på indata. △

Problemet att bestämma kortaste vägen mellan två orter tillhör P. Idag vet man inte om beslutsproblemet för hadelsresanden gör det.

Nästa problemklass kallas för NP som står för 'nondeterministic polynomial', "ickedeterminstiskt polynomiellt". Vi skall förklara vad 'ickedeterminstisk' betyder i sammanhanget i ett följande avsnitt.

Att ett beslutsproblem tillhör NP betyder ungefär att om svaret på frågan i problemet är 'ja' så går det att kontrollera svaret på polynomtid med hjälp av ett *vittne*. Ett vittne i summationsproblemet är en delmängd med summan 0 och ett vittne i handelsresandeproblemet är en rutt som underskrider den angivna gränsen. Alternativt kan man säga att det går att lösa beslutsproblem på exponentiell tid.

Av definitionerna följer att

$$\mathrm{P} \subseteq \mathrm{NP}$$

Idag vet man inte om det finns element i NP som inte tillhör P, dvs man vet inte om $\mathrm{P} = \mathrm{NP}$ eller $\mathrm{P} \neq \mathrm{NP}$. Det är få som tror att det är samma mängd, men ingen har lyckats visa det ena eller det andra. Om det funnes ett Nobelpris i matematik skulle en sådan bedrift säkert belönas.

Det finns en stor mängd av intressanta problem i NP som tillhör de "svåraste" i NP. Om man lyckas visa att ett av dessa problem i själva verket tillhör P så betyder det att alla problemen i NP kan lösas på polynomtid, dvs $\mathrm{P} = \mathrm{NP}$. Denna klass av svåraste problem i NP kallas för NP-kompletta, (eng. NPC, NP complete). De tre exempel på problem i NP som vi diskuterat ingår alla i NPC.

10.4 Turingmaskiner

En *Turingmaskin* är en matematisk modell av mycket primitiv dator. Den består av en tillståndsmaskin som kan läsa och skriva symboler på ett "band". Bandet är Turingmaskinens "primärminne". Tillståndsmaskinen är Turingmaskinens "program". Genom att byta tillståndsmaskin kan man beräkna olika saker.

Det som gör modellen viktig är att allt som går att beräkna med en riktig dator också går att beräkna med en Turingmaskin. Turingmaskinen är i ett avseende mer kapabel än en verklig dator. Turingmaskinens band är obegränsat även om varje terminerande beräkning bara utnyttjar en begränsad del av bandet. Detta kompenseras i viss mån av att man kan komplettera en riktig med mer minne om det behövs så länge tillverkarna av hårddiskar och andra minnen kan producera.

Turingmaskinen är inte lika effektiv som en verklig dator, men exekveringstiderna är polynomiskt relaterade. Om en verklig dator kan lösa ett problem på polynomtid så kan en Turingmaskin också göra det.

Bandet används för att ge indata till en beräkning, vara minne under beräkningen och att presentera resultatet. Indata och resultat anges som strängar på ett alfabet, Σ, som beror på vad som skall beräknas. De symboler som kan finnas på bandet är Σ plus två speciella symboler, en *blanksymbol*, $\sqcup$, och en symbol '$' som används som "markör". Vi kallar detta för *bandalfabetet*, $\Sigma_t \triangleq \Sigma \cup \{\sqcup, \$\}$. Vid varje tillståndsövergång utföres också en *bandoperation*. Maskinen har ett läs/skrivhuvud som alltid finns vid en symbol på bandet. De möjliga operationerna är att skriva en symbol ur Σ på bandet, att flytta huvudet ett steg åt vänster eller ett steg åt höger.

Vi använder symbolerna i Σ_t tillsammans med `L` och `R` för att beteckna dessa operationer, $\Sigma_h = \Sigma \cup \{\texttt{L}, \texttt{R}\}$. Vi förutsätter att $\sqcup, \$, \texttt{L}$ och `R` inte ingår i Σ.

Tillståndsmaskinen har en ändlig mängd tillstånd $Q \triangleq \{q_0, q_1, \ldots, q_n\}$. I mängden ingår alltid ett *starttillstånd* och ett *sluttillstånd*. Vi använder gärna s som namn på starttillståndet och h som namn på sluttillståndet.

Det finns också en övergångsfunktion T som för varje tillstånd i Q och för varje symbol i Σ_t ger ett nytt tillstånd och bandåtgärd. Övergångsfunktionen

$$T \in (Q \times \Sigma_t) \to (Q \times \Sigma_h)$$

Turingmaskinens *konfiguration* beskrivs av tillståndet $q \in Q$ och bandinnehållet och huvudets position. Innehållet och positionen kan beskrivas med en trippel (α, σ, β) där $\alpha \in \Sigma_t^\star$ är den sträng som finns till vänster om huvudet, $\sigma \in \Sigma_t$ är symbolen vid huvudet och $\beta \in \Sigma_t^\star$ är den sträng av symboler som finns till höger om huvudet, till och med den sista symbol som inte är blank. Alternativ kan vi skriva $\alpha\underline{\sigma}\beta$, där vi markerat symbolen vid huvudet med understrykning.

En exekvering av med en Turingmaskin är en sekvens av konfigurationer. Vi återanvänder symbolen $\Rightarrow$ och låter den betyda 'ger efter ett steg'. Om exekveringen kommer till sluttillståndet h terminerar den.

Exempel. En enkel Turingmaskin, E för *erase*, gör alla symboler till $\sqcup$ från läshuvudets position och fram till första blanka symbol. Vi antar att $\Sigma \triangleq \{0, 1\}$. Låt $Q \triangleq \{s, q_1, h\}$ och T given av

q	σ	$T(q, \sigma)$
s	\$	$(q_1, \texttt{R})$
q_1	0	$(q_2, \sqcup)$
q_1	1	$(q_2, \sqcup)$
q_2	$\sqcup$	$(q_1, \texttt{R})$
q_1	$\sqcup$	(h,$\sqcup$)

Om startkonfigurationen är $(s, \underline{\$}01)$ så får vi följande exekvering

$$(s, \underline{\$}01) \Rightarrow (q_1, \$\underline{0}1) \Rightarrow (s, \$\underline{\sqcup}1) \Rightarrow (q_1, \$\sqcup\underline{1}) \Rightarrow (s, \$\sqcup\underline{\sqcup}) \Rightarrow (q_1, \$\sqcup\sqcup\underline{\sqcup}) \Rightarrow (h, \$\sqcup\sqcup\underline{\sqcup})$$

$\Diamond$

Aktivitet 2.
Konstruera en Turingmaskin som kan fortsätta där den vi just konstruerat slutade och återvända till symbolen \$.

Det är inte särskilt svårt att fortsätta på den inslagna vägen och konstruera en Turingmaskin som kan addera två naturliga tal givna i basen 1, dvs en skall kunna utföra exekveringen

$$(s, \underline{\$}111\sqcup 11) \Rightarrow^\star (h, \underline{\$}11111)$$

Med större ansträngning och lite hjälp går det att konstruera en Turingmaskin som kan addera tal givna i basen 2. När man förstått hur detta går till är man nästan övertygad om att alla beräkningar som kan göras med en verklig dator även kan utföras av en Turingmaskin, om än mycket mer omständligt.

Nu kan vi definiera klassen P.

Definition. P är mängden av alla beslutsproblem som kan lösas med Turingmaskiner på polynomtid. Det skall alltså finnas en exekvering

$$(s, \underline{\$}\alpha) \Rightarrow^\star (h, \underline{\$}Y) \text{ eller } (s, \underline{\$}\alpha) \Rightarrow^\star (h, \underline{\$}N)$$

där α är indata till problemet, och antalet exekveringssteg är begränsat av ett polynom i $|\alpha|$ som får bero på problemet men inte på indata.

△

10.5 Ickedeterministiska Turingmaskiner

I en *ickedeterministisk Turingmaskin* använder man en övergångsrelation i stället för en övergångsfunktion, dvs

$$T \in (Q \times \Sigma_t) \leftrightarrow (Q \times \Sigma_h)$$

För en given konfiguration kan alltså maskinen ibland välja mellan flera olika exekveringsalternativ. Man säger att ett beslutsproblem tillhör NP om det finns en Turingmaskin som har någon exekvering som på polynomtid lämnar resultatet `Y` om svaret på beslutsproblemet är 'ja'. Om svaret är 'nej' behöver exekveringen inte ens terminera.

Intuitivt betyder detta att den ickedeterministiska Turingmaskinen på något magiskt sätt kan välja rätt alternativ i varje läge eller göra alla beräkningar parallellt.

Idag finns inga verkliga datorer som har den förmågan, men det pågår forskning kring så kallade *kvantdatorer* som möjligen har förmågan att utföra obegränsat många beräkningar parallellt. I en kvantdator består minnet inte av vanliga bitar utan av *qubits*. En qubit har antingen värdet 0 eller 1 men man vet inte vilket förrän man vill observera värdet. Hypotesen är att man kan göra beräkningar med qubits utan att "observera" värdena och man bara observerar slutresultatet `Y` eller `N`.

Det påstås att man lyckats utföra beräkningar med hjälp av några få quibits. Själv är jag skeptisk till möjligheten att bygga kvantdatorer. Om jag vore mer försiktigt lagd skulle jag säga att det är så svårt att vår generation knappast kommer att kunna köpa en kvantdator på MediaMarkt.

11 Bibliografi

Denna förteckning innehåller referenser som har varit särskilt betydelsefulla för författaren till dessa anteckningar. Jag återvänder ofta till dem; jag har ännu inte förstått allt och jag tränar min hjärna genom att läsa dem. Se den som min tio-i-topp-lista för den som tycker att teori är viktigt och intressant. Listan är sorterad efter svårighetsgrad. Referenserna i början av listan är lättlästa och ofta underhållande, de i slutet tillhör mina absoluta favoriter.

1. D. R. Hofstadter: *Gödel, Echer, Bach — An Eternal Golden Braid*, Penguin Books, ISBN 0-14-005579-7, 1979. En bred beskrivning av datavetenskap och matematik med relationer till konst och musik.

2. D. Harel: *Algorithmics — The Spirit of Computing*, Addison-Wesley, ISBN 0-201-50401-4, 1992. Vad är datavetenskap?

3. R. L. Graham, D. E. Knuth, O. Patashnik: *Concrete Mathematics*, Addison-Wesley, ISBN 0-201-14236-8, 1989. En outsinlig källa för den som vill räkna ut diskreta storheter.

4. R. P. Feynman, R. B. Leighton, M. Sands: *The Feynman Lectures on Physics*, Addison-Wesley, ISBN 0-8053-9046-4, 2006. Detta verk i tre volymer har inte mycket med diskret matematik att göra men desto mer med den värld vi lever i.

5. D. E. Knuth: *The Art of Computer Programming; Fundamental Algorithms, Seminumerical Algorithms, Sorting and Searching, Combinatorial Algorithms*, Addison-Wesley, 1968-2011. En enastående ambitiös men ofullbordad beskrivning datavetenskapen.

6. P. R. Halmos, *Naive Set Theory*, Springer-Verlag, ISBN 0-387-90092-6. En lättillgänglig beskrivning av mängdläran baserad på en djup förståelse.

7. H.R. Lewis, C.H. Papadimitriou: *Elements of the Theory of Computation*, Prentice-Hall, 1981, 1998. En väskriven bok om automater, Turingmaskiner, formella språk och problem. Jag föredrar den första upplagans som mer omfattande.

8. S. C. Kleene: *Introduction to Meta-Mathematics*, North-Holland, 1964. En gedigen introduktion till logik och beräkningsteori.

9. E. Mendelson: *Introduction to Mathematical Logic*, Wadsworth & Brooks/Cole Advanced Books & Software, ISBN 0-534-06624-0, 1987. En mycket noggrann och innehållsrik introduktion till logik och grundläggande diskret matematik.

10. J. H. Conway: *On Numbers and Games*, A. K. Peters, ISBN 1-56881-127-6, 2001. Knuth skriver: "certainly one of the top mathematical creations of the 20th centuary". Jag instämmer. Boken kräver inte mer förkunskaper än denna kurs, men man behöver förtrogenhet med de andra verken i denna lista för att förstå Knuths omdöme.

12 Referenser

1. M. R. Garey, D. S. Johnson: *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

2. D. Prawitz: *Natural Deduction. A Proof-Theoretical Study*, Almqvist & Wiksell, 1965.

3. D. Prawitz: *Natural Deduction. A Proof-Theoretical Study*, Dover Publications, 2006, ISBN 0-486-44655-7.

4. A.N. Whitehead, B. Russell: *Principia mathematica*, Cambridge University Press, 1910-13. Även via `http://quod.lib.umich.edu/u/umhistmath/`