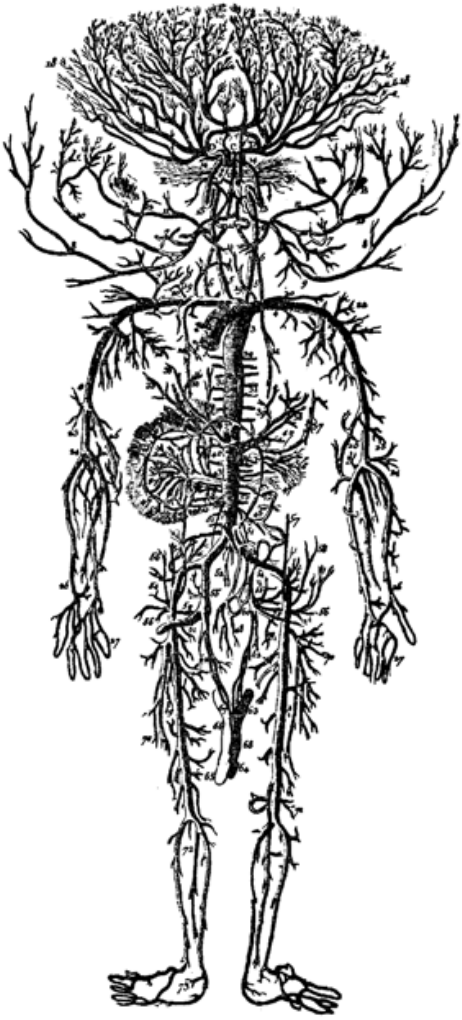


ALGORITMIK

Lennart Salling

Omslagsbild: Människans blodomlopp av William Harvey 1628. Harvey ville med figuren visa hur blodet gick genom artärerna till kroppens alla delar.

Förord

The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

— Donald E. Knuth

Alltsedan antiken har *det algoritmiska*[†] haft en väsentlig plats i alla kulturer. Och i dagens högteknologiska kultur är dess inflytande större än någonsin.

När de första penndragen drogs för att försöka definiera det algoritmiska var *räkning med hela tal* en viktig inspirationskälla. Även denna skrift tar sin utgångspunkt i heltalsaritmetiken. Vi försöker t.o.m. övertyga läsaren om att *allt* algoritmiskt kan beskrivas i termer av heltalsfunktioner, men att det omvända inte är sant.

Närmare bestämt visar vi hur man med utomordentligt enkla komponenter som utgångspunkt kan bygga program kapabla att utföra sysslor av avsevärd svårighetsgrad – ja i princip lösa alla algoritmiska problem. Vad gäller ickealgoritmiska problem, så behandlar vi det mest berömda av dem alla, och försöker förklara vad olika stora oändligheter har med saken att göra.

Skriften innehåller ett stort antal exempel och övningar som tacklar problem inom den diskreta matematiken på ett sätt som kanske kan få läsare med erfarenhet av programmering att uppleva igenkännandets glädje, och alla läsare att förhoppningsvis både erfara lite spänning i en och annan utmanande problemställning, samt förnimma något av skönheten i det algoritmiska hantverket.

[†] Ordet *algoritm* påstås ha sitt ursprung i namnet *al-Khwarizmi* på en persisk matematiker och astronom, år 780-850, som bl.a. är känd för konststycket att uppskatta jordens omkrets.

Innehåll

1

Calculus

Sätten är många när det gäller att räkna. I begynnelsen räknade man sina får genom att lägga upp en hög av små stenar[†], en sten för varje får. Kulramen användes på samma sätt. Handens fingrar likaså. I grunden handlar det om att lägga till eller ta bort en sten.

Introduktion

Addition Jag skall berätta en historia om två fåraherdar, som funderade över hur många får de hade tillsammans. De visste hur många de hade var och en. Ty var och en hade de sina stenar. I en fårskinnspåse fäst vid midjebältet förvarade de sina små stenar vilka alltid överensstämde i antal med de får de hade att vakta.

Om vi lägger ihop våra stenar, så ser vi hur många får det blir tillsammans, föreslog den ene. Ånej, sa den andre. Blandas min stenhög med din, så vet jag inte längre hur många får som jag har. Och lika lite vet du. Vi

[†] liten sten = *calculus* (latin)

gör så här istället. Och så vände han ut och in på de två skinnpåsarna, så att stenarna trillade ut och lade sig i två prydliga högar. Sedan började han att flytta stenar. För varje gång som han tog bort en sten från sin hög och lade till den andres, så tog han en ny sten (det fanns gott om stenar där de stod) och lade i en ny hög. Sålunda flyttade han sin stenhög tillhopa med den förstes, men lade samtidigt upp en extra hög. En hög som var en kopia av den egna originalhögen.

Så här många får fick vi om vi slog oss samman, sa han och pekade på den förstes förstorade stenhög. Och så här många hade jag före sammanslagningen sa han samtidigt som han pekade på extrahögen bredvid sig. Hur många hade du?

Då fick den förste en bestämd känsla av något oåterkalleligt. Den här sammanslagningen av fåren var ju bara avsedd som en tankelek. Det var aldrig meningen att de skulle slå sig samman. Nej, de skulle precis som vanligt gå åt varsitt håll. Var och en med sina får. Var och en med sina stenar. Innan han hade formulerat sina tankar i ord, hade emellertid den andre fortsatt med sina kalkyler.

Subtraktion För att se hur många får du hade före sammanslagningen, måste vi göra så här, sa han. Och så började han att flytta stenar igen.

Varje gång som han tog bort en sten från den sammanslagna högen, tog han bort en sten från extrahögen. Så höll han på tills den sista stenen var bortflyttad från extrahögen. De stenar som han tog bort från den sammanslagna högen, lade han i sin egen skinnpåse, och återstoden i den andres. Sålunda hade han återkallat de två originalhögarna. Och den andre fåraherden stod som förstummad. Men lättad.

Symboler Själv blev du väl knappast förstummad av den stenräknande fåraherdens adderande och subtrahe-

rande. Du skulle naturligtvis ha gjort motsvarande kalkyler på ett till synes mindre omständligt sätt med hjälp av de symboler som generationer av fåraherdar, astronomer, bokhållare och andra har utvecklat fram till idag.

Men det som den här fåraherden visar, är att det går utmärkt att addera och subtrahera med hjälp av en enda symbol (en sten) samt högar av denna. Högar utan inre struktur. För att klara av detta, använder sig fåraherden av ett väl strukturerat arbetssätt. En sten i taget läggs till eller tas bort ifrån en eller flera högar. I en viss ordning. Detta upprepas tills en viss hög är tom.

Stenhögsspråket

Hur avancerade kalkyler kan egentligen fåraherden göra med sitt sätt att räkna? Tro mig eller ej, men ingen nu existerande dator kan göra mer. Och förmodligen ingen framtida heller.

För att lättare kunna tala om dessa ting, skall vi formalisera fåraherdens stenhögsräknande till ett språk – *stenhögsspråket* – i vilket man på ett precist sätt kan beskriva den räknande fåraherdens stenhögsmanipulationer. Instruktionerna 1, 2 och 3 nedanför utgör det här språkets minsta beståndsdelar.

	Instruktion	Innebörd
1	Öka x	En sten läggs till högen x.
2	Minska x	En sten tas bort från högen x, såvida det finns någon sten där.
3	Sålänge x är icketom $\{ \ldots \}$	Det som står innanför parenteserna – vilket förresten bara får vara instruktioner[†] av typ 1, 2 och 3 – upprepas så länge x är icketom.

Man kan likna stenhögarna med en dators minnesceller.

[†] En följd av instruktioner åtskiljer vi med semikolon eller radbrytning. Se exemplen som följer.

↬ EXEMPEL 1.1 (Att tömma en hög.) Om man vill tömma en stenhög, skriver man

Sålänge x är icketom {Minska x}

vars innebörd är att man upprepande skall ta bort en sten i taget från x, så länge x innehåller någon sten. När x i sinom tid blir tom, slutar man.

↬ EXEMPEL 1.2 (Flytta en hög till en annan hög.) För att flytta alla stenar från en hög till en annan, tar man bort sten från den förra högen, och lägger samtidigt sten i den senare. En sten i taget, tills den förra blir tom.

Sålänge x är icketom {Minska x; Öka y}

I stenhögsspråkets (sten)rike är alla stenar likvärdiga.

Om den sten som läggs i y kommer från x eller från en annan hög spelar ingen roll. Det är enbart *antalet* stenar i en hög som är av betydelse.

Lägg märke till att y i slutänden kommer att innehålla summan av det som fanns i x och i y från början. Och i det specialfall att y är tom från början, så kommer y i slutet att vara en kopia av x.

↬ EXEMPEL 1.3 (Addition) Vi påpekade nyss att om x:s stenar flyttas till y, adderas x till y. Eftersom x därvid töms sten för sten försvinner dessvärre ursprungs:x, något som i vissa situationer kan vara katastrofalt. (Se på sidan 2 hur den ene fåraherden reagerar på det till synes oåterkalleliga.)

Hur kan man addera x till y, utan att förlora x?

Svaret är följande. Börja med att skaffa en tom hög, säg e. Flytta sedan x till y samtidigt som du lägger lika många stenar i e. Flytta slutligen e:s innehåll till x:

Sålänge e är icketom {Minska e}
Sålänge x är icketom {Minska x; Öka y; Öka e}
Sålänge e är icketom {Minska e; Öka x}

↬ EXEMPEL 1.4 (Kopiering) I en slutkommentar i EXEMPEL 1.2 nämnde vi att ""flytta x till y" resulterar i en kopiering av x, ifall y är tom från början. Dock blir vi därvid av med original-x.

Hur kan vi kopiera x, och efter kopieringen ha originalet kvar? Efter EXEMPEL 1.3 är det helt klart att "addera x till y, där y är tom från början" löser problemet. Dvs.

Töm y
Addera x till y

Här har vi döpt tömnings- och additionsrecepten på föregående sida till "Töm x" och "Addera x till y".

↬ EXEMPEL 1.5 (En särskild resultatshög.) Om man vill addera två högar och lägga resultatet i en tredje hög, behöver man bara addera först den ena, sedan den andra högen till den tredje, som ursprungligen är tom. Lägg märke till att de två ursprungshögarna bevaras intakta.

Töm r
Addera x till r
Addera y till r

Procedurer och modultänkande Vi har just bevittnat hur de minsta byggstenarna

Öka x, Minska x, Sålänge x är icketom $\{ \ldots \}$

kan användas för att bygga moduler, till exempel

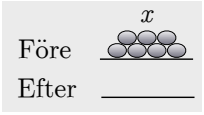
"Töm x" och "Addera x till y",

vilka i sin tur kan användas för att åstadkomma nya moduler.

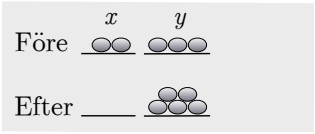
> Det här att använda färdiga moduler – och inte bara de minsta byggstenarna – hjälper oss att bygga. Det hjälper oss också att tänka. Ty att tänka är ju också att bygga, fastän med tankar.

Låt oss presentera modulerna från de tidigare exemplen – men nu med namn, så att vi i fortsättningen kan anropa dem. Dessa moduler är en sorts arbetande byggnadsverk som skall uträtta olika saker åt oss. Vi kallar dem för *procedurer* – ”stenhögsprocedurer”.

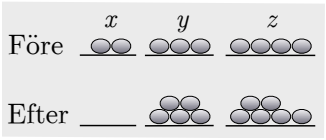
Töm x =
    Sålänge x är icketom {
        Minska x}



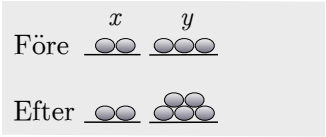
Flytta x till y =
    Sålänge x är icketom {
        Minska x; Öka y}



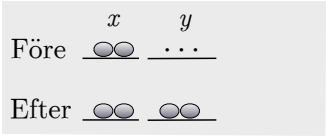
Flytta x till y och z =
    Sålänge x är icketom {
        Minska x; Öka y; Öka z}



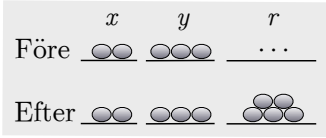
Addera x till y =
    Töm e
    Flytta x till y och e
    Flytta e till x



$y \leftarrow x$[†] =
    Töm y
    Addera x till y



Addera x och y i r =
    Töm r
    Addera y till r
    Addera x till r



[†] ”$y \leftarrow x$” utläses ”gör y till en kopia av x” eller “tilldela y det innehåll som x har”

Flera exempel

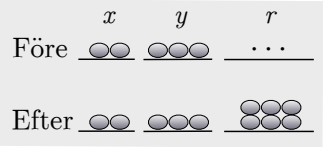
↬ EXEMPEL 1.6 (Multiplikation) Om en hög innehåller x stenar och en annan y stenar, hur kan man då ordna så att en tredje hög får $x \cdot y$ stycken stenar? Här följer ett svar.

```
Töm r
Sålänge x är icketom {
   Minska x; Addera y till r}
```

Proceduren bygger på att $x \cdot y$ avser x stycken y. Således behöver vi x gånger addera högen y till en från början tom hög r. Om vi önskar förhindra den ödeläggning som x blir utsatt för här, kan vi börja med att skaffa oss en kopia x' av x, och sedan arbeta med kopian x' istället för med x. Då får multiplikationsproceduren följande utseende:

Multiplikation är upprepad addition!

```
Multiplicera x och y i r =
    x' ← x; Töm r
    Sålänge x' är icketom {
       Minska x'; Addera y till r}
```



ANM. 1.1 Ursprungshögarna x och y går oförstörda ur den här multiplikationen. I vissa situationer kan man dock medvetet vilja ändra innehållet i någon av multiplikationens ursprungshögar. T. ex. så att en av ursprungshögarna x eller y i slutänden kommer att innehålla resultatet av multiplikationen. I själva verket kan ovanstående multiplikationsprocedur anropas på ett sätt – *Multiplicera x och y i x* – som gör ursprungshögen x till resultatshög för multiplikationen, och som därmed ändrar innehållet i högen x. Eftersom anropet *Multiplicera x och y i x* har den effekten att x multipliceras med y, kallar vi fortsättningsvis detta anrop för *Multiplicera x med y*. Tyvärr gör inte anropet *Multiplicera x och y i y* att ursprungshögen y blir resultatshög. Denna brist på symmetri i behandlingen av högarna x och y beror på att proceduren

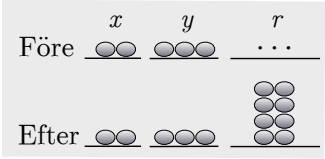
Multiplicera x och y i r är skriven så att x men inte y kopieras. Försök själv ändra proceduren *Multiplicera x och y i r* för att avhjälpa denna brist. Försök även ändra så att proceduren kan anropas med *Multiplicera x och x i x* på ett meningsfullt sätt. I fortsättningen antar vi att lämpliga justeringar har gjorts så att multiplikationsproceduren och andra stenhögsprocedurer kan anropas med sådan flexibilitet. T. ex. så att *Addera x och x i x* betyder *fördubbling* av x och så att *Multiplicera x och x i x* betyder *kvadrering* av x.

↬ EXEMPEL 1.7 (Potensupphöjning) Två högar antas från början innehålla x respektive y stenar. Hur åstadkomma en tredje hög med x^y stenar?

Potensupphöjning är upprepad multiplikation!

Med x^y menas multiplikation av y stycken x, vilket är likvärdigt med att y gånger multiplicera r med x, där r från början innehåller precis en sten. Efter att r har multiplicerats med x första gången kommer således r att innehålla $1 \cdot x = x$ stenar, efter två multiplikationer $1 \cdot x \cdot x = x^2$ stenar, osv..

Potensupphöj x till y i r =
  $y' \leftarrow y$; $r \leftarrow 1$[‡]
  Sålänge y' är icketom {
    Minska y'
    Multiplicera r med x}



Booleska procedurer En procedur som svarar ja eller nej kallas för *boolesk* efter George Boole[†].

↬ EXEMPEL 1.8 (En boolesk procedur.) I detta exempel vill vi skriva en procedur med kompetens att reda ut huruvida en stenhög är tom eller ej.

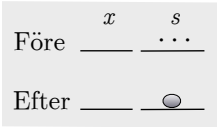
Men – använder du – detta är väl något som stenhögsspråket förutsättes kunna ”medfött”, precis som ökning

[‡] ”1” betecknar en stenhög innehållande en sten. [†] I sitt stora verk *An Investigation of the Laws of Thought* introducerade Boole 1854 en matematisk modell för den logik som synes styra rationellt tänkande.

och minskning med en sten. Kontroll av huruvida en hög är tom eller ej ingår ju i stenhögsspråkets tredje byggsten "*Sålänge x är icketom* { ... }" och måste väl därmed uppfattas som en på förhand given kompetens i stenhögsspråket?

Alldeles riktigt. Men den procedur som detta exempel handlar om är tänkt att kunna mer:

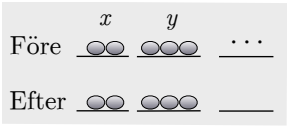
Att kunna "svara" ja eller nej. Detta skall gå till så, att en "svarshög" – benämnd s nedanför – får innehållet en sten (respresenterande svaret ja) om den hög som kontrolleras är tom, annars noll stenar (nej):



x är tom? i s =
    $x' \leftarrow x$; $s \leftarrow 1$
    Sålänge x' är icketom {
        Töm x'; Minska s}

KOMMENTAR Om x' är en tom hög, så kommer Sålänge-slingans repetitionsblock aldrig att besökas. Det enda som proceduren därvid uträttar innan den stannar, är att göra s till en 1-hög – dvs. en hög som representerar svaret *ja*. Om å andra sidan x' är icketom, så körs repetitionsblocket *en* gång. Sedan är det stopp. Och vad ligger nu i s?

↬ EXEMPEL 1.9 (Ytterligare en boolesk procedur.) Antag att vi vill reda ut huruvida den ena av två högar är minst lika stor som den andra. Följande booleska procedur utför uppdraget genom att ta bort en sten i taget från vardera högen tills den ena töms, och i slutänden kontrollera ifall även den andra blev tömd.



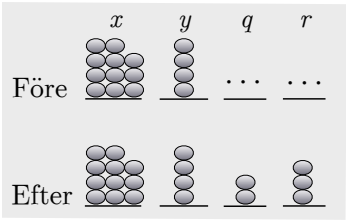
$x \geq y$? i s =
    Sålänge x är icketom {
        Minska x; Minska y}§
    y är tom? i s

↬ EXEMPEL 1.10 (Division) Att *dividera* en hög x med en annan hög y är liktydigt med att, så långt det går, dra bort delhögar av y:s storlek från x. Låt oss kalla sådana delhögar för y-högar. Antalet y-högar som kan dras från x benämnes divisionens *kvot*, och representerar således en sorts *gräns*[†]. När just så många y-högar är dragna från x, så har x reducerats till en hög som är mindre än y. Den resterande x-högen benämnes divisionens *rest*.

Kvoten ger en *gräns* för hur många högar som är möjliga att dra bort.

Notera att r-högen axlar x-högens roll i procedurens inledning, och utsätts sedan för de förändringar som annars skulle ha förstört x-högen.

Dividera x med y i q och r =
  $r \leftarrow x$; $q \leftarrow 0$
  $r \geq y$? i s
  Sålänge s är icketom {
    Subtrahera[‡] y från r
    Öka q
    $r \geq y$? i s}



ANM. 1.2 Varför görs de två "$r \geq y$"-testen?
SVAR: För att kontrollera slingans omtagningar. Det som är avgörande för om vi kan dra bort en y-hög från r och därmed öka kvoten, är nämligen om r är större eller lika med y. Därför utföres ett "$r \geq y$"–test före inträdet i Sålängeslingan där subtraktionen med åtföljande ökning av q skall äga rum. Eftersom r blir mindre varje gång som slingans subtraktionsrad körs, blir emellertid resultatet av det inledande "$r \geq y$"–testet överspelat i samma ögonblick som subtraktionen genomföres. Därför måste ett nytt "$r \geq y$"-test göras efter subtraktionen, något som förklarar förekomsten av det andra "$r \geq y$"-testet.

Om x redan från början råkar vara mindre än y, så kommer förstås slingan aldrig att köras, med följd att q och r

§ Lämplig modifiering så att ursprungshögarna passerar oförändrade genom proceduren överlämnas till läsaren. † Även i samhällslivet används benämningen *kvot* för att uttrycka en *gräns* för någon kvantitet, t. ex. för fisket eller handeln. ‡ Du får själv skriva en lämplig subtraktionsprocedur. Se övning 1.3.

blir lika med noll respektive x, ett uttryck för att x inte kan sönderdelas i några y-högar alls.

Raka rör mellan syntax och semantik

För en programutvecklare är det eftersträvansvärt att kunna få programtexten (programmets krumelurer) att ”stämma” med programmets mening. ”Raka rör” mellan de här två delarna – vilka brukar kallas för *syntax* och *semantik* – förenklar själva programkonstruktionen och gör ett program lättläst.

krumelurerna = syntaxen

innebörden = semantiken

Till att börja med ska vi utrusta stenhögsspråket med en enklare och tydligare kontroll av slingors omtagningar. Vi gör det med hjälp av *alias*[†].

Sålänge Bool Ett program som lämnar mycket övrigt att önska vad gäller läsbarhet är vår nyss tillverkade divisionsprocedur som nedanstående utdrag är taget från.

```
r ≥ y? i s
Sålänge s är icketom {
    Subtrahera y från r
    Öka q
    r ≥ y? i s}
```
(1)

(1) följer en mall:

```
Bool? i s
Sålänge s är icketom {
    P
    Bool? i s}
```
(2)

som f.n. är stenhögsspråkets krystade sätt att få en procedur P repeterad så länge en viss boolesk procedur Bool

[†] Poängen med ett *alias* (täcknamn) är att med en kort och kärnfull beteckning kunna referera till något som upplevs som alltför komplicerat för att presenteras i alla detaljer.

svarar "ja". (Se ANM. 1.2 på sidan 10.) Med tydlighet som ledstjärna inför vi ett *alias* för (2):

Ett *alias*

$$\text{Sålänge Bool}\,\{\text{P}\} \tag{3}$$

Innebörden i detta är att vi kan skriva (3) och få (2) utförd. T. ex. kan vi nu skriva

$r \leftarrow x$; $q \leftarrow 0$
Sålänge $r \geq y$ {Subtrahera y från r; Öka q}

och få divisionsproceduren på sidan 10 exekverad.

Och, eller, inte Antag att vi vill reda ut om två booleska procedurer *båda* svarar ja vid körning. Själv skulle du väl köra de två procedurerna och därefter titta i deras svarshögar.

Men kan man inte lösa problemet med ett program?

Jovisst, med ett program som kör de två procedurerna, och sedan *multiplicerar* deras svar. Ty produkten av två tal är ju lika med 1 om båda talen är 1:or, men lika med 0 om något av talen är 0. Se nedanför, där vi inför täcknamnet

och-alias

$$(\text{Bool}_1\ \textbf{och}\ \text{Bool}_2)?\ \text{i}\ s$$

Svarar ja, om **både** Bool_1 **och** Bool_2 gör det.

för

$\text{Bool}_1?$ i s_1
$\text{Bool}_2?$ i s_2
Multiplicera s_1 och s_2 i r

Antag sedan att vi vill veta om *minst en* av de två procedurerna svarar ja när man kör dem. Lösningen till det problemet finns i summan. Ty summan av två stenhögar är icketom om och endast om minst en av högarna är det. Detta förklarar införandet av täcknamnet

eller-alias

$$(\text{Bool}_1\ \textbf{eller}\ \text{Bool}_2)?\ \text{i}\ s$$

för

Bool_1? i s_1
Bool_2? i s_2
Addera s_1 och s_2 i r
r är icketom? i s

Svarar ja, om Bool_1 **eller** Bool_2 gör det.

Med hjälp av det senaste täcknamnet kan vi t. ex. skriva enradersproceduren

$$(y \text{ är tom } \textbf{eller } y \geq x)? \text{ i } s$$

istället för den gräsliga

y är tom? i s_1
$y \geq x$? i s_2
Addera s_1 och s_2 i r
r är icketom? i s

Till sist inför vi kortnamnet

$$\textbf{inte}(\text{Bool})? \text{ i } s$$

inte-alias

för proceduren

Bool? i s'
s' är tom? i s

som undersöker om Bool svarar nej vid körning. Med dess hjälp kan vi t. ex. skriva följande procedur för att undersöka ifall $x < y$:

$$(x < y)? \text{ i } s = \textbf{inte}(x \geq y)? \text{ i } s$$

$x \ngeq y$ är liktydigt med $x < y$.

Om - så - annars År 1963 föreslog John McCarthy† att man i varje programmeringsspråk borde kunna skriva något i stil med

$$\text{if Bool then } \text{P}_1 \text{ else } \text{P}_2$$

† Se McCarthy (1963), A basis for a mathematical theory of computation, Computer Programming and formal system, Braffort and Hirschberg, NorthHolland, Amsterdam.

med innebörden

Om Bool är sann utför P_1 annars P_2

Med hjälp av två *alias* fullföljer vi McCarthys intentioner.

Närmare bestämt inför vi täcknamnen

Om-alias **Om** Bool **så** P

för

```
Boole? i s
Sålänge s är icketom {
    P
    Minska s}
```

Vad gör *Minska s* för nytta?

och **Om** Bool **så** P_1 **annars** P_2

för

```
Boole? i s
s' ← s
Sålänge s är icketom {
    P1
    Minska s}
Sålänge s' är tom {
    P2
    Öka s'}
```

Varför behöver vi kopiera s?

Inse att P_1 körs igång om Bool svarar ”ja”, och att P_2 körs annars.

↬ EXEMPEL 1.11 (Den största av två högar.) En procedur som levererar den största av två högar är osedvanligt lätt att skriva med hjälp av vårt senaste alias:

Om $x \geq y$ **så** $r \leftarrow x$ **annars** $r \leftarrow y$

Delbarhet

Stenhögar behöver ofta *delas* i mindre högar. Eftersom högars storlek beskrivs av hela tal, måste en stenhögsprogrammerare vara förtrogen med heltalens delbarhetsegenskaper.

När ett naturligt tal x divideras med ett positivt naturligt tal y, får man en *kvot* q och en *rest* r (se sidan 10) som uppfyller

$$x = q \cdot y + r \text{ och } r < y$$

T. ex. $15 = 2 \cdot 6 + 3$.

Om resten r är noll, dvs. om $x = q \cdot y$, så är x lika med något av talen

$$0, y, 2y, 3y, \ldots \tag{4}$$

Talen i (4) sägs vara (*jämnt*) *delbara* med y. Det är väl ett bra namn, eller hur?

ANM. 1.3 Här är ett alternativt sätt att se på saken. Om man ur mängden av naturliga tal $0, 1, 2, 3, \ldots$ väljer vartannat tal, så får man $0, 2, 4, 6, \ldots$. Tar man istället vart sjunde tal, fås $0, 7, 14, 21, \ldots$.

Om man – helt allmänt – tar vart y:te tal, så erhålls (4). Har du skolårens tragglande med multiplikationstabellerna i gott minne, känner du igen talen i (4) som talen i y:ans tabell. Talet 0 finns med i varje tabell, och är alltså delbart med varje tal. Talet 1 finns bara med i en enda tabell – 1:ans, och är bara delbart med sig själv. Talen 0 och 1 har således extrema delbarhetsegenskaper. De övriga talen $2, 3, 4, \ldots$ finns alltid med i minst två tabeller, 1:ans och sin egen tabell – men ofta också i andra tabeller – och är således delbara med minst två tal, 1 och sig själv. Exempelvis är 6 delbart med 1, 2, 3 och 6, medan 7 endast är delbart med 1 och 7. Mer om detta i avsnittet om triviala delare nedanför.

↬ EXEMPEL 1.12 (Är x delbar med y?) Med hjälp av divisionsproceduren *Dividera x med y i q och r* är det lätt att konstruera en delbarhetstestare:

x är delbar med y? i s =
    Dividera x med y i q och r
    r är tom? i s

Triviala delare, prima och sammansatta tal. Om x är delbar med d, sägs d vara en *delare* till x. Alla x

har sig själv och 1 som delare. Därför kallas 1 och x för *triviala*† delare till x.

Vissa x – t. ex. 5 – har *bara* triviala delare. Sådana x sägs vara *prima* (eller *primtal*), om de är större än 1. Vi återkommer till primtalen lite längre ner.


$5 = 1 \cdot 5$

En hög som har enbart triviala delare känns igen på att om man vill formera den i en rektangel så får man bara en "urartad" rektangel: en rad eller en kolumn.


$6 = 2 \cdot 3$

Andra x – t. ex. 6 – har även icketriviala delare, vilka också kallas för *äkta* delare. En hög som har äkta delare kan alltid kan formeras i en "riktig" rektangel, något som förklaras utförligt i anmärkningen nedanför.

ANM. 1.4 Varje sammansatt x har minst *två* äkta delare. Ty, om x har någon äkta delare d, dvs. om $1 < d < x$ och $x = q \cdot d$, så följer att även q är en äkta delare till x. Se till att du förstår det! Således kan varje sammansatt x skrivas som en produkt av två äkta delare. Härav språkbruket "*x kan sönderdelas*" för sådana x som har äkta delare.

Att vara sammansatt är att kunna sönderdelas.

↬ EXEMPEL 1.13 (Har x någon äkta delare?) Här är en boolsk procedur som kontrollerar om x är delbar med något av talen $2, 3, 4, \ldots, x - 1$.

```
x har någon äkta delare? i s =
    s ← 0; d ← 2
    Sålänge d < x och s är tom {
        x är delbar med d? i s
        Öka d
    }
```

ANM. 1.5 I själva verket är det onödigt slösaktigt att som i proceduren ovanför göra delbarhetskontrollen för *alla* tal d i området $2 \leq d < x$. Villkoret $d < x$ kan ersättas med $d \cdot d \leq x$§

† Inom matematiken brukar man kalla en egenskap för *trivial*, om egenskapen ägs av alla. § Försök lista ut anledningen till detta. LEDNING: ANM. 1.4.

– något som reducerar antalet besök i slingans block väsentligt när x saknar äkta delare. För t. ex. $x = 101$ förvandlas antalet besök från 99 till 9.

Primtal Som redan omnämnts kallas ett tal p för *primtal* om

$$p \geq 2 \textit{ och } p \textit{ saknar äkta delare} \tag{5}$$

Exempelvis är $2, 3, 5, 7, 11$ primtal, men inte $4, 6, 8, 9, 10$. De senare talen kan sönderdelas som $2 \cdot 2$, $2 \cdot 3$, $2 \cdot 2 \cdot 2$, $3 \cdot 3$, $2 \cdot 5$. Lägg märke till att faktorerna i dessa produkter inte kan sönderdelas ytterligare, de är s.k. *primtalsfaktorer*. T. ex. är 2 och 5 primtalsfaktorer i 10.

Med (5):s hjälp är det lätt att skriva en primtalstestare:

x är prima? i $s =$
    ($x \geq 2$ **och inte**(x har någon äkta delare))? i s

Övningar

1.1 Skriv procedurer som
   a) gör två kopior av en hög,
   b) adderar tre högar, utan att förstöra deras ursprungliga innehåll, (Lägg resultatet i en fjärde hög.)
   c) låter två högar byta innehåll med varandra.

1.2 Vilka innehåll kommer högarna x, y, z att få efter körning av nedanstående treradersprogram?

$x \leftarrow y$
$y \leftarrow z$
$z \leftarrow x$

1.3 Konstruera subtraktionsprocedurer som uträttar följande uppgifter
   a) Subtraherar en hög från en annan. Använd den andra högen som resultatshög.

b) Som ovan, men så att de två ursprungshögarna bevaras intakta, och resultatet hamnar i en särskild hög.

1.4 Tillverka procedurer som givet att en hög innehåller x stycken stenar ordnar så att antalet stenar i en resultatshög blir
a) $1 + 2 + \ldots + x$,
b) $1 \cdot 2 \cdot \ldots \cdot x$,
c) $1 + x + x^2$,
d) $1 + x + x^2 + \ldots + x^n$.

1.5 Konstruera procedurer som passar till nedanstående namn
a) x är icketom? i s,
b) $x > y$? i s,
c) $x = y$? i s,
d) x är udda? i s.

1.6 Konstruera en procedur som gör att innehållet i dess resultatshög blir lika med skillnaden mellan den största och den minsta av två högar. Jfr. med övning 1.3.

1.7 Talen $0, 1, 4, 9, 16, 25, 36, \ldots$ är s.k. *kvadrattal*.
a) Skriv en procedur Kvadrattal n i r, som, för varje naturligt tal n, kan beräkna det n:te kvadrattalet.
b) Är 720801 ett kvadrattal? Skriv en procedur som för varje naturligt tal x kan reda ut om det är kvadratiskt.

1.8 Talen $0, 1, 3, 6, 10, 15, 21, \ldots$ beskriver storleken på stenhögar med triangulär form. Alltsedan antiken har talen gått under benämningen *triangeltal*. Om du har gjort övning 1.4, så har du redan skrivit en procedur (låt oss kalla den för Triangeltal x i r) som beräknar triangeltal. Här skall du skriva en procedur, x är ett triangeltal? i s, som för varje naturligt tal x kan reda ut om det är ett

triangeltal. Om du redan har gjort b)-uppgiften i förra övningen, kommer denna uppgift att gå som en dans.

1.9 Talen $1, 1, 2, 3, 5, 8, 13, 21$ är exempel på s.k. *Fibonaccital*. De kännetecknas av att varje tal efter de två första är summan av de två föregående. Skriv en procedur som för varje naturligt tal n placerar det n:te Fibonaccitalet i högen r.

1.10 Vad beräknar nedanstående procedurer?

a) Ärtig x i s =
```
y ← 0
Sålänge y < x {
   Addera 3 till y
}
y = x? i s
```

b) Buk x i r =
```
r ← 0; y ← x
Sålänge y är icketom {
   z ← x
   Sålänge z är icketom {
      Addera x till r
      Minska z
   }
   Minska y
}
```

c) Jadhr x i r =
```
r ← 0; k ← 0
Sålänge x ≥ k {
   Öka r
   Multiplicera r och r i k
}
Minska r
```

d) Tardavk x i r =
```
r ← 0; y ← 1; z ← x
Multiplicera z med 2
Sålänge y ≤ z {
   Addera y till r
   Addera 2 till y
}
```

2

De rekursiva funktionerna

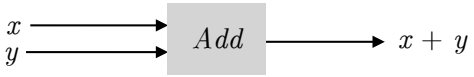
Vad en dator är och vad en dator gör kan man beskriva på olika sätt. Här skall vi inte gå in på vilka elektroniska komponenter som sitter innanför höljet. Inte heller skall vi försöka tala om vilka olika användningsområden som finns eller som kan tänkas komma att finnas i framtiden för dessa elektroniska underverk. Vår utgångspunkt är istället följande:

Input och output

Varje datoranvändning handlar i grund och botten om ”att få något visst ut (output) av något man stoppar in (input)”. Detta output kommer inte utan vidare såsom genom ett trollslag, även om det kan tyckas så ibland. Nej, man behöver ett recept (en *algoritm*) som alldeles precist – ibland steg för steg – talar om vad som skall göras. Vad en dator förmår uträtta kommer därför att ytterst bero på vilka sådana här recept som är möjliga att skriva.

I första kapitlet lärde vi känna ett algoritmiskt koncept som i hög grad byggde på att man tänkte i termer av ”hårdvara” – stenhögar. Om vi bortser från vad som händer med de inblandade stenhögarna under körning av en procedur, kan vi säga att t. ex. en adderande stenhögsprocedur för givna godtyckliga† naturliga tal x, y ”räknar ut” $x + y$.

Man säger att den adderande proceduren beräknar en *funktion* – låt oss kalla den för *Add* – med två naturliga tal x, y som input, och det naturliga talet $x + y$ som output.



† dvs. för *alla* naturliga tal.

Att tänka på en stenhögsprocedur som en funktionsberäknare är att tänka på en högre abstraktionsnivå, där man bortser från stenhögsmanipulerandet, och istället fokuserar på sambandet mellan input och output.

Och frågan

Vad kan man göra med datorer?

får i funktionsberäknarnas värld formuleringen

Vilka funktioner kan man beräkna med datorer?

Den som söker svar på denna senare fråga, tvingar programmeringskonstens innersta väsen att öppna sig, och tvingas själv ut på en del djupa matematiska vatten.

Låt oss – innan vi kastar oss i djupet – berätta lite grann om själva funktionsbegreppet.

Funktionsbegreppet

En *funktion* kan sägas vara en regel som beskriver hur vissa output är relaterade till vissa input. Men man kan också säga att en funktion är alla par av input-output ifråga. Följande exempel får illustrera vad som avses.

↬ EXEMPEL 2.1 (Att dubblera.) Betrakta å ena sidan, input $0, 1, 2, 3, \ldots$ tillsammans med regeln

”output är *dubbelt* så stort som input”, (1)

och å andra sidan följande par av input/output

input	0	1	2	3	$\cdots$	x	$\cdots$
output	0	2	4	6	$\cdots$	$2x$	$\cdots$

(2)

Det är lätt att se att (2) följer av (1), och att (1) följer av (2).

ANM. 2.1 Eftersom mängden av input-output beskriver vilka output som hör ihop med vilka input, är det inte fel att hävda att själva mängden av input-output också är en regel beskrivande hur output är relaterat till input. Och i vissa situationer är detta faktiskt den naturligaste och enklaste regeln.

I alla händelser, så finns funktionsbegreppet för att beskriva att input och output associeras till varandra på sådant sätt att varje input har högst ett output associerat till sig. Observera att jag säger *högst ett*. Det kan vara så att ett input inte har något enda output associerat till sig. Men om det har något, så har det exakt ett. Med andra ord:

En funktion skall för ett input ge *ett bestämt* output, eller inget output alls. En funktion skall t. ex. inte kunna ge både output 3 och output 7 till ett och samma input. Detta kallas ofta för *determinism*. Om en funktion har ett output associerat till ett visst input x, säger vi att funktionen är *definierad* i x och att den *returnerar* det associerade outputelementet. Annars (om funktionen saknar output för ett visst x) säger vi att funktionen är *odefinierad* i x.

determinism

returnera

Lägg märke till att vi inte ställer några krav på hur output skall beräknas. Vi kräver inte ens att output går att beräkna[†].

Många tycker att detta är konstigt.

Genom att funktionsbegreppet är så allmänt hållet kan vitt skilda fenomen beskrivas med funktioner, vilket framgår redan av exemplen nedan.

Några enkla exempel

↬ EXEMPEL 2.2 (Att rita en karta.) En Sverigekarta utgör output till en funktion från ”riktiga Sverige” till ett platt ”kart-Sverige”.

Till varje position i ”riktiga Sverige” hör högst en position på kartan.

[†] Mer om detta i kapitel 7.

↬ EXEMPEL 2.3 (Att älska.) Regeln "*x älskar y*", definierar knappast en funktion från en inputmängd X av personer till en outputmängd av personer Y, eftersom x mycket väl kan älska flera personer. Däremot är förstås "*x älskar bara y*" en korrekt funktionsregel för en funktion med x som input, och y som output.

↬ EXEMPEL 2.4 (Att addera.) *Adderaren* är en funktion som för varje par x_1, x_2 av tal returnerar summan x_1+x_2.

En funktion som i likhet med adderaren tar sitt input i form av par kallas för *tvåställig*. På motsvarande sätt kan en funktion ta ännu flera element som input. De enskilda inputelementen är funktionens s.k. *argument*.

argument

Funktioner som specialiserade arbetare Vi kommer ofta – när vi lämnar den formella aspekten av funktionsbegreppet åt sidan – att se på funktioner som specialiserade arbetare. När det gäller att döpa en funktion, försöker vi välja ett namn som tydligt uttrycker vad funktionen gör. Funktioner som vi kommer att använda ofta ger vi om möjligt korta namn. T. ex. kallar vi adderaren i EXEMPEL 2.4 för *Add*.

Vi använder s.k. *funktionell* beteckning på funktionernas output. T. ex. skriver vi $Add(x, y)$ för det som *Add* returnerar. På motsvarande sätt kommer vi t. ex. att skriva

Funktionell notation

$$Mor(x),\ Sub(x, y),\ Mult(x, y) \text{ och } Pot(x, y)$$

för

$$x\text{:s mor},\ x - y,\ x \cdot y \text{ respektive } x^y$$

Heltalsfunktioner Det man gör när man använder datorer kan på ytan vara väldigt långt ifrån räkning med heltal. Men innerst inne handlar det ändå alltid om att

stoppa in heltal och få ut heltal. Ty de symboler som man matar datorn med, och de som datorn svarar med, kodas ju till olika sekvenser av nollor och ettor – s.k. *bitsträngar*. Sådana sekvenser representerar naturliga tal i 2-systemet. Därför kan man se på datorn som en matematikmaskin, vars förmåga består i att kunna beräkna funktioner med naturliga tal som input och output. För enkelhets skull kallar vi sådana funktioner för *heltalsfunktioner*, fastän bara ickenegativa heltal är inblandade. Det problem som detta kapitel nu skall behandla och vars lösning stakar ut gränserna för vad en dator kan uträtta är följande.

Vilket naturligt tal representerar 1101010 i 2-systemet?

Vilka heltalsfunktioner kan en dator beräkna?

Innan du går vidare, skall du lämpligen försätta dig i en byggmästares situation. Om du gör det, vet du att man för att bygga något behöver *byggstenar* och lämpliga *metoder*, där de senare i bästa fall kan användas för att fogas samman inte bara byggstenarna till större byggnader, utan också de senare till ännu större byggnader.

Märkligt nog intresserade sig t. ex. David Hilbert och Stephen Kleene för dessa funktioner redan innan de elektroniska maskiner som idag kallas för datorer hade gjort sin entré.

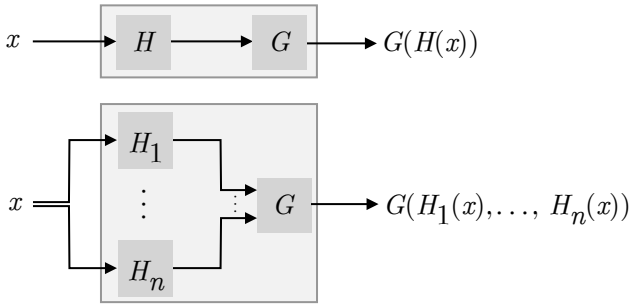
Basfunktionen Möjligheten att beräkna följande funktion tas för given†.

x	0	1	2	$\cdots$	x	$\cdots$
$\ddot{O}ka(x)$	1	2	3	$\cdots$	$x+1$	$\cdots$

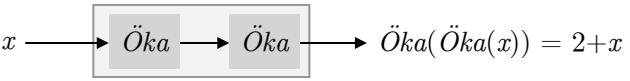
Sammansättning Att bilda sammansättningar – dvs. att låta output från en eller flera funktioner bilda input till en annan – är ett mekaniskt sätt att skapa nya funktioner av gamla.

† Om inte annat, så kan den räknande fåraherden (sid 2) utföra beräkningen åt oss.

Serie- och parallell-kopplingar.

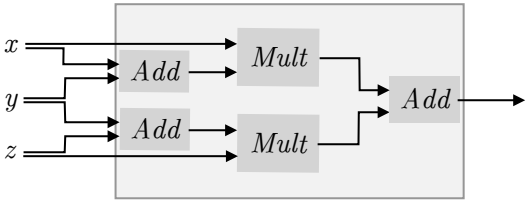


Så länge vi bara har funktionen *Öka* att laborera med kan vi dock inte erbjuda så mycket annat än



Med hjälp av funktioner som adderar och multiplicerar skulle vi däremot kunna komponera† t. ex.

$$x \cdot (x + y) + (y + z) \cdot z$$



.

Rekursion

Nu skall vi tillföra vår byggnadskonst en avgörande byggnadsmetod, som är något av ett trollspö för algoritmiken. Dess utmärkande drag är att den tillåter ”byggnadsverket” att vara med och bygga sig själv.

Funktioner byggda med denna metod använder vanligtvis vissa av sina output för att beräkna andra.

† sammansättning = composition (engelska)

T.e.x. kan det för en sådan funktion vara så att den för att beräkna output för $x = 100$, först måste ta reda på vilket output som svarar mot föregående input $x = 99$. Och för att göra det, måste den först ta reda på output för $x = 98$, och så vidare tills den hittar output för lägsta input – man brukar säga att rekursionen bottnar. Med hjälp av output för lägsta input kan den beräkna output för nästa input, och så vidare tills output för $x = 100$ är färdig att levereras.

Rekursion som ordagrant betyder *återlöpning* är en talande beteckning på denna byggnadsmetod, eller hur.

Vi börjar med ett par enkla exempel, där fullständigt triviala funktioner beskrivs med hjälp av återlöpning.

↬ EXEMPEL 2.5 (Bara nollor.)

x	0	1	2	$\cdots$
$Noll(x)$	0	0	0	$\cdots$

Så här ser en enkel rekursiv beskrivning ut:

$$\begin{cases} Noll(0) = 0 & (\downarrow) \\ Noll(Öka(x)) = Noll(x) & (\curvearrowleft) \end{cases}$$

En rekursiv beskrivning

Enkel? Är inte det här snarare en tillkrånglad beskrivning? En funktion som för alla input returnerar 0, uttrycks väl ändå enklast på följande sätt:

$$Noll(x) = 0 \textit{ för alla } x. \tag{3}$$

Javisst, (3) är förstås – i de flestas ögon – det enklaste och naturligaste sättet att beskriva funktionen *Noll*. Men nu är det så att vår ambition är att konstruera funktioner enbart medelst rekursion och sammansättning. Och eftersom (3) varken utgör något exempel på rekursion eller på sammansättning, så tvingas vi för närvarande att förkasta (3).

Tillbaka nu till vår rekursiva beskrivning.

($\curvearrowleft$) säger att nästa output är lika med nuvarande output, som i sin tur är lika med föregående output osv.. Och tack vare det s.k. *basfallet* ($\downarrow$) blir de alla lika med 0. Följande körprov illustrerar det hela:

$$Noll(3) \overset{(\curvearrowleft)}{=} Noll(2) \overset{(\curvearrowleft)}{=} Noll(1) \overset{(\curvearrowleft)}{=} Noll(0) \overset{(\downarrow)}{=} 0.$$

↬ EXEMPEL 2.6 (Identiteten) Den tabellbeskrivna funktionen nedanför gör till synes ingenting, eftersom dess output är *identiskt* med input. Men något annat än triviala funktioner har ju heller inte utlovats just nu.

x	0	1	2	$\cdots$
$Id(x)$	0	1	2	$\cdots$

Här är en rekursiv beskrivning av *Id*.

$$\begin{cases} Id(0) = 0 & (\downarrow) \\ Id(\ddot{O}ka(x)) = \ddot{O}ka(Id(x)) & (\curvearrowleft) \end{cases}$$

Rekursionssteget ($\curvearrowleft$) beskriver hur två på varandra följande output är relaterade till varandra: om man ökar det ena med en enhet så får man nästa.[†] *Basfallet* ($\downarrow$) beskriver hur output skall vara i botten – dvs. för det lägsta värdet på input.

Följande körexempel illustrerar den mekaniska karaktären i användningen av ($\curvearrowleft$) och ($\downarrow$). Vad för slags mekanisk kompetens som krävs skall vi diskutera lite längre fram.

[†] Man kan även formulera relationen mellan nämnda två output så här: $Id(x) = \ddot{O}ka(Id(x-1))$ för $x \geq 1$. I de flesta programspråk är detta synsätt det förhärskande. Eftersom det förutsätter tillgång till ytterligare funktionalitet (en basfunktion som minskar en enhet, samt villkorsbeskrivning), avstår vi från det uttryckssättet till förmån för det mer renrakade ($\curvearrowleft$). Men det skadar aldrig att i tanken kunna hoppa mellan de två.

($\curvearrowleft$) används för att *skriva om* först $Id(3)$, sedan $Id(2)$ och till sist $Id(1)$. Därefter griper ($\downarrow$) in för att *skriva om* $Id(0)$.

$$
\begin{aligned}
& Id(3) \\
\overset{(\curvearrowleft)}{=}\ & \ddot{O}ka(Id(2)) \\
\overset{(\curvearrowleft)}{=}\ & \ddot{O}ka(\ddot{O}ka(Id(1))) \\
\overset{(\curvearrowleft)}{=}\ & \ddot{O}ka(\ddot{O}ka(\ddot{O}ka(Id(0)))) \\
\overset{(\downarrow)}{=}\ & \ddot{O}ka(\ddot{O}ka(\ddot{O}ka(0))) \\
=\ & \ddot{O}ka(\ddot{O}ka(1)) \\
=\ & \ddot{O}ka(2) \\
=\ & 3
\end{aligned}
$$

↬ EXEMPEL 2.7 (Addition)

x	0	1	2	$\cdots$
$Add(x,y)$	y	$1+y$	$2+y$	$\cdots$

Det som skiljer ett additionsresultat från nästkommande, t. ex. $2+y$ från $3+y$, eller $x+y$ från $(x+1)+y$, är att det senare är en enhet större. Med andra ord,

$$
\begin{cases}
Add(0,y) = y & (\downarrow) \\
Add(\ddot{O}ka(x),y) = \ddot{O}ka(Add(x,y)) & (\curvearrowleft)
\end{cases}
$$

Så här kan en additionsberäkning gestalta sig:

$$
\begin{aligned}
& Add(3,5) \\
\overset{(\curvearrowleft)}{=}\ & \ddot{O}ka(Add(2,5)) \\
\overset{(\curvearrowleft)}{=}\ & \ddot{O}ka(\ddot{O}ka(Add(1,5))) \\
\overset{(\curvearrowleft)}{=}\ & \ddot{O}ka(\ddot{O}ka(\ddot{O}ka(Add(0,5)))) \\
\overset{(\downarrow)}{=}\ & \ddot{O}ka(\ddot{O}ka(\ddot{O}ka(5))) \\
=\ & \ddot{O}ka(\ddot{O}ka(6)) \\
=\ & \ddot{O}ka(7) \\
=\ & 8
\end{aligned}
$$

↬ EXEMPEL 2.8 (Triangeltal) I en gammal historia berättas det om en lärare som gav sina elever i uppgift att räkna ut $1 + 2 + \cdots + 99 + 100$. Till historien hör att Gauss† var en av eleverna. Därför blev det inte alls som läraren hade tänkt sig. Efter endast någon minut var nämligen lille Gauss framme vid katedern och överlämnade en lapp till läraren, en lapp på vilken han tecknat ned talet 5050. Lärarens reaktion blev - enligt sägnen – inte beundran utan vrede, med följd att lille Gauss fick sig en omgång av riskvasten.

> Läraren kunde gissningsvis inte föreställa sig att den här beräkningen kunde göras på annat sätt än att lägga samman talen det ena efter det andra. Och då kunde det ju inte gå så snabbt.
>
> Men Gauss finurliga idé var nog att lägga samman det första talet med det sista, det andra med det näst sista osv., vilket ger upphov till femtio stycken additioner som var och en resulterar i 101. Och $50 \cdot 101$ är ju snabbt uträknat till 5050.

$1 + 2 + 3 + 4$

Talet 5050 är förresten det hundrade *triangeltalet*. De gamla grekerna – som älskade heltal och mönster – kallade tal som ges av $1 + 2 + \ldots + x$ för triangeltal. Ett namn som förklaras av att man kan bygga vackra trianglar om man använder just så många byggstenar som dessa tal anger. Lärarens naiva idé för beräkning av det hundrade triangeltalet $1 + 2 + \cdots + 99 + 100$, dvs. att lägga samman det ena efter det andra av talen, innebär att det avslutande momentet är att addera talet 100 till summan $1 + 2 + \cdots + 99$. Eftersom den senare summan beskriver det nittionionde triangeltalet, så är den naiva idén inget annat än en rekursiv funktionsbeskrivning.

† Carl Friedrich Gauss 1777–1855, en tysk matematiker av enkel härkomst som kom att bli kanske den störste av alla, något som kommer till uttryck i epitetet ”matematikernas konung”.

Nedanför finns en formaliserad version av nämnda beskrivning, samt ett körexempel.

$$\begin{cases} \mathit{Triangeltal}(0) = 0 \\ \mathit{Triangeltal}(\mathit{Öka}(x)) = \mathit{Add}(\mathit{Öka}(x), \mathit{Triangeltal}(x)) \end{cases}$$

$$\begin{aligned} & \mathit{Triangeltal}(3) \\ &= \mathit{Add}(3, \mathit{Triangeltal}(2))) \\ &= \mathit{Add}(3, \mathit{Add}(2, \mathit{Triangeltal}(1)))) \\ &= \mathit{Add}(3, \mathit{Add}(2, \mathit{Add}(1, \mathit{Triangeltal}(0))))) \\ &= \mathit{Add}(3, \mathit{Add}(2, \mathit{Add}(1, 0)))) \\ &= \mathit{Add}(3, \mathit{Add}(2, 1))) \\ &= \mathit{Add}(3, 3)) \\ &= 6 \end{aligned}$$

Här beräknas 3+2+1.

Mönstermatchning

Rekursiva beräkningar av ovanstående slag förutsätter bara en förmåga att mönstermatcha, och att behärska basfunktionen. Närmare bestämt skulle en maskin klara av att utföra dessa rekursiva beräkningar om den hade följande förmågor:

1. att behärska basfunktionen, fram- och baklänges:
   Aha, $\mathit{Öka}(3)$ är detsamma som 4.
   Aha, 4 är detsamma som $\mathit{Öka}(3)$.

2. att kunna mönstermatcha textsträngar:
   $\mathit{Triangeltal}(\mathit{Öka}(3))$ överensstämmer tecken för tecken med $\mathit{Triangeltal}(\mathit{Öka}(x))$, då x sätts till 3.

3. att kunna ersätta en läst sträng – vars mönster sammanfaller med vänsterledssträngen i något recept – med motsvarande sträng i receptets högerled: T. ex

ersätta
Triangeltal(*Öka*(3)) med *Add*(4, *Triangeltal*(3)), eftersom den första matchar vänsterledet och den andra matchar högerledet i receptet för triangeltalsfunktionen.

Önsketänkandeprincipen

Förr eller senare ställs du inför problemet att själv komponera en rekursiv funktion. Då kan detta lilla avsnitt kanske vara till hjälp.

Vågar du?

Hemligheten bakom en rekursiv konstruktion är ofta att våga önska sig att man ”nästan är färdig” med konstruktionen.

Antag t. ex. att du vill skriva en rekursiv funktion för multiplikation.

(i) Önska dig då att du för ett visst par av tal x, y redan har räknat ut $x \cdot y$.

(ii) Försök sedan tänka ut *hur* du skulle kunna använda dig av resultatet från (i) för att beräkna $(1+x) \cdot y$. Om du inser att det som skiljer $x \cdot y$ från $(1+x) \cdot y$ är att den senare är exakt y enheter större än den förra, dvs. lika med y plus den förra, så har du funnit en rekursionsformel för multiplikation.
Se ($\curvearrowleft$) nedanför.

↬ EXEMPEL 2.9 (Multiplikation)

x	0	1	2	3	$\cdots$
$Mult(x, y)$	0	y	$2y$	$3y$	$\cdots$

$$\begin{cases} Mult(0, y) = 0 & (\downarrow) \\ Mult(Öka(x), y) = Add(y, Mult(x, y)) & (\curvearrowleft) \end{cases}$$

Sådana specialfall av multiplikation som

$$x \cdot x = x^2 \text{ och } x \cdot x \cdot x = x^3$$

ombesörjes av

$$Mult(x, x) \text{ och } Mult(x, Mult(x, x)).$$

↬ EXEMPEL 2.10 (Potensupphöjning)

y	0	1	2	$\cdots$
$Pot(x,y)$	1	x	x^2	$\cdots$

Notera att

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y \text{ stycken}}.$$

Vilken relation finns mellan x^y och x^{1+y}?
SVAR: Den senare är lika med den förra multiplicerad med x:

$$\begin{cases} Pot(x,0) = 1 \\ Pot(x, Öka(y)) = Mult(x, Pot(x,y)) \end{cases}$$

En mall

Du har säkert redan noterat att de rekursivt byggda funktionerna i exemplen ovan har varit stöpta efter samma mall. En mall som tycks vara något i stil med

(↓) Beskriv output för lägsta input. — basfall

(↶) Beskriv hur ett output kan beräknas med hjälp† av föregående input och output. — rekursionssteg

Den primitivt rekursiva mallen

$$\begin{cases} f(0) = b & (\downarrow) \\ f(Öka(x)) = g(x, f(x)) & (\curvearrowleft) \end{cases}$$

f utan passiva argument

$$\begin{cases} f(0, y_1, ..., y_n) = b(y_1, ..., y_n) & (\downarrow) \\ f(Öka(x), y_1, ..., y_n) = g(x, y_1, ..., y_n, f(x, y_1, ..., y_n)) & (\curvearrowleft) \end{cases}$$

f med n passiva argument

† Funktionen g i mallen svarar för denna hjälp.

ANM. 2.2 Den primitiva rekursionens kännetecken är dels att rekursionen äger rum enbart i ett av argumenten – eventuella övriga argument är *passiva* – dels att det rekurserande argument löper bakåt en enhet varje gång som rekursionssteget används. Tack vare basfallet bottnar (upphör) rekursionen.

$$\begin{aligned} f(4,y) &\overset{(\curvearrowleft)}{=} g(3,y,f(3,y)) \\ &\overset{(\curvearrowleft)}{=} g(3,y,g(2,y,f(2,y))) \\ &\overset{(\curvearrowleft)}{=} g(3,y,g(2,y,g(1,y,f(1,y)))) \\ &\overset{(\curvearrowleft)}{=} g(3,y,g(2,y,g(1,y,g(0,y,f(0,y))))) \\ &\overset{(\downarrow)}{=} g(3,y,g(2,y,g(1,y,g(0,y,b(y))))) \end{aligned}$$

ANM. 2.3 I den primitivt rekursiva mallen har vi valt att låta f:s första argument vara det rekurserande argumentet, men egentligen tillåter vi vilket som helst av argumenten att spela denna roll.

ANM. 2.4 Av bekvämlighetsskäl skall vi inte tolka mallen bokstavligt. En funktionsbeskrivning skall sägas följa mallen även om funktionerna b och g (se mallen) bara använder en del av de tillgängliga argumenten. Mer om detta i återblicken på nästa sida.

Stacken När mallen används för en beräkning, läggs det upp en s.k. *stack* av operationer som måste utföras i en viss ordning.

> Engelskans ord *stack* står inom datalogin för en ”hög” eller en ”trave” av objekt, åtkomliga enbart från toppen. Tänk på en trave pappersark lagda på varandra. Sådan åtkomlighet innebär t. ex. att det är det sist tillfogade objektet, och inget annat, som kan plockas av först.

Stacken byggs upp på så sätt att en ny operation fogas till stacken för varje gång som rekursionssteget används. De “stackade” operationerna bildar till slut (se ANM.2.2 ovanför) en lång sammansättning där funktionen g sammansätts med sig själv.

Återblick Låt oss granska två gamla exempel för att se på vilket sätt dessa är formade efter mallen.

EXEMPEL 2.7 igen.

$$\begin{cases} Add(0,y) = Id(y) = y \\ Add(\ddot{O}ka(x),y) = \ddot{O}ka(Add(x,y)) \end{cases}$$

mallen

$$\begin{cases} f(0,y) = b(y) \\ f(\ddot{O}ka(x),y) = g(x,y,f(x,y)) \end{cases}$$

Ovanför är konstruktionens g-funktion lika med *Öka* som nyttjar blott *ett* av de tre argument som den primitivt rekursiva mallen tillåter. Nedanför använder konstruktionens g emellertid *båda* argumenten som erbjuds. Däremot är *inte* g den funktion man kanske skulle kunna tro vid första påseendet, utan $g(x,z) = Add(\ddot{O}ka(x),z)$.

EXEMPEL 2.8 igen.

$$\begin{cases} Triangeltal(0) = 0 \\ Triangeltal(\ddot{O}ka(x)) = \\ \quad Add(\ddot{O}ka(x), Triangeltal(x)) \end{cases}$$

mallen

$$\begin{cases} f(0) = b \\ f(\ddot{O}ka(x)) = g(x,f(x)) \end{cases}$$

Några nya exempel

↬ EXEMPEL 2.11 (Minskning) Funktionen som beskrivs i input-output-tabellen nedanför är en sorts motsats till funktionen *Öka*.

n	0	1	2	3	...	$1+x$	...
$Minska(n)$	0	0	1	2	...	x	...

Input-output-tabellen leder oss till formuleringen

$$\begin{cases} Minska(0) = 0 \\ Minska(\ddot{O}ka(x)) = x \end{cases}$$

En rekursiv funktion som aldrig anropar sig själv!?

Vid en jämförelse med den primitivt rekursiva mallen, ser man att vår konstruktion följer ur mallen, om mallens g-funktion sätts lika med identitetsfunktionen.

↬ EXEMPEL 2.12

y	0	1	2	...
$Sub(x, y)$	x	$x-1$	$x-2$	...

Det som (i tabellen) skiljer ett subtraktionsresultat $x-y$ från nästkommande $x-(y+1)$ är att det senare är en enhet lägre:

$$x-(y+1) = (x-y)-1.$$

Med andra ord,

$$\begin{cases} Sub(x,0) = x \\ Sub(x, Öka(y)) = Minska(Sub(x,y)) \end{cases}$$

↬ EXEMPEL 2.13 (Konstanta funktioner.) Funktionen *Noll* är en *konstant* funktion. Med dess hjälp kan man konstruera flera:

$Ett(x) = Öka(Noll(x))$ $\quad x \longrightarrow$ *Noll* $\longrightarrow$ *Öka* $\longrightarrow 1$

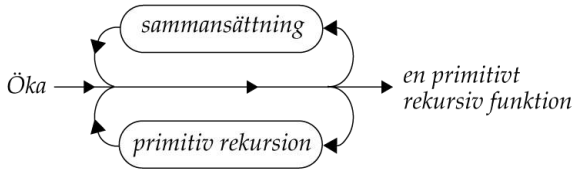
$Två(x) = Öka(Ett(x))$ $\quad x \longrightarrow$ *Ett* $\longrightarrow$ *Öka* $\longrightarrow 2$

$Tre(x) = Öka(Två(x))$ $\quad x \longrightarrow$ *Två* $\longrightarrow$ *Öka* $\longrightarrow 3$

Primitivt rekursiva funktioner

Vi har nu i några exempel (och flera följer) visat hur man genom att tillämpa operationerna sammansättning och/eller primitiv rekursion på basfunktionen kan tillverka mer komplicerade funktioner. Vidare, hur man genom att åter tillämpa operationerna på dessa funktioner kan tillverka ännu mer komplicerade funktioner.

Med en primitivt rekursiv funktion menas en funktion som kan konstrueras på precis detta sätt, dvs. att utgå ifrån basfunktionen och sedan noll eller flera gånger tillämpa de två byggnadsmetoderna.



Så här tillverkas primitivt rekursiva funktioner.

ANM. 2.5 Lägg märke till att en primitivt rekursiv funktion kan skapas utan att den primitivt rekursiva mallen används. Basfunktionen *Öka* är ju t. ex. primitivt rekursiv, precis som sammansättningar av den. Men utan den primitivt rekursiva mallen kan man bara tillverka triviala funktioner.

Booleska primitivt rekursiva funktioner Med en *boolesk* funktion menar vi en funktion vars output tillhör mängden $\{0, 1\}$. Talen 0 och 1 representerar här *falskt* och *sant* (eller *nej* och *ja*).

↬ EXEMPEL 2.14 (Fem booleska funktioner.) Här presenteras en kvintett användbara primitivt rekursiva booleska funktioner. Den första är så populär att den har två namn: *Noll*? och *Icke*.

Ibland avslutas ett namn med tecknet "?".

x	0	1	2	...
$Noll?(x) = Icke(x) = Sub(Ett(x), x)$	1	0	0	...

De övriga fyra är tvåställiga:

$$\begin{aligned} Och(x, y) &= Icke(Noll?(Mult(x, y))) \\ Eller(x, y) &= Icke(Noll?(Add(x, y))) \\ Större(x, y) &= Icke(Noll?(Sub(x, y))) \\ Mindre(x, y) &= Större(y, x) \end{aligned}$$

Funktionen *Och* returnerar 1, när båda argumenten är nollskilda, *Eller* gör det då minst ett argument är nollskilt, och *Större* då första argumentet är större än det andra.

Falldefinitioner och funktionen *Om* Vi skall nu introducera den omtyckta funktionen *Om*, som efter en rekommendation från John McCarthy[†] finns med i de flesta programspråk för att hantera *vägval*. Funktionen ifråga har tre argument och returnerar sitt andra eller tredje argument beroende på om det första argumentet är positivt eller ej:

$$Om(x, y, z) = \begin{cases} y & \text{om } x > 0 \\ z & \text{annars} \end{cases}$$

Om kan lätt tillverkas med hjälp av den primitivt rekursiva mallen:

$$\begin{cases} Om(0, y, z) = z = Id(z) \\ Om(Öka(x), y, z) = y = Id(y) \end{cases}$$

Ett typiskt sätt att använda *Om*-funktionen är att skriva

$$Om(p(x), f(x), g(x))$$

Resultatet är en funktion som returnerar $f(x)$ om $p(x) > 0$, och $g(x)$ om $p(x) = 0$.

↬ EXEMPEL 2.15 (Returnera det största talet.) Följande två konstruktioner kräver nog inga kommentarer.

$$Max(x, y) = Om(Större(x, y), x, y)$$
$$StörstAvTre(x, y, z) = Max(x, Max(y, z))$$

Division och delbarhet Att dividera ett naturligt tal x med ett nollskilt naturligt tal y innebär som bekant (sidan 10) att från x dra bort så många y:n att återstoden blir mindre än y. Antalet bortdragna y:n är divisionens kvot q, och återstoden r är divisionens rest. Det hela kan sammanfattas med

[†] John McCarthy är känd som mannen bakom programspråket LISP.

$$x - q \cdot y = r$$
eller
$$x = q \cdot y + r$$
där $r < y$

$11 - 2 \cdot 4 = 3$
eller
$11 = 2 \cdot 4 + 3$

Om man adderar y till båda sidor av den senare likheten erhålls

$$x + y = (1 + q) \cdot y + r$$

$11 + 4 = 3 \cdot 4 + 3$

När x ökas med y enheter ökar således kvoten med en enhet medan resten förblir oförändrad. Ett konstaterande som visar hur resultatet från x dividerat med y kan användas för att dividera $x+y$ med y. Tabellen nedanför illustrerar hur det ser ut när några tal divideras med 4.

Rekursion!

x	0	1	2	3	4	5	6	7	8	9	10	11	...
q	0	0	0	0	1	1	1	1	2	2	2	2	...
r	0	1	2	3	0	1	2	3	0	1	2	3	...

Här divideras x med 4.

Om vi inför beteckningarna $Kvot(x, y)$ och $Rest(x, y)$ för kvoten respektive resten, kan rekursionen skrivas

$$\begin{aligned} Kvot(x + y, y) &= Öka(Kvot(x, y)) \\ Rest(x + y, y) &= Rest(x, y) \end{aligned}$$

eller

$$\begin{aligned} Kvot(x, y) &= Öka(Kvot(x - y, y)) && (4) \\ Rest(x, y) &= Rest(x - y, y) && (5) \end{aligned}$$

(4) och (5) gäller enbart om $x \geq y$. Ifall $x < y$, kan inget helt y dras från x, med följd att kvoten blir lika med 0 och resten blir lika med hela x. Det följer att

$$\begin{aligned} Kvot(x, y) &= Om(Mindre(x, y), 0, Öka(Kvot(Sub(x, y), y))) \\ Rest(x, y) &= Om(Mindre(x, y), x, Rest(Sub(x, y), y)) \end{aligned}$$

Den här rekursionen är inte *primitiv*. Se vidare övning 2.17.

PROVKÖRNING:

$$
\begin{aligned}
&\mathit{Kvot}(19,4) && \mathit{Rest}(19,4)\\
&= \mathit{Öka}(\mathit{Kvot}(15,4)) && = \mathit{Rest}(15,4)\\
&= \mathit{Öka}(\mathit{Öka}(\mathit{Kvot}(11,4))) && = \mathit{Rest}(11,4)\\
&= \mathit{Öka}(\mathit{Öka}(\mathit{Öka}(\mathit{Kvot}(7,4)))) && = \mathit{Rest}(7,4)\\
&= \mathit{Öka}(\mathit{Öka}(\mathit{Öka}(\mathit{Öka}(\mathit{Kvot}(3,4))))) && = \mathit{Rest}(3,4)\\
&= \mathit{Öka}(\mathit{Öka}(\mathit{Öka}(\mathit{Öka}(0)))) && = 3\\
&= \mathit{Öka}(\mathit{Öka}(\mathit{Öka}(1)))\\
&= \mathit{Öka}(\mathit{Öka}(2))\\
&= \mathit{Öka}(3)\\
&= 4
\end{aligned}
$$

Till sist, med hjälp av funktionen *Rest* kan vi tillverka en enkel delbarhetstestare:

$$\mathit{Delbar}(x,d) = \mathit{Noll?}(\mathit{Rest}(x,d))$$

Äkta, triviala och prima Vissa tal har äkta delare. Andra har det inte. T. ex. har 6 de äkta delarna 2 och 3, medan 5 bara har triviala delare, dvs. 1 och 5.

För en återblick, se sid. 15.

↬ EXEMPEL 2.16 (Äkta delare.) För att reda ut om ett tal x har någon äkta delare, behöver man bara undersöka om x har någon delare d i området $2 \leq d \leq x-1$, vilket leder oss till att formulera

$$\mathit{HarÄktaDelare}(x) = \mathit{HarDelareIOmrådet}(x, 2, x-1)$$

Funktionen $\mathit{HarDelareIOmrådet}(x,y,z)$ som är specialiserad på att undersöka om x har någon delare d i området $y \leq d \leq z$, kan konstrueras med primitiv rekursion:

$$\mathit{HarDelareIOmrådet}(x,y,y) = \mathit{Delbar}(x,y)$$
$$\mathit{HarDelareIOmrådet}(x,y,\mathit{Öka}(z)) =$$
$$\quad \mathit{Eller}(\mathit{Delbar}(x,\mathit{Öka}(z)), \mathit{HarDelareIOmrådet}(x,y,z))$$

PROVKÖRNING:

$$\begin{aligned}
&\mathit{HarDelareIOmrådet}(5,2,4)\\
&= \mathit{Eller}(\mathit{Delbar}(5,4), \mathit{HarDelareIOmrådet}(5,2,3))\\
&= \mathit{Eller}(\mathit{Delbar}(5,4), \mathit{Eller}(\mathit{Delbar}(5,3), \mathit{HarDelareIOmrådet}(5,2,2)))\\
&= \mathit{Eller}(\mathit{Delbar}(5,4), \mathit{Eller}(\mathit{Delbar}(5,3), \mathit{Delbar}(5,2)))\\
&= \mathit{Eller}(\mathit{Delbar}(5,4), \mathit{Eller}(\mathit{Delbar}(5,3), 0))\\
&= \mathit{Eller}(\mathit{Delbar}(5,4), \mathit{Eller}(0,0))\\
&= \mathit{Eller}(0, \mathit{Eller}(0,0))\\
&= \mathit{Eller}(0,0)\\
&= 0
\end{aligned}$$

Med hjälp av funktionen *HarÄktaDelare* är det lätt att konstruera en primtalstestare:

↬ EXEMPEL 2.17 (En primtalstestare.) Vi säger att ett naturligt tal är *prima* om det är större än 1 och saknar äkta delare. Härav,

$$\mathit{Prima}(x) = \mathit{Och}(\mathit{Större}(x,1), \mathit{Icke}(\mathit{HarÄktaDelare}(x)))$$

Primitiv rekursion och iteration

När den primitivt rekurserande variabelns återlöpning är genomförd ända ner till bottenvärdet, och basfallet har applicerats, så startar ett beräkningsflöde som kan uttryckas med *iteration* (repetition). Som en illustration av detta skall vi ännu en gång iaktta en provkörning av funktionen

$$\begin{cases} \mathit{Triangeltal}(0) = 0 \\ \mathit{Triangeltal}(\mathit{Öka}(x)) = \mathit{Add}(\mathit{Öka}(x), \mathit{Triangeltal}(x)) \end{cases}$$

Rekursiva anrop bygger upp en stack. Efter de fem första raderna bottnar rekursionen. Därefter är alla beräkningar av ett och samma slag: *Addera ett ständigt ökande tal till ett ackumulerande resultat.*

$$
\begin{aligned}
&\mathit{Triangeltal}(4)\\
&= \mathit{Add}(4, \mathit{Triangeltal}(3))\\
&= \mathit{Add}(4, \mathit{Add}(3, \mathit{Triangeltal}(2)))\\
&= \mathit{Add}(4, \mathit{Add}(3, \mathit{Add}(2, \mathit{Triangeltal}(1))))\\
&= \mathit{Add}(4, \mathit{Add}(3, \mathit{Add}(2, \mathit{Add}(1, \mathit{Triangeltal}(0)))))\\
&= \mathit{Add}(4, \mathit{Add}(3, \mathit{Add}(2, \mathit{Add}(1, 0))))\\
&= \mathit{Add}(4, \mathit{Add}(3, \mathit{Add}(2, 1)))\\
&= \mathit{Add}(4, \mathit{Add}(3, 3))\\
&= \mathit{Add}(4, 6)\\
&= 10
\end{aligned}
$$

De avslutande additionerna är enkla att göra med en slinga i stenhögsspråket

Sålänge $k \leq x$ {Addera k och r i r; Öka k}

som kan kompletteras till en stenhögsprocedur

Triangeltal x i r =
  $k \leftarrow 1$; $r \leftarrow 0$
  Sålänge $k \leq x$ {Addera k och r i r; Öka k}

vilken ger samma beräkningsresultat som den primitivt rekursiva funktionen *Triangeltal*.

På motsvarande sätt kan man för varje funktion som är stöpt i den primitivt rekursiva mallen,

$$
\begin{cases}
\mathit{Fantasi}(0) = b\\
\mathit{Fantasi}(\mathit{Öka}(x)) = \mathit{GörNågotMed}(x, \mathit{Fantasi}(x))
\end{cases}
$$

finna en iterativ stenhögsprocedur som ger samma beräkningsresultat:

Proceduren GörNågotMed får *inte* förändra k. Varför?

Fantasi x i r =
  $k \leftarrow 0$; $r \leftarrow b$
  Sålänge $k < x$ {GörNågotMed k och r i r; Öka k}

Ovillkorig iteration Notera att när *Fantasifunktionen* och *Fantasiproceduren* ovanför körs igång på inputvärdet x tvingas den förra göra x stycken *rekursionsanrop* och den senare x stycken *iterationer*.

Antalet iterationer i en procedur som härmar primitiv rekursion är således bestämd av inputvärdet, dvs. på förhand känt. Sådan iteration sägs vara *ovillkorlig*.

Låt oss utvidga stenhögsspråkets vokabulär med ett alias för *ovillkorlig* iteration:

Repetera med $k = 1$ **till** n {P}

för

$k \leftarrow 1$
Sålänge $k \leq n$ {P; Öka k}†

Med hjälp av vårt nya alias kan triangeltalsproceduren på sidan 42 skrivas

$r \leftarrow 0$
Repetera med $k = 1$ **till** n {Addera k till r }

Annan rekursion

En öppen fråga i början av 1900-talet inom den *konstruktiva matematiken* var vilka byggnadsmetoder som krävdes för att man skulle kunna bygga alla ”mekaniskt konstruerbara” funktioner (de funktioner vars beskrivningar kan ges i form av precisa recept av något slag).

Det tycktes som om metoderna sammansättning och primitiv rekursion var tillräckliga. Ty ingen kände till någon mekaniskt konstruerbar funktion som var omöjlig att nå via dessa metoder enbart. Inte förrän *Ackermann* presenterade sin funktion ...

† Proceduren P får *inte* förändra k.

Ackermanns funktion, på gränsen till det obeskrivbara År 1928 publicerade Wilhelm Ackermann[†] ett recept på en rekursiv funktion som besvarade en för den tiden aktuell fråga. En fråga som vore lika intressant idag om inte Ackermann redan hade givit svaret. Ackermann hade nämligen – med ett relativt invecklat rekursionsförfarande – lyckats konstruera en märklig funktion (se nedanför) samt bevisa att den var omöjlig att konstruera enbart medelst sammansättning och det enklare rekursionsförfarandet (som därefter kom att kallas för primitiv rekursion).

Nedanför finner du receptet på Ackermanns funktion. Lägg märke till att funktionens *båda* argument rekurserar. I den tredje ekvationens högerled har Ack dessutom sig själv i ett av argumenten. Något som brukar kallas *dubbel* rekursion.

$$\begin{cases} Ack(0, y) = \ddot{O}ka(y) \\ Ack(\ddot{O}ka(x), 0) = Ack(x, 1) \\ Ack(\ddot{O}ka(x), \ddot{O}ka(y)) = Ack(x, Ack(\ddot{O}ka(x), y)) \end{cases}$$

Efter en beskedlig inledning uppvisar Ackermanns funktion plötsligt en förvånande tillväxtstakt.

y	0	1	2
$Ack(0, y)$	1	2	3
$Ack(1, y)$	2	3	4
$Ack(2, y)$	3	5	7
$Ack(3, y)$	5	13	29
$Ack(4, y)$	13	65533	$2.0^{\ddagger} \times 10^{19728}$

På nästa sida finner du en provkörning.

[†] En tysk matematiker som arbetade i Göttingen tillsammans med David Hilbert. [‡] Ett närmevärde.

$$
\begin{aligned}
& Ack(2,2) \\
&= Ack(1, Ack(2,1)) \\
&= Ack(1, Ack(1, Ack(2,0))) \\
&= Ack(1, Ack(1, Ack(1,1))) \\
&= Ack(1, Ack(1, Ack(0, Ack(1,0)))) \\
&= Ack(1, Ack(1, Ack(0, Ack(0,1)))) \\
&= Ack(1, Ack(1, Ack(0,2))) \\
&= Ack(1, Ack(1,3)) \\
&= Ack(1, Ack(0, Ack(1,2))) \\
&= Ack(1, Ack(0, Ack(0, Ack(1,1)))) \\
&= Ack(1, Ack(0, Ack(0, Ack(0, Ack(1,0))))) \\
&= Ack(1, Ack(0, Ack(0, Ack(0, Ack(0,1))))) \\
&= Ack(1, Ack(0, Ack(0, Ack(0,2)))) \\
&= Ack(1, Ack(0, Ack(0,3))) \\
&= Ack(1, Ack(0,4)) \\
&= Ack(1,5) \\
&= Ack(0, Ack(1,4)) \\
&= Ack(0, Ack(0, Ack(1,3))) \\
&= Ack(0, Ack(0, Ack(0, Ack(1,2)))) \\
&= Ack(0, Ack(0, Ack(0, Ack(0, Ack(1,1))))) \\
&= Ack(0, Ack(0, Ack(0, Ack(0, Ack(0, Ack(1,0)))))) \\
&= Ack(0, Ack(0, Ack(0, Ack(0, Ack(0, Ack(0,1)))))) \\
&= Ack(0, Ack(0, Ack(0, Ack(0, Ack(0,2))))) \\
&= Ack(0, Ack(0, Ack(0, Ack(0,3)))) \\
&= Ack(0, Ack(0, Ack(0,4))) \\
&= Ack(0, Ack(0,5)) \\
&= Ack(0,6) \\
&= 7
\end{aligned}
$$

När man jämför den här stackens förvandling med hur det ser ut vid körning av en primitivt rekursiv funktion (Se sidan 34.) kan man notera en väsentlig skillnad. Vid körning av *Ack* sväller och krymper stacken *flera gånger*!

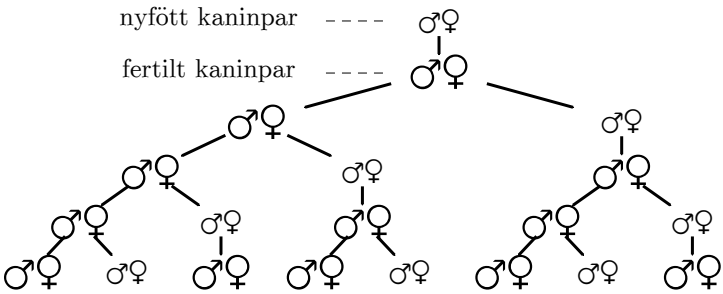
Ackermanns lyckades visa, att en sådan enorm tillväxt på output som hans funktion har, det kan ingen primitivt rekursiv funktion uppvisa – och knappast någon människa föreställa sig. När man tittar på de tre ekvationerna som beskriver Ack, och ser ett enda ställe (den första ekvationen) där output ökar, och då bara med en futtig enhet. Och när man vidare konstaterar att startvärdet $Ack(0,0)$ är 1, ja då är det svårt att föreställa sig att $Ack(4,4)$ är så stort, att om vi fick använda alla pappersark som finns att uppbringa på vår jord för att skriva ut talet, så skulle de inte räcka till.

Rekursion bakåt i flera steg Det finns intressanta funktioner som bara med stort besvär låter sig fångas med primitiv rekursion, men som kan beskrivs mycket enkelt med annan typ av rekursion. Här är två exempel.

↬ EXEMPEL 2.18 (Fibonaccitalen) Antag att en nyfödd kanin blir fertil efter en månad, och att ett kaninpar (hane och hona) en månad senare och fortlöpande varje månad därefter producerar två nya kaniner (hane och hona), vilka i sin tur blir fertila efter en månad och därefter producerar två nya kaniner, osv.. Hur många kaninpar har man efter 12 månader, om man startar med ett nyfött kaninpar?

Ungefär så skrev *Leonardo Pisano* – vanligen kallad *Leonardo Fibonacci* – i sin bok i räknelära *Liber abaci* år 1220.

En tydlig bild av hur det ena kaninparet efter det andra blir till, ges i ett ”släktträd” där ett nyfött kaninpar tronar i toppen, och där mängden kaninpar månad för månad återfinns i trädets olika nivåer.



Antalet kaninpar månad för månad beräknas med en enkel tvåstegsrekurserande† funktion, som vi ger ett kort och kärnfullt namn.

$$\begin{cases} Fib(0) = 0 \\ Fib(1) = 1 \\ Fib(x+2) = Add(Fib(x+1), Fib(x)) \end{cases}$$

x	0	1	2	3	4	5	6	7	8	9	...
$Fib(x)$	0	1	1	2	3	5	8	13	21	34	...

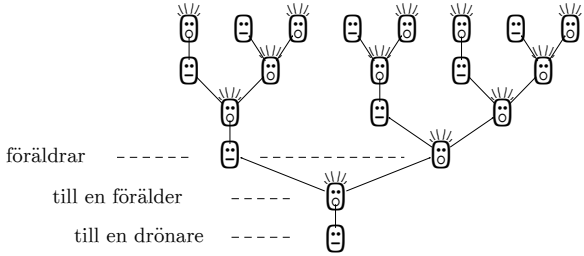
Fibonaccitalen

ANM. 2.6 Kaninparstalen, eller *Fibonaccitalen* som de kom att kallas av den franske 1800-talsmatematikern *Édouard Lucas*, har visat sig dyka upp i allehanda förklädnader. Den mest kända förekomsten av Fibonaccitalen har man nog i växtriket, där de t. ex. finns i grenverket hos ett päronträd, i solrosens blomkorg, i ett blomkålshuvud, i en tallkotte osv..

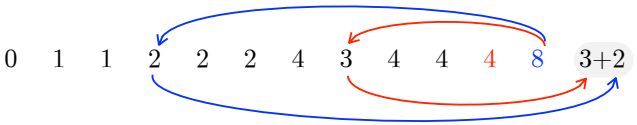
Även i djurriket (det som hyser riktiga djur, inte kaniner av det slag som omnämns i Fibonaccis räknebok), förekommer Fibonaccitalen. T. ex. är det så i ett bisamhälle att förfäderna i den n:te generationen (räknat bakåt i tiden) till en drönare§

† Den speciella formen hos *Fib*-funktionens rekursion följer av (i) att antalet fertila kaninpar en viss månad är lika med totala antalet kaninpar månaden innan, och (ii) att antalet nyfödda kaninpar en viss månad är lika med totala antalet kaninpar två månader tidigare. Försök själv härleda (i) och (ii) från antagandet om kaninparens fortplantning. § drönare = manligt bi.

ges av $Fib(n)$. Något som beror på att en drönare produceras utan sex av en bidrottning, medan en drottning produceras av en drottning och en drönare tillsammans. Se figuren nedanför.



↬ EXEMPEL 2.19 (*Kaos*, en släkting till *Fib.*) Precis som i förra exemplet ska vi i detta exempel tillverka nästa output genom att addera två tidigare output – men nu inte (säkert) de två senaste. Dessa fungerar bara som ett slags pekare till de output som skall adderas. Närmare bestämt anger värdet på de två senaste hur långt (från det senaste) som man skall gå tillbaka för att hitta de två som skall adderas.

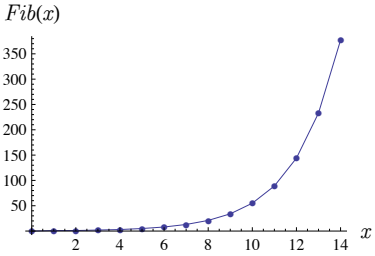


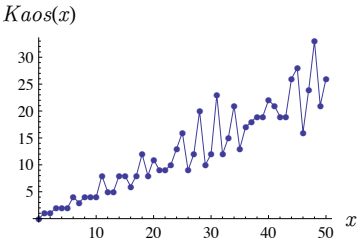
Gå tillbaka 4 resp. 8 steg. Addera talen som du finner.

Precis som i fallet med Fibonaccitalen startar vi med 0 och 1. De output som följer på detta sätt beskrivs av

$$\begin{cases} Kaos(0) = 0 \\ Kaos(1) = 1 \\ Kaos(x+2) = Add(Kaos(Sub(x+1, Kaos(x))), \\ \qquad\qquad\qquad Kaos(Sub(x+1, Kaos(x+1)))) \end{cases}$$

och har, i motsats till Fibonaccitalen, ett oväntat kaotiskt uppförande:





Att en liten förändring i en funktion förvandlar ett mönstergillt beteende till ett kaotiskt, illustrerar att ordning och kaos kan ligga nära varandra.

ANM. 2.7 Den som tycker sig känna igen s.k. *exponentiell* tillväxt i Fibonaccifunktionens graf har rätt – i varje fall för stora värden på argumentet. Exponentialfunktionernas egenskap: $k^{x+1} = k \cdot k^x$ kan jämföras med Fibonaccifunktionens dito för stora värden på x: $Fib(x+1) \approx \varphi \cdot Fib(x)$, där φ är det s.k. *gyllene snittet*. Måhända ligger en del av förklaringen till Fibonaccifunktionens rika tillämpningar i denna egenskap.

En rekursiv fläta I vissa rekursiva konstruktioner anropar två eller flera funktioner varandra – och kanske också sig själva – på ett sätt som gör att ingen av dem står på egna ben.

↬ EXEMPEL 2.20 Här är ett enkelt exempel:

$$\begin{cases} Jämn(0) = 1 \\ Udda(0) = 0 \\ Jämn(Öka(x)) = Udda(x) \\ Udda(Öka(x)) = Jämn(x) \end{cases}$$

Två ihopflätade booleska funktioner.

Ovanstående konstruktion† vilar på det faktum att om ett tal är udda så är nästa tal jämnt, och om ett tal är

† De två funktionerna *Udda* och *Jämn* kan även konstrueras med primitiv rekursion. (Övning 2.8 på sid. 56)

jämnt så är nästa udda. T. ex. beräknas $Jämn(3)$ så här:

$$Jämn(3) = Udda(2) = Jämn(1) = Udda(0) = 0$$

Framåtrekursion Vissa funktioner beskrivs naturligast med en sorts bakvänd primitiv rekursion. Närmare bestämt rekursion *framåt* istället för bakåt. Här är ett par exempel.

↬ EXEMPEL 2.21 (Är x kvadratisk?) Antag att vi vill veta ifall x är kvadratisk, dvs. om x är lika med någon av kvadraterna $0^2, 1^2, 2^2, \ldots$ Eftersom x omöjligen kan vara lika med någon kvadrat som är mindre än x, gäller det förstås att snabbt löpa förbi (se $\curvearrowright$) sådana kvadrater, och stanna först (se $\downarrow$) när man träffar på en kvadrat som *inte* är mindre än x. Den kvadraten är den enda som x möjligen kan vara lika med.

$$\begin{aligned}
&Kvadratisk(x) = LöpFörbiKvadraterna(0, x)\\
&LöpFörbiKvadraterna(n, x) =\\
&\qquad Om(Mindre(Mult(n, n), x),\\
&\qquad\quad LöpFörbiKvadraterna(Öka(n), x), && (\curvearrowright)\\
&\qquad\quad Lika(Mult(n, n), x)) && (T)
\end{aligned}$$

PROVKÖRNINGAR:

$$\begin{aligned}
Kvadratisk(4) &= LöpFörbiKvadraterna(0, 4)\\
&\overset{(\curvearrowright)}{=} LöpFörbiKvadraterna(1, 4)\\
&\overset{(\curvearrowright)}{=} LöpFörbiKvadraterna(2, 4)\\
&\overset{(\downarrow)}{=} Lika(Mult(2, 2), 4)) = 1
\end{aligned}$$

$$\begin{aligned} Kvadratisk(8) &= LöpFörbiKvadraterna(0,8) \\ &\overset{(\curvearrowright)}{=} LöpFörbiKvadraterna(1,8) \\ &\overset{(\curvearrowright)}{=} LöpFörbiKvadraterna(2,8) \\ &\overset{(\curvearrowright)}{=} LöpFörbiKvadraterna(3,8) \\ &\overset{(\downarrow\!\!\!\downarrow)}{=} Lika(Mult(3,3),8)) = 0 \end{aligned}$$

↬ EXEMPEL 2.22 Nu ska vi konstruera en funktion $Primtal(x)$ som returnerar primtalen i tur och ordning:

x	0	1	2	3	...
$Primtal(x)$	2	3	5	7	...

Idé: Utgå från nuvarande primtal, och sök nästkommande genom att stega framåt med en enhet i taget tills ett primtal påträffas. Hjälpfunktionen *NästaPrimtal* nedanför, som rekurserar ett steg framåt, förverkligar denna idé. Huvudfunktionen *Primtal* anropar sig dock med rekursion ett steg *bakåt*.

$$\begin{cases} Primtal(0) = 2 \\ Primtal(Öka(x)) = NästaPrimtal(Öka(Primtal(x))) \end{cases}$$

$$\begin{aligned} NästaPrimtal(y) = Om(&Prima(y), \\ &y, && (\downarrow) \\ &NästaPrimtal(Öka(y))) && (\curvearrowright) \end{aligned}$$

PROVKÖRNING:

$$\begin{aligned} Primtal(4) &= NästaPrimtal(Öka(Primtal(3))) \\ &= NästaPrimtal(Öka(7)) = NästaPrimtal(8) \\ &\overset{(\curvearrowright)}{=} NästaPrimtal(9) \overset{(\curvearrowright)}{=} NästaPrimtal(10) \\ &\overset{(\curvearrowright)}{=} NästaPrimtal(11) \overset{(\downarrow\!\!\!\downarrow)}{=} 11 \end{aligned}$$

Klassen av rekursiva funktioner

De *primitivt rekursiva funktionerna* (Se sidan 36.) är de funktioner som är möjliga att konstruera utifrån basfunktionen med hjälp av sammansättning och den primitiva formen av rekursion.

Med klassen av *allmänt* rekursiva funktioner, eller helt kort *de rekursiva funktionerna*, avses den utvidgning av de primitivt rekursiva funktionerna som åstadkoms av att man tillåts använda andra former[†] av rekursion än den primitivt rekursiva – som t. ex. ”Ackermann-rekursion”, ”Fibonacci-rekursion” eller framåtrekursion.

De rekursiva funktionerna bildar en så rik klass av funktioner att den tycks innehålla varje ”mekaniskt konstruerbar” funktion. I alla händelser har ingen hittills lyckats skriva ihop något recept som accepterats som mekaniskt utan att resultatet ingår i den här klassen av funktioner. Jag skriver *tycks*, ty något bevis i sträng mening finns inte och kommer inte heller att finnas så länge som begreppet ”mekaniskt konstruerbar” vilar på intuitiv grund. Läs mer nedanför om det här begreppets svårfångade men till synes självklara väsen.

Mekaniskt konstruerbar, algoritmisk En intuition för vad som är ”mekaniskt konstruerbart” har nog varje programmerare och varje matematiker. Men att precist formulera begreppet – att ge en definition – ställer sig inte helt självklart.

Svårigheterna att definiera ”mekaniskt konstruerbar”

[†] Till en början – före Ackermanns exempel – fanns inte begreppet *allmänt rekursiva funktioner*. Då sa man *rekursiva funktioner* och menade *primitivt* rekursiva funktioner. Den österikiska matematikern Kurt Gödel var den förste att tala om allmänt rekursiva funktioner (1934), men uttryckte sig intuitivt och lite obestämt. Man kan beskriva den här klassen mer precist och ”tekniskt”. Se t. ex. Kleene, *General recursive functions of natural numbers*, MATH. ANNALEN 112, 340-353.

i allmänna termer ledde fram till olika sorts operationella definitioner† vilka alla visat sig vara ekvivalenta, och som således i grunden avser samma sak. De första definitionerna av detta slag kom 1936. Då beskrev den då blott 22-årige engelsmannen Alan Turing en typ av abstrakta maskiner – Turingmaskiner – och menade att det mekaniskt konstruerbara var precis det som dessa maskiner kunde åstadkomma. Samtidigt talade t. ex. de amerikanska matematikerna Alonzo Church och Stephen Cole Kleene i termer av rekursiva funktioner ungefär som vi just har gjort. Något senare (1943) publicerade Emil Leon Post - som också var en amerikansk matematiker - ett s.k. *strängmanipulerande system*. På 50- och 60-talen kom ytterligare formuleringar genom bl.a. Wang (1957), Smullyan (1962), Shepherdson och Sturgis (1963), Elgot och Robinson (1964) och Minsky (1967).

Alla dessa förslag att via abstrakta maskiner eller symbolspråk definiera ”det mekaniskt konstruerbara” visade sig vara likvärdiga. Att så många forskare har haft samma begrepp i tankarna, men gett det olika formuleringar, kan tas som intäkt på att begreppet är fundamentalt.

Det finns flera olika namn på detta begrepp, t. ex. *effektiv*, *beräkningsbar* eller *algoritmisk*. Ibland försöker man trots allt beskriva begreppet i allmänna termer. Då brukar man tala lite lagom obestämt om en metod eller process som har input och output och som är

- *deterministisk*, dvs. att processen ger samma resultat varje gång som betingelserna är desamma;
- *ändlig*, att processen avslutas inom ändlig tid, och att processen inte kräver oändligt mycket av någon annan kvantitet heller;

† Som jämförelse kan man tänka på hur svårigheter att definiera begreppet intelligens i allmänna termer har framkastat förslaget att definiera intelligens som ”det intelligenstest mäter”.

- *mekanisk*, att processen är så fullständigt och precist beskriven att inget ytterligare behöver tänkas ut. En maskin skall kunna utföra processen;
- *numerisk*, att processen kan beskrivas i termer av heltalen $0, 1, 2, ...$

Förutom de algoritmer som du har mött i denna skrift, så har du säkert sett matrecept eller stickbeskrivningar som uppfyller ovanstående villkor och därför utgör exempel på algoritmer.

Programspråk och rekursiva funktioner I ett programspråk vill man (oftast) ha möjlighet att kunna skriva varje tänkbar algoritm. Eftersom nu alla algoritmer kan formuleras inom klassen av rekursiva funktioner (enligt föregående avsnitt), så ska följaktligen ett programspråk möjliggöra beräkningar av just dessa funktioner.

Alltså bör man i ett programspråk på ett eller annat sätt kunna

- beräkna funktionerna *Öka*,
- bilda *sammansättningar*,
- göra *rekursion*

Vad gäller sammansättningar – att ”först göra en sak” och sedan använda resultatet av detta för att ”göra en ny sak” – så erbjuder programspråken subrutiner och andra procedurella tekniker där en procedur kan användas inuti en annan. Och rekursiva konstruktioner – där program kan anropa sig själva – tillåts i nästan alla programspråk. Annars finns det iterativa konstruktioner som alternativ till de rekursiva.

Övningar

2.1 Visa hur följande uttryck kan beräknas som sammansättningar av $Add(x, y)$, $Mult(x, y)$ och $Pot(x, y)$.

a) $x + y + z$ b) $x \cdot (y + z)$ c) $x^{(y^z)}$
d) $(x^y)^z$ e) x^{y+z} f) $(x + y)^z$
g) $(x + y)^{z \cdot u}$

2.2 Visa (med provkörning) hur *Pot* beräknar 5^4.

2.3 Konstruera $Fakultet(x) = 1 \cdot 2 \cdot \ldots \cdot x$.† med primitiv rekursion.

2.4 I vart och ett av nedanstående tre uttryck adderas x stycken objekt. T. ex. adderas kvadrater i det första uttrycket, jämna tal i det andra och triangeltal i det tredje.

$$1 + 2^2 + 3^2 + \ldots + x^2$$

$$2 + 4 + 6 + \ldots + 2 \cdot x$$

$$1 + (1 + 2) + (1 + 2 + 3) + \ldots + (1 + 2 + 3 + \ldots + x)$$

Funktionen f nedanför kan användas som en mall vid beräkning av ovanstående tre uttryck (och liknande uttryck). Visa hur f – genom lämpliga val av *Vadå* – kan användas för att beräkna just de tre uttrycken ovanför.

$$\begin{cases} f(0) = 0 \\ f(\ddot{O}ka(x)) = Add(\,Vadå(x), f(x)) \end{cases}$$

2.5 Bestäm $f(1)$, $f(2)$, $f(3)$, samt $f(x)$ för godtyckligt x, då

$$f(x) = h(x, x) \quad \text{och} \quad \begin{cases} h(x, 0) = 1 \\ h(x, \ddot{O}ka(y)) = Mult(2, h(x, y)) \end{cases}$$

2.6 Betrakta följande rekursiva konstruktion.

$$\begin{cases} f(0) = 1 \\ f(\ddot{O}ka(x)) = f(Sub(x, f(x))) \end{cases}$$

† Betecknas ofta med $x!$

a) Förklara varför konstruktionen *inte* följer den primitivt rekursiva mallen.
b) Presentera en konstruktion inom klassen av primitivt rekursiva funktioner som beräknar samma sak (som konstruktionen ovanför).

2.7 Beskriv hur funktionen Sub (EXEMPEL 2.12 på sidan 36) följer den primitivt rekursiva mallen.

2.8 Visa hur $Udda$ och $Jämn$ var och en kan tillverkas med den primitivt rekursiva mallen.

2.9 Konstruera följande booleska funktioner inom klassen av primitivt rekursiva funktioner
a) $Lika(x, y)$ som reder ut om $x = y$,
b) $Olika(x, y)$ som reder ut om $x \neq y$,
c) $MindreEllerLika(x, y)$ som reder ut om $x \leq y$,

2.10 Visa att uttrycken nedanför kan beräknas med hjälp enbart av den primitivt rekursiva mallen och funktionerna $Add(x, y)$ och $Mult(x, y)$.

a) 2^x b) $1^3 + 2^3 + 3^3 + \ldots + x^3$

2.11 Betrakta följande stenhögsprocedur

Töm r
Repetera med $k = 1$ till x {
  Potensupphöj k till k i e
  Addera e till r}

a) Vilket värde får r vid körning på $x = 3$?
b) Tillverka en funktion inom klassen av primitivt rekursiva funktioner som beräknar samma sak som stenhögsproceduren ovan.

2.12 Föreställ dig att du endast kan röra dig framåt 1 eller 2 meter i taget. Visa att antalet sätt som du då kan ta dig fram x meter ges av $Fib(x + 1)$.

2.13 Konstruera en procedur i stenhögsspråket som beräknar samma sak som följande primitivt rekursiva funktion.

$$\begin{cases} f(0) = 1 \\ f(\ddot{O}ka(x)) = Add(f(x), Mult(\ddot{O}ka(x), \ddot{O}ka(\ddot{O}ka(x)))) \end{cases}$$

2.14 Betrakta funktionerna i tabellen nedanför

x	0	1	2	3	4	5	6	7	8	9	10	11	...
$f(x)$	1	2	0	0	1	2	0	0	1	2	0	0	...
$g(x)$	1	0	3	2	5	4	7	6	9	8	11	10	...

Konstruera

a) f med rekursion som löper tillbaka fyra steg, och g med rekursion som löper tillbaka två steg,
b) bägge funktionerna inom klassen av primitivt rekursiva funktioner.

2.15 Beräknar följande två funktioner samma sak?

$$\begin{cases} f(0) = 0 \\ f(\ddot{O}ka(x)) = \ddot{O}ka(\ddot{O}ka(f(x))) \end{cases} \qquad g(n) = Mult(2, n)$$

2.16 Konstruera en boolesk funktion som reder ut huruvida x är ett triangeltal eller ej.

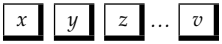
2.17 Tillverka primitivt rekursiva versioner av funktionerna *Kvot* och *Rest*. LEDNING: Studera tabellen på sidan 39.

3

Listor och tabeller

Listor

En (ändlig) lista är en följd av noder (positioner) som kan fyllas med ”vad som helst”.



I praktiken är det *innehållet* i noderna som intresserar oss – inte själva positionerna. Därför menar vi oftast följ-

den av nodinnehåll (element) – istället för följden av noder – när vi talar om en lista. Denna lilla språkförbistring får vi stå ut med.

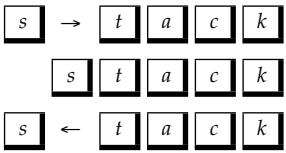
Vad vill man göra med en lista? De manipulationer som man vill göra med listor är ofta av följande slag:

- Välja en position i en lista och undersöka eller byta ut dess innehåll.
- Lägga till eller ta bort en position.
- Foga samman två listor till en.

Nämnda manipulationer behöver du t. ex. göra om du vill byta ut någon bokstav eller något ord i den text som är skriven på den här sidan, eller om du vill sätta in ytterligare ord eller kanske mer ”luft” i texten, eller om du vill sätta samman två avsnitt till ett.

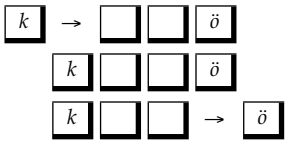
Stack och kö Med hjälp av manipulationer i listans ändpositioner – som är förhållandevis enkla att göra – kan man åstadkomma manipulationer även inuti listan, något som vid en ytlig betraktelse kan framstå som ett mysterium, men som vi snart kommer att ge flera exempel på. Listor som bara tillåter manipulationer i ändpositionerna är därför avsevärt mer flexibla än vad man skulle kunna tro – och mycket flitigt använda. De vanligaste två går under benämningarna *stack* och *kö*.

En lista vars positioner är åtkomliga endast i ena änden kallas för *stack*.



Om positioner läggs till (*pushas*) eller tas bort (*poppas*) sker detta således i stackens ena ände. Tänk t. ex. på en trave böcker. F.ö. har vi redan sett hur rekursiva funk-

tioner organiserar sina arbetsuppgifter i stackar.



En *kö* är en lista där positioner läggs till i ena änden och tas bort i den andra.

Syntax för listor I fortsättningen kommer vi att beteckna listor på följande sätt.

$[x\ y\ \ldots\ z]$	icketom lista
[ ] eller **nil**	tomma listan

Listor av listor Vi skall tillåta listor att innehålla listor, t. ex. som nedanför.

[[ ]]
[2 3 [ ]]
[1 [2 3 [ ]] [ ] 5]

Det utmärkande för sådana listor är att de har djupstruktur. Som exempel kan nämnas ett stycke text som ytterst består av meningar, vilka i sin tur består av ord som består av tecken. Tecknen är här de minsta beståndsdelarna – textens *atomer*. Listors atomer är på motsvarande sätt något slag av element som vi önskar betrakta som odelbara, t. ex. naturliga tal. När vi fortsättningsvis talar om en lista $[x\ y\ \ldots\ z]$ menar vi – om inget annat sägs – att x, y, …, z är listor eller atomer, och att atomerna är naturliga tal.

atomer

[2 3 5 7] är *atomär.*

En lista av atomer kallas i fortsättning för en *atomär* eller *enkel* lista. Även tomma listan hänförs till denna kategori.

Listfunktioner

Med en *listfunktion* menar vi en funktion vars input och output är listor eller naturliga tal. Listfunktioner utgör därmed ett slags generalisering av heltalsfunktioner som vi redan är bekanta med från kapitel 2. Vi ska nu visa hur dessa nya funktioner kan byggas med sammansättning och rekursion på motsvarande sätt som heltalsfunktionerna. Vi behöver emellertid ytterligare en basfunktion.

Basfunktionen En grundläggande algoritmisk verksamhet på en lista är att utöka densamma med en extra nod. Därför definierar vi en basfunktion för en sådan syssla:

basfunktionen

$$\mathit{FogaIn}(\mathit{nod}, [x\ y\ \ldots\ z] = [\mathit{nod}\ x\ y\ \ldots\ z]$$

En populär infixbeteckning för $\mathit{FogaIn}(\mathit{nod}, L)$ – som vi kommer att använda ofta fortsättningsvis – är $\big[\mathit{nod}\big|\, L\big]$.†

Sammansättning Listfunktioner kan sättas samman på motsvarande sätt som heltalsfunktioner. T. ex. är

Om t. ex. $L = [c\ d\ e]$, så är $\big[a\big|\,\big[b\big|\, L\big]\big] = [a\ b\ c\ d\ e]$.

$$\mathit{FogaIn}(a, \mathit{FogaIn}(b, L)) = \big[a\big|\,\big[b\big|\, L\big]\big]$$

en motsvarighet till sammansättningen $\mathit{Öka}(\mathit{Öka}(x))$ från heltalsfunktionernas värld. Se mer i exemplet som följer.

↬ EXEMPEL 3.1 (Är listan tom?) Genom att sätta samman vår gamla funktion *Noll*? med *Längd*-funktionen från exemplet nedanför erhålls en funktion som undersöker om en lista är tom.

$$\mathit{Tom}?(L) = \mathit{Noll}?(\mathit{Längd}(L))$$

Primitiv rekursion Den *primitiva rekursionens* kännetecken för listmanipulation kan sägas vara att manipulationen av listans *återstod*, dvs. den del av ursprungslistan som finns till höger om listans inledande element,

† J.f.r. med infixnotationen $x + y$ för $\mathit{Add}(x, y)$.

används för att manipulera hela listan. Vi börjar med några exempel.

↬ EXEMPEL 3.2 (Längden av en lista.) Med *längden* av en lista avses antalet noder i listan. Tom lista har därför längden 0. Om en lista utökas med ytterligare en nod, så ökar förstås listans längd med en enhet. Härav nedanstående recept:

$$\begin{aligned} &\mathit{Längd}([\,]) = 0 \\ &\mathit{Längd}(\left[\mathit{nod} \,\middle|\, L\right]) = \mathit{Öka}(\mathit{Längd}(L)) \end{aligned}$$

PROVKÖRNING:

$$\begin{aligned} &\mathit{Längd}([[\,]\ 1\ [2]]) \\ &= \mathit{Öka}(\mathit{Längd}([1\ [2]])) \\ &= \mathit{Öka}(\mathit{Öka}(\mathit{Längd}([[2]]))) \\ &= \mathit{Öka}(\mathit{Öka}(\mathit{Öka}(\mathit{Längd}([\,])))) \\ &= \mathit{Öka}(\mathit{Öka}(\mathit{Öka}(0))) \\ &= \mathit{Öka}(\mathit{Öka}(1)) \\ &= \mathit{Öka}(2) \\ &= 3 \end{aligned}$$

↬ EXEMPEL 3.3 (Foga samman.) Givetvis vill vi kunna foga samman två listor till en, t. ex. $[a\ b\ c]$ och $[d\ e]$ till $[a\ b\ c\ d\ e]$. Man kan konstatera att om $[b\ c]$ och $[d\ e]$ redan sammanfogats till $[b\ c\ d\ e]$, så återstår bara att utöka resultatet i vänster ände med a. Följande funktion är byggd på detta kontaterande.

$$\begin{aligned} &\mathit{FogaSamman}([\,], Y) = Y \\ &\mathit{FogaSamman}(\left[x \,\middle|\, X\right], Y) = \left[x \,\middle|\, \mathit{FogaSamman}(X, Y)\right] \end{aligned}$$

PROVKÖRNING:

$$\begin{aligned}
&\mathit{FogaSamman}([a\ b\ c], [d\ e]) \\
&= \big[a \big| \mathit{FogaSamman}([b\ c], [d\ e])\big] \\
&= \big[a \big| \big[b \big| \mathit{FogaSamman}([c], [d\ e])\big]\big] \\
&= \big[a \big| \big[b \big| \big[c \big| \mathit{FogaSamman}([\,], [d\ e])\big]\big]\big] \\
&= \big[a \big| \big[b \big| \big[c \big| [d\ e]\big]\big]\big] \\
&= \big[a \big| \big[b \big| [c\ d\ e]\big]\big] \\
&= \big[a \big| [b\ c\ d\ e]\big] \\
&= [a\ b\ c\ d\ e]
\end{aligned}$$

↬ EXEMPEL 3.4 (Utöka i höger ände.) Ibland vill man utöka en lista L med en extra *nod* i *höger* ände istället för vänster. Det kan man göra genom att foga samman listorna L och $[nod]$. Vi betecknar sådan utökning med $\mathit{FogaTill}(L, nod)$.†

$$\mathit{FogaTill}(L, nod) = \mathit{FogaSamman}(L, [nod])$$

PROVKÖRNING:

$$\begin{aligned}
&\mathit{FogaTill}([a\ b\ c], d) \\
&= \mathit{FogaSamman}([a\ b\ c], [d]) \\
&= [a\ b\ c\ d]
\end{aligned}$$

Sortering Man säger att en enkel lista av tal är *sorterad* om dess element är placerade i storleksordning från mindre till större eller från större till mindre när man läser listan från vänster till höger.

T. ex. är [2 3 5], [5 3 2] och [5 2 2] sorterade, men inte [2 5 3].

† Ibland skriver vi $\mathit{FogaTill}(L, nod)$ med infixnotation $\big[L \,\big\|\, nod\big]$.

↬ EXEMPEL 3.5 (Sortering genom instickning.) Här presenteras en sorteringsfunktion som bygger på det enkla faktum att om man redan har sorterat hela listan förutom det första elementet, så behöver man bara sticka in det första elementet på rätt plats i den redan sorterade delen.

Många sorterar en kortlek på detta sätt.

$$\begin{aligned} &Sortera([\,]) = [\,] \\ &Sortera(\left[nod \middle| L\right]) = StickIn(nod, Sortera(L)) \end{aligned}$$

Instickningen fungerar på så sätt att det element som skall stickas in i den redan sorterade listan jämförs med det första elementet i den sorterade listan. Och beroende på hur jämförelsen utfaller, vidtages relevant åtgärd.

$$\begin{aligned} &StickIn(x, [\,]) = [x] \\ &StickIn(x, \left[y \middle| Y\right]) = Om(Mindre(x, y), \\ &\qquad\qquad \left[x \middle|\, \left[y \middle| Y\right]\right], \\ &\qquad\qquad \left[y \middle| StickIn(x, Y)\right]) \end{aligned}$$

PROVKÖRNING:

$$\begin{aligned} & StickIn(5, [2\ 3\ 7\ 11]) \\ = {} & \left[2 \middle| StickIn(5, [3\ 7\ 11])\right] \\ = {} & \left[2 \middle|\, \left[3 \middle| StickIn(5, [7, 11])\right]\right] \\ = {} & \left[2 \middle|\, \left[3 \middle|\, \left[5 \middle| [7, 11]\right]\right]\right] \\ = {} & \left[2 \middle|\, \left[3 \middle|\, [5\ 7\ 11]\right]\right] \\ = {} & \left[2 \middle|\, [3\ 5\ 7\ 11]\right] \\ = {} & [2\ 3\ 5\ 7\ 11] \end{aligned}$$

Listfunktioners primitiva rekursion

Rekursionen i exemplen ovanför följer ett mönster som går under ett bekant namn.

Den primitivt rekursiva mallen[†]

$$f([\,]) = b$$
$$f(\left[nod \middle| L\right]) = g(nod, L, f(L))$$
$$f([\,], y_1, \ldots, y_n) = B(y_1, \ldots, y_n)$$
$$f(\left[nod \middle| L\right], y_1, \ldots, y_n) = g(nod, L, f(L, y_1, \ldots, y_n))$$

b, nod, y_k avser listor eller atomer.

↬ EXEMPEL 3.6 (En återblick.) Vi visar att *Längd*- och *FogaSamman*-recepten från EXEMPLEN 3.2 och 3.3 följer mallen.

$$f = \mathit{Längd} \qquad f([\,]) = 0$$
$$f(\left[nod \middle| L\right]) = \mathit{Öka}(f(L))$$
$$f = \mathit{FogaSamman} \qquad f([\,], Y) = Y$$
$$f(\left[nod \middle| L\right], Y) = \mathit{FogaIn}(nod, f(L, Y))$$

↬ EXEMPEL 3.7 (Att foga listor av listor.) Inget hindrar basfunktionen *FogaIn* från att utöka en lista med en lista. Detsamma gäller funktionen *FogaTill*. Likaså kan *FogaSamman* slå ihop listor som innehåller listor:

$$\mathit{FogaIn}([\mathbf{nil}\ 1], [2\ 0\ \ [2\ \mathbf{nil}]]) = [[\mathbf{nil}\ 1]\ \ 2\ 0\ \ [2\ \mathbf{nil}]]$$
$$\mathit{FogaTill}([\mathbf{nil}\ 1], [2\ 0\ \ [2\ \mathbf{nil}]]) = [\mathbf{nil}\ 1\ \ [2\ 0\ \ [2\ \mathbf{nil}]]]$$
$$\mathit{FogaSamman}([\mathbf{nil}\ 1], [2\ 0\ \ [2\ \mathbf{nil}]]) = [\mathbf{nil}\ 1\ 2\ 0\ \ [2\ \mathbf{nil}]]$$

[†] Den här mallen skall – precis som den förra (sidan 33) – tolkas så att funktionerna g och B inte nödvändigtvis behöver använda alla sina argument, eller att det rekurserande argumentet är just det första ifall f har flera argument.

Isärtagare Att kunna plocka ut delar av en lista är ett ”måste”. Därför bör du ha följande fyra enkla ”isärtagare” lätt tillgängliga i din verktygslåda. I nästa avsnitt presenteras dessutom en isärtagare som delar en lista mitt itu.

$$\begin{aligned} &\mathit{Första}([\,]) = [\,] \\ &\mathit{Första}(\left[\mathit{nod}\middle|\, L\right]) = \mathit{nod} \end{aligned} \qquad \begin{aligned} &\mathit{UtomFörsta}([\,]) = [\,] \\ &\mathit{UtomFörsta}(\left[\mathit{nod}\middle|\, L\right]) = L \end{aligned}$$

$$\begin{aligned} &\mathit{Sista}([\,]) = [\,] \\ &\mathit{Sista}(\left[\mathit{nod}\middle|\, L\right]) = \\ &\quad \mathit{Om}^{\dagger}(\mathit{Tom?}(L), \\ &\qquad \mathit{nod}, \\ &\qquad \mathit{Sista}(L)) \end{aligned} \qquad \begin{aligned} &\mathit{UtomSista}([\,]) = [\,] \\ &\mathit{UtomSista}(\left[\mathit{nod}\middle|\, L\right]) = \\ &\quad \mathit{Om}(\mathit{Tom?}(L), \\ &\qquad [\,], \\ &\qquad \left[\mathit{nod}\middle|\, \mathit{UtomSista}(L)\right]) \end{aligned}$$

PROVKÖRNINGAR:

$$\begin{aligned} &\mathit{Sista}([a\ b\ c]) \\ &= \mathit{Sista}([b\ c]) \\ &= \mathit{Sista}([c]) \\ &= [c] \end{aligned} \qquad \begin{aligned} &\mathit{UtomSista}([a\ b\ c]) \\ &= \left[a\middle|\, \mathit{UtomSista}([b\ c])\right] \\ &= \left[a\middle|\, \left[b\middle|\, \mathit{UtomSista}([c])\right]\right] \\ &= \left[a\middle|\, \left[b\middle|\, [\,]\right]\right] \\ &= \left[a\middle|\, [b]\right] \\ &= [a\ b] \end{aligned}$$

Annan rekursion än primitiv

Det är ibland naturligt att arbeta med rekursion av annat slag än primitiv. Exemplen med tudelning, reversering och sortering nedanför utgör goda illustrationer.

---

† Ifall rekursionssteget är villkorat väljer man ibland att låta ett extra basfall ta hand om villkorets specialfall. T. ex. skulle receptet för *Sista* kunna skrivas som nedan istället.

$$\begin{aligned} &\mathit{Sista}([\,]) = [\,] \\ &\mathit{Sista}([\mathit{nod}]) = \mathit{nod} \\ &\mathit{Sista}(\left[\mathit{nod}\middle|\, L\right]) = \mathit{Sista}(L) \end{aligned}$$

Att kunna dela en lista *mitt itu* är nödvändigt för några klassiska listfunktioner (Se EXEMPEL 3.9 och övning 3.14). Tudelarfunktionen nedanför använder sig av en hjälpfunktion för att skyffla över hälften av elementen till en från början tom lista.

$$
\begin{aligned}
&\mathit{TuDela}(L) = h([\,], L)\\
&h(X, [\,]) = [X\ [\,]]\\
&h(X, \left[y \middle| Y\right]) = \mathit{Om}(\mathit{Mindre}(\mathit{Längd}(X), \mathit{Längd}(Y)),\\
&\qquad\qquad h(\left[X \,\middle\|\, y\right], Y),\\
&\qquad\qquad \left[X\ \ \left[y \middle| Y\right]\right])
\end{aligned}
$$

Rekursionen går till så att den högra listan förlorar sitt inledande element, medan den vänstra utökas i höger ände med samma element. När de två listorna är lika långa eller när den vänstra är ett element kortare än den högra bottnar rekursionen.

PROVKÖRNINGAR:

$$
\begin{aligned}
&\mathit{TuDela}([a\ b\ c\ d]) = h([\,], [a\ b\ c\ d])\\
&\quad = h([a], [b\ c\ d])\\
&\quad = h([a\ b], [c\ d])\\
&\quad = [[a\ b]\ [c\ d]]
\end{aligned}
\qquad
\begin{aligned}
&\mathit{TuDela}([a\ b\ c]) = h([\,], [a\ b\ c])\\
&\quad = h([a], [b\ c])\\
&\quad = [[a]\ [b\ c]]
\end{aligned}
$$

Provkörningarna illustrerar hur en lista av jämn längd tudelas i två exakt lika långa delar, och hur en av udda längd delas så att vänstra delen blir en enhet kortare än den högra. I alla händelser kallar vi de två delarna för *VänsterHalva* och *HögerHalva*:

$$
\begin{aligned}
\mathit{VänsterHalva}(L) &= \mathit{Första}(\mathit{TuDela}(L))\\
\mathit{HögerHalva}(L) &= \mathit{Sista}(\mathit{TuDela}(L))
\end{aligned}
$$

↬ EXEMPEL 3.8 (Att *reversera* en lista.) Funktionen nedanför vänder på en lista med hjälp av en funktion som påminner om *Tudela*:s hjälpfunktion.

$$
\begin{aligned}
&\mathit{Reversera}(L) = h([\,], L)\\
&h(X, [\,]) = X\\
&h(X, \left[y \middle| Y\right]) = h(\left[y \middle| X\right], Y)
\end{aligned}
$$

Se provkörning nedanför.

PROVKÖRNING:

$$\begin{aligned} \mathit{Reversera}([a\ b\ c]) &= h([\,], [a\ b\ c]) \\ &= h([a], [b\ c]) \\ &= h([b\ a], [c]) \\ &= h([c\ b\ a], [\,]) \\ &= [c\ b\ a] \end{aligned}$$

Sortering igen. Om en listans två halvor redan är sorterade, så behöver man bara ”smälta ihop”† dem:

↬ EXEMPEL 3.9 (Mergesort)

$$\begin{aligned} \mathit{Sortera}(L) = \mathit{Om}(&\mathit{Mindre}(\mathit{Längd}(L), 2), \\ &L, \\ &\mathit{Merge}(\mathit{Sortera}(\mathit{VänsterHalva}(L)), \\ &\quad \mathit{Sortera}(\mathit{HögerHalva}(L)))) \end{aligned}$$

$$\begin{aligned} &\mathit{Merge}(L, [\,]) = L \\ &\mathit{Merge}([\,], L) = L \\ &\mathit{Merge}(\left[x \middle| X\right], \left[y \middle| Y\right]) = \mathit{Om}(\mathit{Mindre}(x, y), \\ &\qquad\qquad \left[x \middle| \mathit{Merge}(X, \left[y \middle| Y\right])\right], \\ &\qquad\qquad \left[y \middle| \mathit{Merge}(\left[x \middle| X\right], Y)\right]) \end{aligned}$$

PROVKÖRNING:

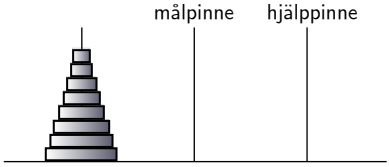
$$\begin{aligned} \mathit{Merge}([1\ 3\ 5], [2\ 3\ 4]) &= \left[1 \middle| \mathit{Merge}([3\ 5], [2\ 3\ 4])\right] \\ &= \left[1 \middle| \left[2 \middle| \mathit{Merge}([3\ 5], [3\ 4])\right]\right] \\ &= \left[1 \middle| \left[2 \middle| \left[3 \middle| \mathit{Merge}([5], [3\ 4])\right]\right]\right] \\ &= \left[1 \middle| \left[2 \middle| \left[3 \middle| \left[3 \middle| \mathit{Merge}([5], [4])\right]\right]\right]\right] \\ &= \left[1 \middle| \left[2 \middle| \left[3 \middle| \left[3 \middle| \left[4 \middle| \mathit{Merge}([5], [\,])\right]\right]\right]\right]\right] \\ &= \left[1 \middle| \left[2 \middle| \left[3 \middle| \left[3 \middle| \left[4 \middle| [5]\right]\right]\right]\right]\right] \\ &= [1\ 2\ 3\ 3\ 4\ 5] \end{aligned}$$

† *Merge* på engelska.

Lite kombinatorik

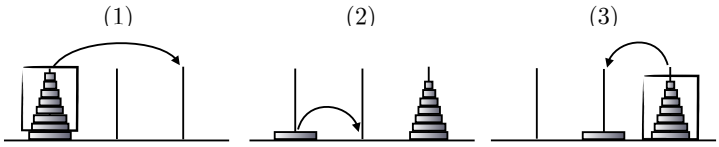
Kombinatorik är en gren av den diskreta matematiken inom vilken man försöker beskriva hur *arrangemang* av olika slag kan se ut och hur många de är. T. ex. är lösningen av pusslet nedanför av kombinatorisk art.

↬ EXEMPEL 3.10 (Hanois torn.) I slutet av 1800-talet presenterade den franske matematikern Édouard Lucas ett slags matematiskt pussel ("recréation mathématiques"), vars mål var att flytta en trave skivor av olika storlekar från en pinne till en annan.



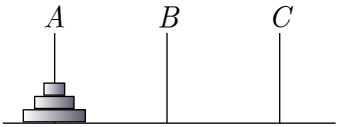
Enligt pusslets regler skulle skivorna flyttas – en i taget – utan att någon skiva hamnar ovanpå en mindre skiva. En tredje pinne fick användas som hjälppinne på vilken en skiva kunde vila medan andra skivor flyttades. En lyckosam strategi som löser pusslet beskrivs nedanför.

STRATEGI För att flytta ett torn med $n+1$ skivor till en viss målpinne, måste man förr eller senare flytta tornets bottenskiva. Men först måste (1) de n översta skivorna mellanlagras på hjälppinnen. Efter förflyttningen (2) av bottenskivan, återstår bara att (3) flytta de mellanlagrade skivorna till målpinnen. Det är allt.



Antag t. ex. att ett 3 skivor högt torn skall flyttas från A till B, med C som hjälppinne.

Ett 3-torn



Strategin uppmanar oss att

Låt oss skriva $A \to B$, $A \to C$, $B \to C$.

(1) först mellanlagra det översta 2-tornet på C. Medelst strategin kan man göra det genom tre enstaka skivförflyttningar: den minsta skivan flyttas från A till B (mellanlagring), nästa skiva från A till C, och till sist flyttas den nyss mellanlagrade minsta skivan en gång till, nu från B till C,

$A \to B$.

(2) sedan flytta bottenskivan från A till B,

$C \to A$, $C \to B$, $A \to B$.

(3) till sist flytta det mellanlagrade 2-tornet från C till B. Här kan vi kopiera förflyttningarna i (1), men med C, B, A i A, C, B:s gamla roller.

En *följd* av $3+1+3$ enstaka skivförflyttningar (Se marginalen ovanför!) löser således pusslet med 3 skivor. Men hur ska vi presentera lösningen? Få se . . . , en *följd* är ju inget annat än innehållet element för element i en lista. Så låt oss paketera nämnda följd i en lista:

$$[A{\to}B \;\; A{\to}C \;\; B{\to}C \;\; A{\to}B \;\; C{\to}A \;\; C{\to}B \;\; A{\to}B]$$

Funktionen *Hanoi* nedanför, som är byggd med hjälp av den nämnda strategin – och därför löser pusslet – returnerar listor just av ovanstående slag. Funktionens första argument beskriver tornets höjd mätt i antal skivor, och de övriga argumenten representerar i tur och ordning ”frånpinnen”, ”tillpinnen” och ”hjälppinnen”.

$$
\begin{aligned}
&\mathit{Hanoi}(0, A, B, C) = \mathbf{nil}\\
&\mathit{Hanoi}(\ddot{O}ka(n), A, B, C)\\
&\qquad = \mathit{FogaSamman}(\mathit{Hanoi}(n, A, C, B),\\
&\qquad\qquad \big[A{\to}B \,\big|\, \mathit{Hanoi}(n, C, B, A)\big])
\end{aligned}
$$

Första argumentet rekurserar ett steg bakåt. Ändå är *Hanoi* *inte* konstruerad med primitiv rekursion. Varför?

PROVKÖRNINGAR:

$$
\begin{aligned}
&\mathit{Hanoi}(1, A, B, C)\\
&= \mathit{FogaSamman}(\mathit{Hanoi}(0, A, C, B), \big[A{\to}B \,\big|\, \mathit{Hanoi}(0, C, B, A)\big])\\
&= \mathit{FogaSamman}(\mathbf{nil}, \big[A{\to}B \,\big|\, \mathbf{nil}\big]) = [A{\to}B]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Hanoi}(2, A, B, C)\\
&= \mathit{FogaSamman}(\mathit{Hanoi}(1, A, C, B), \big[A{\to}B \,\big|\, \mathit{Hanoi}(1, C, B, A)\big])\\
&= \mathit{FogaSamman}([A{\to}C], \big[A{\to}B \,\big|\, [C{\to}B]\big])\\
&= \mathit{FogaSamman}([A{\to}C], [A{\to}B \;\; C{\to}B])\\
&= [A{\to}C \;\; A{\to}B \;\; C{\to}B])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Hanoi}(3, A, B, C)\\
&= \mathit{FogaSamman}(\mathit{Hanoi}(2, A, C, B), \big[A{\to}B \,\big|\, \mathit{Hanoi}(2, C, B, A)\big])\\
&= \mathit{FogaSamman}([A{\to}B \;\; A{\to}C \;\; B{\to}C]\,,\\
&\qquad\qquad \big[A{\to}B \,\big|\, [C{\to}A \;\; C{\to}B \;\; A{\to}B]\big])\\
&= \mathit{FogaSamman}([A{\to}B \;\; A{\to}C \;\; B{\to}C]\,,\\
&\qquad\qquad [A{\to}B \;\; C{\to}A \;\; C{\to}B \;\; A{\to}B])\\
&= [A{\to}B \;\; A{\to}C \;\; B{\to}C \;\; A{\to}B \;\; C{\to}A \;\; C{\to}B \;\; A{\to}B]
\end{aligned}
$$

ANM. 3.1 Man kan visa att pusslet inte går att lösas med färre skivförflyttningar än de som vår lösning ger upphov till.

↬ EXEMPEL 3.11 Vi har nyss löst problemet med hur de enskilda skivorna skall flyttas i Hanoitornspusslet. Ett annat intressant problem om Hanoitornspusslet är följande.

Hur många skivförflyttningar måste man göra när ett n-torn flyttas?

Här är tre olika lösningar.
LÖSNING 1: Eftersom funktionen *Hanoi* returnerar alla skivförflyttningar paketerade i en lista, och ingen kortare lista duger (se ANM. 3.1), behöver vi bara beräkna listans längd.

$$\mathit{AntalSkivförflyttningar}(n) = \mathit{Längd(Hanoi(n,A,B,C))}$$

LÖSNING 2: Nästa lösning använder sig inte explicit av funktionen *Hanoi*, men är inspirerad av densamma.

$$\begin{aligned}&\mathit{AntalSkivförflyttningar}(0) = 0\\ &\mathit{AntalSkivförflyttningar}(\mathit{Öka}(n))\\ &\qquad = 1 + 2\mathit{AntalSkivförflyttningar}(n)\end{aligned}$$

LÖSNING 3: Med hjälp av lösningarna ovanför är det lätt att beräkna några inledande output.

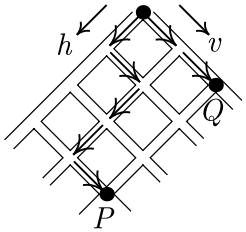
n	0	1	2	3	4	5	6
$\mathit{AntalSkivförflyttningar}(n)$	0	1	3	7	15	31	63

Tycker du att talen $0, 1, 3, 7, 15, 31, 63$ verkar bekanta?
LEDNING: Jämför med talen $1, 2, 4, 8, 16, 32, 64$. dvs. med $2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6$.
Du har rätt om du gissar att

$$\mathit{AntalSkivförflyttningar}(n) = 2^n - 1.$$

Kombinationer av vänster och höger Nästa exempel beskriver alla sätt att ta sig från en punkt till en annan – med vissa restriktioner på förflyttningarna.

↬ EXEMPEL 3.12 (Zickzackvägar) Föreställ dig en stadsdel vars gatunät består av enkelriktade gator med två tillåtna riktningar – betecknade v respektive h i figuren. Från en infart i övre hörnet kan man zickzacka sig fram till stadsdelens olika gatukorsningar. Till Q nedanför går det *en* enda väg, men till P går det flera – fastän bara en är utritad.



Två vägar $[h\ v\ h\ h\ v]$ och $[v\ v]$.

Varje lista med v:n och h:n beskriver en väg – och omvänt.

Hur ser vägbeskrivningen ut för den väg till P som har 2 v:n som avslutning?

Eftersom figurens P ligger 2 kvarter till vänster om och 3 kvarter till höger om infarten måste varje väg till P – likt den utritade – innehålla 2 stycken v:n och 3 stycken h:n.

Nu ska vi konstruera listfunktionen $AllaVägar(n, k)$ som för $n \geqslant k$, returnerar en lista innehållande *alla* vägar av längd n med k stycken v:n (och resten h:n).

(a) Betrakta först en gatukorsning som likt figurens Q ligger längs någon av de två övre kanterna av gatunätet. Till en sådan gatukorsning finns det blott *en* väg. Beroende på om gatukorsningen ligger på den övre vänstra kanten eller den högra, så kommer vägen dit att innehålla enbart h:n eller enbart v:n, dvs. den är av typ $[h\ h\ \ldots\ h]$ eller $[v\ v\ \ldots\ v]$.

En lista vars element är likadana tillverkas enklast av

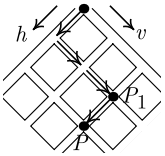
Se övning 3.2, sid 85.

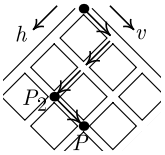
$$Bara(n, x) = [\underbrace{x\ x\ \ldots\ x}_{n \text{ st}}]$$

Varför paketeras $Bara(n, h)$ och $Bara(n, v)$ i listor?

Härav,

$$AllaVägar(n, 0) = [Bara(n, h)]$$
$$AllaVägar(n, n) = [Bara(n, v)]$$





(b) Betrakta sedan en gatukorsning som likt P i marginalen *inte* befinner sig på någon av gatunätets övre två kanter. Till en sådan gatukorsning finns det vägar med ett avslutande h och vägar med ett avslutande v – se illustrationen i marginalen. De förra passerar P_1, och de senare går genom P_2. Vidare gäller det att alla vägar till P_1 kan utökas med ett h, och därmed bli vägar till P. Likaså kan alla vägar till P_2 efter utökning med v bli vägar till P. Några andra vägar till P finns inte.

Det betyder att om man redan känner alla vägar till P_1 och alla vägar till P_2, behöver man följaktligen bara utöka dessa på det nyss beskrivna sättet, och vips så har man alla vägar till P.

Notera att vägarna till P_1 har samma längd som de till P_2, men att de senare har ett v mer än de förra. Det följer att om vägarna till P_1 beskrivs av $AllaVägar(n, k)$, så ges vägarna till P_2 av $AllaVägar(n, k + 1)$, och de till P av

Vi konstruerar *Vänster-Utöka* och *HögerUtöka* på sid. 75.

$$\begin{aligned} &AllaVägar(Öka(n), Öka(k)) = \\ &\quad FogaSamman(VänsterUtöka(AllaVägar(n, k)), \\ &\qquad\qquad HögerUtöka(AllaVägar(n, Öka(k)))) \end{aligned}$$

Det kompletta receptet för funktionen *AllaVägar* blir därmed

$$
\begin{aligned}
&\mathit{AllaVägar}(n,0) = [\mathit{Bara}(n,h)]\\
&\mathit{AllaVägar}(n,n) = [\mathit{Bara}(n,v)]\\
&\mathit{AllaVägar}(\mathit{Öka}(n),\mathit{Öka}(k)) =\\
&\quad \mathit{FogaSamman}(\,\mathit{VänsterUtöka}(\mathit{AllaVägar}(n,k)),\\
&\qquad\qquad \mathit{HögerUtöka}(\mathit{AllaVägar}(n,\mathit{Öka}(k))))
\end{aligned}
$$

PROVKÖRNINGAR:

$$
\begin{aligned}
&\mathit{AllaVägar}(2,1)\\
&= \mathit{FS}(\,\mathit{VU}(\mathit{AllaVägar}(1,0)),\mathit{HU}(\mathit{AllaVägar}(1,1))),\\
&= \mathit{FS}(\,\mathit{VU}([[h]]),\mathit{HU}([[v]]))\\
&= \mathit{FS}([[h\,v]],[[v\,h]])\\
&= [[h\,v]\ \ [v\,h]]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{AllaVägar}(3,2)\\
&= \mathit{FS}(\,\mathit{VU}(\mathit{AllaVägar}(2,1)),\mathit{HU}(\mathit{AllaVägar}(2,2)))\\
&= \mathit{FS}(\,\mathit{VU}([[h\,v]\ \ [v\,h]]),\mathit{HU}([[v\,v]]))\\
&= \mathit{FS}([[h\,v\,v]\ \ [v\,h\,v]]\,,[[v\,v\,h]])\\
&= [[h\,v\,v]\ \ [v\,h\,v]\ \ [v\,v\,h]]
\end{aligned}
$$

***Map*, en högre ordningens funktion.** Nu ska vi som utlovat (sid. 74) konstruera listfunktionerna *VänsterUtöka* och *HögerUtöka*. De två vägutökarna är tänkta att för en given lista av vägar, utöka alla i listan ingående vägar med ett avslutande v resp. h.

Låt oss först konstruera *Map*, som är en mer generell listfunktion. Dess specialitét är att med hjälp av olika funktioner f **m**anipulera **a**lla **p**ositioner i en lista.

$$
\begin{aligned}
&\mathit{Map}(f,[\,]) = [\,]\\
&\mathit{Map}(f,\left[\mathit{nod}\,\middle|\,L\right]) = \left[f(\mathit{nod})\,\middle|\,\mathit{Map}(f,L)\right]
\end{aligned}
$$

PROVKÖRNING:

$$\begin{aligned} Map(f, [a\ b\ c]) &= \big[f(a)\,\big|\,Map(f, [b\ c])\big] \\ &= \big[f(a)\,\big|\,\big[f(b)\,\big|\,Map(f, [c])\big]\big] \\ &= \big[f(a)\,\big|\,\big[f(b)\,\big|\,\big[f(c)\,\big|\,[\,]\big]\big]\big] \\ &= [f(a)\ f(b)\ f(c)] \end{aligned}$$

Notera att *Map* har egenheten att ta olika *funktioner* som argument. Varje funktion som – likt *Map* – kan ta funktioner som argument brukar kallas för en *högre ordningens funktion.*

Vid användning av *Map* gäller det att välja ett f som utför avsedd nodmanipulation. T. ex.

$$\begin{aligned} Map(Öka, [1\ 2\ 3]) &= [2\ 3\ 4] \\ Map(Prima, [1\ 2\ 3]) &= [0\ 1\ 1] \\ Map(Längd, [[1\ 2\ 3]\ \ [100]\ \ [2\ 3]]) &= [3\ 1\ 2] \\ Map(f_1, [[h\,v]\ \ [v\,h]]) &= [[h\,v\,v]\ \ [v\,h\,v]] \\ om\ f_1(väg) &= FogaTill(väg, v) \\ Map(f_2, [[h\,v]\ \ [v\,h]]) &= [[h\,v\,h]\ \ [v\,h\,h]] \\ om\ f_2(väg) &= FogaTill(väg, h) \end{aligned}$$

Följaktligen är

$$\begin{aligned} VänsterUtöka(vägar) &= Map(f_1, vägar) \\ HögerUtöka(vägar) &= Map(f_2, vägar) \end{aligned}$$

↬ EXEMPEL 3.13 (Antalet vägar.) *Beräkna antalet vägar av längd n som innehåller k stycken v:n.*

Vi presenterar två lösningar.

LÖSNING 1: Det sökta antalet är lika med antalet element i listan $AllaVägar(n, k)$.

$$AntalVägar(n, k) = Längd(AllaVägar(n, k))$$

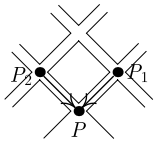
LÖSNING 2: Ovanstående lösning förutsätter att man *konstruerar*† vägarna för att räkna ut hur många de är. Detta kommer att visa sig vara onödigt.

† Det är ju vad $AllaVägar(n, k)$ gör.

Till en gatukorsning som ligger på någon av de två övre kanterna går det bara *en* väg. Härav följande två basfall

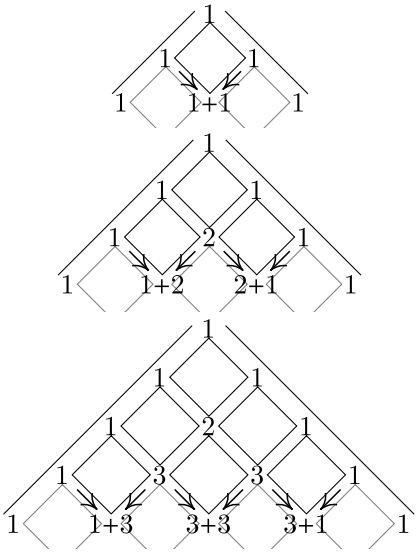
$$AntalVägar(n, 0) = 1$$
$$AntalVägar(n, n) = 1$$

För en gatukorsning P som *inte* ligger på någon av de övre kanterna, kan man konstatera att vägar till P kommer från gatukorsningarna omedelbart ovanför P. Och att varje väg till någon av de senare gatukorsningarna blir efter utökning en väg till P. Följaktligen måste summan av antalet vägar till gatukorsningarna omedelbart ovanför P vara lika med antalet vägar till P. Härav,



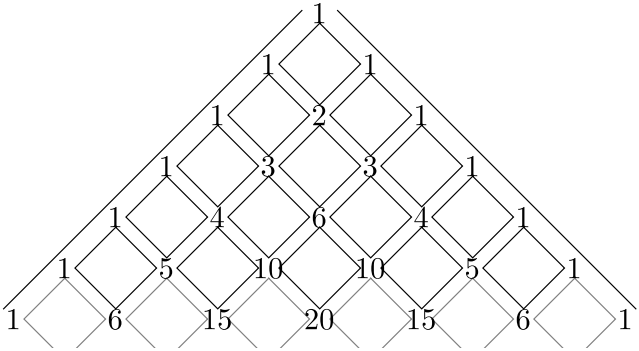
$$\begin{aligned} &AntalVägar(\ddot{O}ka(n), \ddot{O}ka(k)) \\ &\quad = AntalVägar(n, k) + AntalVägar(n, \ddot{O}ka(k)) \end{aligned}$$

Rekursionssteget ovanför är extremt lätt att tillämpa, eftersom det bara innehåller en addition. Här följer några illustrationer där antalet vägar skrivs in i de gatukorsningar till vilka vägarna leder.



Binomialtalen Rad för rad har vi ovanför beräknat hur många vägar det finns till de olika vägkorsningarna på avstånden 2, 3 och 4 från infarten. Nedanför finner läsaren resultatet av ytterligare några beräkningar.

Talen på undre raden är $AntalVägar(6,0)$, $AntalVägar(6,1)$, $AntalVägar(6,2)$, $AntalVägar(6,3)$, $AntalVägar(6,4)$, $AntalVägar(6,5)$, $AntalVägar(6,6)$



Talen i det triangelformade området beskrevs av den franska matematikern Blaise Pascal på 1600-talet. De var dock kända redan på 200-talet f.Kr. i Indien. Skälen till skilda kulturers stora intresse för talen är att de förekommer naturligt i många kombinatoriska sammanhang. Ex.vis vid s.k. *binomialutvecklingar*[†]:

binomial-utvecklingar

$$(1+z)^0 = 1$$
$$(1+z)^1 = 1+z$$
$$(1+z)^2 = 1+2z+z^2$$
$$(1+z)^3 = 1+3z+3z^2+z^3$$
$$(1+z)^4 = 1+4z+6z^2+4z^3+z^4$$
$$(1+z)^5 = 1+5z+10z^2+10z^3+5z^4+z^5$$
$$(1+z)^6 = 1+6z+15z^2+20z^3+15z^4+6z^5+z^6$$

[†] De olika koefficienterna framför z-potenserna kallas *binomialkoefficienter* eller *binomialtal* och har i den matematiska traditionen fått en speciell beteckning, nämligen $\binom{n}{k}$ för den koefficient som står framför z^k i utvecklingen av $(1+z)^n$. T. ex. skriver man $(1+z)^6 = 1+\binom{6}{1}z+\binom{6}{2}z^2+\binom{6}{3}z^3+\binom{6}{4}z^4+\binom{6}{5}z^5+z^6$.

Att $\binom{n}{k}$ överensstämmer med $AntalVägar(n,k)$ får sin förklaring i nedanstående anmärkning.

ANM. 3.2 Man kan konstatera att antalet vägar av längd n som innehåller k stycken v:n är lika med antalet sätt att i en lista av längden n välja k platser (åt v:na), eller helt allmänt ur en mängd med n element av något slag välja k stycken element. T. ex. att ur en låtsamling med n låtar, välja ut k låtar, eller att vid multiplikationen

$$\underbrace{(1+z)(1+z)\ldots(1+z)}_{n\text{ st}}$$

$1+z$ är ett *binom.*

välja z ur k stycken binom (och 1:an ur de återstående binomen). Varje sådant val resulterar i en z^k-term†, när de valda z- och 1-elementen multipliceras ihop. Antalet val är därmed lika med antalet z^k-termer i binomialutvecklingen.

Ett särskilt basfall för atomär nod

Listfunktioner som skall arbeta på listor av listor behöver ibland kunna särskilja en *atomär* nod från en nod som är en *lista*. Det löser vi med hjälp av ett ”atomärt” basfall. Närmare bestämt reserverar vi ordet **atom** för atombeskrivning, och räknar med att varje maskin som kör våra recept känner till detta. Följande exempel illustrerar förhoppningsvis vad som avses.

↬ EXEMPEL 3.14 (En boolesk funktion.) Med hjälp av ovan nämnda reservation av ordet **atom** kan vi konstruera en funktion som särskiljer atomer från listor.

$$\begin{aligned} &Atom?(\mathbf{atom}) = 1\\ &Atom?([\,]) = 0\\ &Atom?(\big[nod\,\big|\,L\big]) = 0 \end{aligned}$$

† Vid multiplikationen $(1+z)(1+z)$ får man t. ex. z-termer när man väljer z ur det första binomet (och 1:an ur det andra), eller z ur det andra (och 1:an ur det första).

↬ EXEMPEL 3.15 (En utslätare.) Antag att du vill avlägsna en del av en texts djupstruktur, t. ex. dess indelning i kapitel, avsnitt och meningar. Då handlar det om att släta ut texten. Nedanför konstrueras en funktion som slätar ut en lista ända ner i botten. (Se även övning 3.18 på sid. 89.) Dvs. den här funktionen tar bort listans hela djupstruktur, så att listans atomer returneras paketerade i all sin nakenhet inuti en tom lista – i samma ordning som de påträffas i den ursprungliga listan.

$$
\begin{aligned}
&\mathit{SlätaUt}([\,]) = [\,]\\
&\mathit{SlätaUt}(\left[\mathit{nod}\middle|L\right]) =\\
&\quad \mathit{Om}(\mathit{Atom?}(\mathit{nod}),\\
&\qquad \mathit{FogaIn}(\mathit{nod}, \mathit{SlätaUt}(L)),\\
&\qquad \mathit{FogaSamman}(\mathit{SlätaUt}(\mathit{nod}), \mathit{SlätaUt}(L)))
\end{aligned}
$$

Som demonstration utslätar vi en lista med oansenlig djupstruktur:

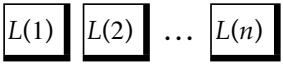
$$
\begin{aligned}
&\mathit{SlätaUt}([3\ [2]])\\
&= \mathit{FogaIn}(3, \mathit{SlätaUt}([[2]]))\\
&= \mathit{FogaIn}(3, \mathit{FogaSamman}(\mathit{SlätaUt}([2]), \mathit{SlätaUt}([\,])))\\
&= \mathit{FogaIn}(3, \mathit{FogaSamman}(\mathit{FogaIn}(2, \mathit{SlätaUt}([\,])), \mathit{SlätaUt}([\,])))\\
&= \mathit{FogaIn}(3, \mathit{FogaSamman}(\mathit{FogaIn}(2, [\,]), [\,]))\\
&= \mathit{FogaIn}(3, \mathit{FogaSamman}([2], [\,]))\\
&= \mathit{FogaIn}(3, [2])\\
&= [3\ 2]
\end{aligned}
$$

Med hjälp av funktionerna *Längd* och *SlätaUt* är det lätt att beräkna antalet atomförekomster i en lista:

$$\mathit{AntalAtomer}(L) = \mathit{Längd}(\mathit{SlätaUt}(L))$$

Tabeller

Endimensionella Antag att man aldrig fogar nya noder till en enkel lista och heller aldrig tar bort noder från den, utan enbart intresserar sig för att undersöka eller ändra nodinnehåll. Då kan man, när man vill beskriva en nods innehåll, referera till nodens position. Om positionerna numreras med t. ex. $i = 1, 2, \ldots, n$, så kan en sådan listas innehåll uppfattas som output $L(i)$ till en funktion L med ändligt många input $1, 2, \ldots, n$.

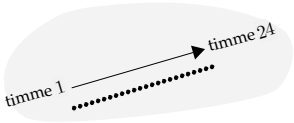


Vi kallar en sådan lista för *endimensionell tabell*, och betecknar den med

$$L(i), i = 1, 2, \ldots, n \quad \text{eller} \quad [L(1)\ L(2)\ \ldots\ L(n)]$$

↬ EXEMPEL 3.16 (En temperaturlista) Tänk dig att vi mäter temperaturen på en viss ort varje timme under ett visst dygn. Dygnets 24 timmar förser oss då med 24 stycken temperaturvärden. Dessa kan bokföras i en endimensionell tabell av längd 24, vilket innebär att temperaturvärdena $T(timme)$ räknas upp det ena efter det andra allt efter vilken timme på dygnet de är uppmätta:

$T(timme)$,
$timme = 1, 2, \ldots, 24$

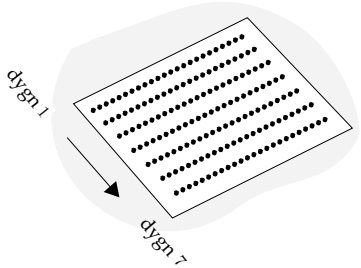


Tabeller av högre dimension

↬ EXEMPEL 3.17 (En tvådimensionell tabell) Tänk dig nu istället att vi har gjort sådana här mätningar varje dygn under en hel vecka. Då har vi $7 \cdot 24 = 168$ temperaturuppgifter att bokföra. Istället för att placera dessa

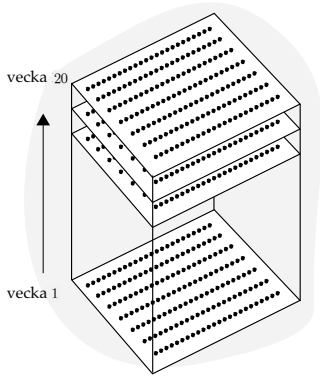
i en endimensionell tabell av längd 168 som ordnar temperaturvärdena enbart efter periodens 168 timmar, kan vi välja att placera dem i en tvådimensionell tabell som håller reda både på vilken veckodag och vilken timme på dygnet som mätningen är gjord. Detta ger mer struktur åt våra mätdata – och mer ordning och reda åt oss. Varje temperaturuppgift $T(dygn, timme)$ blir nu med i två uppräkningar:

$$T(dygn, timme), \begin{cases} dygn = 1, 2, \ldots, 7 \\ timme = 1, 2, \ldots, 24 \end{cases}$$



↬ EXEMPEL 3.18 (En tredimensionell tabell) Låt oss nu gå vidare och tänka oss in i en situation där vi har gjort sådana här temperaturmätningar under flera veckor – t. ex. under årets 20 första veckor. Om vi bokför mätningarna även efter veckonummer får vi en tredimensionell tabell vars element $T(vecka, dygn, timme)$ finns med i tre olika uppräkningar

$$T(vecka, dygn, timme), \begin{cases} vecka = 1, 2, \ldots, 20 \\ dygn = 1, 2, \ldots, 7 \\ timme = 1, 2, \ldots, 24 \end{cases}$$



Man kan åskådliggöra en sådan här tredimensionell tabell som t. ex. ett hus med tjugo våningar, där varje vå-

ning representerar en viss veckas temperaturuppgifter. Vertikal förflyttning i huset motsvarar då en inspektion av de olika veckornas temperaturer en viss veckodag och ett visst klockslag.

↬ EXEMPEL 3.19 (Fyra dimensioner) Vidare, tänk dig att vi gjorde sådana här temperaturmätningar varje år under t. ex. ett decennium (10 år). Då skulle vi kunna bokföra våra mätdata som nedan:

$T(\mathit{år}, \mathit{vecka}, \mathit{dygn}, \mathit{timme})$,

$$\begin{cases} \mathit{år} = 1, 2, \ldots, 10 \\ \mathit{vecka} = 1, 2, \ldots, 20 \\ \mathit{dygn} = 1, 2, \ldots, 7 \\ \mathit{timme} = 1, 2, \ldots, 24 \end{cases}$$

Om du gärna vill tänka i bilder, tänk då t. ex. på 10 stycken 20-våningshus längs en gata, där varje hus innehåller ett års mätdata.

↬ EXEMPEL 3.20 (Fem dimensioner) Om vi fortsätter med insamlingen av mätdata under 10 decennier (ett sekel), så lämpar sig följande tabell för bokföringen:

$T(\mathit{decennium}, \mathit{år}, \mathit{vecka}, \mathit{dygn}, \mathit{timme})$,

$$\begin{cases} \mathit{decennium} = 1, 2, \ldots, 10 \\ \mathit{år} = 1, 2, \ldots, 10 \\ \mathit{vecka} = 1, 2, \ldots, 20 \\ \mathit{dygn} = 1, 2, \ldots, 7 \\ \mathit{timme} = 1, 2, \ldots, 24 \end{cases}$$

”Ett kvarter med 10 gator om vardera 10 stycken 20-våningshus.”

Iteration är en naturlig operation på tabeller Att rekursion är en naturlig operation vid listmanipulering, har vi sett flera exempel på. De operationer som vi kan göra med tabeller utförs mycket naturligt med ovillkorlig (se sid 44) iteration.

↬ EXEMPEL 3.21 Medelvärdet av samtliga temperaturvärden från tabellen $T(vecka, dygn, timme)$ (EXEMPEL 3.18) fås genom att först addera dem och sedan dividera med antalet:

$sum \leftarrow 0$
Repetera med $timme = 1$ **till** 24 {
  Repetera med $dygn = 1$ **till** 7 {
    Repetera med $vecka = 1$ **till** 20 {
      Addera $T(vecka, dygn, timme)$ **till** sum
    }
  }
}
$antal \leftarrow (20 \cdot 7 \cdot 24)$[†]
Dividera sum med $antal$ i $medel$ och r[§]

↬ EXEMPEL 3.22 (Ett annat temperaturmedelvärde) Genom att addera $T(1, 1, 12), T(2, 1, 12), \ldots, T(20, 1, 12)$ och sedan dividera med 20, beräknas medelvärdet av de temperaturer som mättes upp under de 20 veckornas måndagar klockan 12 (här noteras måndag som dygn 1):

$sum \leftarrow 0$
Repetera med $vecka = 1$ **till** 20 {
  Addera $T(vecka, 1, 12)$ **till** sum
}
$antal \leftarrow 20$
Dividera sum med $antal$ i $Medel(dygn, tim)$ och r

[†] Ett förkortat skrivsätt för en stenhögsprocedur som multiplicerar tre högar i högen antal. [§] Divisionsproceduren förser oss med en resthög r som vi inte har någon användning av.

↬ EXEMPEL 3.23 (En medelvärdestabell) Gör för *varje* veckodygns *varje* timme motsvarande beräkning som den för måndagarnas tolvslag ovan. Dvs. fyll en tabell

$$\begin{aligned} &\mathit{Medel}(\mathit{dygn}, \mathit{timme}), \\ &\begin{cases} \mathit{dygn} = 1, 2, \ldots, 7 \\ \mathit{timme} = 1, 2, \ldots, 24 \end{cases} \end{aligned}$$

med medelvärden, ett medelvärde för varje dygn och varje timme:

Repetera med $\mathit{dygn} = 1$ **till** 7 {
  Repetera med $\mathit{tim} = 1$ **till** 24 {
    $\mathit{sum} \leftarrow 0$
    Repetera med $\mathit{vecka} = 1$ **till** 20 {
      Addera $T(\mathit{vecka}, \mathit{dygn}, \mathit{tim})$ **till** sum
    }
    $\mathit{antal} \leftarrow 20$
    Dividera sum med antal i $\mathit{Medel}(\mathit{dygn}, \mathit{tim})$ och r
  }
}

Övningar

3.1 Låt Z vara listan [**nil** 1 [2 **nil**]]. Bestäm
  a) $\mathit{Första}(\mathit{Sista}(Z))$,
  b) $\mathit{Sista}(\mathit{UtomFörsta}(Z))$,
  c) $\mathit{Sista}(\mathit{Sista}(Z))$.

3.2 Tillverka funktionen $\mathit{Bara}(n, x)$ så att den returnerar en lista med n stycken x och inget annat. Så att t. ex. $\mathit{Bara}(3, 5) = [5\ 5\ 5]$ och $\mathit{Bara}(3, [1\ 2]) = [[1\ 2]\ \ [1\ 2]\ \ [1\ 2]]$.

3.3 Antag att L är en atomär lista och att x är en atom (naturligt tal). Konstruera funktionen
   a) *Tillhör*(L, x) så att den reder ut huruvida L innehåller någon x-förekomst eller ej,
   b) *AntalFörekomster*(L, x), som returnerar antalet x-förekomster i L. Så att 3 returneras om ex.vis $x = 4$ och $L = [4\ 2\ 7\ 4\ 1\ 4]$.

3.4 Konstruera funktionen *ListPositioner*(L, x) så att den returnerar en lista med positionerna† till samtliga x-förekomster i L, givet att L och x är som i förra övningen. Så att t. ex. *ListPositioner*$([4\ 2\ 7\ 4\ 1\ 4], 4) = [1\ 4\ 6]$. LEDNING: Tänk dig att du känner positionerna för 4 i $[2\ 7\ 4\ 1\ 4]$, dvs. positionslistan $[3\ 5]$. Vilken manipulation av denna positionslista behöver du göra för att få positionerna för 4 i $[4\ 2\ 7\ 4\ 1\ 4]$?

3.5 Konstruera funktioner som givet en lista och ett godtyckligt naturligt tal n returnerar

Beträffande position, se fotnoten.

   a) listans nod i positionen 1,
   b) listans nod i positionen n,
   c) listan förutom noden i positionen 1,
   d) listan förutom noden i positionen n,
   e) listan efter positionen n.

3.6 Antag att L är en lista. Tillverka en funktion
   a) som utökar L genom att foga in x omedelbart före position 2 i L, så att t. ex. $[a\ b\ c]$ utökas till $[a\ x\ b\ c]$,
   b) *FogaInFöre*(n, L, x) som infogar x omedelbart före position n i L.

3.7 Skriv ett program som med hjälp av tabellen i EXEMPEL 3.18 på sidan 82 beräknar ett temperaturmedelvärde klockan 12
   a) för vecka 5, b) för årets första 20 veckor.

† Låt oss beteckna en listas positioner med $1, 2, \ldots$

3.8 Låt L vara en godtycklig atomär lista. Konstruera funktioner som
   a) tar bort varje x-förekomst från L,
   b) ersätter varje x-förekomst i L med y,
   c) undersöker om L innehåller två eller flera förekomster av något element.

3.9 Låt L vara en godtycklig atomär lista. Konstruera funktioner som
   a) vid förekomst av flera på varandra följande *identiska noder* i L tar bort alla utom en. Så att t. ex. [2 2 1 7 7 7 2] övergår i [2 1 7 2],
   b) vid förekomst av flera identiska noder i L tar bort alla utom en (av de identiska). Så att t. ex. [2 2 1 7 7 7 2] övergår i [2 1 7].

3.10 Konstruera med primitiv rekursion en funktion som
   a) reverserar atomära listor, t. ex. [a b c] till [c b a]),
   b) reverserar godtyckliga listor, men ej dess noder, t. ex. [[a b] c] till [c [a b]]),

3.11 Konstruera en funktion som reverserar godtyckliga listor, och alla dess noder, t. ex. [[a b] c] till [c [b a]]).

3.12 Antag att L är en enkel lista med tal. Tillverka en listfunktion *ListMax*(L) som returnerar L:s största element. Om L är tom, skall **nil** returneras. Jämför med talfunktionen *Max* på sidan 38.

3.13 Heltalsfunktionen *Fib* på sidan 46 returnerar Fibonaccital. T. ex. är *Fib*(6) = 8. Konstruera nu – utan att använda funktionen *Fib* – en listfunktion *Fibonaccitalslista*(n) som returnerar en lista med alla Fibonaccital upp t.o.m. *Fib*(n). Så att t. ex.
Fibonaccitalslista(6) = [0 1 1 2 3 5 8].

3.14 *Binär sökning* efter ett visst element x i en sorterad lista L bygger på det faktum att om x är mindre än det första elementet i L:s högerhalva, så räcker det – om listan är sorterad från mindre till större – att leta efter x i vänsterhalvan av L, annars i högerhalvan. Efter varje *mindre*-jämförelse, halveras således sökrymden. Konstruera en funktion $BinärSök(x, L)$ som implementerar sådan sökning i en lista L som är sorterad från mindre till större.

3.15 En lista kan delas i tre lika långa delar om dess längd är delbar med 3. Annars måste de tre delarnas längder skilja sig åt med åtminstone en enhet. Jfr. med delning mitt itu på sidan 67. Konstruera en funktion *TreDela* som delar en lista i tre delar på nämnda sätt.

3.16 Den kända sorteringsalgoritmen *Quicksort* bygger vidare på den idé som *Mergesort* använder sig av, nämligen den att dela en lista i två delar, sortera delarna var och en för sig, och sedan sätta ihop de sorterade delarna. Men till skillnad från *Mergesort*, så ser *Quicksort* till att de redan sorterade listorna kan sättas ihop till en sorterad lista medelst enkel sammanfogning.

a) Hur ska två sorterade delar vara beskaffade om enkel sammanfogning av dem garanterat skall ge en sorterad lista som resultat?

b) Konstruera en delningsfunktion $Dela(L, m)$ som delar upp L i två delar, så att den ena delens element är mindre än m, och så att den andra delens element är större eller lika med m.

c) Implementera *Quicksort*!

3.17 I övning 3.6 konstruerades en funktion som utökar en lista genom att foga in ett element på godtycklig plats i den. Konstruera en funktion som givet en lista L och ett element x returnerar en lista med alla utökade listor

som erhålls när L utökas på alla sådana sätt med x. T.ex skall $[[x\ a\ b]\ \ [a\ x\ b]\ \ [a\ b\ x]]$ returneras om $L = [a\ b]$.

3.18 Funktionen *SlätaUt* på sidan 80 slätar ut all djupstruktur i en lista. Konstruera en mer nyanserad utslätare *SlätaUtHögst*(L, n) som slätar ut högst ett bestämt antal nivåer n.

Så att t. ex. *SlätaUtHögst*$([[1\ 2]\ \ 3\ \ [[4]]], 1) = [1\ 2\ 3\ [4]]$, men *SlätaUtHögst*$([[1\ 2]\ \ 3\ \ [[4]]], 2) = [1\ 2\ 3\ 4]$.

3.19 Föreställ dig att du skall placera en mängd personer vid ena sidan av ett långbord. Vilka olika placeringslistor (permutationer) är möjliga? Ja, ja, det beror förstås på hur många personerna är. Är de t. ex. tre stycken, A, B och C, så finns det sex olika listor. Se nedanför där de olika permutationerna är paketerade i en tom lista.

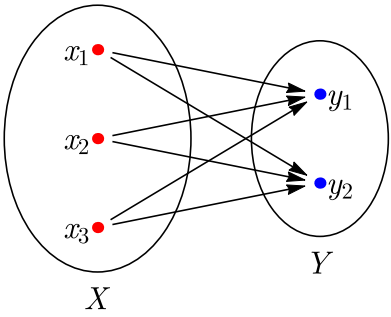
$$[[A\ B\ C]\ \ [A\ C\ B]\ \ [B\ A\ C]\ \ [B\ C\ A]\ \ [C\ A\ B]\ \ [C\ A\ B]]$$

Konstruera en funktion *Permutationer* som för en godtycklig mängd X returnerar alla permutationer av X paketerade på ovanstående sätt. T. ex. så att *Permutationer*$([A\ B\ C])$ ger sexelementslistan ovanför.

3.20 Med *Kartesiska produkten* $X \times Y$ mellan två listor X och Y menas listan av alla tvåelementslistor $[x\ y]$, där x tillhör X och y tillhör Y. Här är ett exempel:

$$[x_1\ x_2\ x_3] \times [y_1\ y_2] =$$
$$[[x_1\ y_1]\ \ [x_1\ y_2]\ \ [x_2\ y_1]\ \ [x_2\ y_2]\ \ [x_3\ y_1]\ \ [x_3\ y_2]]\,.$$

Med andra ord, $X \times Y$ kopplar samman varje element i X med varje element i Y, något som motiverar figuren nedanför.



Skriv en funktion som givet två godtyckliga listor X och Y returnerar den Kartesiska produkten $X \times Y$.

ANM. 3.3 Ordningen mellan de olika tvåelementslistorna i $X \times Y$ är oväsentlig. T. ex. skulle även följande lista vara en korrekt beskrivning av $[x_1\ x_2\ x_3] \times [y_1\ y_2]$.

$$[[x_1\ y_1]\ [x_2\ y_1]\ [x_3\ y_1]\ [x_1\ y_2]\ [x_2\ y_2]\ [x_3\ y_2]]\,.$$

Däremot får ordningen mellan de två elementen x, y i $X \times Y$:s enskilda tvåelementslistor $[x\ y]$ inte kastas om.

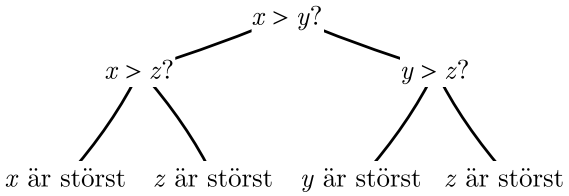
4

Träd

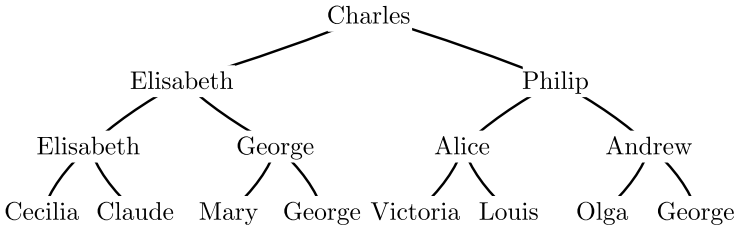
Introduktion

Ett träd växer ut och bildar grenar. Varje gren växer i sin tur ut och skaffar sig sina egna grenar (kvistar), som sedan växer ut och ....

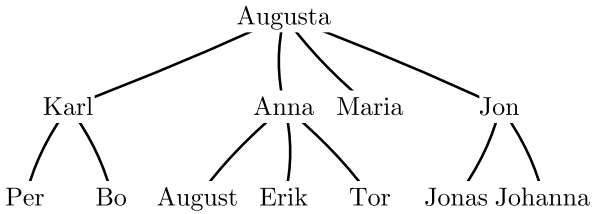
↬ EXEMPEL 4.1 Program förgrenar sig ofta likt träd.



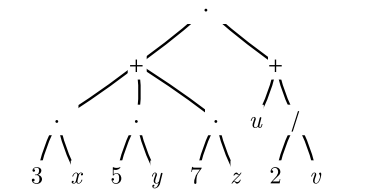
↬ EXEMPEL 4.2 (Antavla) Charles föräldrar, mor- och farföräldrar och deras föräldrar presenteras bäst med nedanstående släktträd. En antavlas förgreningar kommer alltid i *par*[†].



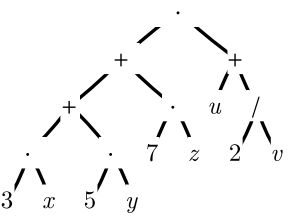
↬ EXEMPEL 4.3 (Stamtavla) Augusta tillsammans med barn och barnbarn bildar en familjedynasti – fastän i litet format – med Augusta som dynastins moder.



↬ EXEMPEL 4.4 Aritmetiska uttryck låter sig beskrivas som om de vore stam- eller antavlor.



$(3 \cdot x \ + \ 5 \cdot y \ + \ 7 \cdot z) \cdot (u \ + \ 2/v)$



$((3 \cdot x \ + \ 5 \cdot y) \ + \ 7 \cdot z) \cdot (u \ + \ 2/v)$

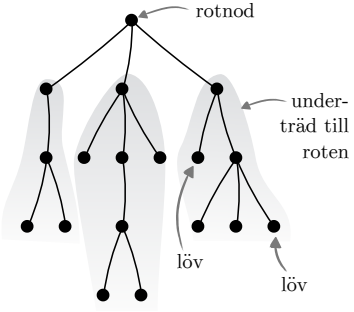
[†] Se mer om *binära* träd på sidan 101.

Träd

Du har just sett exempel på s.k. *rotade trädstrukturer*, dvs. datastrukturer av hierarkisk karaktär, med datats minsta beståndsdelar i *noderna*, och där själva hierarkin visar hur dessa minsta delar är relaterade till varandra. Fortsättningsvis kallar vi en sådan här datastruktur för ett *rotat träd*. Epitetet *rotat* kommer sig av att en av trädets noder är utvald för att vara en sorts urmoder för trädet. Den kallas *rotnod* eller *roten*. Här ”börjar” trädet. På motsvarande sätt ”slutar” trädet i *löven*.

DEFINITION Ett *rotat träd* består av

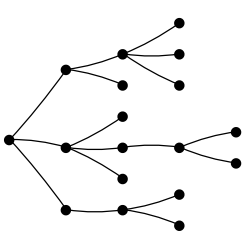
- en rotnod och inget mer eller av
- en rotnod som via utgående bågar är förbunden med andra träd – en båge för varje sådant s.k. *underträd*.

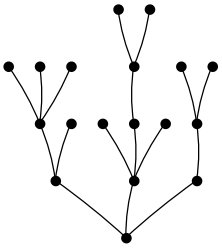


I fortsättningen kallar vi rotade träd kort och gott för *träd*.

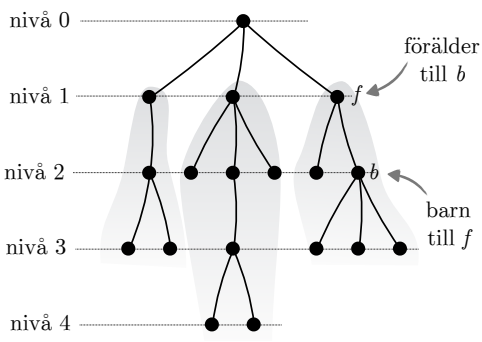
Om definitionens rekursiva karaktär Definitionen är rekursiv, eftersom den använder sig av begreppet träd för att beskriva begreppet träd. Detta kan tyckas gå i cirkel. Men eftersom de “andra träden” som definitionen talar om är åtminstone en nod fattigare än det “första trädet”, så faller definitionen slutligen tillbaka på det fall att trädet består av en enda nod – basfallet. Och då behöver definitionen inte anropa sig själv längre.

Trädens orientering Det är tradition att rita träden “upp och ner”, dvs, så att de förgrenar sig nedåt – precis som rotsystemet till ett träd. Men inget hindrar dig från att rita träden med andra orienteringar, bara det framgår vilken nod som är tänkt att vara rotnod.





Nivåer och släktskapsrelationer



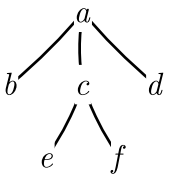
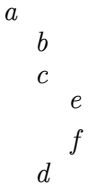
I analogi med ett släktträds olika generationer har ett träd olika *nivåer*. En nod i ett träd sägs vara *förälder* till de noder som den förgrenar sig till i nivån strax under. De senare noderna kallas i sin tur för barn till sin föräldrernod.

Underträd Ett *underträd* T' i ett givet träd T är ett träd som bildas genom att låta en nod i T bilda T':s rotnod, och sedan låta nämnda nods barn, barnbarn osv. ..., ända ner till löven vara med i T'. Speciellt kommer T att vara ett underträd i sig själv (det största underträdet). Och T:s löv bildar de minsta underträden. I figuren ovanför är tre speciella underträd markerade – de som är direktkopplade till roten.

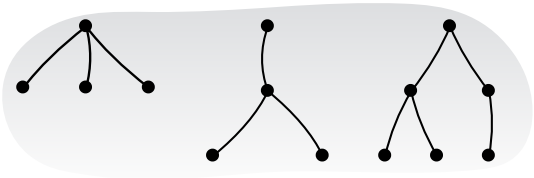
Ordnade respektive oordnade träd Om vi fäster betydelse vid den inbördes ordningen mellan barnen till en nod sägs trädet vara *ordnat*, annars *oordnat*. Alla

våra exempelträd utom det tredje är ordnade. Notera t. ex. hur ordningen mellan barnen i EXEMPEL 4.4 är av största betydelse. Kastar man om ordningen mellan 1 och u, så kommer nämligen $2/v$ att ändras till $v/2$ i det aritmetiska uttrycket som då får en helt annan innebörd.

Ett träds olika ansikten Ett träd kan representeras på andra sätt än med en grafisk bild av ett "riktigt träd". T. ex. med *indragna rader* eller med en *lista*. Den senare representationen kommer du att se mer av i fortsättningen.

Graf	Indragna rader	Lista
		[a [b] [c [e] [f]] [d]]

Skog Ibland har man anledning att betrakta flera träd tillsammans som om de bildade en enhet, vilken då kallas – just det – för en skog.



En skog med tre träd

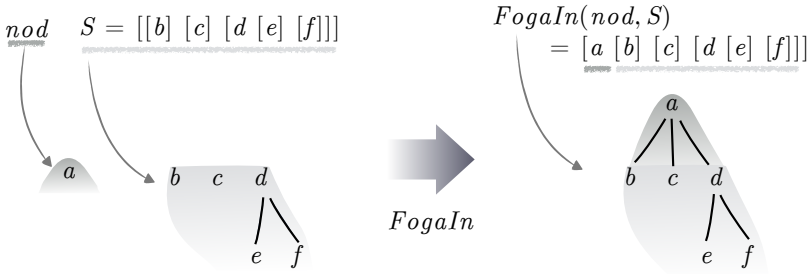
ANM. 4.1 I listmodellen representeras en skog av en lista innehållande noll eller flera trädlistor.

tom skog = [ ]
icketom skog = [trädlista trädlista ... trädlista]

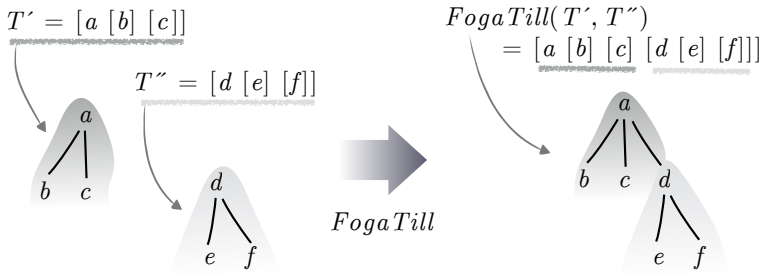
Trädmanipulerande algoritmer

Algoritmer manipulerar alltid helheten genom att manipulera delar av den. Om helheten kan delas sönder i delar som liknar helheten, så är det naturligt att manipulera helheten med rekursiva algoritmer. Träd kan delas sönder i delar som själva är träd. Träd bör därför manipuleras med rekursiva algoritmer. Träd kan representeras med rekursiva listor, och rekursiva listor kan manipuleras med listfunktioner. Listfunktioner utgör därmed algoritmiska verktyg för att manipulera träd.

Definitionen av ett träd (se sidan 93) beskriver hur ett träd tillverkas av en rotnod och ett antal enklare träd. Definitionen kan implementeras i listmodellen med hjälp av *FogaIn*, efter att de enklare träden har samlats ihop till en skog S.



Även funktionen FogaTill kan användas för trädtillverkning. Då utgår man från *två träd*:



Notera att den senare ihopsättaren tar emot träd och lämnar ut träd, men att den första ihopsättarens input och output är av skilda slag – en atom och en skog kommer in, ett träd kommer ut.

Vilken av de två ihopsättningarna som är lämpligast får bedömas från fall till fall, men enligt min erfarenhet är den första att föredra i de flesta fall.

↬ EXEMPEL 4.5 (En lövsamlare) Vår första trädfunktion handlar om ett träds löv. Vi ska konstruera en funktion som för ett godtyckligt träd returnerar dess löv paketerade inuti en lista.

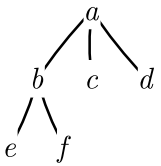
För det minsta trädet (se sid. 93) är roten det enda lövet.

För ett större träd hittar man löven längre ner i trädet, närmare bestämt i den icketomma skog som roten är direktkopplad till. Tack och lov är det lätt att samla löv ur en icketom skog. Man behöver bara foga samman löven ur skogens första träd med löven ur resten av skogen. På detta konstaterande kan vi bygga en rekursiv fläta mellan en trädfunktion och en skogfunktion:

En rekursiv fläta.

$$
\begin{aligned}
&\mathit{LövUrTräd}([\mathit{rot}]) = [\mathit{rot}] \\
&\mathit{LövUrTräd}(\left[\mathit{rot} \,\middle|\, \mathit{skog}\right]) = \mathit{LövUr}(\mathit{skog}) \\
&\mathit{LövUr}([\,]) = [\,] \\
&\mathit{LövUr}(\left[T \,\middle|\, S\right]) = \mathit{FogaSamman}(\mathit{LövUrTräd}(T), \mathit{LövUr}(S))
\end{aligned}
$$

En provkörning på trädet nedanför illustrerar det rekursiva flätandet.



$FS =$
FogaSamman

$$
\begin{aligned}
&\mathit{LövUrTräd}([a\ [b\,[e]\,[f]]\ [c]\,[d]]) \\
&= \mathit{LövUr}([[b\,[e]\,[f]]\ [c]\,[d]]) \\
&= \mathit{FS}(\mathit{LövUrTräd}([b\,[e]\,[f]]), \mathit{LövUr}([[c]\,[d]])) \\
&= \mathit{FS}(\mathit{LövUr}([[e]\,[f]]), \mathit{LövUr}([[c]\,[d]])) \\
&= \mathit{FS}(\mathit{FS}(\mathit{LövUrTräd}([e]), \mathit{LövUr}([[f]])), \\
&\qquad \mathit{FS}(\mathit{LövUrTräd}([c]), \mathit{LövUr}([[d]]))) \\
&= \mathit{FS}(\mathit{FS}([e], \mathit{LövUr}([[f]])), \\
&\qquad \mathit{FS}([c], \mathit{LövUr}([[d]]))) \\
&= \mathit{FS}(\mathit{FS}([e], \mathit{FS}(\mathit{LövUrTräd}([f]), \mathit{LövUr}([\,]))), \\
&\qquad \mathit{FS}([c], \mathit{FS}(\mathit{LövUrTräd}([d]), \mathit{LövUr}([\,])))) \\
&= \mathit{FS}(\mathit{FS}([e], \mathit{FS}([f], [\,])), \\
&\qquad \mathit{FS}([c], \mathit{FS}([d], [\,]))) \\
&= \mathit{FS}(\mathit{FS}([e], [f]), \\
&\qquad \mathit{FS}([c], [d])) \\
&= \mathit{FS}([e\ f], [c\ d]) \\
&= [e\ f\ c\ d]
\end{aligned}
$$

ANM. 4.2 Läge märke till att skogfunktionen *LövUr* ovanför tar en lista med träd som input och manipulerar varje träd på ett och samma sätt, nämligen paketerar dess löv i en lista. Sådan listmanipulering kan man göra med hjälp av funktionen *Map* (Se sidan 75). Dock måste man avsluta med att avlägsna djupstrukturen med hjälp av *SlätaUt*. Det följer att den rekursiva flätan i exemplet ovanför kan ersättas av

$$
\begin{aligned}
&\mathit{LövUrTräd}([rot]) = [rot] \\
&\mathit{LövUrTräd}(\left[rot \,\middle|\, skog\right]) = \mathit{SlätaUt}(\mathit{Map}(\mathit{LövUrTräd}, skog))
\end{aligned}
$$

Att traversera ett träd

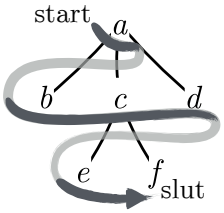
Många trädmanipulerande algoritmer behöver gå igenom (traversera) ett träds alla noder för att söka efter något eller för att ändra innehållet i noderna.

Två huvudlinjer faller sig därvid naturliga:

- Att gå igenom trädet i sidled. bedden först
- Att gå igenom trädet i djupled. djupet först

Bredden först Här tar man noderna nivåvis enligt figuren.

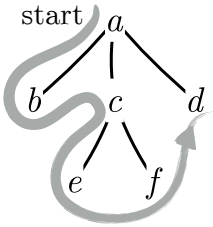
Detta kan synas enkelt. Men eftersom en sådan genomgång inte följer trädens underträdstruktur, som den representeras i trädens listor, så blir denna traversering relativt intrikat – i jämförelse med ”djupet först”. Se övning 4.16 på sidan 108.



Tre olika varianter av djupet först När du läser en bok, så går du igenom ett träd. Bokträdets struktur presenteras vanligtvis i en innehållsförteckning. Om du läser boken från början till slut – boktiteln först, sedan bokens kapitel i tur och ordning, och för varje kapitel, kapitelrubriken först och sedan själva kapiteltexten med eventuella underkapitel i turordning, och för varje underkapitel dess rubrik först osv., så går du igenom boken i s.k. *preordning*.

DEFINITION Att gå igenom ett träd i *preordning* innebär att

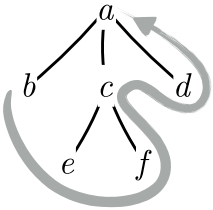
1. först besöka rotnoden,
2. sedan i *preordning* gå igenom underträden i turordning från vänster.



Läser du däremot bokens kapitel innan du läser dess titel, och kapiteltexterna före kapitelrubrikerna osv., så läser du i *postordning*.

DEFINITION Att gå igenom ett träd i *postordning* innebär att

1. först i *postordning* gå igenom underträden i turordning från vänster,
2. sist besöka rotnoden.

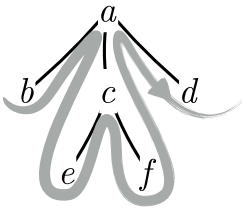


Eller läser du kanske i *inordning*?

Inordning används i huvudsak för binära träd.

DEFINITION Att gå igenom ett träd i *inordning* innebär att

1. i *inordning* gå igenom underträden i turordning från vänster,
2. besöka rotnoden däremellan.



ANM. 4.3 Lägg märke till de tre ”djupet först”-traverseringarnas *rekursiva* karaktär. T. ex. består en *preordningstraversering* av ett rotbesök följt av enklare träds *preordningstraverseringar* – i rotens underträd. Dessa i sin tur består av rotbesök följt av ännu enklare träds *preordningstraverseringar*. Och det hela bottnar med de allra enklaste *preordningstraverseringarna* – rotbesök allena – när man kommer till löven.

ANM. 4.4 Tronföljden i en dynasti överensstämmer vanligtvis med dynastiträdets preordning.

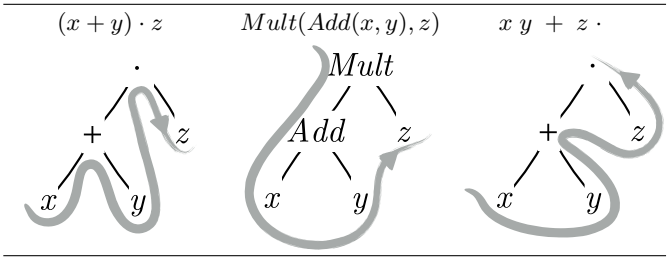
↬ EXEMPEL 4.6 (Funktionen *PreOrdna*.) Om man avlägsnar djupstrukturen i *listmodellen* så får man en utslätad lista med trädets noder i preordning. Härav följande preordningsfunktion.

$$PreOrdna(träd) = SlätaUt(träd)$$

↬ EXEMPEL 4.7 *PostOrdna* kan vi konstruera med hjälp av *Map* på motsvarande sätt som vi konstruerade lövsamlarfunktionen *LövUrTräd* i ANM. 4.2.

$$PostOrdna([rot]) = [rot]$$
$$PostOrdna(\left[rot \,\middle|\, skog\right]) = FogaTill(SlätaUt(Map(PostOrdna, skog)), rot)$$

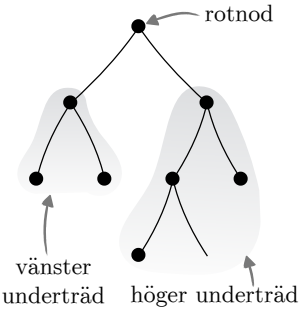
ANM. 4.5 Vid inordningsgenomgång av trädet för ett aritmetiskt uttryck räknas noderna upp i "rätt" ordning. Vid preordnings- och postordningsgenomgång kan man tycka att noderna kommer huller om buller. Men i själva verket är preordningen precis den som den funktionella notationen använder sig av. Och postordningen är RPN-fickräknarnas inmatningsmode.



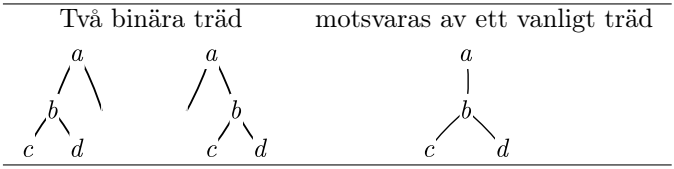
Binära träd

DEFINITION Ett *binärt träd* består av

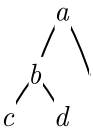
- ingen nod alls (det är tomt) eller av
- en rotnod som via utgående bågar är förbunden med två andra binära träd – *vänster* och *höger* underträd.



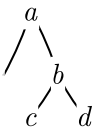
ANM. 4.6 Enligt definitionen har varje nod x i ett binärt träd *exakt* två underträd kopplade till sig. När någon av dessa är tomt, kan man i grafiska illustrationer representera ”det tomma” genom att från x dra en utgående båge som slutar i intet där ”intet” följaktligen representerar ett tomt underträd. Denna grafiska konvention har delvis tillämpats både i figuren som illustrerar definitionen av binära träd och i figuren nedanför.



Listmodellen för binära träd För att kunna bearbeta binära träd algoritmiskt, behöver vi listrepresentera dem. Här är ett förslag. Representera icketomt binärt träd med en lista av längd 3 – $[rot\ vä\ hö]$ – och tomt binärt träd med **nil**.

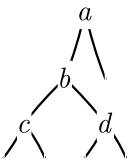


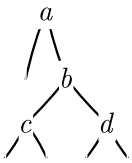
$[a\ \ [b\,[c]\,[d]]\ \ \mathbf{nil}]$



$[\,a\ \mathbf{nil}\ \ [b\,[c]\,[d]]\,]$

Men vänta nu. Som påpekats i ovanstående anmärkning har ju *alla* noder två underträd kopplade till sig – även löven, fastän deras underträd är tomma. Därför bör vi inte skriva $[löv]$, utan $[löv\ \mathbf{nil}\ \mathbf{nil}]$. Ty det algoritmiska maskineriet fungerar bäst om vi representerar alla tomma binärt träd på samma sätt.





$[a\ \ [b\ [c\ \mathbf{nil}\ \mathbf{nil}]\ [d\ \mathbf{nil}\ \mathbf{nil}]]\ \ \mathbf{nil}]\ [a\ \ \mathbf{nil}\ \ [b\ [c\ \mathbf{nil}\ \mathbf{nil}]\ [d\ \mathbf{nil}\ \mathbf{nil}]]]$

↬ EXEMPEL 4.8 Vi skall nu konstruera en listfunktion *InOrdna* som tar ett binärt träds listrepresentation som input, och som returnerar en (atomär) lista innehållande det binära trädets icketomma noder tagna i inordning. T. ex. skall vår funktion vid körning på de två binära träden i figuren ovanför returnera listorna

$$[c\ b\ d\ a] \quad \text{respektive} \quad [a\ c\ b\ d]\,.$$

Innan vi skrider till verket med tillverkningen av *InOrdna*, notera att dess input skall vara en tom lista eller en lista av typ $[rot\ v\ h]$ vars tre element i tur och ordning representerar roten, vänster underträd och höger underträd.

Att inordna går som bekant ut på att först ta vänster underträd (i inordning), sedan roten och sist höger underträd (i inordning). Så om vi redan har inordnat nämnda underträd, dvs. om vi redan har två atomära listor $InOrdna(v)$ och $InOrdna(h)$ med respektive underträds noder tagna i inordning, ja då är det inte särskilt mycket som återstår att göra. Vi behöver bara foga ihop alltsammans på lämpligt sätt:

$$\begin{aligned}
&InOrdna([\,]) = [\,]\\
&InOrdna([rot\ v\ h]) =\\
&\qquad FogaSamman(InOrdna(v), \left[rot \,\middle|\, InOrdna(h)\right])
\end{aligned}$$

PROVKÖRNINGAR:

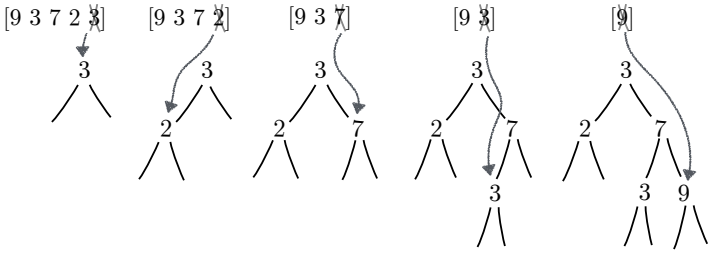
$$\begin{aligned}
InOrdna([[b\ [\,]\ [\,]]) &= FogaSamman(InOrdna([\,]), \left[b \,\middle|\, InOrdna([\,])\right])\\
&= FogaSamman([\,], [b\ [\,]])\\
&= FogaSamman([\,], [b])\\
&= [b]
\end{aligned}$$

$$\begin{aligned}
&InOrdna([c\ \ [d\ [\,]\ [\,]]\ \ [\,]])\\
&\qquad = FogaSamman(InOrdna([d\ [\,]\ [\,]]), \left[c \,\middle|\, InOrdna([\,])\right])\\
&\qquad = FogaSamman([d], [c\ [\,]])\\
&\qquad = FogaSamman([d], [c])\\
&\qquad = [d\ c]
\end{aligned}$$

$$
\begin{aligned}
&InOrdna([a\ \ [b\ [\,]\ [\,]]\ \ [c\ \ [d\ [\,]\ [\,]]\ \ [\,]]])\\
&\quad = FogaSamman([InOrdna([b\ [\,]\ [\,]]),\\
&\qquad\qquad \big[a \big| InOrdna([c\ \ [d\ [\,]\ [\,]]\ \ [\,]])\big])\\
&\quad = FogaSamman([b], \big[a \big|\ [d\ c]\big])\\
&\quad = FogaSamman([b], [a\ d\ c])\\
&\quad = [b\ a\ d\ c]
\end{aligned}
$$

↬ EXEMPEL 4.9 (Att sortera en lista med hjälp av ett binärt träd) Om man har en lista av tal som man vill sortera i storleksordning, så kan man göra det genom att först bygga ett binärt träd av listans tal, och sedan gå igenom det binära trädet i inordning. Det enda som man behöver tänka på är att bygga trädet på lämpligt sätt. Här är en illustration:

Betrakta listan [9 3 7 2 3]. Ta dess tal ett i sänder (börja t. ex. från slutet) och placera in dem på följande sätt i ett växande binärt träd:

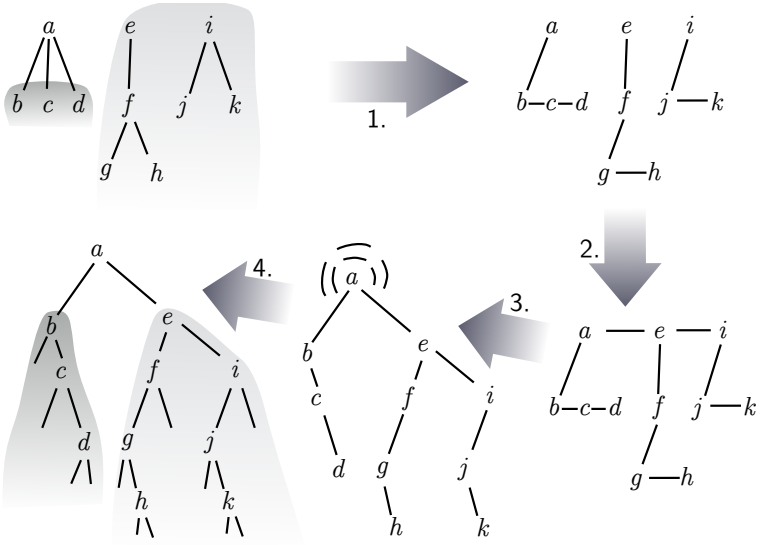


Successiv konstruktion av ett binärt träd vars inordningsgenomgång räknar upp talen i storleksordning.

Trädkonstruktionens idé är följande: När ett tal (från listan) skall läggas in i trädet gäller det att tillfoga en nod. Var i trädet skall noden tillfogas? SVAR: I vänster underträd ifall talet är mindre än rotnodens tal och i höger annars. Sedan upprepas detta vägval i varje underträd som man befinner sig i tills man är i ett löv. Då fogas talet in som vänsterbarn till lövet ifall talet är mindre än lövets tal, och högerbarn annars. Se vidare övning 4.18 på sidan 109.

ANM. 4.7 Om den givna listan redan är sorterad, kommer det binära trädet att få en extrem form. Vilken? Och vilken form får det binära trädet om listan är ”extremt osorterad”?

Från en skog av träd till ett binärt träd Varje skog av vanliga träd kan transformeras till ett binärt träd som nedan. Och detta på ett reversibelt sätt. Dvs. man kan gå från skogen till det binära trädet och sedan tillbaka igen, utan att någonting hos skogen går förlorat. (Vad gäller den tillbakagående algoritmen, se övning 4.19 på sidan 109.)

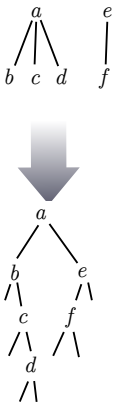


1. Varje yngre syskon frigörs från sin förälder och knyts istället till sitt äldre syskon. 2. Varje yngre träds rotnod förbinds med sitt närmast äldre träds dito. (Äldsta trädet finns längst till vänster.) 3. Håll i äldste roten och skaka! 4. Förbered för listrepresentationen.

En enkel listfunktion kan utföra transformationen.

$$\mathit{SkogTillBinär}([\,]) = [\,]$$
$$\mathit{SkogTillBinär}([[\mathit{rot}\,|\,S_1]\;|\;S_2]) = [\,\mathit{rot}\;\;\mathit{SkogTillBinär}(S_1)\;\;\mathit{SkogTillBinär}(S_2)\,]$$

PROVKÖRNING:



$$
\begin{aligned}
&STB([[a\ [b]\ [c]\ [d]]\ [e\ [f]]]) \\
&= [a\ STB([[b]\ [c]\ [d]])\ STB([[e\ [f]]])] \\
&= [a\ STB([[b]\ [c]\ [d]])\ [e\ STB([[f]])\ STB([\,])]] \\
&= [a\ STB([[b]\ [c]\ [d]])\ [e\ [f\ STB([\,])\ STB([\,])]\ STB([\,])]] \\
&= [a\ STB([[b]\ [c]\ [d]])\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]] \\
&= [a\ [b\ STB([\,])\ STB([[c]\ [d]])]\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]] \\
&= [a\ [b\ \mathbf{nil}\ STB([[c]\ [d]])]\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]] \\
&= [a\ [b\ \mathbf{nil}\ [c\ STB([\,])\ STB([[d]])]]\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]] \\
&= [a\ [b\ \mathbf{nil}\ [c\ \mathbf{nil}\ STB([[d]])]]\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]] \\
&= [a\ [b\ \mathbf{nil}\ [c\ \mathbf{nil}\ [d\ STB([\,])\ STB([\,])]]]\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]] \\
&= [a\ [b\ \mathbf{nil}\ [c\ \mathbf{nil}\ [d\,\mathbf{nil}\ \mathbf{nil}]]]\ [e\ [f\ \mathbf{nil}\ \mathbf{nil}]\ \mathbf{nil}]]
\end{aligned}
$$

Övningar

4.1 Vilka ordnade träd kan man bilda med noderna a, b, c om a är rot?

4.2 Rita binära träd som representerar följande aritmetiska uttryck.

a) $7 \cdot \dfrac{x}{y+z}$ b) $\dfrac{(x-y)+(z \cdot u)}{v}$ c) $\dfrac{2 \cdot x + 3 \cdot (y \cdot z)}{5 \cdot x - x \cdot y}$

4.3 Rita alla träd som har totalt fyra noder utspridda på nivåerna[†] 0, 1, 2.

4.4 Presentera trädlistan $[a\ [b]\ [c\,[d]]]$

a) i modellen ”med indragna rader” b) i grafmodellen

4.5 Antag att listan T beskriver ett träd. Då beräknas antalet barn till trädets rot av *Längd*(*UtomFörsta*(T)).
Vad beräknas av *Minska*(*Längd*(*Första*(*UtomFörsta*(T))))?

4.6 *Horners metod* går ut på att transformera ett uttryck[§] på formen $a_0 + a_1x + ... + a_nx^n$ till en form (*Horners form*) som

[†] Se sidan 94. [§] Sådana uttryck kallas för *polynom*. T. ex. är $2 + 3x + 5x^2$ ett polynom som omskrivet till Horners form blir lika med $2 + (3 + 5x)x$.

innehåller färre multiplikationer än det ursprungliga uttrycket. T. ex. blir $a_0 + a_1x + a_2x^2$ och $a_0 + a_1x + a_2x^2 + a_3x^3$ lika med $a_0 + (a_1 + a_2x)x$ resp. $a_0 + (a_1 + (a_2 + a_3x)x)x$, när de skrivs om till Horners form.

a) Rita ett binärt träd som representerar uttrycket $a_0 + (a_1 + (a_2 + a_3x)x)x$.

b) Konstruera en sammansatt funktion med *Add* och *Mult* som beräknar $a_0 + (a_1 + (a_2 + a_3x)x)x$.

c) Konstruera en listfunktion *Horner* som – givet en lista av tal $[a_0\ a_1\ \ldots\ a_n]$ och en obekant x – returnerar Horners form av $a_0 + a_1x + ... + a_nx^n$. T. ex. skall $Horner([2\ 3\ 5], x)$ ge $2 + (3 + 5x)x$.

4.7 Rita ett binärt träd som när det gås igenom i inordning representerar det aritmetiska uttrycket

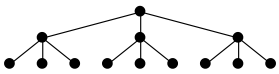
$$3^{2^x} \cdot 5^{3^{2^x} \cdot 5^{2^x}}$$

4.8 Tillverka en funktion $BytLövMotTräd(T, x, B)$ som i ett binärt träd T byter ut varje x-förekomst mot ett binärt träd B, ifall x är ett löv i T.

4.9 Med ett träds *djup* avses antalet nivåer i trädet. Skriv en funktion som för ett godtyckligt träd returnerar dess djup.

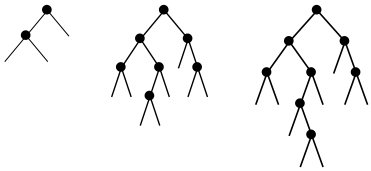
4.10 Skriv en funktion som för ett godtyckligt *binärt* träd returnerar dess djup.

4.11 Ett träd T sägs vara *k-ställigt* och *fullständigt*, om varje förälder i T har k stycken barn, och om föräldrar på samma nivå är rötter till lika djupa underträd.



Konstruera en funktion som för godtyckliga T och k besvarar frågan ”Är T ett fullständigt k-ställigt träd?”.

4.12 Låt oss kalla ett binärt träd T för *balanserat*, om det för varje nod i T skiljer högst en nivå mellan trädjupen hos nodens vänstra och högra underträd. (Se nedanför.) Konstruera en funktion som undersöker om ett binärt träd är balanserat.



Två är balanserade och ett (vilket?) är obalanserat.

4.13 Betrakta en godtycklig skog. Varje träd i skogen har som du vet en rot. Skriv en funktion *Rötter* som returnerar en lista innehållande rötterna till skogens alla träd.

4.14 Detta trädproblem handlar om syskonskaror i ett godtyckligt träd $T = \big[rot \,\big|\, skog\big]$. T. ex. bildar de noder som har rot som förälder en syskonskara som ges av *Rötter*($skog$), där funktionen *Rötter* återfinns ovanför. Men en syskonskara kan förstås även finnas längre ner i T. Din uppgift nu är att konstruera en funktion *SyskoniTräd*(T, x) som för ett godtyckligt x returnerar en lista av alla syskonskaror inuti T i vilka x ingår.

4.15 Tillverka en funktion *NoderPåNivå*(n, T) som, för ett godtyckligt naturligt tal n och ett godtyckligt träd T, returnerar en lista med de av T:s noder som ligger på nivå n.

4.16 Använd *NoderPåNivå*(n, T) ovanför, för att tillverka en funktion som returnerar en lista med T:s noder i den ordning som de påträffas vid ”bredden först”-traversering[†].

4.17 Som bekant är det i första hand det förstfödda barnet som ärver tronen i en dynasti. Skriva en trädfunktion som returnerar hela tronföljden för ett godtyckligt träd, dvs. som returnerar en lista av trädets rot följd av rotens förstfödda barn, det förstfödda barnets förstfödda barn, osv..

[†] Se sidan 99

4.18 Konstruera en funktion som utgående ifrån en enkel tallista bildar ett binärt träd vars inordningsgenomgång räknar upp talen i storleksordning. (Se EXEMPEL 4.9 på sidan 104.)

4.19 Antag att T är det binära träd som blev resultatet av transformationen av en skog S enligt funktionen *FrånSkogTillBinär* på sidan 105. Konstruera en funktion som transformerar tillbaka från det binära trädet T till skogen S.

4.20 Skriv en funktion som evaluerar det binära trädet för ett aritmetisk uttryck byggd av addition, subtraktion och multiplikation av naturliga tal.

5
Gödelnumrering

En förutsättning för att något skall kunna behandlas med hjälp av dator, är att detta något kan aritmetiseras, dvs. beskrivas med tal. Vi skall här presentera en metod – Gödelnumrering – som på sätt och vis är aritmetisering in absurdum. Med denna metod kan man exempelvis – i princip[†] – beskriva innehållet ord för ord i Bibeln med hjälp av ett enda naturligt tal. Att så mycket information kan ”bo inuti” ett tal kan förefalla mystiskt, men beror faktiskt bara på det enkla faktum att de naturliga talen är oändligt många, och att det är viss ordning och reda dem emellan. Gödelnumrering utnyttjar en fundamental egenskap hos de naturliga talen – den *entydiga primtalsfaktoriseringen*.

Primtalen som byggstenar

Primtalen fungerar som en sorts byggstenar för de naturliga talen. Man kan nämligen åstadkomma varje naturligt tal som är större än 1 och som inte själv är primtal, genom att multiplicera ihop lämpliga primtal med varandra.

Ett sådant x kallas *sammansatt*.

Antag nämligen att x är ickeprima och större än 1. Då kan x sönderdelas[§] i faktorer, säg $x = y \cdot z$, där y och z är större än 1 och mindre än x. Endera är y och z prima, och då är primtalsfaktoriseringen av x färdig. Eller

[†] Metoden ger alltför stora heltal för att vara användbar i praktiken, men har ändå en avsevärd principiell betydelse. [§] Se ANM. 1.4 på sidan 16 om sammansatta tal.

> så är någon av dem ickeprima, och kan sönderdelas, säg $z = u \cdot v$. Härav $x = y \cdot u \cdot v$. Endera är y, u, v prima varvid primtalsfaktoriseringen av x är färdig. Eller så är de inte prima alla tre, och då kan någon primtalsfaktoriseras ....
>
> Nu kan ju inte den här sönderdelningen hålla på hur länge som helst, dvs. x:s faktorer kan inte bli hur många som helst, ty då skulle deras produkt växa över alla gränser eftersom varje faktor är större än 1. Speciellt över det givna talet x som produkten skulle vara lika med. Alltså, förr eller senare går ingen av faktorerna att sönderdela ytterligare. Då är primtalsfaktoriseringen av x ett faktum.

Att generera naturliga tal genom att multiplicera ihop primtal kan jämföras med ett i viss mening enklare sätt att generera naturliga tal. Det att addera samman ettor. Ettor är additionens primtal i meningen att de ej kan sönderdelas som summor av mindre naturliga tal. Så här genereras exempelvis talet 120 på de två olika sätten:

$$
\begin{aligned}
120 = {} & 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
& \qquad\qquad\qquad +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
\end{aligned}
$$

$$120 = 2 \cdot 3 \cdot 3 \cdot 5$$

Vid ettsummering behöver man bara en sorts byggsten (talet ett), men man måste använda många byggstenar varje gång. Vid primtalsfaktorisering måste man ha tillgång till många olika sorts byggstenar (oändligt många primtal), men man behöver inte använda så många åt gången. Talet 120 genereras i första fallet av 120 ettor, och i andra fallet av 3 tvåor, 1 trea och 1 femma. Observera att talet vi genererar vid ettsummeringen beskriver (är lika med) antalet byggstenar. I nästa

avsnitt skall vi driva tesen att talet vi genererar vid primtalsfaktoriseringen också beskriver antalen byggstenar i en viss mening – t. ex. att 120 är en sorts beskrivning av [3 1 1], och omvänt att listan beskriver talet 120.

Gödeltal De naturliga talen används vanligen till att räkna antal – ettsummeringen är grunden för detta självklara bruk. I primtalsfaktoriseringen finns hemligheten bakom en annorlunda användning. Vi ger en enkel illustration:

> Om man använder de tre första primtalen 2, 3 och 5 som multiplikativa byggstenar, så kan man bygga vissa naturliga tal, t. ex.
>
> $$2 \cdot 3 \cdot 5 = 30$$
> $$2 \cdot 5^2 = 50$$
> $$2 \cdot 3 \cdot 3 \cdot 5 = 120$$

Kort sagt, genom att multiplicera ihop ett visst antal tvåor, ett visst antal treor och ett visst antal femmor, åstadkommer man vissa tal – som vi kallar för *Gödeltal*[†]. Sådan talproduktion kan förstås beskrivas i termer av $(input, output)$ på följande sätt:

input	[1 1 1]	[1 0 2]	[2 1 1]	[*x y z*]	...
output	30	50	60	$2^x \cdot 3^y \cdot 5^z$	...

Vi har här att göra med en funktion som tar emot en lista av *tre* tal och returnerar *ett* tal – listans Gödeltal. Funktionen har den viktiga egenskapen att den kan "köras baklänges" – den är *reversibel*. Och detta beror på att primtalsfaktorisering är entydig. Tar vi exempelvis talet 30, så är det så att 30 inte kan faktoriseras i andra primtal än just 1 tvåa, 1 trea och 1 femma. Den här möjligheten att gå från listan [1 1 1] till Gödeltalet 30 – via primtalsmultiplikation – och sedan från Gödeltalet 30 tillbaka till listan [1 1 1] – via primtalsfaktorisering – innebär att Gödeltalet 30 kan ses som ett namn eller en kod för listan [1 1 1]. För att på motsvarande sätt beräkna Gödeltal för heltalslistor som är längre än 3, använder man flera primtal

[†] Kurt Gödel introducerade s.k. *Gödeltal* i beviset av sin ofullständighetssats.

än 3, men förfar i övrigt på samma sätt.

$$\mathit{gödel}([x_0\ x_1\ x_2\ \ldots\ x_n]) = 2^{x_0} \cdot 3^{x_1} \cdot 5^{x_2} \cdot \ldots \cdot {p_n}^{x_n} \qquad (1)$$

där p_n är det n:te primtalet[†].

Givetvis kan man ge en rekursiv konstruktion av (1):

$$\begin{aligned}
&\mathit{gödel}([\,]) = 1\\
&\mathit{gödel}(L) = \mathit{Mult}(\mathit{gödel}(\mathit{UtomSista}(L)),\\
&\qquad\qquad \mathit{Pot}(\mathit{Primtal}(\mathit{Längd}(\mathit{UtomSista}(L))),\\
&\qquad\qquad\quad \mathit{Sista}(L)))
\end{aligned}$$

Vid hantering av olika långa listor kan ett återskapningsproblem uppstå, nämligen att olika listor kan ha likadana Gödeltal. Följande exempel illustrerar vad som åsyftas:

L	$[1\ 1\ 1]$	$[1\ 1\ 1\ 0]$
$\mathit{gödel}(L)$	$2^1 \cdot 3^1 \cdot 5^1 = 30$	$2^1 \cdot 3^1 \cdot 5^1 \cdot 7^0 = 30$

En kodning för vilken skilda objekt kan ha en och samma kod, är inte reversibel, och kan knappast förtjäna epitetet *kodning*. Olika lösningar till det här problemet är tänkbara. Snart presenteras *en* sådan lösning.

En lista som innehåller andra tecken än heltal – exempelvis bokstäver – kan också ges ett Gödeltal, bara man först ser till att tecknen kodas till tal.

Ger man exempelvis A kodnumret 1, B kodnumret 2, osv., så översätts $ABBA$ först till $[1\ 2\ 2\ 1]$ som sedan på vanligt sätt tilldelas $\mathit{gödel}([1\ 2\ 2\ 1]) = 2^1 \cdot 3^2 \cdot 5^2 \cdot 7^1 = 3150$, vilket får vara $ABBA$:s Gödeltal.

Om man vill ge Gödeltal till listor som innehåller både tal och bokstäver, där de senare tas ur ett alfabet med säg 29 bokstäver, så kan man börja med att koda de 29 bokstäverna till heltal. Exempelvis så att A blir 1, B blir 2, osv.... Detta innebär att man reserverar talen $1, \ldots, 29$ för bokstäverna. Och då måste man förpassa talen $1, \ldots, 29$ till andra boplatser – t. ex. flytta alla tal (utom 0 som kan stanna kvar) 29 steg

[†] $p_0 = 2,\ p_1 = 3,\ p_2 = 5,\ \ldots$

framåt, dvs. så att ett tal x får kodnumret $x+29$. När sålunda tal och bokstäver har fått sina respektive kodnummer, så kan en lista bestående av både tal och bokstäver översättas till en ren tallista som sedan kan ges ett Gödeltal på vanligt sätt. Exempelvis skulle $ADA1$ först översättas till $[1\ 4\ 1\ 30]$, och sedan skulle man beräkna $\mathit{gödel}[1\ 4\ 1\ 30] = 2^1 \cdot 3^4 \cdot 5^1 \cdot 7^{30} = 182568656354607290511692316 90$.

Listor av listor kan också tillordnas Gödeltal. Ta exempelvis listan av alla böcker som redan är skrivna. Varje enskild bok kan ses som en lista av ord och varje ord kan ses som en lista av tecken. Om man först ger varje ord ett Gödeltal som ovan, så blir varje bok en lista av tal. Och en lista av tal kan ju ges ett Gödeltal. Därmed blir listan av böcker en lista av tal, som slutligen tillordnas sitt Gödeltal. Enkelt eller hur? Ett enda heltal kan således vara bärare av all den text som är skriven. Och med hjälp av detta heltal – textens Gödeltal – kan man i princip återskapa varje boks innehåll ord för ord. Tanken svindlar.

> Det finns ett problem förbundet med Gödelnumreringen av listor vars element själva kan vara listor. Ännu ett reversibilitetsproblem. Antag nämligen att man har ett Gödeltal i sin hand, och att man ur detta vill återskapa det som en gång är kodat. Då kan man få svårigheter att ur Gödeltalet läsa ut huruvida det var en lista av listor eller en lista av tal som var kodat. t. ex. är 16 Gödeltalet för listan [[2]] men också för listan [4].

En modifiering av Gödeltalsdefinitionen Vi har stött på två reversibilitetsproblem med Gödeltalen. Ett som berodde på förekomst av nollor i slutet av listorna, och ett som dök upp när vi ville koda såväl listor av tal som listor av listor. Båda dessa problem försvinner om man modifierar *gödel* till *Gödel* på följande sätt:

$$\mathit{Gödel}(\mathit{tal}) = 2^{\mathit{tal}}$$

$$\mathit{Gödel}([x_1\ x_2\ \ldots\ x_n]) = 3^{\mathit{Gödel}(x_1)} \cdot 5^{\mathit{Gödel}(x_2)} \cdot \ldots \cdot {p_n}^{\mathit{Gödel}(x_n)}$$

↬ EXEMPEL 5.1 Listorna $[1\ 1\ 1]$ och $[1\ 1\ 1\ 0]$ har likadana Gödeltal när man kodar med hjälp av *gödel* (Se förra sidan.),

men tilldelas skilda Gödeltal av *Gödel*. Det beror på att när man fogar en nolla till slutet av en lista så multipliceras listans Gödeltal i första fallet med 1, och i det senare fallet med ett primtal. Nedanför får du svart på vitt på hur det ser ut vid kodning med *Gödel*.

$$\begin{aligned}
\mathit{Gödel}([1\ 1\ 1]) &= 3^{\mathit{Gödel}(1)} \cdot 5^{\mathit{Gödel}(1)} \cdot 7^{\mathit{Gödel}(1)} \\
&= 3^{2^1} \cdot 5^{2^1} \cdot 7^{2^1} \\
&= 3^2 \cdot 5^2 \cdot 7^2 \\
&= 9 \cdot 25 \cdot 49 \\
&= 11025
\end{aligned}$$

$$\begin{aligned}
\mathit{Gödel}([1\ 1\ 1\ 0]) &= 3^{\mathit{Gödel}(1)} \cdot 5^{\mathit{Gödel}(1)} \cdot 7^{\mathit{Gödel}(1)} \cdot 11^{\mathit{Gödel}(0)} \\
&= 3^{2^1} \cdot 5^{2^1} \cdot 7^{2^1} \cdot 11^{2^0} \\
&= 3^2 \cdot 5^2 \cdot 7^2 \cdot 11^1 \\
&= 9 \cdot 25 \cdot 49 \cdot 11 \\
&= 11025 \cdot 11 \\
&= 121275
\end{aligned}$$

↬ EXEMPEL 5.2 Redan måttligt komplexa listor tilldelas tal av avsevärd storlek.

$$\begin{aligned}
\mathit{Gödel}([2\ \ [[1\ 0]\ 4]]) &= 3^{\mathit{Gödel}(2)} \cdot 5^{\mathit{Gödel}([[1\ 0]\ 4])} \\
&= 3^{2^2} \cdot 5^{(3^{\mathit{Gödel}([1\ 0])} \cdot\ 5^{\mathit{Gödel}(4)})} \\
&= 3^{2^2} \cdot 5^{(3^{3^{\mathit{Gödel}(1)} \cdot 5^{\mathit{Gödel}(0)}} \cdot\ 5^{2^4})} \\
&= 3^{2^2} \cdot 5^{(3^{3^{2^1} \cdot 5^{2^0}} \cdot\ 5^{16})} \\
&= 3^4 \cdot 5^{(3^{45} \cdot\ 5^{16})} \\
&= 81 \cdot 5^{295431270655083369864 3 \cdot 152587890625} \\
&= 81 \cdot 450792344139226333411102294921875 \\
&= 36514179875277333006299285888671875.
\end{aligned}$$

↬ EXEMPEL 5.3 (Program och Gödeltal) Eftersom ett program kan betraktas som en lista, kan varje program tilldelas ett unikt Gödeltal med hjälp av *Gödel*. Detta faktum kommer i efterföljande kapitel att visa sig vara en begränsande faktor för programutvecklare.

Övningar

5.1 Primtalsfaktorisera 924.

5.2 Är 924 ett Gödeltal? Enligt *gödel*? Enligt *Gödel*?

5.3 Koda listorna [[2]], [4], [2 0] och [0 [2 0]] med *Gödel*.

5.4 Ge strängen AHA ett Gödeltal.

5.5 Finns det listor som har följande Gödeltal enligt *Gödel*? Beräkna i förekommande fall nämnda listor
a) 48 b) $27 \cdot 5^{45}$

5.6 Ge en rekursiv konstruktion av *Gödel*. Låt dig inspireras av motsvarande konstruktion av *gödel* på sidan 113.

5.7 Beskriv informellt hur man kan ge Gödeltal till följande objekt, så att dessa kan återskapas (mer eller mindre exakt) utgående ifrån deras Gödeltal.
a) En melodi, t. ex. *Blinka lilla stjärna där*, given i en viss tonart b) ett foto c) en stumfilm
LEDNING: Börja med att ange hur objekten kan beskrivas med tallistor.

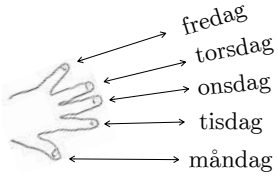
6

Olika stora oändligheter

Cantors idé

Vi skall här presentera en idé som idag är accepterad – bland matematiker – men som möttes med stor misstro när den framkastades i en teori om oändliga mängder av Georg Cantor i slutet av 1800-talet. En för vår del intressant konsekvens av Cantors teori är – som vi snart skall se – att programmen är färre än heltalsfunktionerna.

Ett vedertaget synsätt är att två ändliga mängder *innehåller lika många element* – eller är lika *mäktiga* – om man kan para ihop varje element i den ena mängden med ett element i den andra på ett sätt sådant att inget element blir över i någondera mängden. Man kallar en sådan ihopparning för en 1–1-*motsvarighet*.



Handens fingrar är lika många som veckans arbetsdagar.

1–1-*motsvarighet*

Cantor menade – i sin teori – att ovanstående synsätt kunde tillämpas även på oändliga mängder.

↬ EXEMPEL 6.1 Sålunda kunde Cantor hänvisa till nedanstående 1–1-motsvarighet

$$\begin{array}{cccccc} 1 & 2 & 3 & \dots & n & \dots \\ \updownarrow & \updownarrow & \updownarrow & \dots & \updownarrow & \dots \\ 100 & 200 & 300 & \dots & n \cdot 100 & \dots \end{array}$$

och hävda att hundratalen $100, 200, 300, \dots$ är lika många som de positiva naturliga talen $1, 2, 3, \dots$.

Nu protesterar väl kanske någon och menar att de förra talen (hundratalen) måste vara färre, eftersom de omfattar blott vart hundrade element av de senare, något som nedanstående ihopparning illustrerar:

$$\begin{array}{cccccccccccc} 1 & 2 & \dots & 99 & 100 & \dots & 199 & 200 & \dots & n \cdot 100 & \dots \\ & & & & \updownarrow & & & \updownarrow & & \updownarrow & \dots \\ & & & & 100 & & & 200 & & n \cdot 100 & \dots \end{array}$$

För övrigt – fortsätter den protesterande – kan man ju hävda att hundratalen är flera än de positiva naturliga talen, genom att t.ex. som nedanför para ihop vart tionde hundratal med de positiva naturliga talen:

$$\begin{array}{cccccccc} 100\ 200 \dots & 10 \cdot 100 & \dots & 20 \cdot 100 & \dots & 10n \cdot 100 & \dots \\ & \updownarrow & & \updownarrow & & \updownarrow & \dots \\ & 1 & & 2 & & n & \dots \end{array}$$

Du inser nog nu varför Cantors idé om oändliga mängders mäktighet hade svårt att bli accepterad. Det ligger nära till hands att döma ut hela idén. Att betrakta den som osund och hänvisa till de till synes motstridiga konsekvenserna här ovan.

Men man kan också säga ungefär så här: Om två personer A och B slåss, och A vinner ibland, B andra gånger, och om det också händer att det blir oavgjort, så finns det skäl att hävda att A och B är lika starka.

Låt oss anamma detta senare synsätt när det gäller oändliga mängders mäktighet, och därmed acceptera Cantors idé.

Vi säger således att mängden av hundratalen är lika mäktig som mängden av alla naturliga tal, eftersom det finns *något sätt* att para ihop de två mängdernas element så att inget element blir över i någondera mängden. Dvs. att det finns en 1–1-motsvarighet mellan mängderna. Att man även kan para ihop på andra sätt – så att det blir element över i den ena eller andra mängden – det bortser vi ifrån, eller också ser vi det som högst naturligt.

Uppräkneligt

En mängd som är lika mäktig som mängden av naturliga tal

$$0, 1, 2, \ldots$$

består alltså av element

$$a, b, c, \ldots$$

som kan paras ihop med de förra. Man brukar säga att en sådan mängd är *uppräknelig* eller *numrerbar* – en uttryckssätt som avser att leda tankarna till att man kan ”sätta nummerlappar” på mängdens element, och därmed tala om det 0:e, 1:a, 2:a elementet osv.

$$\begin{array}{cccc} 0 & 1 & 2 & \ldots \\ \updownarrow & \updownarrow & \updownarrow & \ldots \\ a & b & c & \ldots \end{array}$$

En ”numrering”.

Mängden av naturliga tal är förstås själv uppräknelig. Så är också varje oändlig delmängd av de naturliga talen, eftersom delmängdens element alltid kan numreras i storleksordning – som exempelvis hundratalsmängden nyss, eller mängden av primtal 2, 3, 5, ... Varje oändlig mängd vars enskilda element kan ges Gödeltal är också uppräknelig. Ty elementen i en sådan mängd kan ju numreras allt efter hur stora deras Gödeltal är – elementet med det minsta Gödeltalet först.

Några exempel på uppräkneliga mängder

↬ EXEMPEL 6.2 Mängden av alla par $[x\ y]$ av naturliga tal är uppräknelig, eftersom varje sådant par kan ges ett Gödeltal.

↬ EXEMPEL 6.3 Mängden av alla ickenegativa rationella tal x/y (bråktal) är uppräknelig, eftersom varje sådant tal kan representeras med $[x\ y]$.

↬ EXEMPEL 6.4 Alla tänkbara program-input – skrivna medelst tangentbordets teckenuppsättning eller i något annat alfabet – bildar en uppräknelig mängd. Ty varje input kan ses som en ändlig lista $[a_1\ \ldots\ a_n]$ av tecken ur det aktuella alfabetet, och kan således Gödelnumreras.

↬ EXEMPEL 6.5 Mängden av alla program-output är uppräknelig av samma skäl.

↬ EXEMPEL 6.6 Mängden av alla program – korrekta, intelligenta eller dumma, men också felaktiga – är uppräknelig, eftersom program är texter och har därmed Gödeltal. Se EXEMPEL 5.3 på sidan 116.

↬ EXEMPEL 6.7 Mängden av *alla* texter likaså. Romaner, dikter, tidningstexter, bloggar, reklamtexter, ja alla texter (korta eller långa) som redan är skrivna men även de som ännu inte är det. En till synes ofantligt stor mängd.

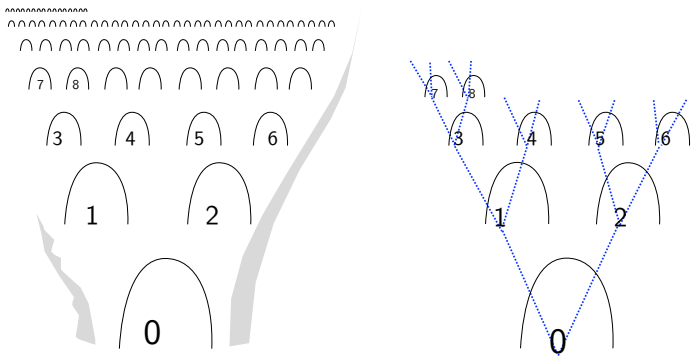
Hilberts hotell

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

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

Vidare har hotellet formen av ett binärt träd, med rum 0 i rotnoden och de övriga rummen i de övriga noderna – numrerade som i figuren nedan. De första gästerna anländer. Av någon oförklarlig anledning är de numrerade med nummer 1, nummer 2, osv. Hotellportieren Hilbert som uppmärksammar deras nummerlappar hälsar dem välkomna och säger.

”Ni skall få de bästa rummen, eftersom ni är de första gästerna. Den som har nummer n, var god intag rum n.”



Hilberts hotell

Rum 0 – tänkte han – kan alltid vara bra att ha kvar.

Just då kommer det ytterligare två gäster till det till synes fullbelagda hotellet. Hilbert som var van att formulera svåra problem, men även att snabbt lösa enklare dito, funderade en mycket kort stund. Sedan visste han.

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

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

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

Innan Hilbert hunnit slå sig till ro, dyker det upp ytterligare en samling gäster. Oändligt många. Och numrerade likt de första gästerna. På väg ut för att möta de nya gästerna vänder han tillbaka in i hotellet igen, för att till hotellets alla rum låta sända ut ännu ett meddelande.

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

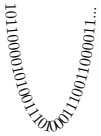
Därigenom blev alla rum med udda rumsnummer lediga. Och Hilbert kunde lugnt gå ut och hälsa de nya gästerna välkomna.

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

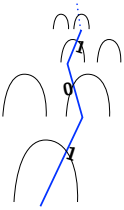
Innan Hilbert hunnit sätta sig ned – nöjd över att ha ordnat

rum åt alla gäster och ändå ha ett rum över – inträffar något mycket märkligt. Dagsljuset som nyss silade in genom hotellets vackra entrédörrar blev allt svagare. Utanför hotellet tornade något upp sig, som satte Hilberts tankar i brand. Han kände hur det hettade innanför pannloben. I nattliga diskussioner med Cantor, hisskonduktören, hade denne påtalat möjligheten att det till hotellet en dag skulle kunna komma så förfärligt oändligt många gäster att hotellet inte kunde ta hand dem alla. Att rummen icke skulle räcka trots att de var oändligt många. Med ett resonemang vars bärande idé var en diagonal hade Cantor kommit fram till detta. Först nu insåg Hilbert att Cantor hade haft rätt hela tiden. Just då viks entrédörrarna undan och in i vestibulen väller en enorm massa. En individ ur massan lyckas tränga sig fram till Hilbert och säger:

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



När den trugande gästen böjer huvudet i en blidkande gest ser Hilbert hur en sträng av nollor och ettor dinglar fram och tillbaka runt dennes hals. En snabb blick övertygar Hilbert om att varje individ i massan har en sådan sträng runt halsen. Han granskar den trugandes sträng igen så noga han kan. Och så vitt han kan bedöma så finns det inte något slut på strängen. Då ser han med ens allting klart för sig. Varje sådan sträng kan ses som en vägbeskrivning. En karta över en oändlig väg nerifrån vestibulen och uppåt i hotellets korridorer, våning efter våning, ända upp till takterassen! Den n:te siffrans värde anger åt vilket håll uppåt (0=vä och 1=hö) som man skall välja när man befinner sig på hotellets n:te våningsplan. Med denna idé i bakhuvudet griper Hilbert nu tag i kommunikationssystemets mikrofon och säger med lugn och säker stämma:



”Hjärtligt välkomna, jag förstår att Ni önskar varsitt rum. Det skall Ni få. Inte nog med det. Ni skall få de bästa rummen – de som finns på takterassen, och som därför har en fantastisk utsikt. Med vår accelererande hiss som för varje våning fördubblar hastigheten passerar Ni hotellets första våning på 1 sekund, nästa våning på en halv sekund osv., varför Ni efter totalt $1+1/2+1/4+1/8+\ldots = 2$ sekunder har nått Edra rum.

För att göra Er vistelse hos oss så personlig och säker som bara vi kan, har vi dessutom ordnat så att vår hisskonduktör Cantor tar dig till just ditt rum med hjälp av den sträng som du bär runt Din hals. Ingen annan sträng leder till ditt rum. Var så goda, tag plats i hissen."

Mängden av booleska funktioner

I anekdoten ovanför antyds det att det finns en större oändlighet än den som de naturliga talen beskriver. Närmare bestämt att mängden av alla oändliga bitsträngar skulle vara en sådan mängd. Vi ska strax bevisa att det är på det sättet. Men låt oss först konstatera att en oändlig bitsträng[†] kan betraktas som outputrad till en boolesk funktion definierad på de naturliga talen, och omvänt.

T.ex. att bitsträngen med 1:or i udda positioner, den med 1:or i jämna positioner, och den med 1:or i primtalspositionerna är outputrader till funktionerna

n	0	1	2	3	4	5	...
$Udda(n)$	0	1	0	1	0	1	...
$Jämn(n)$	1	0	1	0	1	0	...
$Prima(n)$	0	0	1	1	0	1	...

Eftersom mängden av oändliga bitsträngar och mängden av booleska funktioner kan paras ihop på det just beskrivna sättet, är de lika mäktiga. Vi bevisar nu att denna mäktighet är större än de naturliga talens mäktighet. Det vackra beviset går under namnet *Cantors diagonalbevis* och publicerades första gången 1891.

Cantors diagonalbevis

Cantors diagonalbevis

> Det naturliga sättet att bevisa att en mängd är större än en annan är att visa att varje försök att para ihop dess element med den andra mängdens element resulterar i att det blir element över i den förstnämnda mängden.

Antag nu att någon försöker numrera booleska funktioner med naturliga tal:

[†] Notera att varje oändlig väg i Hilberts hotell definierar en boolesk funktion och omvänt.

Ett försök att numrera booleska funktioner.

funktions-nummer	
0	en boolesk funktion
1	en annan boolesk funktion
2	ytterligare en boolesk funktion
⋮	⋮

För att göra det kommande resonemanget mer konkret, så kan vi tänka oss att de första tre funktionerna i numreringen är lika med *Udda*, *Jämn* och *Prima*:

funktions-nummer	input	0	1	2	3	4	5	...
0	output	0	1	0	1	0	1	...
1	output	1	0	1	0	1	0	...
2	output	0	0	1	1	0	1	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Med hjälp av de hypotetiskt numrerade funktionerna ska vi nu se att man kan bilda en funktion som inte finns med bland de förra. Betrakta diagonalen, men framförallt den *inverterade* diagonalen - dvs. varje 1:a i diagonalen ändrad till en 0:a och vice versa:

input	0	1	2	...
diagonalen	0	0	1	...
inverterade diagonalen	1	1	0	...

Den inverterade diagonalen definierar en boolesk funktion som omöjligen kan vara lika med någon av de från början numrerade funktionerna. Ty betraktas vilken som helst av dessa från början numrerade funktioner, säg den med nummer n, så är dess output hörande till inputvärdet n - p.g. av inverteringen - skild från den nya funktionens output för samma input.

Detta bevisar att varje försök att numrera booleska funktioner med naturliga tal lämnar åtminstone en boolesk funktion utanför. □

ANM. 6.1 De booleska funktionerna utgör således en mäktigare mängd än mängden av naturliga tal, och är ett exempel på en s.k. *överuppräknelig* mängd.

En intressant konsekvens av detta är, att det visar på förekomsten av booleska funktioner som inte kan beskrivas med ord[†].

> Om varje boolesk funktion kunde beskrivas med en text (dvs. av ord), skulle de booleska funktionerna kunna paras ihop med de texter som beskriver dem. Och då skulle nämnda funktioner inte vara flera än texterna. Men det är de ju, eftersom de är överuppräkneligt många, medan texterna är uppräkneliga. (EXEMPEL 6.7 på sidan 120.)

Det finns således heltalsfunktioner som inga program kan beräkna!

Funktioner som inte kan beskrivas med text kan förstås inte heller *beräknas* av program, eftersom program är texter, och ett program som beräknar en funktion följdaktligen skulle vara en text som beskriver funktionen ifråga.

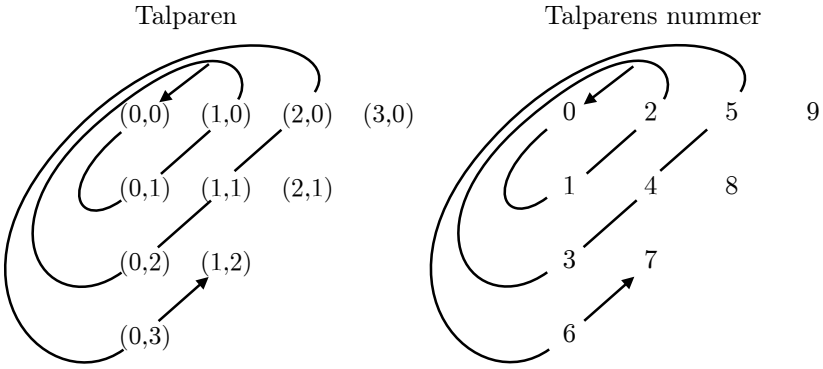
Övningar

6.1 Hitta på en *uppräkning*[§] av talen $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$.

6.2 Hitta på en uppräkning av de positiva decimaltal som har 6 eller färre decimaler. (0.1 och 3147526.136619 är två sådana tal.)

6.3 Nedanför visas ett klassiskt sätt att räkna upp alla ordnade par av naturliga tal. Konstruera en rekursiv funktion $Nummer(x, y)$ som returnerar (x, y):s nummer i denna uppräkning.
LEDNING Triangeltalen är involverade.

[†] "*There are indeed things that cannot be put into words.*" (Ludvig Wittgenstein) [§] Dvs. en 1–1-motsvarighet mellan mängden av naturliga tal och uppgiftens mängd.


Talparen
Talparens nummer
(0,0)
(1,0)
(2,0)
(3,0)
(0,1)
(1,1)
(2,1)
(0,2)
(1,2)
(0,3)
0
2
5
9
1
4
8
3
7
6

7

Stopproblemet

Att det finns heltalsfunktioner som inte kan beräknas med program, det har vi lyckats visa. (Se sidan 125.) Men inget enda exempel har vi presenterat ännu. Nu ska vi lätta på den förlåten.

Program och deras funktioner

När man intresserar sig för den funktion som ett program beräknar, bortser man från hur programmet är utformat. Det enda som har betydelse är programmets output i relation till dess input. Visserligen är det ofta helt oproblematiskt att tala om programmet och funktionen i samma andetag, men i detta kapitel är det av största vikt att man kan hålla isär de två begreppen. Observera också att det inte är ovanligt att två skilda program – dvs. program som är skrivna på olika sätt – beräknar en och samma funktion. Vidare kan det vara så att ett program av en eller annan anledning inte ger output för varje input.

> Den som har minsta erfarenhet av programmering, vet t. ex. hur lätt det är att koka ihop en oändlig slinga i sitt program, och därmed få ett program som ”försöker beräkna output, men aldrig lyckas”.

Om ett program – för ett visst input – inte förmår ge output, så gäller förstås samma sak för den funktion som programmet beräknar. Ofta säger man att programmet och dess funktion är *odefinierade* för ett sådant input.

↬ EXEMPEL 7.1 Programmet

$$NollStopp(x) = Om(Noll?(x), 1, NollStopp(x))$$

och dess funktion är odefinierade för alla input utom 0.

x	0	1	2	3	...
$NollStopp(x)$	1	–	–	–	...

En funktion som inte kan beräknas

Av ett programs determinism följer det att varje program beräknar en funktion. Men omvänt följer det inte från funktionsbegreppet (se sidan 22) att det till varje funktion måste finnas ett program som kan beräkna funktionen. I funktionsbegreppet finns nämligen inget krav på hur funktionens output skall beräknas eller att de alls måste gå att beräkna. I själva verket går det att definiera funktioner som inga program kan beräkna. Vi ska strax ge ett par exempel.

Som utgångspunkt för det första exemplet ligger det faktum (se sidan 120) att mängden av alla program är uppräknelig. Låt

$$P_0, P_1, P_2, \ldots$$

vara en uppräkning av nämnda mängd. Varje redan skrivet program såväl som varje ännu inte påtänkt program finns med i denna uppräkning.

Vi definierar nu en funktion *CANDI* med naturliga tal som input såväl som output, och visar sedan att inget av programmen $P_0, P_1, P_2, \ldots$ kan beräkna den.

$$CANDI(n) = \begin{cases} 1 + P_n(n), \text{ om } P_n(n) \text{ är definierad} \\ 1, \text{ om } P_n(n) \text{ är odefinierad} \end{cases} \quad (1)$$

Innan vi diskuterar *CANDI*:s oberäkningsbarhet, får du en fråga att noga reflektera över. Texten som beskriver *CANDI* kan ju, med vårt generösa synsätt på vad program är, betraktas som ett program. Eller hur! Låt oss kalla detta program för *CANDI-texten*.

Varför kan inte CANDI-texten beräkna CANDI?

Här är ett svar.

Ett nödvändigt och tillräckligt villkor för att *CANDI-texten* för ett visst n, skall kunna beräkna $CANDI(n)$ är

$$\textit{att kunna reda ut ifall } P_n(n) \textit{ är definierad.} \qquad (2)$$

Har *CANDI-texten* den förmågan?

Inte ensam, ty i *CANDI-texten* står ju inget om hur (2) skulle kunna tänkas gå till.

Men det kanske ändå finns någon beskrivning någonstans av hur det kan gå till? Om inte, kanske det går att tänka ut en sådan? En beskrivning som vi i så fall skulle kunna lägga in i *CANDI-texten* som därvid i ett slag skulle förvandlas till ett program som beräknar *CANDI*. Men denna tanke måste tyvärr förbli ett önsketänkande. Ty – som vi snart skall inse – finns det inget program som kan beräkna den märkliga funktionen *CANDI*. I nästa avsnitt diskuterar vi sedan en intressant konsekvens av att det är på det sättet.

Vi visar nu att inget av programmen $P_0, P_1, P_2, \ldots$ kan beräkna *CANDI*.

Funktionen *CANDI* är avsiktligt definierad på ett så djävulskt sätt att inget P_n ensam skall kunna beräkna den. Av (1) följer nämligen att för varje n är

$$P_n(n) \neq CANDI(n). \qquad (3)$$

Och av (3) följer att varje P_n har åtminstone *en position* i sin outputrad där innehållet skiljer sig från innehållet i samma position i *CANDI*:s outputrad, nämligen den n:te. Se mer om detta nedanför.

ANM. 7.1 Föreställ dig att du känner outputraderna, den ena efter den andra, till $P_0, P_1, P_2, \ldots$. T. ex. som nedanför. Innebörden i (1) är att *CANDI*:s outputrad definieras som en modifiering av diagonalen.

x	0	1	2	...
$P_0(x)$	-	1	2	...
$P_1(x)$	3	3	3	...
$P_2(x)$	2	3	5	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
$CANDI(x)$	1	4	6	...

Jfr. med Cantors diagonalbevis sid. 123.

Läsare som menar att funktionen *CANDI* på sin höjd är av akademiskt intresse, kan kanske hitta något av intresse i nästa avsnitt.

Stopproblemet

Ifall något program kunde lösa problemet

stopproblemet

att för varje program P_n och varje input x,
reda ut om P_n någonsin stannar vid körning på x.

så skulle funktionen *CANDI* kunna beräknas (se (2)).

Men *CANDI* kan de facto inte beräknas.

Således finns det inget program som kan lösa stopproblemet, som förresten kan ges en funktionsbeskrivning:

En funktion som ”löser” stopproblemet

$$Stannar?(n,x) = \begin{cases} 1, \text{ om } P_n(x) \text{ är definierad} \\ 0, \text{ om } P_n(x) \text{ är odefinierad} \end{cases}$$

Att inget *program* kan lösa stopproblemet, betyder att funktionen *Stannar?* inte kan beräknas av något program. Och det är onekligen mycket trist, eftersom oändliga slingor och andra hängningsproblem är programutvecklares värsta gissel.

Enstaka stopproblem

Stopproblemets olösbarhet innebär att det inte finns något program som för *varje* program och *varje* input kan räkna ut om programmet stannar vid körning på nämnda input.

Med enstaka program – till exempel Kalles program – är det annorlunda. Det kan vara fullt *möjligt* att reda ut ifall det stannar. Om man kör igång Kalles program, och det visar sig att detta stannar efter en tid, så är ju saken klar. Och om det ännu inte har stannat, så kan det ändå vara så att det stannar nästa sekund eller nästa sekel, och då (när det har stannat) är saken klar. Men det kan också vara *omöjligt* att få reda på om det stannar eller ej. Åtminstone är det omöjligt att få visshet via ett resonemang som är så mekaniskt att man kan göra ett program av det. stopproblemets olösbarhet innebär nämligen att man medelst program inte kan få visshet för alla program – och Kalles program kan ju vara ett av de omöjliga.

Lägg märke till att vi säger att Kalles program *kan* vara ett av de omöjliga. Ty hur än Kalles program är beskaffat – om det till exempel är ett oerhört komplext program, svårt att förstå, eller om det är enkelt och lätt att förstå – så kan ändå ingen bevisa att Kalles program tillhör de omöjliga. Ty antag att man kunde bevisa att det var omöjligt att få reda på ifall Kalles program någonsin stannar eller ej. Då skulle Kalles program inte kunna stanna –eftersom man då skulle få visshet! Men att det inte kan stanna innebär ju också att man har visshet! Alltså, ifall man kunde bevisa att det var omöjligt att få visshet, så skulle man få visshet.

Att tala om sig själv är svårt

Med program kan man göra många till synes komplexa uträkningar, exempelvis beräkna planeternas banor eller de första tio millionerna decimaler för π. Men när program skall räkna ut saker om program, så dyker det upp logiska paradoxer av olika slag som gör programmen till Stålmän med kryptonit runt halsen. I själva verket är de här svårigheterna en yttring av Gödels ofullständighetssats[†] – som löst talat säger följande:

> *Inom varje motsägelsefritt axiomsystem för matematiken som inkluderar elementär aritmetik går det att formulera påståenden som det – inom systemets ram – inte går att bevisa eller motbevisa.*

[†] Kurt Gödel publicerade satsen 1931 i Monatshefte für Mathematik und Physik 38, 173-198. Flera utmärkta populärvetenskapliga skrifter har satsen som ett huvudtema. T. ex. Douglas R. Hofstadters Pulitzerprisbelönta bok GÖDEL ESCHER BACH, och Raymond Smullyans *Forever undecided, a puzzle guide to Gödel*, 1988.

Lösningar till övningar

1

1.1 a) $x' \leftarrow x$
$x'' \leftarrow x$

b) Töm r
Addera x till r
Addera y till r
Addera z till r

c) $e \leftarrow y$
$y \leftarrow x$
$x \leftarrow e$

1.2 x, y, z kommer att få de innehåll som ursprungligen fanns i y, z, y.

1.3 a) Subtrahera y från x =
$y' \leftarrow y$
Sålänge y' är icketom {
Minska y'; Minska x}

b) Subtrahera y från x i r =
$r \leftarrow x$; $y' \leftarrow y$
Sålänge y' är icketom {
Minska y'; Minska r}

1.4 a) Töm r
Sålänge x är icketom {
Addera x till r
Minska x}

b) $r \leftarrow 1$
Sålänge x är icketom {
Multiplicera r med x
Minska x}

c) Töm r
Potensupphöj x till 2 i t
Addera t till r
Addera x till r
Addera 1 till r

d) Töm r
Sålänge n är icketom {
Potensupphöj x till n i t
Addera t till r
Minska n}

1.5 a) **inte**(x är tom)? i s

b) **inte**($y \geq x$)? i s

c) ($x \geq y$ **och** $y \geq x$)? i s

d) Dividera x med 2 i q och r
$r = 1$? i s

1.6 **Om** $x \geq y$ **så** (Subtrahera y från x i r)
annars (Subtrahera x från y i r)

1.7 a) Kvadrattal n i r =
Multiplicera n och n i r

b) x är ett kvadrattal? i s =
Töm n; Töm k
Sålänge $k < x$ {
Kvadrattal n i k; Öka n}
$x = k$? i s

1.8 x är ett triangeltal? i s =
Töm n; Töm t
Sålänge $t < x$ {Triangeltal n i t; Öka n}
$x = t$? i s

1.9 Fibonacci n i r =
$r \leftarrow 1$; $r' \leftarrow 0$; $k \leftarrow 1$
Sålänge $k < n$ {
Öka k; $e \leftarrow r$
Addera r' till r
$r' \leftarrow e$}
KOMMENTAR: Om $n = 1$, dvs lika stor som ursprungs-k, så körs slingan aldrig igenom, och då kommer r att innehålla 1, det första av de uppräknade Fibonaccitalen. Om $n = 2$ så körs slingan igenom en gång, osv.. För att generera nya Fibonaccital skall de två senaste talen adderas. Detta inträffar inuti slingan där r alltid innehåller det allra senaste Fibonaccitalet och r' det näst senaste. Extrahögen e används som behållare för r:s Fibonaccital när r uppgraderas. Sedan lämnar e över detta tal – som nu är det näst senaste – till r'.

1.10 a) Ärtig x i s? undersöker om x ligger i treans multiplikationstabell. b) Buk x i r lägger x^3 i högen r. c) Jadhr x i r lägger heltalsdelen av $\sqrt{x}$ i högen r. (*Jadhr* är ett gammalt arabiskt namn for kvadratrot och användes redan av al-Khwarizmi i dennes klassiska text Al-Jabr Wa-Al-Muqabala från år 820.)
d) Tardavk x i r lägger summan av de x första udda talen i r. Och denna summa är lika med x^2.
KOMMENTAR till d) Att summan av de x första udda talen är lika med kvadraten på x kan bevisas mycket övertygande med en figur. Närmare bestämt med hjälp av en kvadrat med sidan x som delas in i rutor med sidan 1.

2

2.1 a) $Add(x, Add(y, z))$ eller $Add(Add(x, y), z)$
b) $Mult(x, Add(y, z))$ c) $Pot(x, Pot(y, z))$
d) $Pot(Pot(x, y), z)$ e) $Pot(x, Add(y, z))$
f) $Pot(Add(x, y), z)$ g) $Pot(Add(x, y), Mult(z, u))$

2.2

$$\begin{aligned}
&Pot(5,4)\\
&= Mult(5, Pot(5,3))\\
&= Mult(5, Mult(5, Pot(5,2)))\\
&= Mult(5, Mult(5, Mult(5, Pot(5,1))))\\
&= Mult(5, Mult(5, Mult(5, Mult(5, Pot(5,0)))))\\
&= Mult(5, Mult(5, Mult(5, Mult(5,1))))\\
&= Mult(5, Mult(5, Mult(5,5)))\\
&= Mult(5, Mult(5,25))\\
&= Mult(5,125)\\
&= 625
\end{aligned}$$

2.3 $\begin{cases} Fakultet(0) = 1 \\ Fakultet(\ddot{O}ka(x)) = Mult(\ddot{O}ka(x), Fakultet(x)) \end{cases}$

2.4

$$\begin{cases} f(0) = 0 \\ f(\ddot{O}ka(x)) = Add(Kvadrat(\ddot{O}ka(x)), f(x)) \end{cases}$$

$$\begin{cases} f(0) = 0 \\ f(\ddot{O}ka(x)) = Add(Mult(2, \ddot{O}ka(x)), f(x)) \end{cases}$$

$$\begin{cases} f(0) = 0 \\ f(\ddot{O}ka(x)) = Add(Triangeltal(\ddot{O}ka(x)), f(x)) \end{cases}$$

2.5 $f(0) = 1, f(1) = 2, f(2) = 4, f(3) = 8$, och $f(x) = 2^x$ för godtyckligt x. Motivering: Anropet $f(x)$ drar (med hjälp av $h(x,x)$) igång en (primitivt) rekursiv kalkyl där h anropar sig själv x gånger. Vid varje anrop fördubblas föregående output, och rekursionens botten är lika med 1. Resultatet efter alla dessa fördubblingar blir $2 \cdot \ldots \cdot 2 \cdot 1 = 2^x$, eftersom antalet 2:or är lika med x.

2.6 a) ”Felet” ligger i att högerledet $f(Sub(x, f(x)))$ är av otillåtet slag. Om man jämför med högerledet i den primitivt rekursiva mallen, ser man att g–funktionen *inte* får konstrueras med hjälp av f, utan måste vara en redan konstruerad funktion – inom klassen primitivt rekursiva funktioner. b) Man ser att f:s samtliga output är 1. Därför kan man t. ex. presentera f som

$$\begin{cases} f(0) = 1 \\ f(\ddot{O}ka(x)) = f(x) \end{cases}$$

2.7 I den primitivt rekursiva mallen

$$\begin{cases} Sub(x, 0) = b(x) \\ Sub(x, \ddot{O}ka(y)) = g(Sub(x, y)) \end{cases}$$

väljer man b som identitetsfunktionen, och g som funktionen $Minska$.

2.8

$$\begin{cases} Udda(0) = 0 \\ Udda(\ddot{O}ka(x)) = Icke(Udda(x)) \end{cases}$$

$$\begin{cases} J\ddot{a}mn(0) = 1 \\ J\ddot{a}mn(\ddot{O}ka(x)) = Icke(J\ddot{a}mn(x)) \end{cases}$$

2.9

$a)\ Lika(x, y) = Och(Icke(Mindre(x, y), Icke(Mindre(y, x))))$
$b)\ Olika(x, y) = Icke(Lika(x, y))$
$c)\ MindreEllerLika(x, y) = Icke(Större(x, y))$

2.10 a) $\begin{cases} TvåPotens(0) = 1 \\ TvåPotens(\ddot{O}ka(x)) = Mult(2, TvåPotens(x)) \end{cases}$

$b)$ $\begin{cases} KubSumma(0) = 1 \\ KubSumma(\ddot{O}ka(x)) = Add(Pot(\ddot{O}ka(x), 3), KubSumma(x)) \end{cases}$

2.11 a) $1^1 + 2^2 + 3^3 = 32$
b) $1^1 + 2^2 + 3^3 + \ldots + x^x$ kan beräknas av $f(x)$, där

$$\begin{cases} f(0) = 0 \\ f(\ddot{O}ka(x)) = Add(Pot(\ddot{O}ka(x), \ddot{O}ka(x)), f(x)) \end{cases}$$

2.12 Det finns onekligen blott ett sätt att gå 1 meter. Nämligen att ”ta ett enmeterskliv”. Låt oss också enas om att det finns ett sätt att gå 0 meter, nämligen ”att stå stilla”.

$$SättAttGå(0) = 1 \text{ och } SättAttGå(1) = 1$$

Om man vill gå längre än 1 meter, så finns det två sätt att inleda promenaden.

FALL 1: Med ett enmeterskliv,

FALL 2: med ett tvåmeterskliv.

I båda fallen återstår det att avverka ytterligare ett antal meter. (Eventuellt 0 meter.) Antag att återstoden av promenaden i FALL 2 mäter x meter, vilket innebär att hela promenadens längd är $x + 2$ meter. Antalet sätt att gå hela promenaden i nämnda fall (med ett inledande tvåmeterskliv) är då lika med antalet sätt att gå de återstående x metrarna. Motsvarande återstod i det första fallet mäter $x + 1$ meter. Det följer att

$$SättAttGå(x + 2) = Add(SättAttGå(x + 1), SättAttGå(x))$$

Man ser (eller hur!) att den här funktionen för $x = 0, 1, 2, \ldots$ returnerar $Fib(1), Fib(2), Fib(3), \ldots$.

2.13 Funktionen f beräknar

$$f(x) = 0 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 + 3 \cdot 4 + \ldots + x \cdot (x + 1).$$

Samma sak beräknas av stenhögsproceduren

Töm r
Repetera med $a = 1$ till x {
  Addera a och 1 i b
  Multiplicera a och b i c
  Addera c till r}

2.14 a) f upprepar sig efter fyra steg, medan g:s output blir två enheter större efter två steg. Härav,

$$\begin{cases} f(0) = 1 \\ f(1) = 2 \\ f(2) = 0 \\ f(3) = 0 \\ f(x+4) = f(x) \end{cases}$$

$$\begin{cases} g(0) = 1 \\ g(1) = 0 \\ g(x+2) = g(x) + 2 \end{cases}$$

b) När de givna tabellerna skärskådas, så ser man följande mönster:

Om f:s nuvarande output är lika med 0, så är nästa lika med 1 ifall nuvarande input är udda, annars (vid jämnt input) 0. Och om f:s nuvarande output är lika med 1, så är nästa lika med 2, annars 0. Det följer att

$$\begin{cases} f(0) = 1 \\ f(\mathit{Öka}(x)) = \mathit{Om}(\mathit{Noll?}(f(x)), \mathit{Udda}(x), \mathit{Om}(\mathit{Ett?}(f(x)), 2, 0)) \end{cases}$$

Vad beträffar g, så kan man konstatera att för udda x är $g(x) = x - 1$ och för jämna x är $g(x) = x + 1$. Härav,

$$g(x) = \mathit{Om}(\mathit{Udda}(x), \mathit{Minska}(x), \mathit{Öka}(x))$$

2.15 Ja, funktionerna beräknar samma sak

2.16 Med framåtrekursion (jfr. sid 50):

$$\begin{aligned} &\mathit{Triangeltal?}(x) = \mathit{LöpFörbiTriangeltalen}(0, x) \\ &\mathit{LöpFörbiTriangeltalen}(n, x) = \\ &\qquad \mathit{Om}(\mathit{Mindre}(\mathit{Triangeltal}(n), x), \\ &\qquad\quad \mathit{LöpFörbiTriangeltalen}(\mathit{Öka}(n), x), \\ &\qquad\quad \mathit{Lika}(\mathit{Triangeltal}(n), x)) \end{aligned}$$

Med primitiv rekursion:

$$
\begin{aligned}
&\mathit{Triangeltal?}(x) = \mathit{LöpFörbiTriangeltalenBakåt}(x, x)\\
&\mathit{LöpFörbiTriangeltalenBakåt}(0, x) = \mathit{Lika}(\mathit{Triangeltal}(0), x)\\
&\mathit{LöpFörbiTriangeltalenBakåt}(\mathit{Öka}(n), x) =\\
&\qquad\qquad \mathit{Om}(\mathit{Större}(\mathit{Triangeltal}(\mathit{Öka}(n)), x),\\
&\qquad\qquad\quad \mathit{LöpFörbiTriangeltalenBakåt}(n, x)),\\
&\qquad\qquad\quad \mathit{Lika}(\mathit{Triangeltal}(\mathit{Öka}(n)), x)
\end{aligned}
$$

2.17
$$
\begin{aligned}
&\mathit{Rest}(0, y) = 0\\
&\mathit{Rest}(\mathit{Öka}(x), y) = \mathit{Om}(\mathit{Mindre?}(\mathit{Öka}(\mathit{Rest}(x, y)), y),\\
&\qquad\qquad\qquad\qquad \mathit{Öka}(\mathit{Rest}(x, y)),\\
&\qquad\qquad\qquad\qquad 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Kvot}(0, y) = 0\\
&\mathit{Kvot}(\mathit{Öka}(x), y) = \mathit{Om}(\mathit{Noll?}(\mathit{Rest}(\mathit{Öka}(x), y)),\\
&\qquad\qquad\qquad\qquad \mathit{Öka}(\mathit{Kvot}(x, y)),\\
&\qquad\qquad\qquad\qquad \mathit{Kvot}(x, y))
\end{aligned}
$$

3

3.1 a) 2 b) [2 **nil**] c) **nil**

3.2
$$
\begin{aligned}
&\mathit{Bara}(0, x) = [\,]\\
&\mathit{Bara}(\mathit{Öka}(n), x) = \left[x \,\middle|\, \mathit{Bara}(n, x)\right]
\end{aligned}
$$

3.3 a)
$$
\begin{aligned}
&\mathit{Tillhör}([\,], x) = 0\\
&\mathit{Tillhör}(\left[\mathit{nod} \,\middle|\, L\right], x) = \mathit{Eller}(\mathit{Lika}(\mathit{nod}, x),\ \mathit{Tillhör}(L, x))
\end{aligned}
$$

b) $AntalFörekomster([\,], x) = 0$
$AntalFörekomster(\left[nod \,\middle|\, L\right], x) =$
$Om(Lika(x, nod),$
$Öka(AntalFörekomster(L, x)),$
$AntalFörekomster(L, x))$

3.4 $ListPositioner([\,], x) = [\,]$
$ListPositioner(\left[nod \,\middle|\, L\right], x) =$
$Om(Lika(nod, x),$
$\left[1 \,\middle|\, Map(Öka, ListPositioner(L, x))\right],$
$Map(Öka, ListPositioner(L, x)))$

3.5 a) $Andra(L) = Första(UtomFörsta(L))$

b) $Element(L, 1) = Första(L)$
$Element(L, Öka(n)) = Element(UtomFörsta(L), n)$

T. ex. är

$$\begin{aligned} Element(L, 2) &= Element(UtomFörsta(L), 1) \\ &= Första(UtomFörsta(L)) \\ &= Andra(L) \end{aligned}$$

c) $UtomAndra(L) = \left[Första(L) \,\middle|\, UtomFörsta(UtomFörsta(L))\right]$

d) $\begin{cases} Utom(L, 1) = UtomFörsta(L) \\ Utom(L, Öka(n)) = \left[Första(L) \,\middle|\, Utom(UtomFörsta(L), n)\right] \end{cases}$

Exempelvis är

$$\begin{aligned} Utom(L, 2) &= \left[Första(L) \,\middle|\, Utom(UtomFörsta(L), 1)\right] \\ &= \left[Första(L) \,\middle|\, UtomFörsta(UtomFörsta(L))\right] \\ &= UtomAndra(L) \end{aligned}$$

$$e)\begin{cases} \mathit{Efter}(L,1) = \mathit{UtomFörsta}(L) \\ \mathit{Efter}(L, \mathit{Öka}(n)) = \mathit{UtomFörsta}(\mathit{Efter}(L,n)) \end{cases}$$

3.6 a) $\big[\mathit{Första}(L)\,\big|\,\big[x\,\big|\,\mathit{UtomFörsta}(L)\big]\big]$

$$b)\begin{cases} \mathit{FogaInFöre}(1,L,x) = \big[x\,\big|\,L\big] \\ \mathit{FogaInFöre}(\mathit{Öka}(n),L,x) = \\ \qquad\qquad \big[\mathit{Första}(L)\,\big|\,\mathit{FogaInFöre}(n,\,\mathit{UtomFörsta}(L),x)\big] \end{cases}$$

3.7 a)
$sum \leftarrow 0$
Repetera med $dygn = 1$ **till** 7 {
  Addera $T(5, dygn, 12)$ **till** sum
}
Dividera sum med 7 i $medel$ och r

b)
$sum \leftarrow 0$
Repetera med $vecka = 1$ **till** 20 {
  Repetera med $dygn = 1$ **till** 7 {
    Addera $T(vecka, dygn, 12)$ **till** sum
  }
}
Multiplicera 20 och 7 i $antalDagar$
Dividera sum med $antalDagar$ i $medel$ och r

3.8 a)
$$\mathit{TaBort}(x, [\,]) = [\,]$$
$$\mathit{TaBort}(x, \big[nod\,\big|\,L\big]) = \mathit{Om}(\mathit{Lika}(x, nod), \mathit{TaBort}(x, L), \big[nod\,\big|\,\mathit{TaBort}(x, L)\big])$$

b)
$$\mathit{Ersätt}(x, y, [\,]) = [\,]$$
$$\mathit{Ersätt}(x, y, \big[nod\,\big|\,L\big]) = \mathit{Om}(\mathit{Lika}(x, nod), \big[y\,\big|\,\mathit{Ersätt}(x, y, L)\big], \big[nod\,\big|\,\mathit{Ersätt}(x, y, L)\big])$$

c) $FinnsKopior([\,]) = 0$
$FinnsKopior(\left[nod \middle| L\right]) = Eller(\,Tillhör(L, nod), FinnsKopior(L))$

3.9 a) $TaBortGrannKopior([\,]) = [\,]$
$TaBortGrannKopior([nod]) = [nod]$
$TaBortGrannKopior(\left[nod \middle| L\right]) =$
$Om(Lika(nod, Första(L)),$
$\left[nod \middle|\, UtomFörsta(\,TaBortGrannKopior(L))\right],$
$\left[nod \middle|\, TaBortGrannKopior(L)\right])$

b) $TaBortKopior([\,]) = [\,]$
$TaBortKopior(\left[nod \middle| L\right]) =$
$Om(\,Tillhör(L, nod),$
$TaBortKopior(L),$
$\left[nod \middle|\, TaBortKopior(L)\right])$

3.10 a) $Reversera([\,]) = [\,]$ b) = a)
$Reversera(\left[nod \middle| L\right]) = \left[Reversera(L) \middle\| nod\right]$

3.11 $TotalReversera(\mathbf{atom}) = \mathbf{atom}$
$TotalReversera([\,]) = [\,]$
$TotalReversera(\left[nod \middle| L\right]) =$
$\left[\,TotalReversera(L) \middle\|\, TotalReversera(nod)\right]$

3.12 $ListMax(\mathbf{nil}) = \mathbf{nil}$
$ListMax([nod]) = nod$
$ListMax(\left[nod \middle| L\right]) = Max(nod, ListMax(L))$

3.14 $BinärSök(x, [\,]) = 0$
$BinärSök(x, [nod]) = Lika(x, nod)$
$BinärSök(x, L) = Om(Mindre(x, Första(HögerHalva(L))),$
$BinärSök(x,\, VänsterHalva(L)),$
$BinärSök(x, HögerHalva(L)))$

3.15 $\mathit{TreDela}(L) = h([\,], L)$

$$h(X, [y|Y]) = \mathit{Om}(\mathit{Mindre}(\mathit{Dubbla}(\mathit{Längd}(X)), \mathit{Längd}(Y)),$$
$$h([X \| y], Y),$$
$$[X, \mathit{TuDela}([y|Y]\ |)])$$

där $\mathit{Dubbla}(n) = \mathit{Mult}(2, n)$.

KOMMENTAR: Tredelningsreceptet går ut på att först (precis som i tudelningsreceptet) steg för steg flytta över element (från vänster ände av listan som skall delas) till en från början tom lista. På detta sätt avskiljs en del av den givna listans vänstra ände. Nu vill vi att den avskiljda delen skall vara en tredjedel av ursprungslistan. Därför skall den icke avskiljda delen av listan vara dubbelt så lång som den avskiljda. Detta förklarar villkoret som styr rekursionen. Efter avskiljningen behöver man bara tudela den del som inte avskiljdes.

3.18 $\mathit{SlätaUtHögst}([\,], n_)) = [\,]$

$$\mathit{SlätaUtHögst}(L, 0) = L$$
$$\mathit{SlätaUtHögst}([nod|L], \mathit{Öka}(n)) =$$
$$\mathit{Om}(\mathit{Atom?}(nod),$$
$$\mathit{FogaIn}(nod, \mathit{SlätaUtHögst}(L, \mathit{Öka}(n))),$$
$$\mathit{FogaSamman}(\mathit{SlätaUtHögst}(nod, n),$$
$$\mathit{SlätaUtHögst}(L, \mathit{Öka}(n))))$$

3.19 LEDNING: Antag (önsketänkande) att man redan har gjort alla möjliga placeringslistor för en mängd X av personer. Ifall ytterligare en gäst x skall vara med vid bordet, så behöver man modifiera varje redan gjord placeringslista L, så att även x får plats. Nämnda modifiering är nyckeln till problemets lösning. Tänk på att modifieringen av L inte är en ny lista med ett extra element, utan flera nya listor. Lika många som det finns platser för x i L. I och med detta har du fått ett embryo till lösning att själv bygga vidare på. Det skulle förvåna mig om inte de två föregående övningarna skulle komma till användning.

3.20 $KartesiskaProdukten([\,], L) = [\,]$

$KartesiskaProdukten(\left[x \,\middle|\, X\right], Y) =$

$FogaSamman(ParaIhop(x, Y),$

$KartesiskaProdukten(X, Y))$

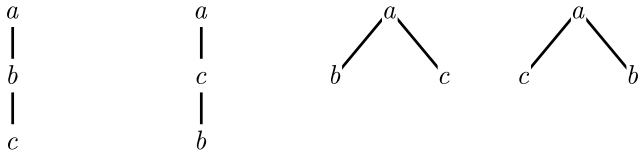
där $ParaIhop(x, L)$ parar ihop ett fixerat godtyckligt x med varje element i en lista L. Dvs. returnerar listan av alla tvåelementslistor $[x\ y]$ där y tillhör L:

$ParaIhop(x, [\,]) = [\,]$

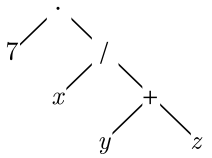
$ParaIhop(x, \left[y \,\middle|\, Y\right]) = \left[[x\ y] \,\middle|\, ParaIhop(x, Y)\right]$

4

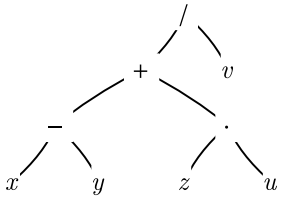
4.1



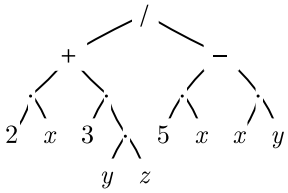
4.2 a)



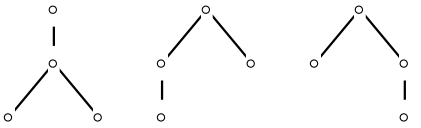
b)



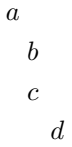
c)



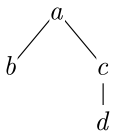
4.3



4.4 a)

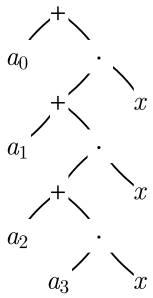


b)



4.5 Om T:s längd är större än 1 så beräknas antalet barn till trädets förste son.

4.6 a)



b)

$$
\begin{aligned}
&Add(a_0,\\
&\quad Mult(Add(a_1,\\
&\qquad\qquad Mult(Add(a_2,\\
&\qquad\qquad\qquad\qquad Mult(a_3,x)),\\
&\qquad\qquad\qquad\quad x)),\\
&\qquad\quad x))
\end{aligned}
$$

c) Vi skall beräkna

$$Horner([\,a_0\ \ldots\ a_n\,],x) = a_0+(a_1+(\ldots+(a_{n-1}+a_nx)x\cdots)x)x,$$

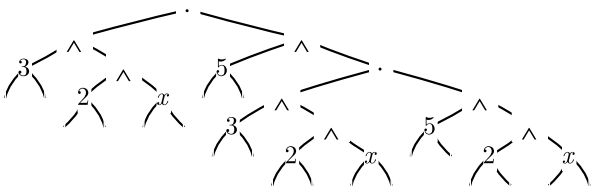
och noterar att det enda som behöver göras ifall vi redan har beräknat

$$Horner([\,a_1\ \ldots\ a_n\,],x) = a_1+(\ldots+(a_{n-1}+a_nx)x\cdots)x,$$

är att multiplicera med x och sedan lägga till a_0. Det följer, om $L = [\,a_1\ \ldots\ a_n\,]$, att

$$
\begin{aligned}
&Horner([\,],x) = 0\\
&Horner(\left[a_0 \,\middle|\, L\right],x) = Add(a_0, Mult(Horner(L,x),x))
\end{aligned}
$$

4.7



4.8 $BytLövMotTräd([\,], x, B) = [\,]$
$BytLövMotTräd([rot\ [\,]\ [\,]], x, B) = Om(Lika(rot, x), B, [rot\ [\,]\ [\,]])$
$BytLövMotTräd([rot\ v\ h], x, B) =$
$\quad [rot\ BytLövMotTräd(v, x, B)\ BytLövMotTräd(h, x, B)]$

4.9 $TrädDjup([rot]) = 1$
$TrädDjup(\left[rot \,\middle|\, skog\right]) = Öka(ListMax(Map(TrädDjup, skog)))$

4.13 $Rötter(skog) = Map(Första, skog)$

4.14 $SyskoniTräd(\left[rot \,\middle|\, skog\right], x) =$
$\quad Om(\,Tillhör(Rötter(skog), x),$
$\quad\quad \left[Rötter(skog) \,\middle|\, SyskoniSkog(skog, x)\right],$
$\quad\quad SyskoniSkog(skog, x))$

$SyskoniSkog([\,], x) = [\,]$
$SyskoniSkog(\left[träd \,\middle|\, skog\right], x) =$
$\quad FogaSamman(SyskoniTräd(träd, x), SyskoniSkog(skog, x))$

4.15 $NoderPåNivå(0, [rot \,|\, skog]) = [rot]$
$NoderPåNivå(Öka(n), [rot \,|\, skog]) =$
$\quad NoderiSkogPåNivå(n, skog)$

$NoderiSkogPåNivå(0, [\,]) = [\,]$
$NoderiSkogPåNivå(n, [träd \,|\, skog]) =$
$\quad FogaSamman(NoderPåNivå(n, träd),$
$\qquad\qquad NoderiSkogPåNivå(n, skog))$

4.16 $BreddenFörst(träd) =$
$\qquad TraverseraNerTillNivå(TrädDjup(träd), träd)$
där
$TraverseraNerTillNivå(0, [rot \,|\, skog]) = [rot]$
$TraverseraNerTillNivå(Öka(n), träd) =$
$\quad FogaSamman(TraverseraNerTillNivå(n, träd),$
$\qquad\qquad NoderPåNivå(Öka(n), träd))$

4.17 $Tronföljden([rot]) = [rot]$
$Tronföljden([rot \,|\, skog]) = [rot \,|\, Tronföljden(Första(skog))]$

4.18 $SkapaBinärtTrädAv([\,]) = [\,]$
$SkapaBinärtTrädAv([nod \,|\, L]) =$
$\qquad SättIn(nod, SkapaBinärtTrädAv(L))$

$SättIn(nod, [\,]) = [nod\ \mathbf{nil}\ \mathbf{nil}]$
$SättIn(nod, [rot\ v\ h]) = Om(Mindre(nod, rot),$
$\qquad\qquad [rot\ SättIn(nod\ v)\ h],$
$\qquad\qquad [rot\ v\ SättIn(nod\ h)])$

4.19 $TillSkog([\,]) = [\,]$
$TillSkog([rot\ v\ h]) = [[rot \,|\, TillSkog(v)] \,|\, TillSkog(h)]$

4.20 $Eval([tal\ \mathbf{nil}\ \mathbf{nil}]) = tal$
$Eval([+\ v\ h]) = Add(Eval(v), Eval(h))$
$Eval([-\ v\ h]) = Sub(Eval(v), Eval(h))$
$Eval([\cdot\ v\ h]) = Mult(Eval(v), Eval(h))$

5

5.1 $924 = 2 \cdot 462 = 2 \cdot 2 \cdot 231 = 2 \cdot 2 \cdot 3 \cdot 77 = 2 \cdot 2 \cdot 3 \cdot 7 \cdot 11 = 2^2 \cdot 3^1 \cdot 5^0 \cdot 7^1 \cdot 11^1$

5.2 924 uppstår som Gödeltalet för tex. listan [2 1 0 1 1] när man använder *gödel*. Den modifierade definitionen *Gödel* kan inte ge upphov till 924, eftersom 924 är en primtalsprodukt av såväl 2:or som större primtal, medan de primtalsprodukter som *Gödel* tillverkar endera innehåller enbart 2:or eller enbart större primtal.

5.3 $G\ddot{o}del([[2]]) = 3^{3^{2^2}}$, $G\ddot{o}del([4]) = 3^{2^4}$.
$G\ddot{o}del([2\ 0]) = 3^{G\ddot{o}del(2)} \cdot 5^{G\ddot{o}del(0)} = 3^{2^2} \cdot 5^{2^0} = 3^4 \cdot 5 = 405$.
$G\ddot{o}del([0\ \ [2\ 0]]) = 3^{G\ddot{o}del(0)} \cdot 5^{G\ddot{o}del([2\ 0])} = 3 \cdot 5^{405}$, ett tal som utskrivet skulle uppta drygt fem rader på denna sida.

5.4 Om AHA först transformeras till [1 8 1], så kan vi sedan Gödelnumrera med $2^1 \cdot 3^8 \cdot 5^1 = 65610$.

5.5 a) $48 = 2^4 \cdot 3$ kan inte uppstå vid kodning med *Gödel*. (Se lösningen till övning 5.2.)
b) [[0] [1 0]].

5.6 $G\ddot{o}del(tal) = Pot(2, tal)$
$G\ddot{o}del([\,]) = 1$
$G\ddot{o}del(lista) = Mult(G\ddot{o}del(UtomSista(lista),$
$Pot(Primtal(L\ddot{a}ngd(lista)),$
$G\ddot{o}del(Sista(lista))))$

5.7 a) En smula förenklat kan vi se en melodi som en lista av toner och pauser. Varje ton är i sin tur en lista av två tal vilka beskriver tonens frekvens respektive tonens längd. (Att tonstyrkan kan variera från ton till ton och även under en och samma ton bortser vi från nu.) Varje paus är i sin tur bestämd av ett tal, pausens längd. En melodi ges således av en lista av listor och tal. Därmed kan den tillordnas ett Gödeltal.
b) Ett fotografi kan delas in i rader av små rutor, där varje ruta har en viss färg eller gråton som kan beskrivas med ett tal. Varje rad av rutor blir därmed en lista av tal, och hela fotografiet en lista av listor av tal.
c) En stumfilm är en sekvens av fotografier. Alltså en lista av listor av listor av tal.

6

6.1 En uppräkning som faller sig naturlig är att börja med 0 och sedan omväxlande ta positiva respektive negativa tal:

0	1	2	3	4	5	6	...
↕	↕	↕	↕	↕	↕	↕	...
0	1	-1	2	-2	3	-3	...

6.2 Det minsta talet är i detta fall 0.000000, dvs. 0. Nästa tal i storleksordning är först 0.000001, sedan 0.000002, osv.. Den uppräkning i storleksordning, som vi just påbörjat är vårt enkla svar.

0	1	2	...	9	10	...
↕	↕	↕	...	↕	↕	...
0.000000	0.000001	0.000002	...	0.000009	0.000010	...

6.3 Lägg märke till att den givna numreringen räknar upp talparen diagonalvis. I samma stund som en viss diagonal är genomgången har man gått igenom samtliga talpar i den *triangel* som har nämnda diagonal som bas och $(0, 0)$ som topp. Som en följd av detta faktum kommer det första talparet på den n:te diagonalen att numreras med $Triangeltal(n)$. Varje annat talpar ges ett nummer som är en enhet högre än numret

som grannen nedanför på samma diagonal har fått.

$$\begin{cases} Nummer(0, y) = Triangeltal(y) \\ Nummer(\ddot{O}ka(x), y) = \ddot{O}ka(Nummer(x, \ddot{O}ka(y))) \end{cases}$$

Funktionsregister

Sakregister