Datorteknik
Strukturerad programmering

Michael Josefsson

Version 0.5

Innehåll

1. Planering

Det[1] är faktiskt inte så stor skillnad på att till exempel planera semestern och att planera ett datorprogram. All planering inför en uppgift kan delas upp i tre steg:

1. Vilken är uppgiften?
2. Vilket är läget nu?
3. Kraftansamla för att lösa uppgiften!

Dessa tre punkter är förvånansvärt användbara, lär dig dem utantill och använd dem alltid! Med lite träning kommer du lära dig att omedelbart och närmast omedvetet tänka i dessa steg.

Exempel Inför uppgiften tentamen behöver du veta *när* den går och *vilket ämne* det handlar om (uppgiften). Men du måste också känna dig själv och bedöma vad du behöver kunna för att lyckas med tentamen, vilka dina förkunskaper är och vad du behöver lära dig (läget). Slutligen måste du studera det som behövs för att klara tentamen (kraftansamla).

De flesta av dessa uppdelningar ter sig rätt naturliga, eller hur? Varför det? För att de *är* naturliga så klart! Vi har genom åren vant oss att tänka på det här sättet helt automatiskt. *Vi tänker så helt utan att tänka på att vi tänker så.* Om vi blir medvetna om att vi tänker på detta sättet kan vi också tillämpa detta tänkesätt även i konstruktioner och utvecklingsprojekt och på så sätt öka effektiviteten och tillförlitligheten i resultatet.

1.1. Deluppgifter

En stor uppgift kan förstås delas upp i mindre, mer hanterbara deluppgifter. Det är svårt, och till och med direkt olämpligt, att inte dela upp en stor uppgift i flera små. En stor uppgift är ohanterlig — naturligtvis, vore den inte ohanterlig vore det ju inte en stor uppgift?

En deluppgift är ofta så stor att man lätt kan se att den har en lösning, även om man inte just vid uppdelningen *exakt* vet *hur* den ska kunna lösas. Den stora fördelen med deluppgifter är faktiskt att man inte ska behöva bry sig om exakt hur de ska lösas, det tar man tag i senare.

[1] *Document:* JSP_AVR, rev 2017-10-17

I högnivåspråk utför vi deluppgifterna i *funktioner* och *procedurer*. Det finns en teoretisk uppdelning mellan funktioner och procedurer som säger att funktioner returnerar något medan en procedur inte gör det. Båda kan dock ha indata i form av argument.

I assemblerprogrammering använder vi proceduren *subrutin* för dessa deluppgifter. Om vi vill returnera ett värde kan vi till exempel använda ett register för det. Subrutiner i sig har inget stöd varken för argument eller funktionsresultat.

Det är inte bara uppgifter i vardagslivet som kan delas upp i deluppgifter: Vi kan dela upp en komplex hårdvara i mindre delar med var för sig enklare funktion: Tangentbord, temperatursensorer, trafikljus,... Och vi kan dela upp ett större program i deluppgifter typiskt funktioner i C eller subrutiner i assembler.

När väl programstrukturen är klar på en första hög och grov nivå kan vi fortsätta med att dela upp subrutinerna i mindre subrutiner tills varje deluppgift verkar hanterbar.

Nu handlar det förstås inte om att allt blir bättre bara genom att dela upp allt i mindre delar. Det gäller att dela upp i begripliga, logiska delar med tydlig och entydig deluppgift.

En deluppgift ska inte delas upp bara för sakens skull. När dess uppgift är entydig och har enkel kommunikation med omgivningen kan den vidare uppdelningen upphöra.

Exempel Ett dataprogram kan nästan alltid delas upp i tre delar: inmatning, databehandling och utmatning. Var och en av dessa delar kan sedan delas upp i lagom stora deluppgifter efter erfarenhet, tycke och smak. Olika programmerare delar upp en uppgift på olika sätt.

Erfarna programmerare kan göra större uppgifter direkt, medan mindre erfarna programmerare gör sig själv en otjänst om de inte delar upp uppgiften.

Det är bättre att dela upp för mycket än för lite innan man är säker på hur bra programmerare man är. Det finns studier som visar att man inte kan ha överblick över mer än 5–7 rader kod samtidigt, vilket man skulle kunna tolka som att en funktion eller subrutin inte bör vara längre än så.

Exempel Att subrutiner är bra kan vi förstå om vi betraktar succén med hämtpizza. Hämtpizza uppfyller nämligen alla krav på en subrutin:

- det är en tydligt avskiljbar uppgift,
- man vill ha pizzan men bryr sig egentligen inte om detaljerna i produktionen, och
- den levereras när den är klar:

```
:
:
while( self->mode() != mätt)
     call hämtpizza();
:
:
```

2. Strukturera, strukturera, strukturera!

Vid programutveckling underlättar det väsentligt att ha en klar bild för sig av vad man egentligen vill göra! Det låter som en självklarhet — och är det naturligtvis också — men många tycker att programmering är något man gör vid tangentbordet.

Faktum är att programmering är något man gör när man hittat 1) lämpliga datastrukturer och 2) en metod att lösa problemet givet denna datastruktur.

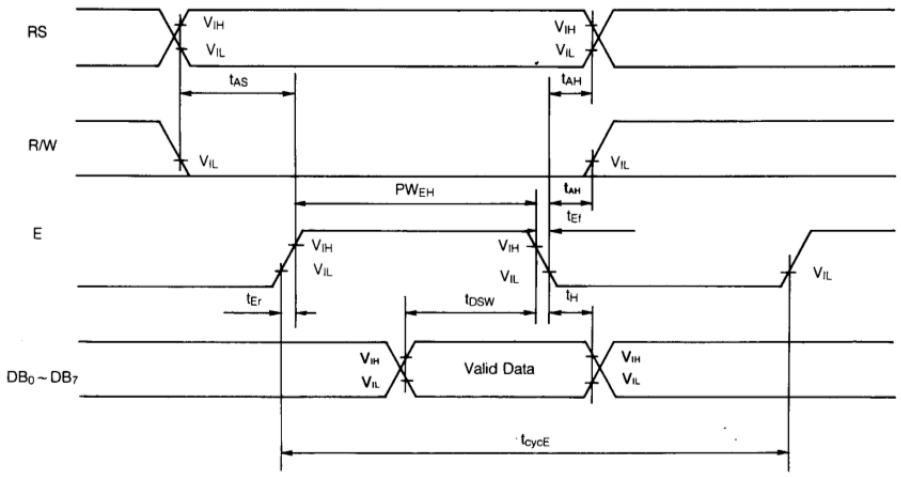
2.1. Exempel på kodförfining: Att programmera en LCD

Till en LCD-display kan man skriva ut ASCII-tecken. Det går också att skicka kommandon till den för att exempelvis rensa displayen, välja teckenposition, välja markör mm. Databladet ger vid handen att LCD:n måste initieras innan man kan skriva tecken till den.

En skrivning innebär att signalerna RS (*register select*), R/W (*read/write*), E (*enable*) och D0–D7 (databitar 0–7) används på rätt sätt enligt ett förutbestämt protokoll. Vid en skrivning måste R/W vara låg, E läggas hög, datat läggas ut och sedan sker själva skrivningen när E går låg igen. RS bestämmer om datat är ett tecken som ska skrivas ut på displayen (RS=0) eller ett kommando till displayhårdvaran (RS=1).

5.3 Timing Characteristics

Fig. 3 Write Operation Timing Diagram
(For data sent from the external microprocessor to the LCD unit)



Ett pulsschema ur databladet[1] visas ovan. Varje signal omgärdas av tidskrav (t_{AH} osv) som sekvensen måste uppfylla. Dessa tidskrav är ofta i storleksordningen nanosekunder, varför vi lämnar dem därhän. Här och nu koncentrerar oss på att varje skrivning ska ske enligt pulsschemat.

Ur pulsschemat kan skrivningen uttryckas[2] (där RS=0 och RW=1 antages vara normal- och viloläget):

```
Hitta datat som ska skrivas
Sätt RS=0
Sätt RW=0
Lägg ut datat
Sätt E=1
Sätt E=0
Ta bort datat
Sätt RS=0
Sätt RW=1
```

Sammanlagt nio programrader för varje tecken som skall skrivas. För att skriva ut ordet `Glass` kan alltså koden bli:

```
Data= 'G'              ; G       Sätt E=0
Sätt RS=0                        Ta bort datat
Sätt RW=0                        Sätt RS=0
Lägg ut datat                    Sätt RW=1
Sätt E=1                         Data= 's'              ; s
Sätt E=0                         Sätt RS=0
Ta bort datat                    Sätt RW=0
Sätt RS=0                        Lägg ut datat
Sätt RW=1                        Sätt E=1
Data= 'l'              ; l       Sätt E=0
Sätt RS=0                        Ta bort datat
Sätt RW=0                        Sätt RS=0
Lägg ut datat                    Sätt RW=1
Sätt E=1                         Data= 's'              ; s
Sätt E=0                         Sätt RS=0
Ta bort datat                    Sätt RW=0
Sätt RS=0                        Lägg ut datat
Sätt RW=1                        Sätt E=1
Data= 'a'              ; a       Sätt E=0
Sätt RS=0                        Ta bort datat
Sätt RW=0                        Sätt RS=0
Lägg ut datat                    Sätt RW=1
Sätt E=1
```

[1] LCD-drivkretsen 44780 är ymnigt förekommande. Se till exempel https://docs.isy.liu.se/pub/VanHeden/DataSheets/jm162a.pdf

[2] Lägg märke till att vi inte specificerar något programspråk här! Beskrivningen är helt oberoende av språket och utgör en sorts *pseudokod*.

Här har man gjort det enkelt för sig genom s.k. *copy-paste*-programmering. Detta ger dålig kod och metoden har flera nackdelar, bland annat:

- Det blir lång kod. Total $5 \cdot 9 = 45$ programrader i exemplet ovan.
- Det är oöverskådligt. En längre textsträng, menyalternativ e dyl, blir flera sidor lång.
- Programmeringen blir lidande eftersom man tvingas tänka på flera nivåer samtidigt, så fort något skall hända måste man ner i varje detalj och gröta.
- Det drar programminne, som i varje fall när man programmerar mikrocontrollers, är en knapp resurs.
- Det blir svårändrad kod. Tänk om man vill införa en liten fördröjning mellan `Sätt E=1` och `Sätt E=0`. Ändringen måste införas på hela fem ställen. Inte bra.

I det givna kodavsnittet är upprepningarna många och resultatet blir svåröverskådlig, svårändrad, svårunderhållen och helt enkelt *svår* kod.

Faktorisera En kort blick på programmet ger att flera sekvenser återkommer. Genom att *faktorisera*[3] ut dessa till funktionsanrop eller subrutiner får vi mycket kortare kod. Sekvensen `Sätt E=1` och `Sätt E=0` är en tydlig kandidat för faktorisering. Vi skriver alltså en rutin, **`do_E`**, för att göra just detta:

```
do_E {
     Sätt E=1
     Sätt E=0
}
```

Nu kan vi skriva koden som

```
Data= 'G'  ; G
Sätt RS=0
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RS=0
Sätt RW=1
Data= 'l'          ; l
Sätt RS=0
Sätt RW=0
Lägg ut datat
do_E
```

```
Ta bort datat
Sätt RS=0
Sätt RW=1
Data= 'a'            ; a
Sätt RS=0
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RS=0
Sätt RW=1
Data= 's'             ; s
Sätt RS=0
```

[3] I assembler: Tänk subrutin!

```
Sätt RW=0                          Sätt RS=0
Lägg ut datat                      Sätt RW=0
do_E                               Lägg ut datat
Ta bort datat                      do_E
Sätt RS=0                          Ta bort datat
Sätt RW=1                          Sätt RS=0
Data= 's'              ; s         Sätt RW=1
```

Fördelen är att vi skiljt ut detaljerna från helheten något. Exakt *vad* som händer i **do_E** är inte intressant för oss här, bara rätt sak händer. Det är intressant *någon* gång men just nu ska vi skriva ut ett ord och är inte överdrivet intresserade av om E går hög → låg eller låg → hög eller något annat. Bara rätt sak händer.

Med denna nya syn på koden kan vi se fler faktoriseringskandidater. En sådan är sekvensen

```
Sätt RS=0
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RS=0
Sätt RW=1
```

som återkommer gång efter annan. Vi kallar denna sekvens för **do_LCD_write**:

```
do_LCD_write {
     Sätt RS=0
     Sätt RW = 0
     Lägg ut datat
     do_E
     Ta bort datat
     Sätt RS = 0
     Sätt RW = 1
}
```

Nu blir programkoden väsenligt kortare:

```
    Data = 'G'  ; G
    do_LCD_write
    Data = 'l'  ; l
    do_LCD_write
    Data = 'a'  ; a
    do_LCD_write
    Data = 's'  ; s
    do_LCD_write
    Data = 's'  ; s
    do_LCD_write
```

Lägg märke till hur kodens själva huvuduppgift nu också framträder mycket tydligare. Ordet *Glass* är ju faktiskt uttydbart direkt ur koden.

Genom att på detta sätt faktorisera ut undernivåer underlättas läsligheten väsentligt. Med frasen *Är det värt att göra så är det värt att göra det i en subrutin* menar man att urskiljbara ”uppdrag” i en kod gör sig bäst i funktioner/procedurer/metoder.

En framtida vinst kan också visa sig då det oftare är lättare och renare att lägga till ytterligare funktionalitet i en subrutin än att göra det i huvudprogrammet. Det är till exempel inte svårt att hitta var man ska lägga till det utökade kravet ”Vid varje skrivning ska en lysdiod blinka till” om koden är faktoriserad.

Parametrisering Önskade vi bara skriva ut några få enstaka ord skulle vi kunna vara nöjda nu, men ofta vill man skriva ut mer text än så. Ett menysystem brukar till exempel innehålla ganska många texter och instruktioner, varför även detta sätt att koda visar sig vara onödigt svårhanterbart, även om den är väl faktoriserad.

Det som saknas är *parametrisering*: att förse funktionen eller proceduren med *argument*.

Hela rutinen hittills är ett enda upprepande av att hämta data och skriva ut den. Om vi anger data separat i minnet i form av en *sträng* kan vi låta programmet indexera över den strängen tills allt är skrivet.

```
    Text: 'Glass',0
LCD_ascii_print {
    Peka ut första tecknet
    Så länge tecken skiljt från 0 {
       Hämta tecknet
       do_LCD_write
       Peka ut nästa tecken
    }
}
```

Här har man varit tvungen att avsluta strängen med ett speciellt tecken, NUL-tecknet[4], för att markera att strängen är slut.[5]

Vi noterar nu att vi gjort oss helt fria från *texten* som skall skrivas, det anges på ett annat ställe. Detta betyder också att rutinen nu kan användas för *alla sorts texter*, inte bara *Glass*-texten. Vi har vunnit generalitet! Generalitet är bra!

Vi har också vunnit, men kanske inte lika uppenbart, att rutinen kommer användas flitigare och därmed också debuggas mer. Ju mer vi använder rutinen i olika sammanhang desto mer och mer säkra blir vi på att den gör rätt oavsett invärden. Denna inbyggda kvalitetstestning av rutinen gör hela programmet mer pålitligt!

Exempel Av någon anledning behöver man vid några tillfällen skriva ut två plustecken efter varandra, ”++”. För att åstadkomma detta kan man förstås skriva rutinen **`Print_plus`**:

```
Print_plus {
     Data='+'
     do_LCD_write
     Data='+'
     do_LCD_write
}
```

Det kanske duger — om man säkert inte kommer behöva skriva exempelvis *tre* tecken någon senare gång!

En annan lösning är att använda den färdiga rutinen **`LCD_ascii_print`** ovan med tillhörande sträng ”++” respektive ”+++”. Men hur är det om vi behöver skriva sju sådana tecken? Eller fem eller något annat antal? Det blir lätt en hel radda strängar. Klart opraktiskt.

Är det då inte bättre att skriva rutinen **`LCD_plusses`** med ett argument som talar om hur många tecken som ska skrivas ut:

```
LCD_plusses (N) {
     För alla N gör {
          Data = '+'
          do_LCD_write
     }
}
```

[4] ASCII-värdet `$00`, alltså inte ASCII-koden för `0` som ju är `$30`.

[5] En annan metod vore till exempel att låta strängen innehålla ett tal som anger strängens längd. Detta är dock inte lika smidigt eftersom man då också behöver hålla reda på strängens längd.

eller till och med

```
LCD_plusses (N) {
     Data = '+'
     För alla N gör {
          do_LCD_write
     }
}
```

(varför är denna senare snabbare att exekvera?) eller med ytterligare parametrisering, för att tillåta även andra tecken än plustecknet:

```
LCD_plusses (N,D) {
     Data=D
     För alla N gör {
          do_LCD_write
     }
}
```

Den parametriserade varianten kan användas till alla utskriftsönskemål men behöver bara skrivas en (1) gång. Det är en fördel.

2.2. JSP, Jackson Structured Programming

Hittills har vi gått igenom ett kodexempel där vi med lite eftertanke skrev om koden så den blev både kortare, överskådligare och enklare att modifiera om behovet skulle uppstå. Visst vore det bra om vi kunde komma fram till detta slutresultat direkt, utan att behöva gå omvägen att skriva om koden i flera steg? Det finns sådana metoder. Här beskrivs en av dem, JSP.

JSP-metoden för strukturerad programmering utvecklades och populariserades av engelsmannen Michael A. Jackson under 1960- och 1970-talen. I korthet handlar det om att hitta en *datastruktur* för problemet och sedan konstruera sitt program så att det följer den givna datastrukturen samtidigt som det löser problemet.

Att programmera tvärs emot datastrukturen blir nämligen ett elände. Varför simma uppströms och göra det krångligare för sig? Det tar säkert mer tid att lösa uppgiften och man försvårar för sig själv också.

Här ska vi presentera en ”light’-version av JSP med betoning på vikten av en bra programstruktur och hur man kan tänka då man ska lösa en programmeringsuppgift.

Notera att metoden innebär att en större del av programmeringsjobbet utförs redan *innan* man bestämt programspråk, eller ens närmat sig tangentbordet!

Detta kan kännas improduktivt[6] men är en förutsättning för att senare kunna koda lösningen.

JSP-metoden baserar sig på två huvudtankar:

1 Varje program består av tre olika sorters element, nämligen

- *Sekvens*. Sekvens innebär att saker händer efter varann. Tänk ”först händer A, sedan B och sist C” till exempel. Programrader som ligger efter varann är exempel på detta.

- *Iteration*. Att iterera betyder att göra någonting om och om igen. En iteration beskriver alltså något som sker upprepade gånger eller består av flera saker av samma sort. Då en iteration *kan* utföras *noll* gånger lämpar sig ofta `while`-satsen för detta. Asterisken anger att det är en iteration.

- *Selektion*. Selektion står för ett val. Ett val betecknar något som bryter rytmen. I ett program betyder selektion ett hoppvillkor (ofta `if`- eller `case`-sats). Ringen anger att det är en selektion. God programmeringssed i JSP säger att det också skall finnas ett villkorslöst alternativ vid en selektion. Denna är tänkt att fånga upp de fall som villkoren inte täcker upp.

2 Programstrukturen skall anpassa sig till den datastruktur som problemet innehåller. Ett första steg är alltså att identifiera den datastrukturen. Att finna datastrukturen är samtidigt bland de svåraste momenten. Har man, å andra sidan, funnit en lämplig datastruktur blir programmeringen lättare.

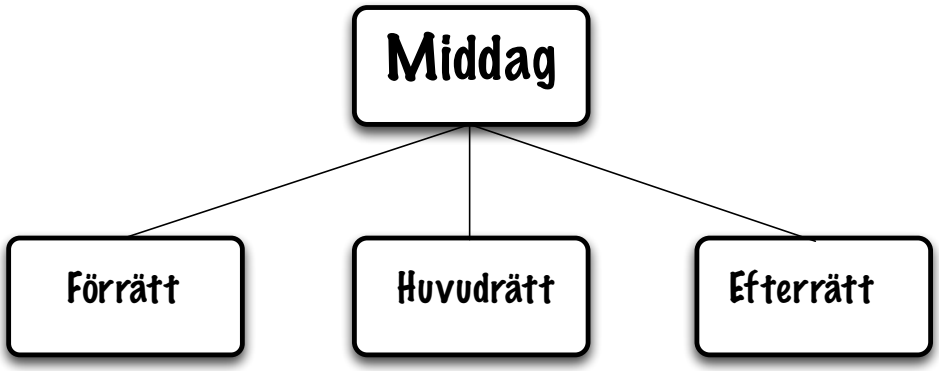
Med en datastruktur som beskriver problemets in- och utdata med de grundläggande elementen *sekvens*, *iteration* och *selektion* kan man sedan beskriva alla förekommande programmeringsuppgifter.

[6] För visst är det så att man inte känner att något *egentligen* händer innan fingrarna är på tangentbordet och kod börjar skrivas?

Vi tar några exempel på begreppen *sekvens*, *iteration* och *selektion*.

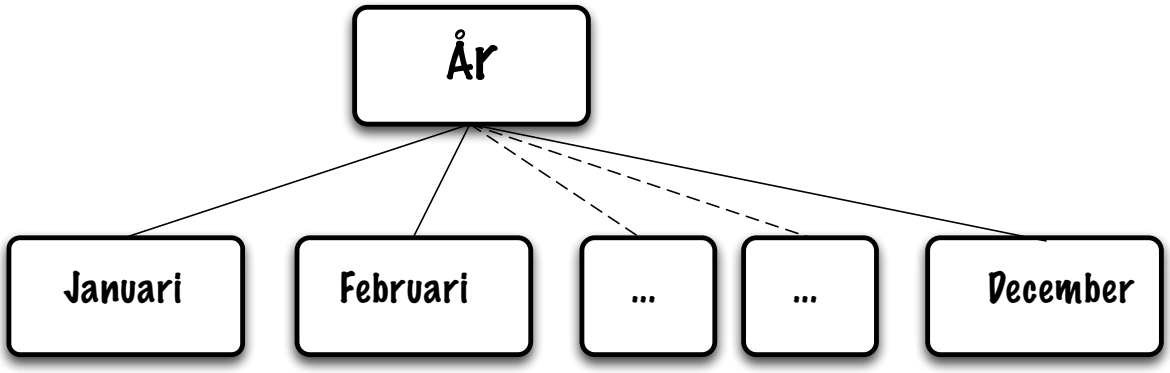
Exempel: Sekvens

- Datastrukturen över dagens tre huvudmåltider kan beskrivas som en *sekvens* av `Frukost`, `Lunch`, `Middag`.
- En `Middag` kan i sig beskrivas som en *sekvens* av `Förrätt`, `Huvudrätt` och `Efterrätt`.

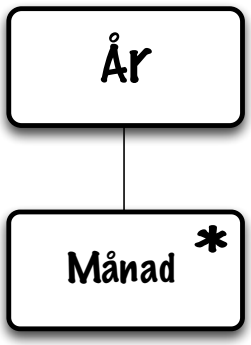


  Denna beskrivning betonar *sekvensen* mellan de olika delarna: Först `Förrätt`, därefter `Huvudrätt`, och sist `Efterrätt`. Enligt detta skrivsätt finns inga middagar som utelämnar `Förrätten` eller några andra delar. Vill man kunna utesluta förrätten måste det finnas ett val att göra det, se selektion nedan.

- Ett år består av månader:

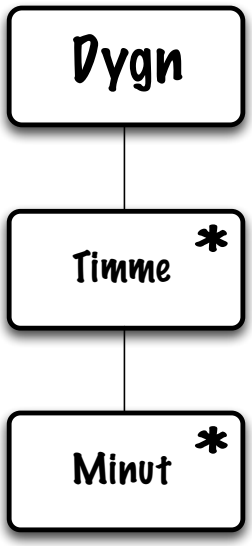


  . . . eller. . .

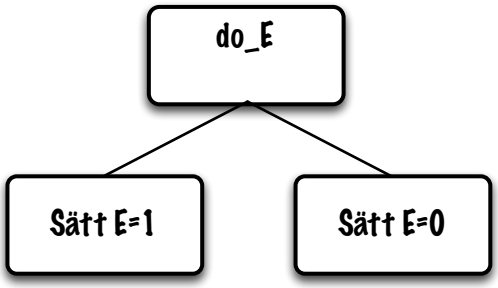


Det första sättet betonar *sekvensen*, den ger ju till och med *ordningen* (januari, februari osv). Det andra sättet anger bara att året består av ett antal månader, en *iteration* av månader. Den senare är användbar för att beräkna årssumma av hyran, ty då behöver man veta att den består av månatliga räkningar, det är inte nödvändigt att känna till månadernas namn för att lösa uppgiften. ”För alla månadsräkningar, summera!”.

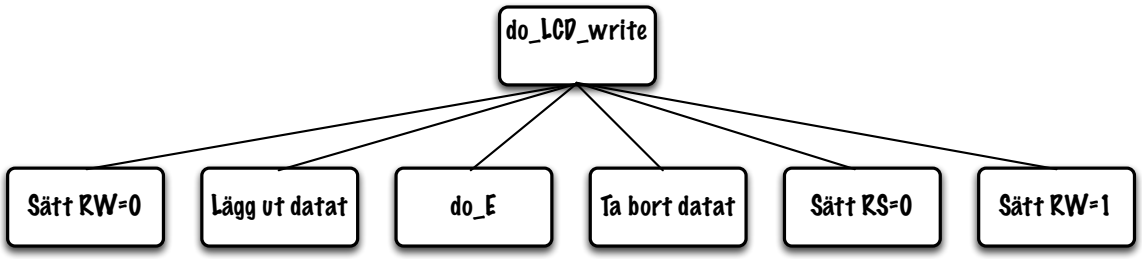
- Ett dygn består av (en iteration av) timmar som i sin tur består av (en iteration av) minuter.



- Den tidigare beskrivna proceduren **`do_E`** (sid. 11) kan skrivas

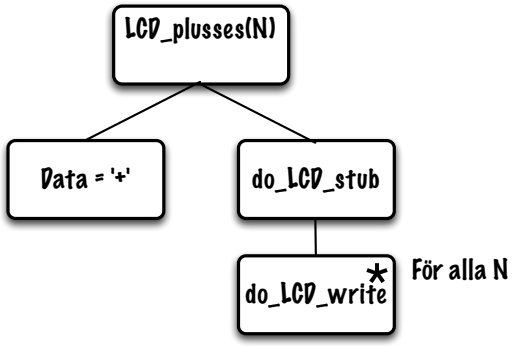


- På liknande sätt kan vi skriva **`do_LCD_write`**



Exempel: Iteration

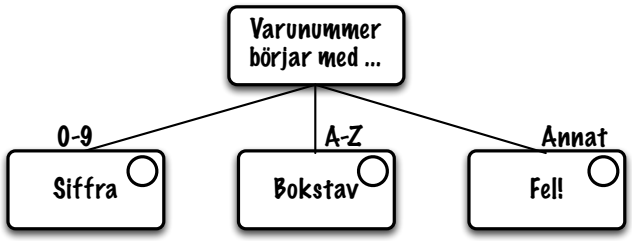
- Ett år kan ses som en *iteration* av månader.
- En promenad kan ses som en *iteration* av steg.
- Ett telefonnummer kan ses som en *iteration* av siffror.
- Vår tidigare **`do_LCD_Plusses`** (sid. 15) blir, med **`do_LCD_write`** *iterationen*:



Här kan man också se hur *iterationen och iterationsvillkoret ses som den itererade komponenten*. Med detta tankesätt skjuter man iterationen framför sig när man har fullt upp med att designa sekvensen. Ingen komponent kan vara två saker samtidigt, det vill säga den kan inte vara *både* sekvens och iteration.

Exempel: Selektion

- En vecka är en iteration av dagar men dagarna kan vara arbetsdagar eller lediga dagar. Här kan alltså dagarna vara av två typer dvs *selektion*.
- ASCII-tecken kan *antingen* vara bokstäver eller skiljetecken eller ”icke-skrivbara”.
- Ett varunummer kan börja med *antingen* en siffra eller en bokstav, inte båda. Och om det inte är någon av det vi förväntar oss måste programmet signalera `Fel!`.



Lägg märke till uttrycket *kan ses som* på några ställen ovan. Ett år skulle kunna ses som en iteration av dagar också, beroende på vad man har för avsikt att göra med informationen i sitt program.

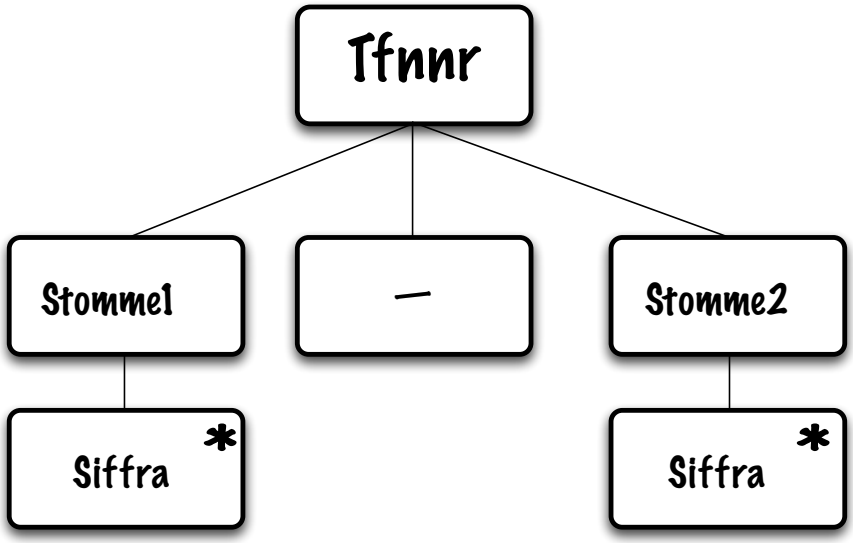
På samma sätt som det är förnuftigt att dela upp sitt program eller projekt i små delar[7] är det förnuftigt att dela upp sin datastruktur i delar *som är av betydelse för uppgiften*.

Om vi ska skriva en lista över årets månader är det uppenbart ointressant att låta månadens dagar ingå i datastrukturen, eller dagarnas namnsdagar eller annan helt överflödig information, även om den finns tillgänglig.

2.3. Ytterligare exempel

När vi nu studerat strukturdiagrammets komponenter kommer här några fler exempel och avslutningsvis övningar.

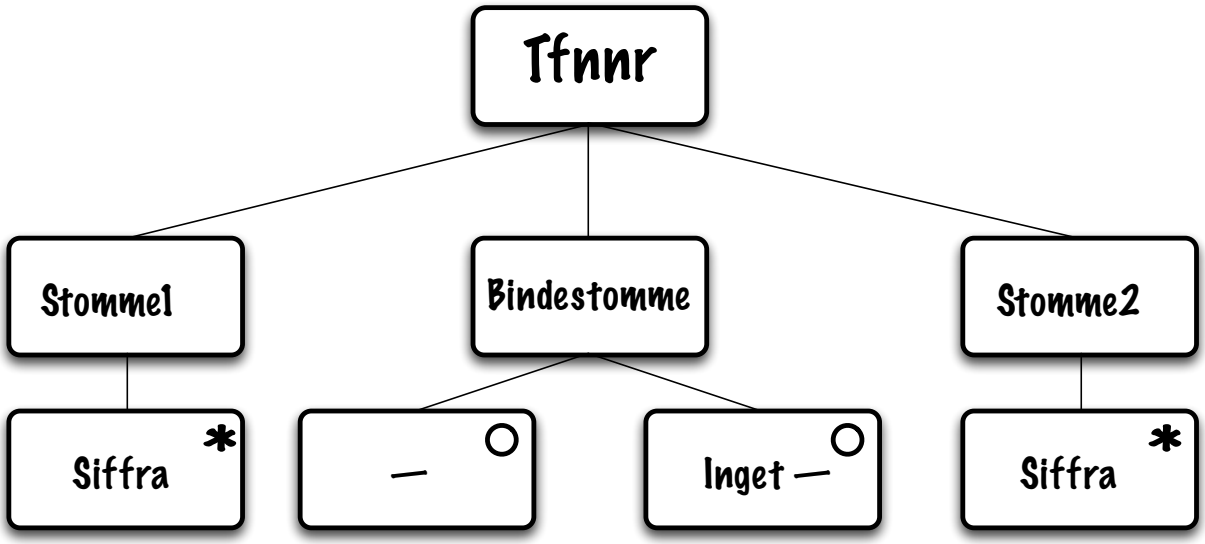
Exempel Ett telefonnummer ser ut som 013-281200, 0142-85000 osv. Datastrukturen blir då



Skrivsättet *tvingar* numret att ha ett bindestreck som separator mellan riktnummer och lokalnummer. Strängt taget passar denna beskrivning för alla nummer med ett bindestreck någonstans i sig. Men det får bara vara ett (1) bindestreck. Datastrukturen duger som vi förstår också för *alla* sifferraddor som består av siffror med ett enda bindestreck inom sig.

[7] men inte *för* små delar!

Om telefonnumret däremot kan se ut som 013-281200, 0142-85000 *eller* som 013281200, 014285000 blir datastrukturen:



Här har man infört möjligheten att separatorn (”—”) mellan riktnummer och lokalnummer finns eller inte finns. Båda möjligheterna är lika godkända.[8]

Vi ser att samma data kan ge olika datastrukturer beroende på vilken uppgift som programmet skall lösa. Häri ligger själva grunden till en bra programstruktur!

Med felaktig datastruktur — eller datastruktur som är motstridig mot uppgiften — kan lätt programmeringen bli en samling ”räddande av specialsituationer”. Är det något man vill undvika är det hantering av speciella fall. Går de att inlemma i en övergripande struktur blir programmeringen enklare.

För att konstruera en bra och användbar datastruktur är en förståelse för uppgiften och dess förutsättningar helt avgörande. Lägg därför ner tid på att hitta en lämplig datastruktur som är så förenklad som möjligt.

Med lite träning hittar man datastrukturer i allt möjligt i omgivningen. Träna på att göra sådana beskrivningar av saker omkring dig!

Övning Hur ser en labsal på universitet ut enligt JSP? På en översiktlig nivå kanske som en iteration av labplatser, men är det viktigt för uppgiften är den ju faktiskt delad i en höger och en vänstersida. Det finns två stolar vid varje labplats, var ska dessa in i JSP-grafen? Vad om det finns tre stolar (ibland)?

Övning En bok beskrivs enligt JSP som en iteration av sidor. Men det finns ju pärm fram och bak också! Alltså sekvens ”frampärm, iteration av sidor, bakpärm”. En sida beskrivs uppifrån och ned som ”marginal, text, marginal”.

[8] Anledningen till den ”onödiga” elementet `Bindestomme` är att ha något att haka fast *selektionen* bindestreck/inget bindestreck på, Inget element kan ju vara två saker. I detta fall skulle selektionen samtidigt vara sekvens om inte stommen infördes på detta sätt. Sådana här *stommar* införs emellanåt för att hålla strukturen ren.

Var skall sidnumret längst ner till höger fästas i JSP-grafen? Texten består av ord som i sig består av tecken. Tecken är bokstäver och siffror. Bokstäver är versaler och gemener. Versaler är A–Ä, gemener a–ö osv.

Detaljeringsgraden kan ökas *om det behövs för uppgiften*. Skall man skriva ett program som räknar antal böcker är kunskapen om pärmarna ointressant och skall inte ingå i datastrukturen.

Övning Beskriv din dag från morgon till kväll enligt JSP. Försök öka detaljeringsnivån till finare och finare detaljer. Vilken information behövs för att beräkna antalet timmar på universitetet? För att beräkna antalet hemarbetstimmar? Behövs samma information för att beräkna total studietid under dagen?[9]

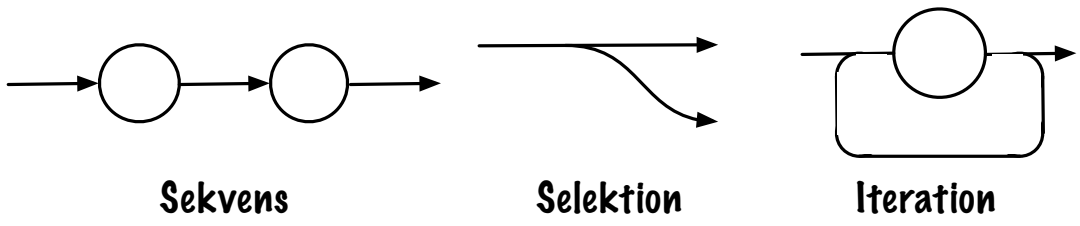
2.3.1. Övningar

För att låta det här sjunka in ordentligt är det lämpligt att träna på några exempel. Först kommer några *syntaxdiagram* som desperat vill översättas till strukturdiagram enligt JSP. Syntaxdiagram kanske kan vara bra att ha sett någon gång men här finns de med enbart som underlag till JSP-övningarna.

Syntaxdiagram

Ett syntaxdiagram är en beskrivning av vilka komponenter som bygger upp ett programmeringsspråk. Diagrammet kan vara ett steg i processen att skriva en kompilator för språket. Här ska vi använda syntaxdiagrammen för att konstruera motsvarande strukturdiagram.

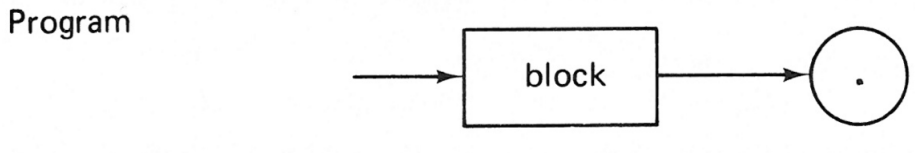
Man läser ett syntaxdiagram från vänster till höger och strukturkomponenterna *sekvens*, *iteration* och *selektion* återfinns i diagrammet fast inte i JSP-form.



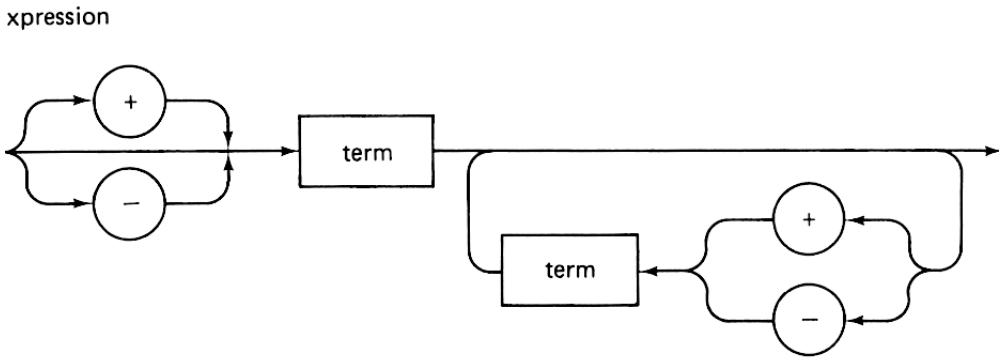
[9] Nej. Uppdelningen mellan undervisningstid och hemarbetstid är onödig information i detta fall.

PL/0

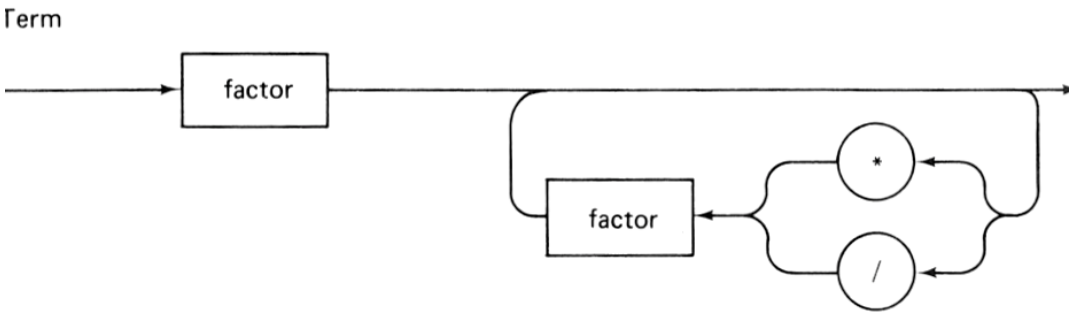
Nedan visas några syntaxdiagram att översätta till JSP-notation. Diagrammen definierar exempelspråket PL/0.[10]



*Ett PL/0-program består av ett **block** följt av en punkt.*

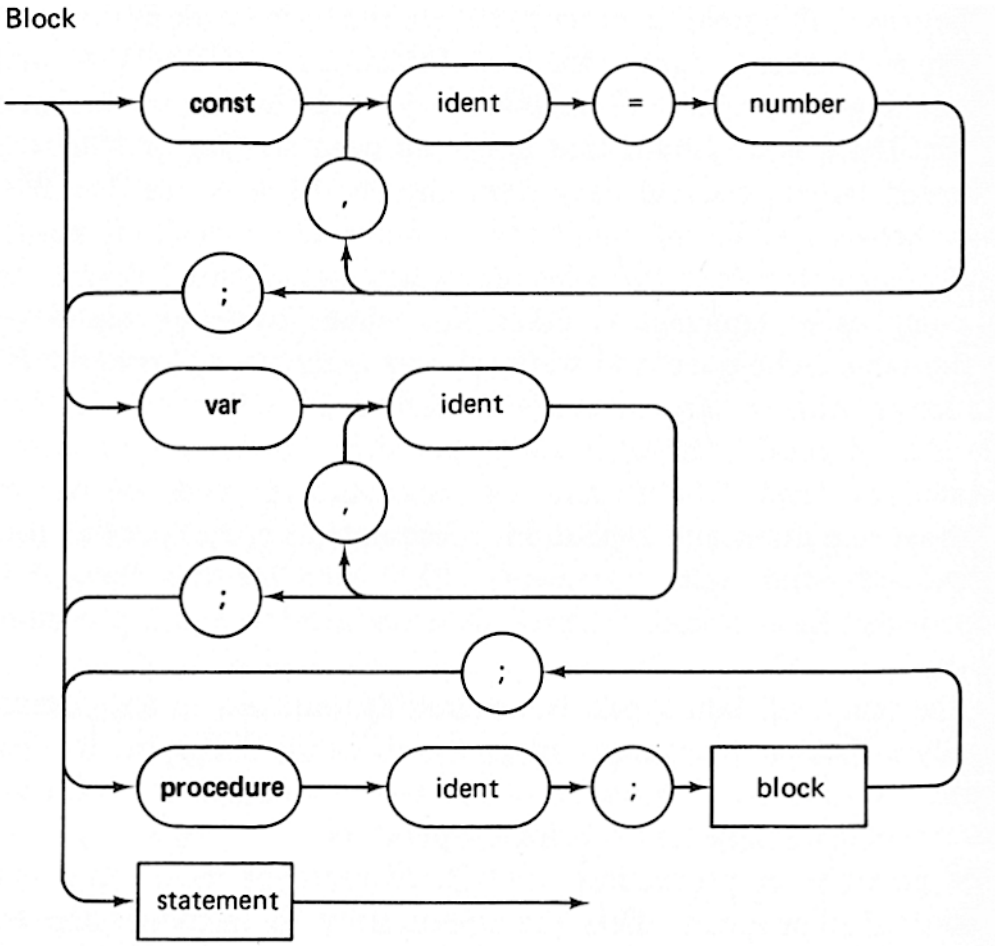


*Ett **expression** inleds med **+**, **-** eller en **term** och kan följas av **+** eller **-** och en term.*

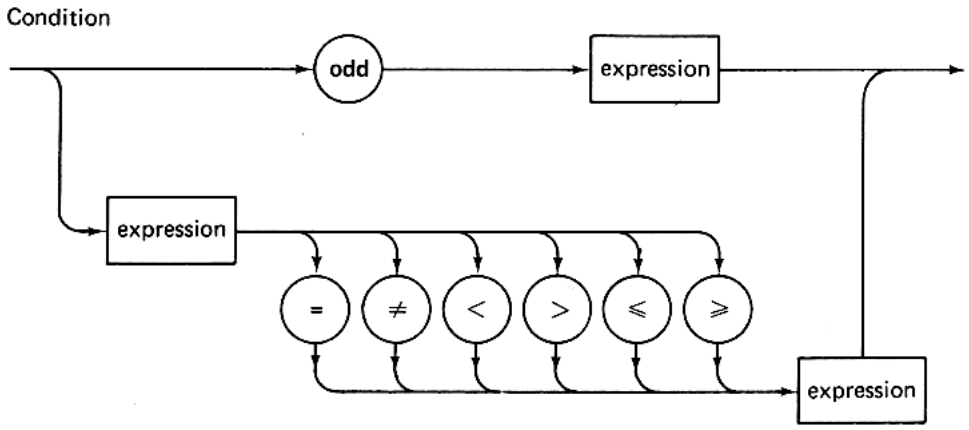


*Syntaxdiagram för en **term**.*

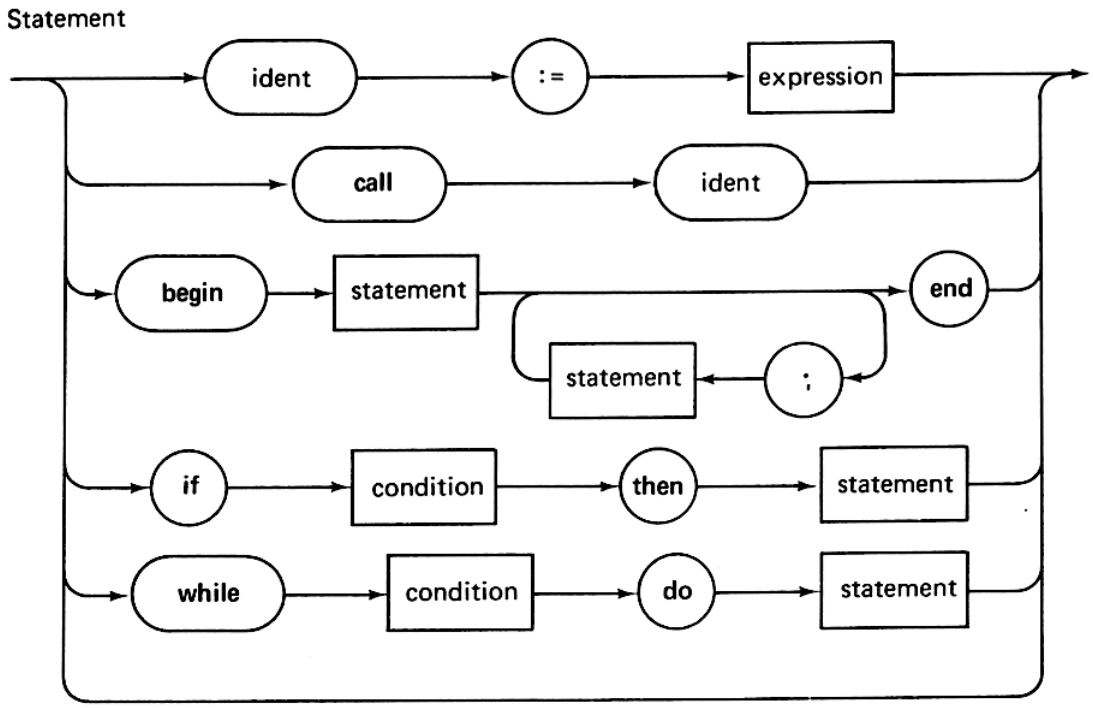
[10]Wirth, *Algorithms + Datastructures = Programs*



*Ett **block** börjar antingen med en konstantlista efter **const**, variabellista efter **var**, procedur **procedure** eller ett **statement**. Variabellistan utgörs av kommaseparerad lista av identifierare **ident** och raden avslutas med ett semikolon ;. Övriga språkelement konstrueras analogt.*



*Syntaxdiagram för ett **condition**.*



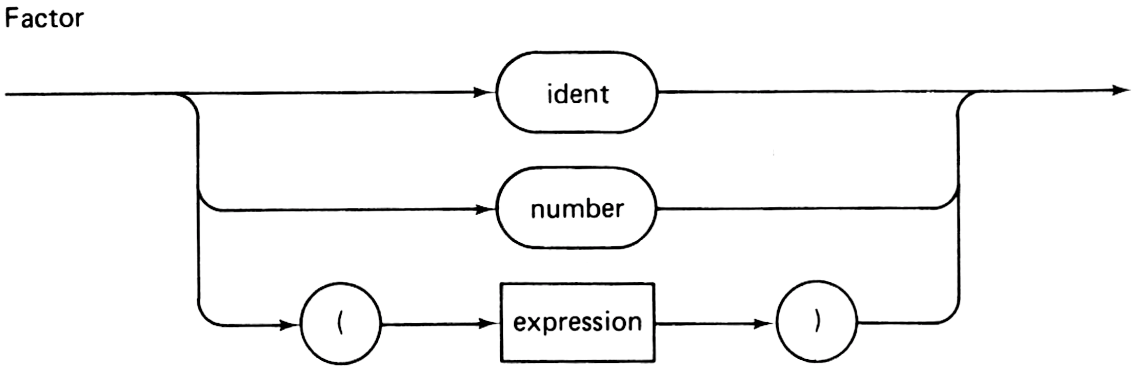
*Syntaxdiagram för ett **statement**. Här känner vi igen språkelement tillhörande ett högnivåspråk.*

2.4. Skriva kod

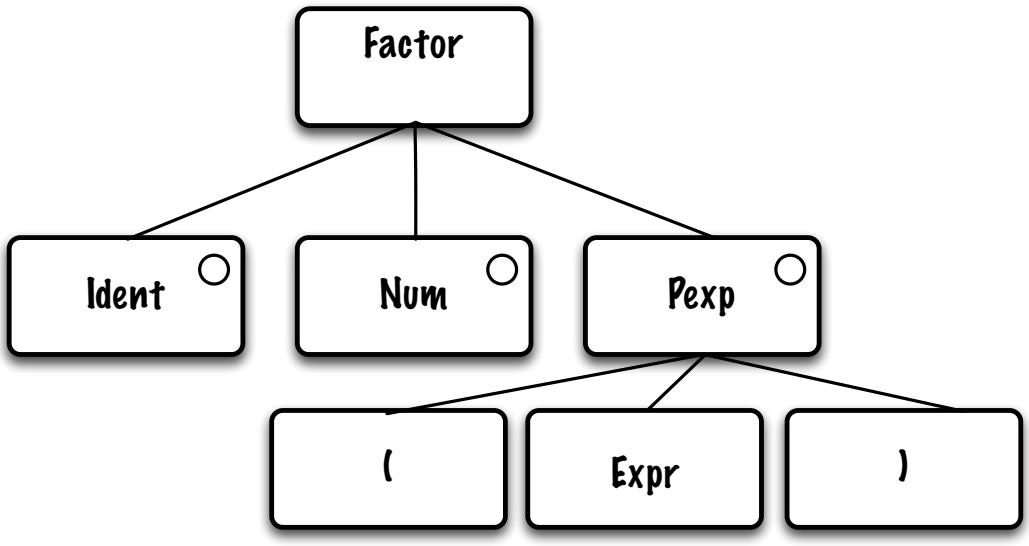
När hela konstruktionen är färdigtänkt och formulerad i JSP-notation återstår att skriva den motsvarande koden i det önskade programmeringsspråket. Med korrekt förarbete är den slutliga översättningen till kod en relativt enkel affär. Det räcker som bekant att kunna skriva kod för de tre elementen *sekvens*, *selektion* och *iteration*!

Selektion och sekvens

I PL/0 beskrivs programkomponenten **factor** med ett syntaxdiagram



Vi ska använda detta syntaxdiagram som exempel för att visa hur *selektion* och *sekvens* kan översättas till ett C-liknande språk och slutligen ända ner till AVR-/assemblerkod.



Vi ser att en `factor` kan bestå av antingen en *ident*, *number* eller ett ”*expression* inom parentes”.

I ett högnivåspråk kan vi översätta detta diagram till kod på några olika sätt. En möjlighet är nästlade `if`-satser, eller sekvensiella om man tycker att nästlade `if`-satser blir för svårlästa eller oöverskådliga om nästlingsdjupet blir stort, en annan möjlighet är en `switch/case`-sats.

```
void factor(int s){
   if (s == IDENT)
      ident;
   else if (s == NUM)
           num;
        else if (s == PEXP){
                lparen;
                expr;
                rparen;
             }
         }
}
```

```
void factor(int s){
   if (s == IDENT) ident;
   if (s == NUM) num;
   if (s == PEXP){
      lparen;
      expr;
      rparen;
   }
}
```

Det högra alternativet ger enklare kod men ställer krav på att villkorsuttrycket inte får matcha flera `if`-satser.

I vissa språk finns `switch`-satsen som ger ett ännu mer lättläst intryck (lägg märke till användningen av `default` för att fånga upp oväntade värden på `s`):

```
void factor(int s){
   switch(s) {
      case IDENT: ident;
         break;
      case NUM: num;
         break;
      case PEXP: {
         lparen;
         expr;
         rparen;
      }
         break;
      default:   error;
   }
}
```

Utan `switch`-sats, men med mer manuellt arbete, kan exakt samma funktion erhållas med omsorgsfullt använda `if` och `goto`:

```
void factor(int s){
   if (s != IDENT) goto facNum
   ident
   goto facDone
facNum:
   if (s != NUM) goto facPexp
   num
   goto facDone
facPexp:
   if (s != PEXP) goto facError
   lparen
   expr
   rparen
   goto facDone
facError:
   error
facDone:
}
```

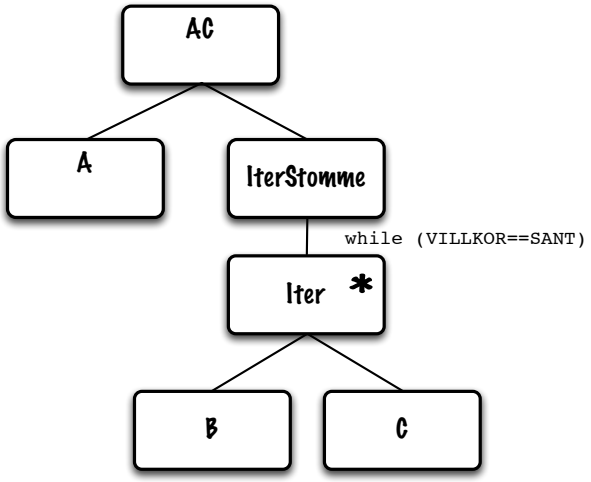
Detta skrivsätt är direkt översättningsbart till AVR-assembler där vi för enkelhets skull antar att `r16` innehåller argumentet `s` ovan och att värdena `cIDENT`, `cNUM` och `cPEXP` är definierade tidigare:[11]

```
factor:
   cpi    r16,cIDENT
   brne   facNum
   call   ident
   jmp    facDone
facNum:
   cpi    r16,cNUM
   brne   facPexp
   call   num
   jmp    facDone
facPexp:
   cpi    r16,cPEXP
   brne   facError
   call   lparen
   call   expr
   call   rparen
   jmp    facDone
facError:
   call   error
facDone:
   return
```

[11] Detta assemblerspråk skiljer tyvärr inte på versaler och gemener så för att inte konstanten `IDENT` skall kollidera med rutinen `ident` markeras konstanten med inledande `c`.

Iteration

I JSP motsvaras komponenten *iteration* av en `while`-sats. I en `while`-sats utförs det som följer så länge ett villkor är sant men kan också utföras *noll* gånger om villkoret inte skulle vara logiskt sant. Som exempel på hur kod skrivs för iteration tar vi följande strukturdiagram där rutinen `A` antas ge något villkor att testa på för *iter*-delen. (Övning: Skriv själv den sekvens av `A`, `B` och `C` som kan genereras av strukturdiagrammet. Vi ser att `AC` måste inledas med ett `A` följt av en iteration och så vidare.)



I ett C-liknande högnivåspråk kan diagrammet översättas till

```
void AC(void){
     A;
     while (VILLKOR){
          call iter;
     }
}
```

eller mer direkt, för att undvika ett extra `call`:

```
void AC(void){
     A;
     while (VILLKOR){
          B;
          C;
     }
}
```

Här måste vanligtvis `iter`, det vill säga, `B` eller `C`, påverka villkoret så programmet kan avslutas någon gång.

Och om vi går ner på AVR-assemblernivå ser vi att `while`-satsen motsvaras av kombon `cp/brne` och en avslutande `jmp`.[12]

```
AC:
     call A
ACAgain:
     cp   r16,cVILLKOR
     brne ACDone
     call B
     call C
     jmp  ACAgain
ACDone:
     return
```

Vi har nu sett hur JSP-strukturerna *sekvens*, *selektion* och *iteration* kan skrivas på några olika sätt ända ner till assemblernivå. Slutsatsen är att själva kodningen blir närmast ett standardförfarande om bara strukturdiagrammen är korrekta. Vi har samtidigt erhållit fördelen att koden är strukturerad, enklare att modifiera och troligen lättare att följa.

I allmänhet: *Man löser problem **bättre** och man felsöker **bättre** om man vet **mer** om systemet än om man vet precis så mycket som behövs för att lösa uppgiften.*

2.5. Några programmeringstips

Indentering Något av det viktigaste vi kan göra med vår kod är att *indentera* den på ett begripligt sätt. Med indentering menas att sätta in mellanslag, eller ännu hellre tabbar (vanligen motsvarande 3–8 mellanslag), så att liknande språkelement börjar i samma kolumn. För programspråk som C, ADA, JAVA med flera, finns speciella skrivregler man skall hålla sig till. I assemblerprogram brukar man däremot inte ha mer än en enda indenteringsnivå varför uppdelning i subrutiner (faktorisering) är mycket viktig.

Symboliska adresser (*labels*) skall börja i kolumn 0 och avslutas med ett kolon ':'. För tydlighets skull brukar man låta en sådan ha en egen rad.

Instruktioner börjar ett tab-indrag från vänsterkolumnen och därefter kommer instruktionens argument, ofta även dessa efter ytterligare ett tab-indrag.

[12] `cp` jämför register med register, `cpi` jämför register med en konstant.

```
AC:
     call A
ACAgain:
     cp   r16,cVILLKOR
     brne ACDone
     call B
     call C
     jmp  ACAgain
ACDone:
     return
```

Indenterad kod är lättläst.

```
AC: call A
ACAgain: cp r16,cVILLKOR
brne ACDone
call B
call C
jmp  ACAgain
ACDone: return
```

Icke-indenterad kod svårläst.

DRY, *Don´t Repeat Yourself* Ett program bör vara lätt att förstå och modifiera. En väg mot detta är konceptet **DRY** som betyder *Don´t Repeat Yourself*. Med detta menas att man inte ska sprida programfunktioner över koden utan låta en programfunktion utföras på ett och endast ett ställe. Behöver man utföra samma sak på flera ställen löser man det med anrop till rutinen där funktionen är definierad. Man kan lätt föreställa sig de problem som uppstår om en åtgärdad bugg i en funktion senare visar sig dyka upp på helt annat ställe i koden — i en annan funktion. Uppenbarligen finns samma bugg på två ställen. Kanske finns den på fler? Tre, fyra ställen? Fler? Ska man behöva gå igenom hela kodbasen för att leta efter samma bugg? Naturligtvis inte: En bugg skall kunna rättas på ett och endast ett ställe.

Parametrisera så långt det går Exempel: Du behöver vänta tre olika tider: 2, 40 och 1000 ms. En första ansats är att skriva en rutin som tar 1 ms och sedan använda den två gånger för att få 2 ms-fördröjningen. Med denna kunskap kanske man kan räkna sig fram till hur de övriga rutinerna ska se ut och man skriver dessa tre, **`wait2ms`**, **`wait40ms`** och **`wait1000ms`**.

En alternativ, och bättre, lösning är att göra en *generell* vänteloop som tar 1 ms och sedan loopa denna i de tre rutinerna och använda dem med argumenten `2`, `40` respektive `1000`. En ytterligare förenkling är att låta dessa argument definieras som konstanter i början av programmet. Fördelarna är tre:

1. Det är lätt att hitta värdena om de skulle ändras,
2. Det är lätt att lägga till ytterligare vänterutiner, det är egentligen bara att definiera upp en konstant till, och slutligen

3. Om till exempel processorns klockfrekvens skulle behöva ändras är det bara att ändra konstanterna proportionellt och programmet kommer fungera *exakt* som förut.

Man skulle till och med dra det ända till att argumenten *beräknas* utifrån den nuvarande klockfrekvensen — då behöver man inte ens tänka på konstanterna om man skulle behöva ändra klockfrekvens, bara ändra klockfrekvensen och konstanterna räknas om med automatik.

Lokala variabler Lokala variabler är sådana som bara existerar under en funktion eller procedurs livslängd. Förutom i särskilda minnesplatser kan dessa lokala variabler finnas i processorns interna register. I en högnivspråksfunktion finns redan stöd för lokala variabler medan man i assembler själv måste hålla ordning på dem.

Generellt skall man eftersträva att låta variabler vara lokala så ofta som möjligt. Det är onödigt och förvirrande att låta funktionsinterna variabler deklareras som globala. Det är ju bara funktionen som behöver känna till dem. I många högnivåspråk är detta enkelt att åstadkomma.

I assembler är man själv ansvarig för hanteringen. En lämplig strategi är att varje subrutin lagrar undan de processorregister rutinen ”kontaminerar” (`push`) på returstacken vid rutinens start. Registren måste återställas innan subrutinens avslutande retur (`pop`).

På detta sätt behöver inte den *anropande* koden känna till något om underliggande kods registerutnyttjande. Speciellt tillåter detta att en *anropad* rutin modifieras eller helt skrivs om utan att *anropande* koden behöver förändras.

Med både DRY och väl utnyttjade lokala variabler i sitt program erhåller man en isolation mellan olika funktioner/rutiner i sitt program som närmar sig ett objekt-orienterat programmeringstänk.

Övergripande programlayout Förutom att *labels* börjar i kolumn 0 och instruktionerna ett tab-indrag (motsvarande 3–8 mellanslag) in från vänsterkolumnen skall koden utföras enligt följande:

- Globala konstanter och definitioner skall specificeras i programmets början med `.equ` respektive `.def`
- Huvudprogrammet inleds ofta med ett hopp till kallstart, `COLD:`
- Omstart av programmet som inte kräver hårdvaruinitialisering, s k varmstart, går ofta till `WARM`:
- En subrutin skall ha endast **en** ingång (*entry point*) och **en** utgång (*exit point*) med gemensam uppsamlingspunkt innan retur.
- Lokala register skall frigöras genom att använda stacken (`push` och `pop`).
- Absoluta hopp (`br*`, `jmp`, `rjmp`) är förbjudna **mellan** subrutiner och får endast användas **inom** en subrutin

Typutseende för ett program med inledande definitioner, kall- och varmstartsadresser blir enligt ovan:

```
    .equ    ...     <--- Inledande konstanter
    .def    ...
    |
COLD:               <--- Kallstart
    |
    call    INIT    <--- Initieringar
    |
WARM:
    call    THIS    <--- Subrutinanrop
    call    THAT    <--- Subrutinanrop
    |
    |
    jmp     WARM    <--- Varm omstart
```

Strukturerad programmering enligt JSP underlättar även för att få en snygg och läsbar programkod då strukturdiagrammet i huvudsak översätts till subrutiner.

Lägg speciell vikt vid namngivning av subrutiner. En subrutin måste göra exakt det den heter. Om det visar sig att att en subrutin med tiden fått flera uppgifter kan en uppdelning i flera subrutiner med lämpliga namn lösa den knuten. Om denna situation uppstår kan det också vara idé att se över om det ursprungliga strukturdiagrammet faktiskt löste uppgiften eller måste arbetas om.

En korrekt sammanställd och uppsnyggad subrutin har en ingång, en utgång, lokala register och innehåller hopp enbart inom rutinen:

```
OK:
    push    r16     <--- Spara lokala register
    |
OK_2:
    |
    breq    OK_1    <--- Hopp inom rutin
    |
    jmp     OK_3    <--- Hopp inom rutin
    |
    jmp     OK_2    <--- Hopp inom rutin
    |
OK_1:
    |
    brne    OK_3
    |
    jmp     OK_2    <--- Hopp inom rutin
    |
OK_3:               <--- Uppsamlingspunkt
    |
    pop r16         <--- innan retur
    ret
```

Det är **inte** korrekt att ovanstående rutin fördelas på åtskiljda platser i den färdiga koden även om den en gång i tiden liknade detta:

```
OK:
    push    r16                 OK_3:
    |                               |
OK_2:                               pop r16
    |                               ret
    breq    OK_1
    |                           OK_1:
    pop     r16                     |
    ret                             brne    OK_3
    |                               |
    jmp     OK_2                    jmp     OK_2
```

När en rutin fungerar och liknar det ovan måste den i ett avslutande steg snyggas upp. Härvid uppnår man flera saker: rutinen blir **en** enda, upprepade kodavsnitt elimineras (`pop r16, ret` ovan) och lämpliga kommentarer tilläggas.

Optimera registeranvändning och få ner antalet hopp inom rutinen om det är möjligt. Vid tveksamhet: Hellre lättläst, underhållsbar och modifierbar kod än *för* optimerad kod.

3. Referenser

När det gäller JSP har vi nosat på en del av metoden. Hela JSP kräver en kurs i sig.

Läs vidare om JSP:

- `http://proceedings.informingscience.org/ /IS2003Proceedings/docs/091Ourus.pdf`
- `https://www.scribd.com/document/49518016/JSP`

I bokform finns dessa:

- Jackson M A, *Principles of program design*, Academic Press 1975
- Ingevaldsson L, *JSP — En praktisk metod för programkonstruktion*, Studentlitteratur, 1984
- Mattson H, Rejler O, *JSP från början*, Studentlitteratur 1991.

När det gäller god programmeringsstil och programmeringstänk har *Uncle Bob*, Robert C. Martin, många åsikter. Han är författare till den läsvärda boken *Clean Code*.

*Uncle Bob*s framträdanden på olika konferenser brukar läggas ut på nätet och är väl värda att lyssna på!

- Allmänt om god programmeringstil
  `https://vimeo.com/12643301`
- Funktionell programmering i perspektiv
  `https://vimeo.com/97514630`
- Om programmeringsparadigmer
  `https://vimeo.com/43659004`
- Hans bok *Clean Code* finns som pdf
  `http://ricardogeek.com/docs/clean_code.pdf`
- ”The Wizard”-boken han nämner finns här
  `https://mitpress.mit.edu/sicp/`

A. Kodexempel

Med bra namn på variabler och funktioner kan koden bli så lättläst att kommentarer ofta är överflödiga. Med en bra programstruktur underlättas både samarbete kring en programmeringsuppgift och felsökning samtidigt som man kan vara rätt säker på att programmet faktiskt löser den uppgift man vill.

A.1. En sak i taget

Som exempel på en funktion som griper över för mycket och kan uppdelas används här exempel G30 ur *Clean Code* (Robert C Martin). Språket är java men samma resonemang gäller i andra språk och *inte minst i assembler*.

Funktionen (eller *metoden* för att prata java) går igenom en lista på anställda och, om det är avlöningsdag för denne, beräknar lönen:

```
public void pay(){
    for (Employee e : employees){
        if (e.isPayday()){
            Money pay = e.calculatePay();
            e.deliveryPay(pay);
        }
    }
}
```

Vet man vad funktionen gör känns den ”rimlig” men tittar man noggrannare kan man konstatera att den gör flera olika saker och bör delas upp för att dessa olika saker ska bli frikopplade från varann[1].

Man kan urskilja några separata moment i koden:

1. För varje anställd gör...
2. Om det är avlöningsdag gör...
3. Beräkna lönen.
4. Betala lönen

En möjlig uppdelning är då

```
public void pay(){
    for (Employee e : employees)
        payIfNecessary(e);
}
```

[1] Motsvarande *ortogonalitet* i lineär alebra.

```
public void payIfNecessary(Employee e){
    if (e.isPayday())
        calculateAndDeliverPay(e);
}

public void calculateAndDeliverPay(Employee e){
    Money pay = e.calculatePay();
    e.deliveryPay(pay);
}
```

Här har man valt att slå ihop punkterna 3 och 4 ovan till en funktion. Argumentet för detta kan vara att man inte gör en beräkning av lönen utan att också betala ut den? Köper man inte det argumentet kan naturligtvis även de två sista funktionerna sönderdelas:

```
public void payIfNecessary(Employee e){
    if (e.isPayday())
        calculatePay(e);
}

public Money calculatePay(Employee e){
    Money pay = e.calculatePay();
    return pay
}

public void calculateAndDeliverPay(Employee e){
    e.deliveryPay(calculatePay(e));
}
```

Det finns ingen anledning att faktorisera bara för faktoriseringens skull. Den sista faktorisering är så långt man kan gå, men har läsbarheten blivit bättre? Det tycker inte jag.

Personligen skulle jag nöjt mig med en grövre uppdelning från början:

```
public void pay(){
    for (Employee e : employees)
        payIfNecessary(e);
}

public void payIfNecessary(Employee e){
    if (e.isPayday()){
        Money pay = e.calculatePay();
        e.deliveryPay(pay);
    }
}
```

Resonemanget bakom detta designbeslut är att ett visst jobb ska utföras per anställd. Utbetalningsdag hör ihop med beräkning av lön och faktisk utbetalning så det är en lämplig hophållen programfunktion. Genom att utföra den i

en enda funktion är också *tidsberoendet* tydligt angivet, programmet beräknar lönen `e.calculatePay()` innan utbetalning `e.deliveryPay(pay)`. Omvänd ordning är omöjlig, till skillnad från den mest finfördelade versionen ovan.

Som synes finns det olika sätt att koda och man utvecklar med tiden en personlig stil. Ingen kodstil är nödvändigtvis ”rätt” och ingen nödvändigtvis ”fel” om man håller sig inom ramarna. Programmet ska dock alltid vara så lätt att läsa som möjligt, felsökningsbart och utbyggbart vid senare tillfälle (kanske av en annan programmerare).

A.2. Faktorisering och bra namn?

Inte sällan är man så inne i programmeringen att man inte lägger särskilt stor vikt vid variabel- och funktionsnamn. I vissa fall följer koden en matematisk beskrivning och då kan det matematiska språket skina igenom till exempel med korta indexvariabler `i, j` för $x_{i,j}$.

Här kommer ett exempel i C++[2]. En klass för en *sist-in-först-ut*-stack för ASCII-tecken kan skrivas:

```
#include <iostream>
using namespace std;

class Stack {
public:
    void emptyStack() {
        top = EMPTY;
    };
    void pushToStack(char c) {
        s[++top] = c;
    };
    char popFromStack(){
        return s[top--];
    };
    bool isEmptyStack() const {
        return (top == EMPTY);
    };
    bool isFullStack() const {
        return (top == FULL);
    };
private:
    enum {
        MAX_LEN = 100,
        EMPTY   = -1,
        FULL    = MAX_LEN - 1
    };
```

[2]Kod insprirerad från *C++ by Dissection* (Ira Pohl), sid 187.

```
    char s[MAX_LEN];
    int top;
};
```

Redan klassens namn `Stack` avslöjar att det handlar om en stack (duh!). I detta fall är det en stack avsedd för `char`s så man kan överväga om namnet `Char_Stack` skulle vara lämpligare? Här används bara klassen till en sak så det extra förtydligandet känns onödigt.[3]

De ingående metoderna är namngivna för tydlighet som

- `emptyStack`
- `pushToStack`
- `popFromStack`
- `isEmptyStack`
- `isFullStack`

Metodnamnen beskriver exakt vad de gör och varje metod gör bara en sak. Att metoden `isEmptyStack` returnerar ett logiskt värde som svar på frågan om stacken är tom är också tydligt.

Efter en tid blev det dock tjatigt att alla metoder heter ”någonting`Stack`”, hela klassen handlar ju om en stack så denna namngivning känns pratig och övertydlig. Efter lite övervägande reducerades namnen till

- `empty`
- `push`
- `pop`
- `isEmpty`
- `isFull`

som är kortare och otvetydiga då de är standardmanipulationer för en stack.

Klassen ska användas för att skriva ut en textsträng baklänges. Ett program som löser denna uppgift är:

```
int main(){
    Stack s;
    char string[40] = { "Hej hopp i lingonskogen" };
    int  i = 0;

    cout << string << endl;
    s.empty();
    while (string[i] && !s.isFullStack())
        s.push(string[i++]);
    while (!s.isEmpty())
```

[3] Tycker *jag* i alla fall.

```
        cout << s.pop();
    cout << endl;
};
```

En dissektion av programmet ger dess funktion stegvis:

- Skriv ut strängen med en avslutande radbrytning

  ```
  cout << string << endl;
  ```

- Töm stacken

  ```
  s.empty();
  ```

- Pusha strängen på stacken

  ```
  while (string[i] && !s.isFullStack())
          s.push(string[i++]);
  ```

- Skriv ut stackinnehållet tecken för tecken

  ```
  while (!s.isEmpty())
          cout << s.pop();
  ```

- Avsluta med en radbrytning

  ```
      cout << endl;
  ```

En körning av programmet ger alltså strängen följt av samma sträng skriven bakänges:

```
Hej hopp i lingonskogen
negoksnognil i ppoh jeH
```

Så långt allt väl. Programmet fungerar. Men lite strukturerad programmering i ryggen ser man att programmet, som det står, dock är rätt fult: Det blandar saker på olika nivåer. På den övre nivån instansieras `s` (`Stack s`), strängen definieras och skrivs. På nästa nivå initieras sedan stacken och med ett index överförs strängen till stacken så länge strängen inte är slut och stacken inte är full. På denna nivå skrivs sedan strängen ut från stacken. Avslutningsvis är programmet tillbaka på övre nivån — för att skriva ut en radbrytning.

Som vanligt vill vi undvika detaljer[4] varför vi kan formulera om koden. Programmets uppgift är att skriva ut en sträng baklänges, låt då det framgå på den översta nivån, det vill säga skriv om `main()` till:

```
:
cout << string << endl;
printStringReversed(string);
cout << endl;
:
```

[4]Här `string[i] && !s.isFull()` och `s.push(string[i++])` till exempel

Faktoriseringen gav den nya funktionen:

```
void printStringReversed(char string[]){
    Stack s;

    s.empty();
    int  i = 0;
    while (string[i] && !s.isFull())
        s.push(string[i++]);
    while (!s.isEmpty())
        cout << s.pop();
}
```

som har fördelarna

- funktionen gör vad den heter,
- variabeln `i` är nu lokal och
- detaljerna är borta från `main()`.

På liknande sätt kan man faktorisera ut även den första utskriften, kanske som funktionen `printString()`:

```
void printString(char string[]){
    cout << string << endl;
}
```

Återstår till slut den obekväma `cout « endl;`-instruktionen som inte hittar någon naturlig plats i tillvaron.

Slutversion(er) Sammantaget kan nu `main()` skrivas med bra val av funktionsnamn så att den är självförklarande och befriad från störande klutter:

```
int main(){
    char string[40] = { "Hej hopp i lingonskogen" };

    printString(string);
    printStringReversed(string);
    cout << endl;
}
```

Jag stör mig fortfarande på sista raden `cout « endl;`. Man kan använda `printString(newLine)` till denna men då med en ytterligare deklaration `char newLine[2] = "\n";` som ger fler rader i källkoden. Så en ”sista” slutversion blir:

```
int main(){
    char newLine[2] = "\n";
    char string[40] = { "Hej hopp i lingonskogen" };

    printString(string);
    printStringReversed(string);
    printString(newLine);
}
```

B. Erfarenhetslista

Detta appendix var från början tänkt att heta ”God programmeringssed” eller något ditåt, men benämningen **erfarenhetslista** är bättre då man erfarenhetsmässigt får bättre program, färre fel och enklare felsökningar om man tänker och skriver kod på detta sätt.

Bra kod skall:

1. vara strukturerad och använda subrutiner
   - skriv en rutin för varje specifik sak
   - faktorisera där det är meningsfullt
   - använd så få argument som möjligt (helst noll!)
   - en subrutin ska ha en (1) ingång (”`LABEL:`”) och enbart en (1) utgång (”`ret`”)
2. vara indenterad och lämpligt kommenterad
   - symboliska adresser ska börja i kolumn 0
   - instruktioner ska börja ett TAB-steg in
   - ibland är en subrutin bättre än en kommentar
   - undvik onödiga kommentarer av typen

     ```
     "inc ZL     ; increment ZL"
     ```

     bättre är

     ```
     "inc ZL     ; next element"
     ```
3. använda meningsfulla variabel- och subrutinnamn
   - en rutin ska göra det den heter
   - sträva efter självdokumenterande kod
4. använda pekare
   - pekare underlättar avancerade datastrukturer
   - pekararitmetik är kraftfullt i assembler och C/C++.
5. parametriseras
   - använd globala parametrar (`.def` och `.equ` i filens start) och preprocessorn
6. vara relokerbar
   - programmet ska inte ställa krav på att ligga på vissa minnesplatser

7. vara resurssnål
    - använd inte onödigt mycket variabler
    - undvik onödig minnesåtgång, både i kod och data
    - använd loopar där detta är möjligt
    - rulla ihop repetitiva kodstycken
8. inte ha sidoeffekter
    - använd subrutiner för att isolera funktioner
    - undvik globala varaiabler
    - använd lokala variabler
    - överväg parameteröverföring via stack