Grundläggande struktur

- Generell struktur

```
COBOL Program.
  Division
    Section
      Paragraph
        Sentence
          Statement
```


COBOLSKOLAN IBM-UTBILDNING 1

Ett COBOL-program har i grunden en tydlig struktur.
De olika logiska delarna har klara och tydliga uppgifter och avgränsningar.
En mer detaljerad beskrivning följer på efterföljande sidor.

Grundläggande struktur

- Division
  - en logisk indelning av programmet

```
Identification Division.
      . . . .
Environment Division.
      . . . .
Data Division.
      . . .
Procedure Division.
      . . . .
```


COBOLSKOLAN IBM-UTBILDNING 2

Ett normalt COBOL-program består av ovanstående fyra divisioner, även om t.ex. `Environment Division` och `Data Division` inte är obligatoriska. De tre första sektionerna är beskrivande, där varje beskrivning avslutat med en punkt (.).

`Procedure Division` innehåller själva programlogiken och består av olika typer av COBOL-verb och uttryck, som är samlade under Paragrafer i COBOL-meningar (Procedurer).

Nivåer

- Nivåindikatorer 01 - 49 används för att indikera inbördes förhållande
- Gruppnamnet avser samtliga underliggande element
- Grupper kan förekomma på flera nivåer
  - en grupp består av andra variabler eller grupper
  - en variabel kan inga i en grupp, eller kan vara fristående

```
* Detta är en Kund
  01 Kund.                                  *> Gruppnamn
     05 Namn.                               *> Gruppnamn
        10 Fornamn            Pic X(20).
        10 Efternamn          Pic X(30).
     05 Adress.                             *> Gruppnamn
        49 Postnr             Pic X(05).  *> Max-nivå
        49 Postadress         Pic X(25).
     05 Kundstatus            Pic 9(01).

  01 Meddelande               Pic X(30). *> Fristående
```


COBOLSKOLAN IBM-UTBILDNING 3

Nivåindikatorerna 01-49 indikerar olika nivåer. Ett lägre nummer indikerar en överordnad nivå. I exemplet ovan så ser du att ”stegen” är 5 mellan nivåerna, men det behöver inte vara så.

Vid referens till ett gruppnamn, t.ex. `Namn`, så avses samtliga element som är underordnade denna grupp, i detta fall både `Fornamn` och `Efternamn`.

Variabeln med namnet `Meddelande` är helt fristående.

Deklarationer

- Alfanumeriska variabler kan vara dynamiska
  - storleken anpassas efter innehållet
  - utrymme allokeras i exekveringsläge vid behov

```
01 dvar Pic X Dynamic [Limit maxl ][Value '..']
```

✓ **Dynamic** beskriver att variabeln kan variera i storlek
✓ **Limit maxl** anger variabelns maximala längd
✓ **Value** anger eventuellt inledande värde


```
01 Var1   Pic X Dynamic.                  *> (1)
77 Var2   Pic X Dynamic Value 'Peter'.    *> (2)
01 Va3    Pic X Dynamic Limit 500.        *> (3)
```


När du beskriver att en variabel skall vara dynamisk så skall storleken anpassas till innehållet.

(1) Denna variabel har inget inledande värde. Dess längd är inledningsvis noll (0). När du flyttar in data till variabeln så anpassas storleken så att den får plats för de antal tecken som du flyttar.

(2) Denna variabel har ett inledande värde som är fem (5) tecken. Längden är inledningsvis fem. Om du flyttar in färre än fem (5) tecken till variabeln så blir längden kortare, flyttar du in fler tecken ärn fem så blir variabeln längre.

(3) Denna variabel har samma beteende som de andra, men den kan inte bli större än 500 tecken. Skulle Du flytta in fler än 500 tecken så får deu en trunkering. Detta återkommer vi till senare.

Deklarationer

- Tilldela inledande värde - alfanumerisk
  - värdet är vänsterjusterat och utfyllt med blanktecken (+, X'40')

```
01 Meddelande      Pic X(20) Value 'COBOL'.
```

```
Pos     :12345678901234567890
Tecken  :COBOL+++++++++++++++
EBCDIC  :CDCDD444444444444444
         36263000000000000000
```

  - noll-terminerad sträng
    - ✓ strängar i C-språket avslutas med X'00'
      - vid överföring av strängar till C-program behöver detta indikeras

```
01 C-Meddelande    Pic X(20) Value z'COBOL'.
```

```
Tecken  :COBOL ++++++++++++++
EBCDIC  :CDCDD044444444444444
         36263000000000000000
```

      - motsvarar Low-Value

Vid beskrivning av en variabel, så beskriver du hur du vill betrakta data, vilken bild (*picture*) du vill ha. Det finns flera tecken än de vi ser här, men dessa är de vanligaste.

X anger att du tänker lagra alfanumeriska tecken i variabeln, medan 9 däremot innebär att data är tänkt att vara numeriskt. Med numeriska variabler kan du göra beräkningar, men det är trots allt viktigt att innehållet i variabeln verkligen är numeriskt.

Storleken på variabeln avgörs av hur många tecken, X eller 9, du har beskrivit. Om du vill så kan du ha en repetitionsfaktor inom parentes som avgör hur många tecken eller siffror som variabeln skall bestå av.

Numeriska variabler kan lagras i flera interna format, med underförstått är att data lagras i s.k. *display-format*. Varje tecken upptar en byte. Mer om detta får du veta senare i kursen.

Redigering

- Decimalkommats placering
  - , anger var decimalkommat skall placeras

```
Environment Division.
Configuration Section.
Special-Names.
    Decimal-Point is Comma.
. . . . . .
77 Summa2      Pic 9999V99
                   Value 1,50.
                   Value 25.
                   Value 125,75.

77 R-Summa2    Pic ***9,99.
                  '***1,50'
                  '**25,00'
                  '*125,75'
```


COBOLSKOLAN IBM-UTBILDNING 6

Decimalpunkten kan nu ersättas med ett decimalkomma, när beskrivningen under `Special-Names`-paragrafen i sektionen `Configuration Section` i divisionen `Environment Division` har gjorts.

Redigering

- Valutatecken (2)
  - egna tecken kan beskrivas (restriktioner förekommer)

```
Environment Division.
Configuration Section.
Special-Names.
    Decimal-Point is Comma
    Currency Sign is 'Kr ' with Pic Symbol 'K'
    Currency Sign is 'EUR' Pic Symbol 'U'.
 . . . . . .
 77 Summa      Pic 9999V99 Value 1,50.

 77 R-Summa1   Pic KZZZ9,99.
                   Kr    1,50

 77 R-Summa2   Pic UZ(03)9,99.
                   EUR   1,50
*-----------------------------------
     Move Summa  to R-Summa1 R-Summa2
```


COBOLSKOLAN IBM-UTBILDNING 7

Man kan beskriva egen teckensymbol, men det finns restriktioner för vilka tecken som kan användas. Detaljer finns i manualen COBOL Language Reference.

Add

- Addera en variabel till en annan variabel
  - lagra svaret i en tredje variabel med avrundning

```
77 Summa       Pic 999V999 Value 125,456. *> Numeric
77 Delsumma    Pic 999V999 Value  75,402. *> Numeric
77 Totalsumma  Pic ZZ99,99.            *> Numeric Edited

    Add
       Summa to Delsumma
       Giving Totalsumma Rounded
    End-Add

    Add
       Summa to Delsumma
       Giving Totalsumma
    End-Add
```

```
Med avrundning
Totalsumma: 200,86
```

```
Utan avrundning
Totalsumma: 200,85
```


COBOLSKOLAN IBM-UTBILDNING 8

Exemplet visar avrundning.

Utan uttrycket `Rounded` så sker ingen avrundning och du får en trunkering (*truncation*), vilket innebär att tusendelarna tapps bort i detta fall.

Add

- Corresponding
  - adderar variabler från en grupp till variabler med motsvarande namn i en annan grupp

```
01 Dagens-Belopp.
   05 Belopp2   Pic 9(06)V99.          *> Value 0 ?
   05 Belopp1   Pic 9(06)V99.
   05 Belopp3   Pic 9(06)V99.
01 Summa-Belopp.
   05 Belopp1   Pic 9(06)V99 Value Zero. *> Value Zero ?
   05 Belopp2   Pic 9(06)V99 Value 0.
   05 Belopp3   Pic 9(06)V99 V

   Add Corresponding
      Dagens-Belopp to
      Summa-Belopp
   End-Add
```

```
Add Belopp1 in Dagens-Belopp
   to Belopp1 in Summa-Belopp
Add Belopp2 in Dagens-Belopp
   to Belopp2 in Summa-Belopp
Add Belopp3 in Dagens-Belopp
   to Belopp3 in Summa-Belopp
```


COBOLSKOLAN IBM-UTBILDNING 9

Antag att ett program läser en fil eller en databas och behöver summera belopp som hämtas från olika poster.

Genom att placera de olika numeriska variablerna i olika grupper, så kan en addering göras enkel. Notera dock att de numeriska variablerna måste ha samma namn (*corresponding*) i de båda grupperna.

Vid deklaration av numeriska värden kan det ibland vara viktigt att de har kända startvärden, t.ex. 0.

I detta exempel kan man tänka sig att variablerna i gruppen `Dagen-Belopp` erhåller sina värden från aritmetiska uttryck (*Giving*) eller *Move*, och inte en Add to. I så fall behövs ingen Value-deklaration.

Variablerna i gruppen `Summa-Belopp` får sina värden genom en addition. Då behövs givetvis kända startvärden, t.ex. noll.

Felhantering

- On Size Error

```
01 Varden.
   05 Antal          Pic 9(03).
   05 Pris           Pic 9(04)V99.
   05 Totalsumma     Pic Z(05),99.

    Multiply Antal By Pris
            Giving Totalsumma
       On Size Error
          Display '*** Allvarligt fel ***'
          Display '*** Variabeln Totalsumma är för liten'
          Display '*** Antal: ' Antal
          Display '*** Pris:  ' Pris
          Move 99 To Return-Code
          Stop-Run
    End-Multiply
    Display 'Resultatet är: ' Totalsumma
```


COBOLSKOLAN IBM-UTBILDNING 10

Problemet här är dimensionering av variabeln `Totalsumma`. Det finns ingen anledning till att ’snåla’ med storleken på värdevariabler. Genom att ändra till `Z(08),99` så skulle problemet vara löst.

Det hade varit ännu bättre att förorsaka en regelrätt Abend i stället för ett normalt avslut med Retur-kod.

I Language Environment finns s.k. *Callable Services* och här finns en tjänst för att förorsaka en *abend*. Mer i avsnittet om *Language Environment*, LE.

Flytta dataelement

- högerjustering
  - ✓ *Just Right*-beskrivning i mottagande variabel
  - ✓ blankutfyllnad om mottagande fält är större

```
Working-Storage Section.
01 Texten      Pic X(05) Value 'COBOL'.
01 NyText      Pic X(10).
01 JustText    Pic X(10) Just Right.

Procedure Division.
    Move Texten to NyText, JustText

    Display 'NyText   *' NyText '*'
    Display 'JustText *' JustText '*'

    GoBack
    .
```

```
NyText   *COBOL     *
JustText *     COBOL*
```


COBOLSKOLAN IBM-UTBILDNING 11

Vid Move av innehållet i alfanumeriska variabler gäller regeln att tecken för tecken från vänster i den sändande variabeln flyttas till den mottagande variabeln och placeras en efter en från vänster till höger efter varandra. Om den mottagande variabeln är större, d.v.s. att den har plats för fler tecken, så sker utfyllnad med blanktecken till höger där det finns utrymme.

Left justification and padding with blanks.

Skulle däremot den mottagande variabeln ha plats för färre tecken, så sker en trunkering (*truncation*) och du tappar tecken. Detta är ett enkelt sätt att korta av en variabel, t.ex. vid listutskrifter.

Flytta dataelement

- Alfanumerisk till alfanumeriskt - Dynamic
  - flyttning tecken för tecken från vänster
  - Längdanpassning av mottagande dynamiska variabel

```
77 Enlang Pic X(25) Value All '*'.
77 Fnamn  Pic X(05) Value 'Peter'.
77 Enamn  Pic X Dynamic.  *> Ingen längd
77 Namn   Pic X Dynamic Limit 8.

    Move Fnamn to Enamn          'Peter'
    Move 'Sterwe' to Enamn       'Sterwe'
    Move Enlang to Enamn         '****. . .***'
    Move Fnamn to Enamn          'Peter'.
    Move Enlang to Namn          '********'
```


COBOLSKOLAN IBM-UTBILDNING 12

Vid Move av innehållet i alfanumeriska variabler gäller regeln att tecken för tecken från vänster i den sändande variabeln flyttas till den mottagande variabeln och placeras en efter en från vänster till höger efter varandra.

Om den mottagande variabeln är en dynamisk variabel, så anpassas längden till samma längd som den sändande variabeln, oberoende av den mottagande dynamiska variabeln nuvarande längd.

Om den dynamisk variabeln är beskriven med `Limit maxlen` så sker trunkering om den sändande variabeln har en längd som överstiger värdet av `Limit`.

Reference Modification (1)

- Position och längd kan anges vid referens till ett dataelement i display-format (`Usage is Display`)

```
datanamn [startpos:[längd]]
```

  - `startpos` anger teckennummer räknat från 1
  - `längd` anger antalet tecken som avses
  - `startpos` och längd kan båda vara konstanter, numeriska dataelement eller aritmetiska uttryck
  - utvärderingen av `startpos` måste vara större än noll(0) och ej utanför posten
  - standardvärde för `längd` är ”resten av dataelementet”

COBOLSKOLAN IBM-UTBILDNING 13

När du behöver referera en del av en variabel kan du använda det som kallas *Reference Modification*. Detta kallas oftast för *Substring* i andra programspråk.

När du använder denna teknik vid en `Move`, så ligger styrkan i att både sändande och mottagande variabel kan vara en delmängd. Du anger endast vilken position du avser samt eventuellt hur många tecken, separerat med ett kolon-tecken. Skulle den mottagande variabeln ha plats för fler tecken än de du angivit, så sker en utfyllnad, alternativt att du får en trunkering.

Längdangivelsen för den mottagande variabeln, om du anger en längd, får då ej vara så stor att du skulle nå utanför den mottagande variabeln. Detta ger ett kompileringsfel, förutsatt att längden är angiven som ett konstant värde, d.v.s en siffra.

Om du inte anger en längd (avser både sändande och mottagande) så tolkas detta som att du avser hela variabeln från den angivna startpositionen.

Reference Modification (2)

- flytta innehållet i **FaltA** till position 5 i **FaltB**
  - trunkering av innehållet från FaltA om det är längre än 7 tecken

```
Move FaltA to FaltB(5:7)
```

```
AAAAAAAAAA  →  ----AAAAAAA--
```

  - annars blankutfyllnad

```
AAAAA  →  ----AAAAAbb--
```

  - ange langden av FaltA vid flytt till FaltB

```
Move FaltA to FaltB(5:Length of FaltA)
```

```
----AAAAA----
```


COBOLSKOLAN IBM-UTBILDNING 14

I exemplet kopieras hela innehållet i variabeln `FaltA` till position 5 i en variabel med namnet `FaltB`. Längden anges till 7. Detta förutsätter att variabeln `FaltB` är så lång att den rymmer denna sträng. Skulle `FaltA` ha en längd som överstiger sju (7) så blir det en vanlig trunkering, skulle den vara kortare så blir det blankutfyllnad, enligt vanliga regler.

Det undre exemplet visa användandet av ett s.k. *Special Register* med namnet `Length Of varnamn`. Detta uttryck kommer ersättas av den verkliga längden av variabeln `varnamn`.

Stränghantering - String

- Ny strängar kan skapas från flera variabler och/eller konstanter

```
String
    varA/litA                                  *> Källa
        [Delimited By var1|lit1|Size]          *> Avgränsare
    varB/litB                                  *> Källa
        Delimited By var2|lit2|Size
        . . . .
    Into
        varC                                   *> Mottagare
    [With Pointer(numvar1)]                    *> Start/Slut
    [On Overflow (cobolverbx)]                 *> Ej plats
    [Not On Overflow (cobolverby)]             *> OK
End-String
```

- källan flyttas tecken för tecken till mottagaren (Into)
- avgränsningen avgör hur mycket av källan som flyttas (Delimited By)
- pekare anger startposition i den skapade strängen (With Pointer)

COBOLSKOLAN IBM-UTBILDNING 15

När du behöver skapa en sträng som består av innehållet i flera variabler, eller delar av, så kan du använda COBOL-verbet `String`. Du avslutar alltid uttrycket med `End-String`.

För varje referens till en variabel eller konstant/literal, så kommer COBOL att flytta tecken för tecken tills dess att uttrycket du beskrivit med `Delimited By` är uppfyllt.

Du kan använda vilka tecken som helst som beskrivning av avgränsningen.

Det finns inget underförstått värde för `Delimited By`, utan detta måste anges, men det angivna värdet gäller för tidigare beskrivningar som beskrivits utan `Delimited By`.

`With Pointer` kan anges och anger då var i mottagande strängen som den första strängen skall placeras. När String-verbet är klar så pekar i `With Pointer(pekare)` på nästa position i den mottagande strängen. Om utrymmet inte skulle räcka till, så signaleras villkoret `On Overflow`.

Om Du inte har med villkoret så blir Du ej medveten om att den mottagande strängen är för liten.

Into-variabeln kan även vara `Dynamic`. Detta innebär att den mottagande into-variabeln anpassas och är alltid tillräckligt stor. Den kan begränsas med `Limit`-beskrivning.

Stränghantering - String

- Skapa ett ny sträng

```
Working-Storage Section.
01 Kund.
   05 Fnamn    Pic X(20) Value 'Peter Ivan'.
   05 Enamn    Pic X(30) Value 'Sterwe'.
77 Meddelande Pic x(80).
77 Tvablanka  Pic X(02) Value Space.

Procedure Division.
    String
       'Mitt namn är ' Delimited by Size
        Fnamn          Delimited by Tvablanka
        Space          Delimited by Size
        Enamn          Delimited by Space
        Into Meddelande
    End-String
    Display Meddelande
```

- Mitt namn är Peter Ivan Sterwe..

COBOLSKOLAN IBM-UTBILDNING 16

Här skapas en ny sträng med namnet `Meddelande` som består av både konstanter/literaler och variabler. Notera att avgränsningen för variabeln med namnet `Fnamn` är två blanktecken (`Tvablanka`), eftersom det ingår ett blanktecken efter strängen `Peter`.

Stränghantering

- Inspect Tallying
  - för att räkna antalet tecken eller förekomster av ett eller flera tecken i en variabel

```
Inspect var1 Tallying var2 for
   Characters |
       Before|After Initial var3|lit3
   All | Leading var4|lit4
       Before|After Initial var5|lit5
```

  - `var1` är den undersökta strängvariabeln
  - `var2` räknas upp med antalet träffar
  - **`Characters`** innebär att antalet tecken räknas
    - ✓ `Before|After|Initial` anger var sökningen skall göras
    - ✓ `var3|lit3` är det eller de tecken som avgränsar sökningen
  - **`All`** innebär att flera förekomster efter varandra accepteras
  - **`Leading`** innebär samtliga inledande förekomster
    - ✓ `var4|lit4` anger sökta tecken
    - ✓ `Before|After|Initial` anger var sökningen skall göras
    - ✓ `var5|lit5` är det eller de tecken som avgränsar sökningen

Ange antingen:
- Characters
- All
- Leading

COBOLSKOLAN IBM-UTBILDNING 17

`Inspect` är ett ganska komplett/komplext COBOL-verb för att undersöka och förändra strängar.

`Inspect Tallying` används för att räkna efter förekomster av angivet tecken i en sträng. Du kan även ange uttryck som beskriver var sökningen skall påbörjas eller att du vill leta efter alla förekomster i hela strängen.

Beskrivningen ovan torde vara tillräcklig, tillsammans med efterföljande exempel, för att du skall förstå hur `Inspect Tallying` fungerar.

För ytterligare förklaring refereras till Google.

Stränghantering

```
1-Strangen: DETTA AR ETT COBOLMEDDELANDE
2-Siffror:  D5TT101R05TT0C2B2LM5DD5L1ND5
3-Omvända   W5GG101J05GG0X2Z2PN5WW5P1MW5
4-Original: DETTA AR ETT COBOLMEDDELANDE
```

- Inspect Converting

```
01 Strangen  Pic X(28) Value 'Detta ar ett COBOLmeddelande'.
01 Tecknen.
   49 Vokaler     Pic X(10) Value 'AOUÅEIYÄÖ '.
   49 Siffror     Pic X(10) Value '1234567890'.
   49 Konstanter  Pic X(20) Value 'BCDFGHJKLMNPQRSTVWXZ'.
   49 Rkonstanter Pic X(20).
```

```
    Move Function Upper-Case(Strangen) to Strangen          *> Inbyggd funktion
    Move Function Reverse(Konstanter)  to Rkonstanter       *> Inbyggd function
    Display '1-Strangen: ' Strangen
    Inspect Strangen Converting Vokaler to Siffror
    Display '2-Siffror:  ' Strangen
    Inspect Strangen Converting Konstanter to Rkonstanter
    Display '3-Omvända   ' Strangen
    Inspect Strangen Converting Siffror to Vokaler
    Inspect Strangen Converting Rkonstanter to Konstanter
    Display '4-Original: ' Strangen
```


COBOLSKOLAN IBM-UTBILDNING 18

I detta exempelprogram används `Inspect Converting` för att ersätta en grupp av tecken med en annan grupp av tecken.

Den inbyggda funktionen `Reverse` används för att enkelt vända på innehållet i en variabel.

Initiera variabler och grupper

- Initialize
  - för att återställa innehållet i en variabel eller en struktur

```
Initialize var1 | group1

                 Alphabetic
                 Alphanumeric
    [Replacing   Numeric                  [Data] By
                 Alphanumeric-Edited       varx | litx]
                 Numeric-Edited
```


  - om Replacing anges så initieras endast den nämnda datatypen
    - ✓ övriga förblir oförändrade
  - om Replacing utesluts vid initiering av en grupp så initieras:
    - ✓ alla *Numeric* och *Numeric-Edited* till noll (0)
    - ✓ alla *Alphabetic*, *Alphanumeric*, *Alphanumeric-Edited* till blank (*space*)

COBOLSKOLAN IBM-UTBILDNING 19

När en struktur fyllts med data och använts, så kan man vilja återställa innehållet till kända värden eller initialvärden. Detta kan göras med verbet `Initialize`.

Notera att om Du använder `Replacing`, så måste du ange samtliga datatyper i din struktur, annars förblir de som inte nämns i `Replacing` oförändrade.

Använt INTE `Initialize` i onödan. Om strukturen fylls med data vid en filläsning eller vid ett databasanrop, så är `Initialize` överflödig.

Initiera variabler och grupper

```
Texten  :Ett COBOL-meddelande
Antal   :05
Belopp  :0012550
Summa   :  627,50

Texten  :************************
Antal   :99
Belopp  :9999900
Summa   :    0,00
```

- Initialize

```
01 Posten.
   05 Texten  Pic X(25)     Value Space.
   05 Antal   Pic 9(02)     Value 0.
   05 Belopp  Pic 9(05)V99 Value 0.      *> Behövs Value i fälten ?
   05 Summa   Pic Z(04)9,99.             *> Error på hur de används
```

```
      Move 'Ett COBOL-meddelande' to Texten
      Move 5 to Antal
      Move 125,50 to Belopp
      Multiply Antal by Belopp Giving Summa
      Perform Display-Posten
      Initialize Posten
         Replacing Alphanumeric by All '*'
                   Numeric By 9999999,99
                   Numeric-Edited by  0
      Perform Display-Posten
```

```
Display-Posten.
     Display 'Texten  :' Texten
     Display 'Antal   :' Antal
     Display 'Belopp  :' Belopp
     Display 'Summa   :' Summa
     Display Space
```


COBOLSKOLAN IBM-UTBILDNING 20

I detta exempelprogram används `Initialize` för att ersätta variabler med egna valda värden.

Perform

- Varying

```
Perform [paragrafnamn]
        [With Test Before | After]
         Varying varA from var1|int1
         By var2|int2 Until villkor
```

- förändra innehållet i en variabel automatiskt
  - ✓In-Line

```
Perform Varying Antal from 1 by Varde
        Until Antal = Maxantal
   cobol-kod
   cobol-kod
End-Perform
```


COBOLSKOLAN IBM-UTBILDNING 21

I denna konstruktion så kan du beskriva att innehållet i en variabel skall förändras automatiskt enligt de förutsättningar du anger.

Variabeln `Varde` förväntas vara en numerisk variabel och innehålla ett lämpligt värde.

Perform

- *Varying* - programexempel

```
Identification Division.
Program-Id. PERFTEST.
Data Division.
Working-Storage Section.
77 Antal     Pic 99.
77 Varde     Pic 99 Value 1
77 Maxantal Pic 99 Value 10.
Procedure Division.
    Perform With test Before
         Varying Antal from 1 by Varde
         Until Antal > Maxantal
       Display 'Detta är inline perform nr ' Antal
    End-Perform

    Perform Paragraf1 With test After        *> Minst en gång
         Varying Antal from 1 by 1
         Until Antal = Maxantal

    GoBack
    .
Paragraf1.
    Display 'Detta är out-of-line perform nr ' Antal
    .
```


COBOLSKOLAN IBM-UTBILDNING 22

Som du ser i detta exempel så innebär villkorsformuleringen att den numeriska variabeln `Antal` skall uppnå ett värde som är större än innehållet i variabeln `Maxantal` för att villkoret skall vara uppfyllt. Om villkoret hade varit `Antal = Maxantal`, så hade koden bara exekverats 9 gånger.

I exemplet med out-of-line så finns uttrycket `With Test After` med, vilket förändrar förutsättningarna.

Exit

- För att avsluta en paragraf i förtid
  - In-Line Perform
    - ✓Exit Perform
      - hopp till slutet av paragrafen (*avslut*)
    - ✓Exit Perform Cycle
      - hopp till Perform-uttrycket (*omstart*)
  - Out-Of-Line Perform
    - ✓Exit Paragraph
      - hopp till slutet av paragrafen (*avslut*)
  - Exit-konstruktionerna påminner mycket om Go To-verbet
    - ✓stämmer inte med bra strukturerad kod
    - ✓en räddningsplanka för mindre bra struktur
    - ✓tänk två gånger!

COBOLSKOLAN IBM-UTBILDNING 23

Cobol-standarden COBOL -85 introducerade mängder av förändringar för att kunna skriva strukturerade COBOL-program, och eliminerade helt behovet av det tidigare ofta använda verbet `Go To`, vars enda uppgift är att flytta position från en plats i programmet till en annan. Frekvent användning kunde skapa det vi normalt kallar för ’spagetti-kod’.

Cobol-standarden COBOL 2002 introducerar ovanstående Exit-konstruktioner, vilket inte är något annat än förtäckta `Go To`-verb, och såldes ett steg tillbaka på vägen mot välstrukturerade COBOL-program.

Använd helst inte – eller med försiktighet. Det finns ALLTID ett annat, bättre alternativ.

Exit

- Programexempel

```
Working-Storage Section.
01 Antal Pic 99 Value 0.
   88 Klar      Value 10.
Procedure Division.
   Perform Until Klar
      Add 1 to Antal
      If Antal = 4
         Display 'Perf-Antal: ' Antal
         Display 'Exit Perform Cycle
         Exit Perform Cycle
      Else
         Display 'Perf-Antal: ' Antal
         If Antal = 6
            Display 'Exit Perform'
            Exit Perform
         End-If
      End-If
      Perform Paragrafen
   End-Perform
   Display 'Har Avslutat, Antal = '
      Antal
   GoBack
```

```
Perf-Antal: 01
- Paragrafen, Normal, Para-Antal=01
- Slut pa paragrafen
Perf-Antal: 02
- Paragrafen, Normal, Para-Antal=02
- Slut pa paragrafen
Perf-Antal: 03
- Paragrafen, Normal, Para-Antal=03
- Slut pa paragrafen
Perf-Antal: 04
Exit Perform Cycle
Perf-Antal: 05
- Paragrafen, Exit, Para-Antal=05
Perf-Antal: 06
Exit Perform
Har Avslutat, Antal = 06
```

```
Paragrafen.
   If Antal = 5
      Display ' - Paragrafen, Exit,'
         'Para-Antal=' Antal
      Exit Paragraph
   Else
      Display ' - Paragrafen, Normal,'
         ' Para-Antal=' Antal
   End-If
   Display ' - Slut pa paragrafen'
   .
```


COBOLSKOLAN IBM-UTBILDNING 24

I det första exemplet med `Exit Perform Cycle` set Du att kontrollen går tillbaka till `Perform`-uttrycket. Vad som sedan sker beror ju på hur villkoret tolkas. I detta fall så kommer programmet att gå i i ’loopen’ igen eftersom villkorsnamnet Klar ej är uppfyllt/sannt.

När variabel `Antal` har värdet 6, så sker ett uthopp ut In-Line-Perform-satsen = loopen, och den blir avslutad.

`Perform Paragrafen` tar programmet till paragrafen `Paragrafen`, och beroende på värdet i variabeln `Antal` så sker ett uthopp ur paragrafen med `Exit Paragraph`, eller så blir det ett normalt avslut.

If syntax

- End-If
  - skall alltid användas för att markera slut (*end-of-scope*) på If-uttrycket
    - punkt är ett giltigt tecken för att markera slut på uttryck, men bör endast användas för att markera slutet på en COBOL-mening (*sentence).*
    - en punkt (. *period*) stänger alla öppna scope

```
* FIN-COBOL
     If villkor1
        cobol-kod1
     End-If
     If villkor2
        cobol-kod2
     Else
        cobol-kod3
     End-If
```

```
FUL-COBOL
     If villkor1
        cobol-kod1[.]?

     If villkor2
        cobol-kod2
     Else
        cobol-kod3.
```


COBOLSKOLAN IBM-UTBILDNING 25

End-If används för att markera slutet på If-uttryck. End-If skall alltid användas. I tidigare versioner av COBOL-språket så var punkten det enda sättet att markera avslut, men sedan COBOL-85 så är detta förändrat.

Punkten är fortfarande giltig för att markera avslut, men skall inte användas. Avsaknaden av en punkt, eller en punkt på fel ställe, kan få, eller får nästan alltid, allvarliga konsekvenser för programlogiken.

Det finns inställningar för COBOL-kompilatorn för att beskriva om man tillåter en punkt som avslutsmarkering. (RULES=NOENDP). Detta bör vara standardvärde.

If syntax

- COBOL-kompilatorn kan kontrollera
  - inställningen RULES(NOENDP)
    - vid utelämnad *scope-terminator* ger kompilatorn ett varningsmeddelande när End-uttrycket har utelämnats

```
cbl rules(noendp)
Identification Division.
Program-Id. TESTIF.
Data Division.
Working-Storage Section.
01 Variabler.
   49 Vara     Pic X(01).
   49 Varb     Pic X(01).
   49 Varc     Pic X(01).
```

```
Procedure Division.                    (10)
  If Vara = Varb                       (11)
     Display 'Vara = Varb'             (12)
  Else If Varb = Varc                  (13)
          Display 'Varb = Varc'        (14)
       Else                            (15)
          Display 'Ett Meddelande'     (16)
  .                                    (17)
```

```
11 IGYPS2159-W    **RULES(NOENDPERIOD)** The scope of conditional
                  statement "IF" was terminated by a period on
                  line 17 instead of by an explicit scope terminator.
```


COBOLSKOLAN IBM-UTBILDNING 26

I detta exempel ser Du hur `End-If` (2 st) har utelämnats och ersatts med en punkt.

Om COBOL-kompilatorn har instruerats att kontrollera att inga punkter (*period*) används som ersättning för en uttrycklig *scope-terminator*, så ges returkod 4 (Varning (*W*)). Det gåt även att konfigurera COBOL-kompilatorn så att detta meddelande blir allvarligare = större returkod, så att en punkt anses som ett allvarligt fel. (Bra grej!!)

Detta skall givetvis vara en standardinställning för kompilatorns tillval (*options*), men på första raden i detta program ser du uttrycket `CBL RULES(NOENDP)`, som har samma effekt.

Alla s.k. *Options* kan påverkas via detta uttryck i programmet.

Villkorsnamn

- Ett villkorsnamn
  - när man vill jämföra en variabel mot olika kända värden kan villkoret tilldelas ett namn
    - ✓Kundkod = 12 - ett villkor som kan ges ett beskrivande namn, t.ex. Privatkund
    - ✓Antal >= 100 - ett villkor som kan ha namnet Maxantal

```
If Kundkod = 12                    *> Privatkund
   Perform Behandla-Privatkund
Else
   If Antal >= 100                 *> Maxantal
      Perform MaximaltAntal
End-If
```

```
If Privatkund                      *> Kundkod = 12
   Perform Behandla-Privatkund
Else
   If Maxantal                     *> Antal >= 100
      Perform MaximaltAntal
End-If
```


COBOLSKOLAN IBM-UTBILDNING 27

En villkorsvariabel används bl.a. för att ge namn till värden som en variabel kan innehålla. I stället för att utvärdera mot värden så kan utvärderingen sket mot villkorsnamnet.

På detta sätt kan man enkelt ge namn till koder eller andra uttryck.

Villkorsnamn

- Ett villkorsnamn
  - ✓ beskrivs med nivåindikator 88
  - ✓ ett villkor tilldelas ett namn
  - ✓ är alltid associerat med en villkorsvariabel på valfri nivå
  - ✓ beskriver ett eller flera värden som en villkorsvariabel kan innehålla

```
nn Variabelnamn      Pic  . . .
                   [ Value  ] [      [          ]      ]
   88 Villkorsnamn [ Values ] [nn    [ Through  ] nnn  ]
                   [        ] [      [ Thru     ]      ]
```

  - ✓ villkorsnamnet är sant/uppfyllt när ett av värdena finns i `Variabelnamn`

En villkorsvariabel används bl.a. för att ge namn till värden som en variabel kan innehålla. I stället för att utvärdera mot värden så kan utvärderingen sket mot villkorsnamnet.

På detta sätt kan man enkelt ge namn till koder eller andra uttryck.

Villkorsnamn - Indikatorer

- Används som indikatorer/*switchar*

```
Set VillkorsNamn  to True | False
```

```
nn VariabelNamn     Pic  . . Value v0.
      88 Villkorsnamn1 Value v1
                  [When Set to] False v2].
      88 Villkorsnamn2 Value v3.
```

- programmet bestämmer sjalv under vilka förutsättningar som villkorsnamnet skall vara sant med *Set to True* eller med standardvärde
- *Set to False* är bara giltigt för villkorsnamn med *When Set to False* angivet

COBOLSKOLAN IBM-UTBILDNING 29

När villkorsvariabler används som indikatorer eller switchar, så baseras inte värdet på innehållet i någon variabel med verkligt data, utan beskriv endast som en variabel, med eller utan namn, samt det värde som variabeln skall inta när villkorsnamnet sätts i sant (*True*) eller falskt (*False*) läge.

Villkorsnamn - Indikatorer

- Exempel (1) – ett villkor sant

```
77 Villkoren Pic 9(01) Value 0.  *> Standardvärde
   88 Saknas  Value 0.           *> Sant initialt
   88 Start   Value 1 When Set To False 0.
   88 Finns   Value 2 False 0.
   88 Slut    Value 3 False 0.
```

- Tilldelning av villkorsvärden

```
   Move 1 to Villkoren
* Mindre lyckat men fungerar, avstå från detta

   Set Start to True
* COBOL flyttar 1 till Villkoren
* Programmet behöver ej känna till värdet
   Set Start to False  [= Set Saknas to True]
* COBOL flyttar 0 till Villkoren
```


COBOLSKOLAN IBM-UTBILDNING 30

När du använder villkorsvariabler som indikatorer för något läge, så ändrar du villkorets tillstånd med uttrycket `Set 88-namn to True`. Notera dock att endast ett villkor i din villkorslista kan vara uppfyllt vid varje tidpunkt. Skulle du behöva flera villkor som skall kunna vara uppfyllda/sanna vid samma tidpunkt så kan du deklarera dessa under olika överordnade nivåer. Dessa behöver inte ens ha något namn.

Evaluate

```
Evaluate True
   When Antal = 1
      Perform ParagrafEtt
   When Antal = 4          *> (Antal = 4) OR (Var1 = Var2)
   When Var1 = Var2
      Perform ParagrafMulti
   When Saknas
      Perform ParagrafSaknas
   When Felkod Not Equal to Zero
      Perform ParagrafFelkod
   When Other
      Perform ParagrafAnnatFel
End-Evaluate
```

```
Evaluate True
   When (Antal = 1 Or 2) And Maxvarde
      Perform ParagrafEttEllerTvaMax
   When Other
      Perform ParagrafAnnatFel
End-Evaluate
```

Omdefinition - dataelement

- Ett dataelementen beskrivs med `Picture`
- Det kan finnas behov av att betrakta ett dataelement på flera sätt
  - gör en ny definition av samma utrymme med annan Picture

```
nn VarnamnA Pic . . .
nn VarnamnB Redefines VarnamnA Pic . . .
```

  - exempel

```
01 Redefexempel.
   05 VardeText1     Pic X(08).          *> t.ex '00012575'
   05 VardeNum       Redefines VardeText1 Pic 9(06)V99.
   05 Resultat       Pic ZZZ.ZZZ,99.
```

```
If VardeText1 is Numeric
   Compute Resultat = Vardenum * 12   *> bara ex på aritm.
Else
   Perform SkrivMeddelandeEjNum
End-If
```


COBOLSKOLAN IBM-UTBILDNING 32

När du beskriver en variabel eller ett dataelement, så görs detta med en `Picture`-beskrivning. Ett program kan ha fått tillgång till en sträng med data, som separerats med `Unstring`. Om man vet att en av dessa variabler innehåller numeriskt data med två decimaler, så kan man göra en omdefinition för att kunna använda datat i aritmetik.

En textvariabel kan inte användas i aritmetik, även om den innehåller siffror.

Exemplet förutsätter att det finns inledande nollor i `VardeText1`.

Omdefinition - dataelement

- om antalet siffertecken kan variera i `Vardetext1`
  - ✓ högerjustera och fyll ut med nollor
- exempel

```
01 Redefexempel.
   05 VardeText1    Pic X(08) Value '12575'.

   05 VardeText2    Pic X(08) Just Right.

   05 VardeNum      Redefines VardeText2 Pic 9(06)V99.
   05 Resultat      Pic ZZZ.ZZZ,99.
   05 Numvar        Pic 9(02) Value 10.
```

12575___

```
Move VardeText1 to VardeText2

Inspect Vardetext2 Replacing All Space by '0'

If VardeText1 is Numeric
   Compute Resultat = Vardenum * Numvar
Else
   Perform SkrivMeddelandeEjNum
End-If
```

___12575

00012575

COBOLSKOLAN IBM-UTBILDNING 33

Om variabeln `VardeText1` kan innehålla ett varierande antal siffror så måste detta hanteras.

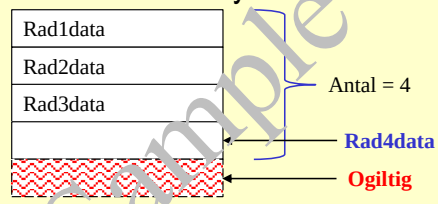
En textvariabel är vänsterjusterad med blankutfyllnad, och om det i exemplet inte är åtta (8) siffror så finns det ett antal blanktecken efter sista siffran. Dessa blanktecken måste tas bort.

Siffrorna måste högerjusteras. Detta kan göras enkelt genom att flytta innehållet i variabeln `VardeText1` till en annan variabel, `VardeText2,` som har attributet `Just Right` i `Picture`-beskrivningen

Efter detta så finna det ett antal inledande blanktecken innan siffrorna. Dessa blanktecken måste ersättas med nollor. Detta kan göras med `Inspect Replacing`.

Tabell

- En tabell består av en eller flera tabellrader
- Varje tabellrad och tabellelement refereras med ett radnummer
  - rader i tabeller kan även ha en nyckel (behandlas i senare avsnitt)
    - nr 1 ==>
    - nr 2 ==>
    - nr 3 ==>
    - nr 4 ==>



  - en tabell har ett bestämt antal rader
    - referens till en rad utanför tabellen är en felaktig referens
  - en tabell måste fyllas med data
    - kan vara konstanta värden som anges vid tabellbeskrivningen eller data som programmet hämtar från en fil eller databas

COBOLSKOLAN IBM-UTBILDNING 34

En tabell kan innehålla många tabellrader. I det enklaste fallet finns endast en ’kolumn’ med data. För att referera en viss rad används radnummer. Du måste på något sätt förse tabellen med innehåll för att den skall vara meningsfylld.

Alla tabeller har alltid ett bestämt antal rader, och om du försöker referera en rad som inte finns i tabellen, så är detta ett allvarligt fel.

För att fylla en tabell med data, så kan du redan vid beskrivningen av tabellen förse den med information, eller så kan du med programkod i programmets `Procedure Division` hämta data från en fil eller en databastabell och placera i tabellraderna.

Beskriva COBOL-tabell

- COBOL-tabellrader
  - tabell med en (1) dimension – en (1) occurs
    - ✓kan vara ett enkelt dataelement

```
nn varnamn  Pic . . (nn) Occurs nn [Times].
```

```
01 Tabnamn1.                    *> Occurs tillåts ej på nivå 1
   05 Namnet      Pic X(35) Occurs 100.      *> 100 rader
```

    - ✓kan vara en grupp med ett eller flera dataelement

```
nn tabellnamn Occurs nn [Times] [Value . . ].
   nn varnamn1   Pic . . [Value . . ]
   nn varnamn2   Pic . . [Value . . ]
```

```
01 Tabnamn2.
   05 Namn        Occurs 100.                *> 100 rader
      10 Fnamn    Pic X(10).
      10 Enamn    Pic X(25).
      10 Belopp   Pic 9(05)V99.
```

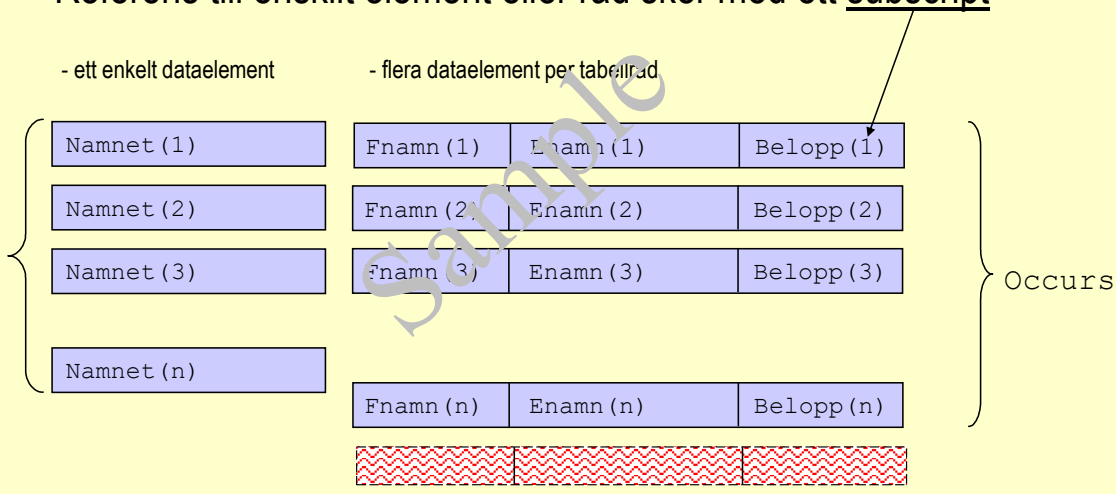
COBOLSKOLAN IBM-UTBILDNING 35

När du beskriver en tabell, så kan tabellraderna bestå av ett eller flera dataelement. Med uttrycket `Occurs` anger du hur många tabellrader som skall finnas i tabellen.

Antalet rader är alltid fast och kan inte förändras annat än att göra förändring i beskrivningen av tabellen.

Referera COBOL-tabell

- En tabell med en (1) dimension
- Referens till enskilt element eller rad sker med ett subscript



COBOLSKOLAN IBM-UTBILDNING 36

När du beskriver en tabell så ger du olika namn till de olika fälten i tabellraden. För att referera ett element så i tabellen, så använder du något vi brukar kalla för *index*. Det korrekta namnet, för denna typ av tabell, är egentligen *subscript*, som är ett numeriskt värde eller en numerisk variabel med korrekt värde. Det finns en skillnad mellan begreppen *index* och *subscript* som jag kommer att beröra i senare kapitel.

Det namn du gett till ett fält refereras med detta *subscript* inom parentes efter namnet t.ex. `Namnet(Numx)`.

Notera speciellt att det är du själv som ansvarar för att ditt numeriska *subscript*/indexvärde värde inte överskrider maxvärde. Du kan låta kompilatorn generera kod för att kontrollera alla referenser till *subscript/index*. Om referensen skulle vara ogiltig så kommer COBOL att avsluta programmet med en *user-abend* samt meddelande som i klartext anger felorsaken.

Du kan ange kompilator-direktiv direkt i din programkod på första raden med uttrycket CBL *direktiv*.

Mer om detta senare.

COBOL-tabell

- Ingen automatisk kontroll av giltigt subscript-värde
  - kompilator-option SSRANGE
    - ✓kompilatorn genererar kod för att kontrollera att alla referenser innehåller giltiga *subscript* ('index')-värden
    - ✓felaktigt värde vid kompilering ger kompileringsfel
      - – gäller bara referenser som angivits som siffror
    - ✓felaktigt värde vid exekvering medför abend U4038 eller meddelande
      - – när referenser angivits som numeriska variabler - värdet är inte känt förrän vid exekveringen

```
CBL SSRANGE(ABD ! MSG)
```


COBOLSKOLAN IBM-UTBILDNING 37

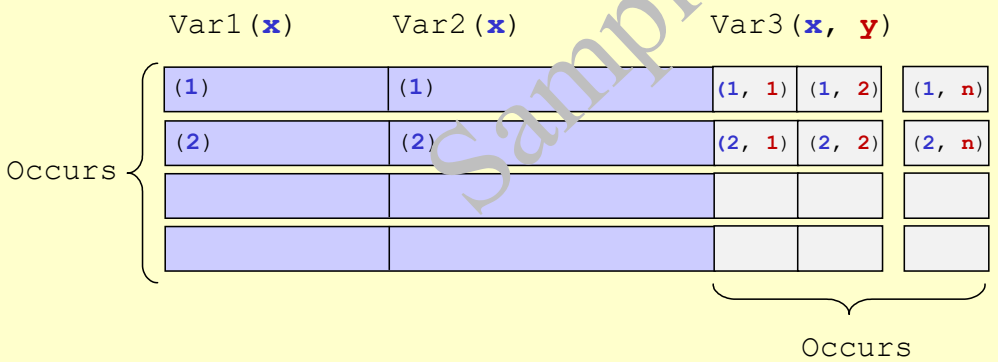
För att referera ett element i en tabell, så använder du något vi brukar kalla för *index*. Det korrekta namnet, för denna typ av tabell, är egentligen *subscript*, som är ett numeriskt värde. Det finns en skillnad mellan begreppen *index* och *subscript* som jag kommer att beröra i senare kapitel.

Notera speciellt att det är du själv som ansvarar för att ditt numeriska *subscript*/*index*-värde inte överskrider maxvärde. Du kan låta kompilatorn generera kod för att kontrollera alla referenser till *subscript/index*. Om referensen skulle vara ogiltig så kommer COBOL att avsluta programmet med en *user-abend* samt meddelande som i klartext anger felorsaken eller ett meddelande beroende på SSRANGE.

Du kan ange kompilator-direktiv direkt i din programkod på första raden med uttrycket CBL *direktiv*.

Tabeller i tabeller

- Ett tabellelement i en tabell kan vara en annan tabell
  - tabellen har flera dimensioner
  - referens till enskilt element sker med flera subscript ('index')
    - ✓ett för dimension1, ett för dimension2




```
Display Var1(1) '; ' Var3(1,4)*> Rad1
```


Denna bild visar schematiskt en tabell med flera rader, där ett element på varje rad är en egen tabell.

Read

- Läsning görs med READ

```
Read filnamn [Into postarea ]
   At End
      cobol-kod
   Not At End
      cobol-kod
End-Read
```

- `filnamn` är namnet i Select-satsen
- `Into` kan anges om man vill ha en post flyttad till en struktur i Working-Storage Section
- **`At End`** villkoret är uppfyllt vid läsning efter sista post (EOF)
- **`Not At End`** är uppfyllt vid lyckad läsning

När du skall läsa en post så gör du detta med uttrycker `Read`. Detta uttryck har två (2) inbyggda villkor där antingen det ena eller det andra kommer att vara uppfyllt.

`At End` innebär att du kommit till slutet på filen. Om det är den första läsningen, så innebär det att filen är tom.

`Not At End` är uppfyllt vid en lyckad läsning.

Var posten finns efter läsningen beror på vilken teknik du använder; har du en detaljerad beskrivning i File Section eller i Working-Storage Section?

Om du har en detaljerad postbeskrivning i WS med namnet `Postarea` så anger du `Into Postarea` vid läsningen. Denna teknik kallas för *Move Mode*. Det andra sättet kallas för *Locate Mode.*

Read

- Exempel

```
Procedure Division.
    Perform Initiera
    Perform LasFil Until Slut
    Perform Avsluta
    Goback
    .
LasFil.
    Read Artiklar Into WS-Artikelpost
       At End
          Set Slut to True
       Not At End
          Perform BearbetaPost
    End-Read
    .
BearbetaPost.
```


Detta är ett exempel på Move Mode, där posten läses in till en struktur med namnet `WS-Artikelpost`. Du ser också exempel på ett villkor med namnet `Slut` som jag sätter till sant läge när jag kommit till slutet på filen.

```
77 Villkoren   Pic 9 Value 0.
   88 Slut           Value 1.
   88 . . .          Value ..
```

Seriell sökning med Search

- Tabeller med index kan sökas med Search
  - villkoren kan vara enkla eller sammansatta

```
Search tabnamn [Varying index1]
   At End
      cobol-kod
   When villkor1 [or villkor1A]
      cobol-kod1
  [When villkor2 [and villkor2B]
      cobol-kod2]
End-Search
```


Sökningen är en seriell sökning som startat med aktuellt indexvärde. Därför får du inte glömma att sätt indexvärdet till ett (1) innan du börjar. Som du ser kan du skriva ett eller flera `When`-uttryck, som kan referera valfria dataelement i tabellraden som `index1` pekar ut. Du behöver inte hålla reda på när hela tabellen är sökt, utan detta fångar du med `At End`-uttrycket.

Strukturen blir tydlig och koden lättare att förstå än om du använder egen kod som vi såg tidigare.

Seriell sökning med Search

- Exempel

```
01 Priser.
   05 Prisantal        Pic 9(03) Value 0.
   05 Pristabell Occurs 1 to 100 Times
                 Depending on Prisantal
                 Indexed by PrisIndex.
      10 Artikel-nr    Pic X(05).
      10 Artikel-pris  Pic 9(05)V99.
```

```
Set PrisIndex to 1        *> Start på sökningen

Search Pristabell [Varying PrisIndex]
   At End
      Set Saknas to True
   When Artikel-nr(PrisIndex) = S-Artikel-nr
      Set Funnet to True
End-Search
. . .                     *> Sökning klar
```


Som du ser i exemplet så har jag skrivit [`Varying PrisIndex`] inom hakparenteser, vilket innebär att det inte är obligatoriskt. När en tabell endast har ett index så kan du utelämna denna konstruktion, vilket du även kan göra om du avser det första indexet bland flera som du kan ha beskrivit i tabellen.

`At-End` ger dig tydlig signal att du kommit till slutet av tabellen utan att fått någon träff med ditt `When`-uttryck. I `When`-uttrycket refererar du dina tabellelement med indexvärdet på vanligt sätt.

Om du skulle behöva göra flera utvärderingar, så kan du ha flera på varandra följande `When`-uttryck, och även sammansatta villkor.

Binär sökning med Search

- Tabeller med nyckel och sorterat data kan sökas med binärsökning
  - ✓genom jämförelser med sökt nyckelvärde och nycklars placering i tabellen så kan sökningen kraftigt förenklas

```
Search  All tabnamn [Varying index1|var1]
    At End
       cobol-kod
    When keyA(index1) = varx
       cobol-kod1
End-Search
```


Nyckelordet `All` anger att binärsökning skall göras.

COBOL kommer att ta sökvärdet i ditt `When`-uttryck och jämföra detta värde med nyckelvärdet för en rad i mitten av tabellen. Beroende på om nyckelvärdet är högre eller lägra än det sökta värdet, så kan ena halvan av tabellen förkastas. Sedan halveras den återstående tabellen och samma process utförs sedan flera gånger.

Notera att du endast kan ha en (1) `When`-formulering och att du måste söka på hela nyckelvärdet.

Skulle detta inta passa dig så kan du ju alltid använda seriell sökning, och flexibiliteten blir större, men sökningen blir långsammare.

Hexadecimala talsystemet

- Basen är sexton
  - siffersymboler 0-9 samt tecknen A-F
    - representerar sexton olika värden
  - en siffras position bestämmer dess värde
    - förflyttning en position till vänster innebär en multiplikation med sexton (16)
  - en hexadecimal siffra motsvarar 4 bitar

1	1	1	1	Maxvärde	Hex
2^3	2^2	2^1	2^0	8+4+2+1 = 15	= F
1	0	1	1	8+0+2+1 = 11	= B

COBOLSKOLAN IBM-UTBILDNING 44

I det hexadecimala systemet är basen 16. För att åskådliggöra de olika värdena så används siffersymbolerna noll (0) t.o.m. nio (9) samt tecknen A t.o.m. F, där de sistnämnda då representerar värdena tio (10) t.o.m. femton (15). Tillsammans blir det sexton olika värden.

På motsvarande sätt så innebär en förflyttning i sidled en multiplikation med basen sexton, så siffrorna 0 t.o.m. F har sina grundvärden i första positionen. På andra plats har siffran ett (1) värdet sexton, siffran två har då värdet 2*16 = 32. Hexadecimala siffror i tredje positionen har då ett värde motsvarande den hexidecimala siffrans värde (0-15) multiplicerat med 256.

Et hexadecimalt tal 10B skulle då ha det decimala värdet 1*256 + 0*16 + 11*1 = 267

Alfanumeriska tecken

- Bokstäver och siffror lagras i ett kodat format i en *Byte*
  - **E**xtended **B**inary **C**oded **D**ecimal **I**nterchange **C**haracter
    - EBCDIC är stordatorns lagringsform
  - **A**merican **N**ational **S**tandard **C**ode for **I**nformation **I**nterchange
    - ASCII är persondatorns m.fl. lagringsform
  - översättningstabeller beskriver koder för olika tecken

EBCDIC	Tecken	ASCII
C1	A	41
C2	B	42
C3	C	43
F1	1	31
F2	2	32
F3	3	33

För att representera siffror och bokstäver används åtta (8) bitar, vilket innebär 256 olika värden. I COBOL motsvara detta `High-Value`, alla bitar ”på” = FF.

I COBOL motsvara `Low-Value`, alla bitar ”av” = 00.

Detta innebär att man kan beskriva 256 olika tecken med hexadecimala tal. I IBMs stordatorarkitektur är lagringsformen s.k. EBCDIC, där teckentabeller beskriver vilket underliggande hexadecimalt värde som används för att representera tecknet.

Som exempel kan nämnas att bokstaven A har koden C1, medan bokstaven a har koden 81. Våra specialtecken Å Ä samt Ö, i både versal och gemen form har man hittat speciallösning för. Rent alfabetiskt så kommer ju tecknet Å efter Z i vårt alfabet, men det gör det inte i teckentabellen.

Zonat Decimalt

- Usage Is Display
  - tecken lagras i den högra bytens vänstra halvbyte
    - utan tecken = F
    - med tecken, positivt värde = C
    - med tecken, negativt värde = D
    - ✓ zon

```
Pic  9999  Display  Value 1234        F1 F2 F3 F4
Pic  9999  Display  Value 75          F0 F0 F7 F5
Pic  S9999 Display  Value +1234       F1 F2 F3 C4
Pic  S9999 Display  Value -1234       F1 F2 F3 D4
```


Numeriska variabler i displayformat kan även innehålla tecken. Detta gör du genom att ange ett S (*Sign*) framför första nian (9) i deklarationen.

Det krävs inget extra utrymme för att lagra information om tecknet. Den vänstra halvbyten i den högra byten förändras från att ha zonen F till antingen D eller C beroende på tecken.

Packat Decimalt

Usage Is Packed-Decimal

```
nn namn PIC [S]9(ant) Usage Is Packed-Decimal|Comp-3
                      [Value värde].
```

- hur mycket plats
  - 2 decimala siffror = 1 byte
  - en siffra + tecken = i den högra byten
- till vad
  - den mest effektiva lagringsformen, optimalt för udda antal siffror
  - konverteras lätt till displayformat, bättre än från binärt format
- begränsningar
  - -

COBOLSKOLAN IBM-UTBILDNING 47

Flera av datorns interna instruktioner baseras på att data vid numeriska operationer måste vara i `Packed Decimal`, eller `COMP-3` format. Formatet bygger på att numeriska värden representeras i sitt decimala format. Detta innebär i princip att två (2) decimala siffror kan beskrivas i en byte, som ju består av två (2) hexadecimala siffror. För att lagra tecken så åtgår en halvbyte. För detta används den högra halvbyten i talet.

Detta format är effektivt ur lagringssynpunkt eftersom du får plats med nästan dubbelt så mycket information som t.ex. Displayformat. Utöver detta så blir aritmetiken betydligt effektivare än vid andra format, beroende på datorns arkitektur.

Packat Decimalt

- Usage Is Packed-Decimal
  - tecken lagras i den högra siffran i den högra byten
    - utan tecken = F
    - med tecken, positivt värde = C
    - med tecken, negativt värde = D

```
Pic   99999   Comp-3          Value   12345        12 34 5F
Pic   S99999  Packed-Decimal  Value  +12345        12 34 5C
Pic   S9999   Packed-Decimal  Value  +1234         01 23 4C
Pic   S99999  Packed-Decimal  Value  -12345        12 34 5D
```


I denna bild ser du tydligt hur decimala tal lagras samt hur tecknet representeras. Du behöver ju egentligen inte bry dig, men kommer att se att flertalet numeriska variabler i filstrukturer och databaser kommer att vara i detta format. Det är ju lättare att förstå värdet, om du skulle titta in i en fil eller en databas, eftersom det är i decimal form. I binärform måste man ju konvertera för att förstå.

Binära tal

- hur stora värden
  - ✓Comp, Binary med Trunc(Bin) eller Comp-5
    - 2 bytes med tecken = 2**15 = 7FFF = 32767= 32k
      utan tecken = 2**16 = FFFF = 65535= 64k
    - 4 bytes med tecken = 2**31 = 7FFFFFFF = 2147483647 = 2g
      utan tecken = 2**32 = FFFFFFFF = 4294967295 = 4g
    - 8 bytes med tecken = 2**63 = ...
      utan tecken = 2**64 = . .
  - ✓Comp, Binary, med Trunc(Std)
    - antalet nior avgör maxvärde och blir då det maximala numeriska värdet
    - slöseri med utrymme att använda Trunc(Std)

Om Du beskriver en binär variabel med `Comp`, `Binary` och har kompileringsoption Trunc(Bin) eller har beskrivit variabeln med typen `Comp-5`, så är det storleken å variabeln som avgör hur stora tal som kan lagras.

När man beskriver att variabeln skall ha tecken, d v s beskriver ett inledande S innan nior, t.ex. `S9(05)` så används den bit som finns längs till vänster som tecken-indikator.

Anropa subprogram

- Anrop av subprogram görs med Call

```
Call pgmnamn [Using By Reference param1 |
                    By Content param2]
    [On Exception
        cobol-kod ]
    [Not On Exception
        cobol-kod ]
[End-Call]
```

- `pgnmamn` anger namnet på det anropade programmet
- `paramx` refererar det data som skall överföras till `pgmnamn`
- `On Exception` om programmet saknas
- `Not On Exception` vid lyckat anrop
- anropet kan vara dynamiskt eller statiskt

COBOLSKOLAN IBM-UTBILDNING 50

När du från ett program behöver anropa ett annat program så gör du det med verbet `Call`. Programnamnet anger du i en alfanumerisk variabel eller som en literal. Jag kommer att berätta med om detta senare.

När du skall överföra parametrar, eller argument, så använder du `Using By Reference` eller `By Content`, vilket beskrivs på kommande sidor, och sedan namnet på ett dataelement eller en grupp.

Du kan givetvis överföra flera parametrar.

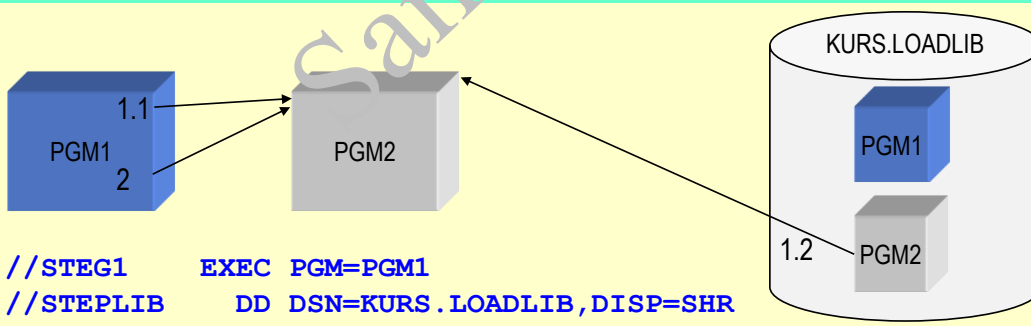
Villkoret `On Exception` är uppfyllt när det anropade programmet saknas och villkoret `Not On Exception` är uppfyllt när anropet har genomförts.

Anropa subprogram

- Dynamic Call
  - det anropade programmet är en fristående laddmodul
    - ✓namnet skall beskrivas som `Pic X(08)Value` 1-8 tecken
    - ✓subprogrammet laddas in vid första anrop

```
77 Pgmnamn  Pic X(08) Value 'PGMNAMN'.

     Call Pgmnamn [Using . . .]
```




```
//STEG1     EXEC PGM=PGM1
//STEPLIB     DD DSN=KURS.LOADLIB,DISP=SHR
```


Anrop till subprogram kan göras på två olika sätt, dynamiskt eller statiskt.

Med dynamiskt anrop (*dynamic call*) avses att det anropade programmet hämtas in från ett laddmodulbibliotek när det anropas första gången. Detta program är alltså en egen, hoplänkad laddmodul. Du uppnår detta genom att ha programnamnet i en variabel.

Programmet kan även vara ett ”inbakat” program och finnas i samma källkod/modul. Detta beskrivs senare i avsnittet om Inbakade Program.

Programnamnet på det anropade programmet anges som en literal/konstant.

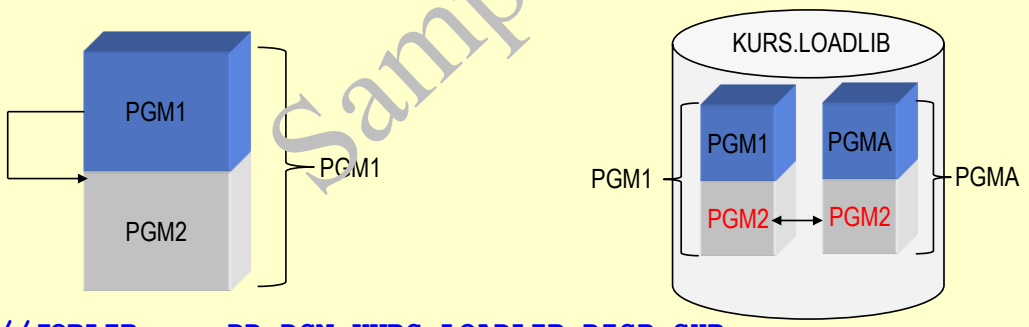
Not.

Vissa kompilatordirektiv kan påverka statiskt eller dynamiskt beteende.

Anropa subprogram

- Static Call
  - det anropade programmet ingår i samma laddmodul
    - ✓finns tillgängligt när huvudprogrammet startar
    - ✓omlänkning av samtliga program som innehåller subprogrammet vid ändring

```
Call 'PGMNAMN' [Using . . .]
```




```
//JOBLIB     DD DSN=KURS.LOADLIB,DISP=SHR
//STEG1    EXEC PGM=PGM1
```


Statiskt anrop, (*static call*) innebär att det anropade programmet finns i samma laddmodul som det anropande programmet. Innan du länkar det första programmet så måste du konstruera det andra programmet och kompilera detta så att det finns en objektmodul som sedan, när det första programmet länkas, hämtas in av länkaren (*Program Binder, Linkage Editor*).

Programnamnet på det anropade programmet anges som en literal/konstant.

Tekniken med statiska anrop var vanligt förr, eftersom man ’minskade’ antalet I/O-operationer mot programbiblioteket, eftersom ’alla’ program laddades in samtidigt.

I dag skulle man överväga denna lösning p g a de negativa konsekvenserna med att ALLA laddmoduler som innehåller ett berört subprogram måste länkas om för att inkludera den nya ändringen i subprogrammet.

Not.

Vissa kompilatordirektiv kan påverka statiskt eller dynamiskt beteende.

Överföra parametervärden

- Parameteröverföring - By Content
  - mottagande program har en pekare i *Linkage Section* som kommer att peka ut en kopia av överfört data
  - förändringar av det anropade programmet sker i kopian av anroparens data, som förblir oförändrat

```
Program-Id. PGM1.
Data Division.
Working-Storage Section.
01 Pgmnamn  Pic X(08) Value 'PGM2'.
01 WS-Datat . .

Procedure Division.
    Call Pgmnamn
    Using By Content WS-Datat
```

```
01 WS-Datat . .

Program-Id. PGM2.
Data Division.
Linkage Section.
01-LS-Datat. . .

Procedure Division Using LS-Datat.

    Move 'Ändring' to LS-Datat
```


Här vill jag visa att det mottagande programmet har en bild över parametern som sändande programmet överför.

Bilden av elementet eller gruppen skall ha samma utseende i de båda programmen, men det mottagande programmet har sin bild i `Linkage Section`. Namnet på strukturen eller elementen behöver dock ej vara samma. Som du ser så har det mottagande programmet en `Using`-formulering i `Procedure Division` som refererar en 01-nivå i `Linkage Section`. Denna 01 är egentligen en pekare som pekar på en kopia av det sändande programmets parameter.

Local-Storage Section

- Hela sektionen återställs till initialvärden vid varje anrop

```
Identification Division.
Program-ID. PGM1.
Data Division.
Working-Storage Section.
01 Prog  Pic X(08) Value 'PGM2'.
01 WS-Parm1.
   . . .
77 WS-Summa Pic 9(05)V99.

Procedure Division.
    Call Pgm2 Using WS-Parm1
              WS-Summa   Ny Local-Storage
                         Ny WS
    . . .
    Call Pgm2 Using WS-Parm1,
              WS-Summa   Ny Local-Storage
                         Samma WS
    GoBack
    .
```

```
Identification Division.
Program-ID. PGM1.
Data Division.
Working-Storage Section.
01 WS-Data . .
Local-Storage Section.
77 Resultat   Pic 999V99 Value Zero.
Linkage Section.
01 LS-Parm1.
   . . .
77 LS-Summa Pic 9(05)V99.
Procedure Division Using
          LS-Parm1 LS-Summa.
    Add . . . giving Resultat
    . . .
    Move Resultat to LS-Summa
    Exit Program
    .
```

Local-Storage Section.
77 Resultat Pic 999V99 Value Zero.

Beskrivningar som görs i ett subprograms `Working-Storage Section`, är bestående mellan olika anrop till programmet.

Beskrivningar i `Local-Storage Section` är inte bestående eftersom detta utrymme initieras automatiskt vid varje nytt anrop. På detta sätt så kan du alltså ha både bestående och icke bestående variabler i ett subprogram, beroende på var du beskriver variablerna.

Nästlade program

- Flera program kan ingå i samma källkod
  - ett program påbörjas med `Identification Division` och avslutas med `End Program` *pgmnamn*

```
Identification Division.
Program-Id. PGM1
. . .
Identification Division.
Program-Id. PGM11.
. . .
End Program PGM11.
End Program PGM1.
```


  - övriga `Division` är valfria

Nästlade program, eller ”inbakade program” är program som ingår i andra program.

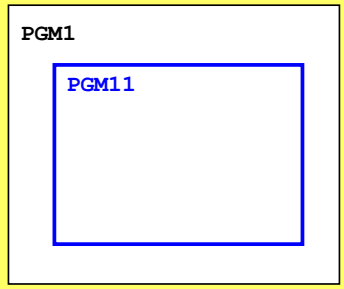
Varje enskilt programs omsluts av uttrycken `Identification Division` och `End Program pgmnamn`.

Ett yttre program kan bestå av flera underordnade program. För att exekvera koden i dessa programs `Procedure Division`, gör du ett normalt `Call`-anrop. Du kan givetvis överföra parametrar på vanligt sätt.

Nästlade program

- Directly Contained
  - ett nästlat program som omsluts av ett yttre program

```
Identification Division.
Program-Id. PGM1.
. . .
  Identification Division.
  Program-Id. PGM11.
  . . .
  End Program PGM11.
End Program PGM1.
```



  - endast det yttre programmet får innehålla *Configuration Section*

Ett *Directly Contained Program* är ett program som omsluts av ett yttre program. Det är bara det yttre programmet som kan innehåll en `Configuration Section`.

Översikt

- Inbyggda funktioner (2)
  - - identifieras med funktionsnamn

```
funktionsnamn [(arg1, arg2,...)]
```

```
Compute Res = En-Funktion(Data1,Data2)
```

✓Kräver beskrivning i Environment Division – Configuration Section

```
Repository.
 Function(func1, func2,...)|All Intrinsic.
```

```
Environment Division.
Configuration Section.
Repository.
    Function All Intrinsic.
```


Om `Environment Division` har kompletterats med paragrafen `Repository` samt uttrycket `Function (….)` så kan en funktion identifierat med endast sitt funktionsnamn.

Översikt

- Inbyggda funktioner – syntaxregler
  - eventuella argument omsluts av parentes

```
Function  funktionsnamn [(arg1,arg2,...)]
```

  - funktionsanropet blir ett numeriskt värde eller en sträng
    - hela eller delar av den returnerade strängen kan användas

```
Move Function Funktionen(Data1,Data2)(n1:n2) to Res
```

Värde

Ref Mod

COBOLSKOLAN IBM-UTBILDNING 58

Vanligen så kräver en funktion att du anger ett eller flera argument eller parametrar. Det finns dock undantag till detta.

I de fall när funktionsanropet returnera en sträng, så kan du använda *Reference Modification* för att ange att du endast vill använda en del av den returnerade strängen (*substring*)

Översikt

- Inbyggda funktioner – returvärden
  - alfanumerisk
    - ✓tillhör klassen *Alphanumeric*, underförstått *Display-format*
    - ✓kan användas som alfanumeriska argument till andra funktioner
    - ✓*reference modification* är tillåtet
  - numeric
    - ✓tillhör klassen *Numeric*, numeriskt hel- eller decimaltal med tecken
    - ✓kan användas som numeriska argument till andra funktioner
    - ✓kan INTE användas där Integer-argument krävs
  - integer
    - ✓tillhör klassen *Numeric*, numeriskt heltal med tecken
    - ✓kan användas som Integer-argument till andra funktioner
  - se COBOL Language Reference för detaljer

COBOLSKOLAN IBM-UTBILDNING 59

Inbyggda funktioner används i olika uttryck och ger ett returvärde i exekveringsläge. Värdet kan vara av olika typ beroende på funktion som används.

Alfanumeriskt returvärde är av typen *Alphanumeric*. Returvärdet kan även användas som argument till andra funktioner som kräver alfanumeriska argument. Detta innebär att en funktion kan ta en annan funktion som argument, vilket gör uttrycken kraftfulla.

Alfanumeriska returvärden kan användas i sin helhet, eller bara en del av det genom *Reference Modification*.

Funktioner som ger numeriska resultat kan användas i olika aritmetiska operationer. Vissa funktioner returnerar numeriska tal med tecken och kanske även decimaler.

Vissa funktioner ger som resultat en heltalsvariabel d v s en `Integer`.

Beskrivningen av funktionerna (*Table of Functions)* anger vilka datatyper som argumenten kräver samt returnerar.

Komplett beskrivning finns i manualen *Enterprise COBOL for z/OS – Language Reference*

Översikt

- COBOL använder tre (3) datumformat
  - Gregorian Date, Standard Date
    - åtta siffror i formatet **YYYYMMDD**
    - området 1 januari (Måndag) 1601 till 31 December 9999
    - **MM** från 1 till 12
    - **DD** från 1 till 31 beroende på månad
  - Integer Date
    - ett numeriskt värde mellan 1 och 3.067.676
    - representerar antalet dagar sedan den 31 December 1600
  - Julian Date
    - ett värde med sju siffror i formatet **YYYYDDD**
    - **DDD** är ett värde mellan 1 och 366 beroende på årtal

COBOLSKOLAN IBM-UTBILDNING 60

Grunden till en enkel och effektiv användning av datum och tid är att det finns en bra grundreferens. COBOL använder sig av den Gregorianska kalendern, vars första dag, en måndag, var den 1 januari 1601.

Det gregorianska datumformatet är YYYYMMDD, och är resultat från eller ett argument till flera datumfunktioner. Formatet kallas *Gregorian Date* eller ibland även *Standard Date*.

Integer Date är ett annat numerisk format och representerar dagnummer i den gregorianska kalendern.

Dag nummer 1 kalendern var Måndagen den 1 januari 1601.

Flera funktioner ger som resultat detta format, eller kräver det som argument.

Formatet *Julian Date*, YYYYDDD ser vi ofta i jobbloggar etc.

Från manualen IBM Enterprise COBOL for z/OS: Programming Guide

"INTDATE(ANSI) instructs the compiler to use the 85 COBOL Standard starting date for integer dates used with date intrinsic functions. Day 1 is Jan 1, 1601. INTDATE(LILIAN) instructs the compiler to use the Language Environment Lilian starting date for integer dates used with date intrinsic functions. Day 1 is Oct 15, 1582."

Current-Date

- Exempel

```
Working-Storage Section.
01 CurrDate          Pic X(08).
01 DatumMeddelande   Pic X(80).
Procedure Division.
    Move Function Current-Date to CurrDate  *> 21 Automatisk trunk 8
    String
        'I dag är det den '
        CurrDate(7:2)
        '/'
        CurrDate(5:2)
        ', '
        CurrDate(1:4)       Delimited by size
        Into DatumMeddelande
    End-String
    Display DatumMeddelande
    GoBack
    .
```

```
I dag är det den dd/mm, 202x
```


I detta exempel visar jag hur du kan använda delar av `Current-Date` för att bygga en ny datumformulering.