Föreläsning 5

- Mer om funktioner och procedurer
  - (se bilder från föreläsning 4)
  - Kommandon och uttryck
  - Referentiell transparens
  - Funktionsprocedurer
  - Sidoeffekter

  - Parameteröverföringsmetoder
  - Begränsning på funktioners resultattyp
- Moduler
  - Varför moduler?
  - Vad är en modul?
  - Kompilering av moduler
  - Ett programmeringsexempel
  - Moduldiagram
  - Något om bra modulindelning

Funktioner och procedurer

Parameteröverföringsmetoder

- Det finns två sätt att skicka argument till procedurer och funktioner. Det första är

Värde-parametrar

Det "normala" sättet. Samma som i funktionella språk.

- Funktionsdefinition:

```
PROCEDURE f(x:REAL);
```

- Anrop:

Argumentet är ett godtyckligt uttryck (av rätt typ)

```
f(3.0*t+2.3)
```

- Vad som händer vid anrop:

1. Argumentuttryckets *värde* beräknas. Kalla det v.
2. Parametern (**x** ovan) blir en lokal variabel med v som startvärde.
3. Kommandona i procedurkroppen utförs.

Funktioner och procedurer

Parameteröverföringsmetoder

- Det finns två sätt att skicka argument till procedurer och funktioner. Det andra är:

Variabel-parametrar

Finns i någon variant i de flesta imperativa språk.

- Funktionsdefinition:

```
PROCEDURE f(VAR x:REAL);
```

- Anrop:

Argumentet måste vara en variabel (av rätt typ)

```
f(t)
```

- Vad som händer vid anrop:

2. Parametern (**x** ovan) blir ett smeknamn för variabeln i anropet (**t** ovan).
3. Kommandona i procedurkroppen utförs.

Funktioner och procedurer

Funktioners resultattyp

Problem

- I Modula-2 får funktioners resultattyp inte vara en sammansatt typ.

```
Date = RECORD
          year:Year;
          month:Month;
          day:Day
         END;

PROCEDURE NextDay(date:Date):Date;  (* FEL! *)
  VAR nextday:Date;
BEGIN
  ...
  ...
  RETURN nextday
END NextDay;
```

Lösning

- Gör om funktionen till en procedur och lämna svaret via en VAR-parameter

```
PROCEDURE NextDay(date:Date; VAR nextday:Date);
BEGIN
  ...
  ...
  (* RETURN tas bort *)
END NextDay;
```

- Varför? ...

Moduler

Hur organiserar man stora program?

- Det måste finnas ett sätt att dela upp program i delar och ge delarna namn.
  - Stora problem löses genom att dela in dem i delproblem. Detta måste kunna avspeglas i programstrukturen.
  - Namngivning underlättar också återanvändning.

Namngivning av programdelar

- Hur ska program delas upp i delar?
- Imperativt program = följd av kommandon som datorn ska utföra.
- Procedur = namngiven följd av kommandon.

  Räcker detta?

- Programmet manipulerar data av något slag.
- Det är bra att kunna definiera datastrukturer och ge dem namn.
- Programstruktur i språket Pascal...

  (state of the art 1970)

Moduler

Mjukvarukrisen

- Det räcker inte att dela upp programmen i procedurer som skrivs i en lång rad efter varandra.
- Procedurer är inte tillräckligt bra som byggklossar.
- Ofta innehåller program ett antal datatyper och för varje typ ett antal tillhörande procedurer.
- Ett naturligt sätt att organisera ett program är att gruppera en datatyp med tillhörande procedurer till en enhet. Man får då vad som kallas en *modul*.

Moduler

Vad är en modul?

- Modul = en samling av konstanter, typer, variabler och procedurer, betraktad som en enhet (med ett namn).
- Moduler är bra byggklossar!
- En modul kan också ses som en "svart låda".

  Det finns vissa bestämda sätt att komma åt innehållet.

  Man behöver inte veta hur en modul ser ut inuti för att kunna använda den.

  Man kan ändra inuti en modul utan att det syns utifrån.

Moduler

Moduler består av två delar

Definitionsdelen

- Beskriver vad man kan göra med "lådan" utifrån.
- Konstanter, typer, variabler & procedurer som ska synas utifrån (*exporteras*) räknas upp.
- Procedurkroppar skrivs ej här.
- Det som exporteras kan importeras i en annan modul.

```
DEFINITION MODULE m;
  ...
END m.
```

Implementeringsdelen

- Beskriver hur "lådan" ser ut inuti.
- Fullständiga definitoner av de exporterade procedurerna ges här.
- Det kan också finnas definitioner av konstanter, typer, variabler och procedurer som inte ska exporteras.

```
IMPLEMENTATION MODULE m;
  ...
  ...
END m.
```

Moduler

Praktiska detaljer (gäller Mocka-systemet)

Filnamn

- Definitionsdelen till en modul m ska ligga i filen m.md.
- Implementeringsdelen till m ska ligga i filen m.mi.

Kompilering

- Definitionsdelen kompileras först

```
mcs m
```

Filen *m*.d skapas.

- Sedan implementeringsdelen

```
mcc m
```

Filen *m*.o och *m*.r skapas.

Omkompilering efter ändringar

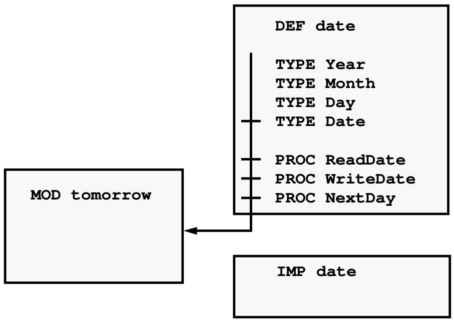
- Ändrar man i implementeringsdelen m.mi måste denna kompileras om, men inga andra moduler.
- Ändrar man definitionsdelen m.md måste den, implementeringsdelen (m.mi) och alla moduler som importerar m kompileras om.

- Kompilatorn håller själv reda på vad som behöver kompileras!

Moduler

Programmeringsexemplet

Moduldiagram



- Programmet, något förkortat, finns sist i detta häfte.
- Hela programmet finns i WWW

http://www.cs.chalmers.se/kurser/naptv/del2/forel/f05/

Moduler

Om bra modulindelning

(Avsnitt 11.4 i boken)

- Syftet med modulindelning är att underlätta utveckling och underhåll av stora program.

Samhörighet (cohesion)

- Saker som ligger i samma modul ska "höra ihop":

En datatyp + operationer på denna

Procedurer för ett visst ändamål (t ex InOut, ...)

Beroende (coupling)

- Moduler ska vara så fristående från varandra som möjligt.

Ju mer man kan förändra inuti en modul utan att andra moduler påverkas, desto lättare blir det att rätta fel eller göra utökningar i ett stort program.

Datatyper bör vara abstrakta (så kan man byta ut typen...).

Variabler bör normalt inte exporteras.

Moduler

Värt att notera om moduler

- Definitionsdelen talar om vad modulen exporterar (= vad som kan importeras i andra moduler).
- Definitioner av konstanter, typer och variabler i definitionsdelen blir automatiskt kända i implementeringsdelen.
- Exporterade procedurer måste ges fullständiga definitioner i implementeringsdelen.
- Den som använder en modul ska inte behöva titta i implementeringsdelen.
  - Om inte namn & typ säger allt måste man komplettera med en kommentar!

Moduler

Exemplet

Definitionsdelen för datum-modulen

```
DEFINITION MODULE date;
(* Modul för hantering av datum. *)

TYPE Year = CARDINAL;

     Month = (Jan,Feb,Mar,Apr,Maj,Jun,
              Jul,Aug,Sep,Oct,Nov,Dec);
     Day = [1..31];

     Date = RECORD
              year:Year;
              month:Month;
              day:Day
            END;

PROCEDURE ReadDate(VAR date:Date);
  (* Denna procedur läser in ett datum.
   * Årtal, månadsnummer och dagnummer
   * matas åtskiljda av mellanslag
   *)

PROCEDURE WriteDate(date:Date);
  (* Denna procedur skriver ut ett datum. *)

PROCEDURE NextDay(date:Date; VAR nextday:Date);
  (* Denna "funktion" räknar ut datumet för
   * dagen efter ett givet datumet. *)

END date.
```


Moduler

Exemplet

Implementeringsdelen för datum-modulen

```
IMPLEMENTATION MODULE date;
(* Implementering av modulen för hantering av datum. *)

FROM InOut IMPORT WriteString,WriteLn,WriteCard,ReadCard;

PROCEDURE ReadDate(VAR date:Date);
VAR n:CARDINAL;
BEGIN
  ...
END ReadDate;

PROCEDURE LastDayInMonth(year:Year; month:Month):Day;
BEGIN
  ...
END LastDayInMonth;

PROCEDURE NextDay(date1:Date; VAR nextday:Date);

BEGIN
  (* beräkna nästa dags datum *)
  ...
END NextDay;

PROCEDURE WriteDate(date:Date);
BEGIN
  ...
END WriteDate;

END date.
```


Moduler

Exemplet

Huvudprogrammet

```
MODULE tomorrow;

FROM InOut IMPORT WriteString,WriteLn;
FROM date IMPORT ReadDate, WriteDate, NextDay, Date;

VAR date,nextday:Date;(* inmatat datum och nästa dags datum *)


BEGIN
  (* skriv ut ledtext *)
  WriteString("Mata in ett datum (år mån dag): ");
  (* läs in ett datum *)
  ReadDate(date);

  (* Skriv ut nästa dags datum *)
  WriteString("Nästa dags datum är: ");
  NextDay(date,nextday);
  WriteDate(nextday);
END tomorrow.
```