Föreläsning 9

- Dynamiska datastrukturer (forts)

  Några exempel på länkade strukturer:

  listor, träd, köer, dubbellänkade listor, ...
- Abstrakta typer

  "Hemliga" typdefintioner (eng: *opaque types*)

  Moduler med internt tillstånd

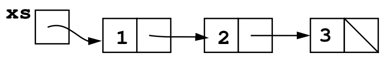
Dynamiska datastrukturer

Förra gången såg vi länkade listor...

```
        TYPE List = POINTER TO ListNod;
             ListNod = RECORD
                         head: INTEGER;
                         tail: List
                       END;

    VAR xs:List;

    xs = [1,2,3]
```

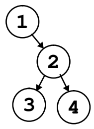


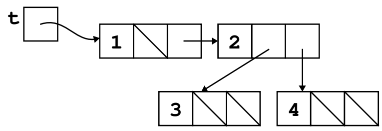
Binära träd

```
    TYPE Tree = Pointer TO TreeNode;
         TreeNode = RECORD
                      info: Info;
                      left, right: Tree
                    END;

    VAR t: Tree;

    t =
```





Dynamiska datastrukturer

Köer

Operationer

Skapa en tom kö

Testa om en kö är tom

Sätt in ett nytt element sist i kön

Ta ut det första elementet i kön

SML

- En kö kan lagras som en lista, första elementet först

  Insättning: insert x queue = queue @ [x], långsamt!

  Uttagning: hd, snabbt
- Första elementet sist istället:

  Insättning, insert x queue = x::queue, snabbt

  Uttagning blir långsamt

Modula-2

- Lätt att göra köer där både insättning och uttagning går snabbt!

  vgv

Dynamiska datastrukturer

Köer

Modula-2

- Använd en lista + en pekare till sista elementet.

```
TYPE Queue =
          RECORD
            first, last: POINTER TO ListNod
          END;

     List = POINTER TO ListNod;
     ListNod = RECORD
                 head: Info;
                 tail: List
               END;
```

Listoperationerna

- Destruktiva

```
PROCEDURE emptyQueue(VAR q:Queue);
...
PROCEDURE isEmpty(q:Queue);
...
PROCEDURE insert(x: Info; VAR q:Queue);
...
PROCEDURE remove(VAR x:Info; Var q:Queue);
...
```

Dynamiska datastrukturer

Dubellänkade strukturer

Dubellänkade listor

...

```
TYPE List2 = POINTER TO ListNod2;
     ListNod2 = RECORD
                  info: Info;
                  next, previous: List2
                END;
```

Träd med bakåtpekare

...

```
TYPE Tree2 = POINTER TO TreeNod2;
     TreeNod2 = RECORD
                  info: Info;
                  up, left, right: Tree2
                END;
```


Abstrakta datatyper

Motivering

- Stora program är normalt uppdelade i ett antal moduler.
- 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...).

Abstrakta datatyper

Exempel: en modul för listor, konkret typ

- Definitionsdelen

```
DEFINITION MODULE KonkretaLlistor;

TYPE Info = INTEGER; (* Typen på elementen i listan *)
     List = POINTER TO ListNod;
     ListNod = RECORD
                 head:Info;
                 tail:List
               END;

PROCEDURE nil():List;  (* tomma listan *)
PROCEDURE cons(x:Info; xs:List):List; (* cons lägger... *)

...
END KonkretaLlistor.
```

- Implementeringsdelen

```
IMPLEMENTATION MODULE KonkretaLlistor;
FROM Storage IMPORT ALLOCATE,DEALLOCATE;

PROCEDURE nil():List;
BEGIN
  RETURN NIL
END nil;

PROCEDURE cons(x:Info; xs:List):List;
BEGIN
  ...
END cons;

...
END KonkretaLlistor.
```


Abstrakta datatyper

Exempel: en modul för listor, abstrakt typ

- Definitionsdelen

```
DEFINITION MODULE AbstraktaListor;

TYPE Info = INTEGER; (* Typen på elementen i listan *)
TYPE List; (* Typen List är abstrakt (hemlig). *)

PROCEDURE nil():List;  (* tomma listan *)
PROCEDURE cons(x:Info; xs:List):List; (* cons lägger... *)
...
END AbstraktaListor.
```

- Implementeringsdelen

```
IMPLEMENTATION MODULE AbstraktaListor;
FROM Storage IMPORT ALLOCATE,DEALLOCATE;

TYPE List = POINTER TO ListNod;
     ListNod = RECORD
                 head:Info;
                 tail:List
               END;

PROCEDURE nil():List;
BEGIN
  RETURN NIL
END nil;

PROCEDURE cons(x:Info; xs:List):List;
BEGIN
  ...
END cons;
...
END AbstraktaListor.
```

Operationer på konkreta & abstrakta typer

Operationer som fungerar på alla typer

- Deklaration av variabler

  VAR xs,ys:List;

- Tilldelning

  xs:=ys;

- Skicka argument till procedurer & funktioner

  IF null(xs) THEN ...

- Ta emot resultat från funktioner

  ys:=cons(3,xs);

Operationer som kräver att typen är känd

xs:=NIL;

- Det måste vara känt att **xs** är en pekare.

xs^.head:=5;

- Det måste vara känt att **xs** är en pekare till en post med en del som heter **head**.

y:=2*x+1;

- Det måste vara känt att **y** och **x** har någon heltalstyp.

Exempel: En modul för mängder

```
DEFINITION MODULE SetOfInt;
(* En modul för mängder av heltal *)

TYPE Set; (* Typen är abstrakt (hemligt). *)

TYPE  Info = INTEGER; (* Typen på elementen i mängden *)

PROCEDURE emptySet():Set;  (* tomma mängden *)

PROCEDURE insert(x:Info; VAR xs:Set);
(* insert lägger till x till mängden xs *)

PROCEDURE isEmpty(xs:Set):BOOLEAN;
(* isEmpty testar om en mängd är tom *)

PROCEDURE member(x:Info; xs:Set):BOOLEAN;
(* member undersöker om x ingår i mängden xs *)

PROCEDURE delete(x:Info; VAR xs:Set);
(* delete tar bort x ur mängden xs *)


PROCEDURE union(xs1,xs2:Set):Set;
(* unionen av två mängder *)

PROCEDURE intersection(xs1,xs2:Set):Set;
(* snittet av två mängder *)

END SetOfInt.
```


Implementering av modulen för mängder

Representation av mängder (ett sätt)

- En mängd lagras som en lista

  TYPE Set=List; (* importera List... *)

  Listan ska vara sorterad

  Listan får inte innehålla dubbletter

- Villkoren kallas en *invariant*.

Operationerna

- **insert** ser till att listan är sorterad och att inga dubbletter uppstår.
- Genom att utnyttja att listan är sorterad kan **member**, **delete**, **union** och **snitt** göras snabbare...
- De uppsnabbade operationerna skulle kunna ge fel resultat om listorna inte var sorterade och dubblettfria.
- Om man utnyttjar att mängder representeras som listor kan man göra godtyckliga list-operationer på dem...

Poängen

- Genom att göra typen **Set** abstrakt kan man garantera att invarianten alltid är uppfylld (dvs att ...)

Ett annat sätt att representera mängder

- Om endast tal mellan 0 och 1000 är intressanta:

```
TYPE Number = [0..1000];
     Set = ARRAY Number OF BOOLEAN;
```

- Idé: **s[i]** är **TRUE** om **i** ingår i mängden **s**.
- **insert**, **delete** & **member** blir enkla och snabba:

```
PROCEDURE insert(i:Number; VAR s:Set);
BEGIN
  s[i]:=TRUE
END insert;

PROCEDURE member(i:Number; s:Set);
BEGIN
  RETURN s[i]
END member;
```

Problem

På grund av en irriterande begränsning i Modula-2 kan inte typen **Set** ovan göras abstrakt.

- Endast pekartyper, **CARDINAL**, och **INTEGER** kan göras abstrakta!

Lösning!? Göra typen konkret?

Abstrakta datatyper

Moduler med internt tillstånd

- Antag att ett program använder modulen **`SetOfInt`**, men bara innehåller *en* variabel av typen **`Set`**.

  Då kan man lägga variabeln i mängd-modulen istället!

  - Man slipper skicka med mängden som argument.
  - Man behöver inte exportera typen för mängder från mängd-modulen!
- Variablerna i en modul kallas modulens *tillstånd*.

Abstrakta datatyper

En modul med en mängd som internt tillstånd

```
DEFINITION MODULE SetOfNum;
(* En modul för mängder av heltal mellan 0 och 1000 *)

TYPE  Number = [0..1000];
      Info = Number;   (* Typen på elementen i mängden *)

(* Denna modul hanterar en enda mängd som lagras internt. *)

PROCEDURE emptySet();
(* tömmer mängden *)

PROCEDURE insert(x:Info);
(* Lägger till x till mängden. *)

PROCEDURE isEmpty():BOOLEAN;
(* isEmpty testar om mängden är tom *)

PROCEDURE member(x:Info):BOOLEAN;
(* member undersöker om x ingår i mängden *)

PROCEDURE delete(x:Info);
(* delete tar bort x ur mängden *)

END SetOfNum.
```


Abstrakta datatyper

Implementeringdelen

```
IMPLEMENTATION MODULE SetOfNum;

TYPE Set = ARRAY Number OF BOOLEAN;

(* Här är variabeln som innehåller den internt lagrade
   mängden. *)

VAR theSet : Set;

PROCEDURE emptySet();   (* tömmer mängden *)
VAR n:Number;
BEGIN
  FOR n:=MIN(Number) TO MAX(Number) DO
    theSet[n]:=FALSE
  END
END emptySet;

PROCEDURE insert(x:Info); (* Lägger till x till mängden. *)
BEGIN
  theSet[x]:=TRUE;
END insert;
```

... implementering av isempty, member & delete ...

```
BEGIN
  (* initiering av mängden *)
  emptySet()
END SetOfNum.
```

- **`MAX`** & **`MIN`** ger det största resp minsta värdet i en typ.

Abstrakta datatyper

Angående moduler med internt tillstånd

- Mängdmodulen hanterar endast *en* mängd.

  Även om modulen importeras på många ställen i ett stort program finns det bara en enda mängd i programmet.
- Moduler utan internt tillstånd kan användas av flera moduler i ett program utan att de stör varandra.
- Istället för använda internt tillstånd kan man göra typen känd, och ange med kommentarer att typen egentligen borde vara abstrakt.

```
DEFINITION MODULE SetOfNum;
(* En modul för mängder av heltal mellan 0 och 1000 *)

TYPE Number = [0..1000];
     Info = Number;

     Set = ARRAY Number OF BOOLEAN;
(* Typen Set ska betraktas som abstrakt (hemligt)!! *)

PROCEDURE emptySet():Set;
(* tomma mängden *)

PROCEDURE insert(x:Info; VAR xs:Set);
(* insert lägger till x till mängden xs *)

...

END SetOfNum.
```

- Man får ingen kontroll på att typen bara används abstrakt.