Föreläsning 7

- Datastrukturer
    - Datastrukturer i Modula-2 jämfört med SML
    - Statiska datastrukturer
    - Samband mellan data- och programstrukturer
    - Dynamiska datastrukturer
    - Länkad lagringsstruktur

Datastrukturer

Jämförelse mellan SML och Modula-2

Exempel 1

```
datatype Frukt = Apple | Paron | Banan

TYPE Frukt = (Apple,Paron,Banan)
```

Exempel 2

```
datatype Complex = Rect of real*real

TYPE Complex = RECORD
                  re,im: REAL
               END
```

re
im

Datastrukturer

Jämförelse mellan SML och Modula-2

Exempel 3

- SML

```
datatype Vehicle = Car of Make*Regnum*real
                 | Truck of Regnum*real
                 | Bike
```

...

- Modula-2

```
TYPE VehicleKind = (Car, Truck, Bike);

TYPE Vehicle =
  RECORD
    CASE kind:VehicleKind OF
      Car:   m:Make; r:Regnum; cweight:REAL
    | Truck: r:Regnum; tweight:REAL
    | Bike:
    END
  END
```

kind		
m	r	
r	tw	
cw		

Datastrukturer

Samband mellan program- och datastruktur

- Tre grundläggande strukturer: sekvens, upprepning & val

Struktur	Program	Data
Sekvens	*kommando*$_1$; *kommando*$_2$; ... *kommando*$_n$	**RECORD** v_1: *typ*$_1$; v_2: *typ*$_2$; ... v_n: *typ*$_n$ **END**
Upprep...	**FOR i:=1 TO N DO** *kommandon* **END**	**ARRAY [1..N] OF** *typ*
Val	**CASE** *uttryck* **OF** k_1: *kommandon*$_1$ | k_2: *kommandon*$_2$ ... | k_n: *kommandon*$_n$ **ELSE** *kommandon* **END**	**RECORD** **CASE v:**U **OF** k_1: *delar*$_1$ | k_2: *delar*$_2$ ... | k_n: *delar*$_n$ **ELSE** *delar* **END** **END** - Poster med varianter

Datastrukturer

Rekursiva typer?

- ML-typen

```
datatype Tree = Leaf
              | Node of int*Tree*Tree
```

- blir på samma sätt

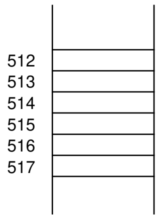
```
TYPE TreeKind = (Leaf, Node);

TYPE Tree =
  RECORD
    CASE kind:TreeKind OF
      Leaf:
    | Node: n:INTEGER; left,right:Tree
    END
  END;
```

- FUNGERAR INTE!!!

Datastrukturer

Minnet i en von Neumann-dator

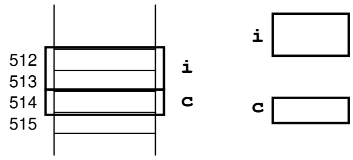


- Minnet linkar en ARRAY!

```
VAR minnet : ARRAY [0..Max] OF BYTE
```

- Varje byte har en egen *adress.*
- Dataobjekt i Modula-2 lagras i en sammanhängande följd av bytes med början på en bestämd adress.

```
VAR i:INTEGER;
    c:CHAR;
```



- Kompilatorn väljer adresser för alla variabler i programmet.

Datastrukturer

- Alla objekt av en viss typ har samma storlek.

  SIZE(T) ger storleken på objekt av typen **T**

- I vårt Modula-2-system

```
SIZE(CHAR)=1
SIZE(INTEGER)=4
SIZE(REAL)=4

SIZE(ARRAY [1..20] OF REAL) = 20 * 4 = 80

SIZE(RECORD x,y:REAL END) = 4 + 4 = 8
```

- Storleken på poster med varianter bestäms av den största varianten

```
SIZE(Vehicle) = 4 + MAX(10+6+4,6+4,0) = 24
```

- Det är lätt att räkna ut adressen till delarna i en RECORD eller elementen i en ARRAY.

  ...

- Storleken på **Tree**?

```
SIZE(Tree) = 4 + MAX(0,4+2*SIZE(Tree))
```

  Ekvation som saknar lösning!

Datastrukturer

Dynamiska datastrukturer

- Listor och träd är exempel på *dynamiska datastrukturer.*
  - Listors längd kan variera. Det finns ingen övre gräns.
  - Träds form kan variera.

Statiska datastrukturer

- RECORD- & ARRAY-typer är exempel på *statiska datastrukturer.* Alla värden av samma typ är lika stora har exakt samma struktur.
- Modula-2 tillhandahåller bara typer för statiska datastrukturer.

  Poster med varianter tillåter viss variation.

Dynamiska datastrukturer i Modula-2

- Hur hanterar man dynamiska strukturer när det bara finns typer för statiska strukturer?

Listor

- Kanske inte behöver lagras alls

  Ex: ett program som läser in och summerar en talföljd

- Man kan använda en ARRAY + en extra variabel med index till sista använda elementet.

  Ex: ett program som läser in en talföljd och skriver ut den baklänges

- Man kan använda en ARRAY och lagra ett speciellt värde efter det sista använda elementet.

  Ex: strängar

- Index: första elementet = 0, elementet efter i = i +1
- Problem: måste bestämma en övre gräns
- Hur implementerar man de vanliga list-operationerna?

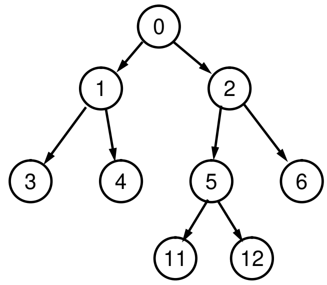
  cons, hd, tl, append, ...

  Övning 2, uppgift 8.

Dynamiska datastrukturer i Modula-2

Träd

- Träd är krångligare än listor, eftersom de förgrenar sig.
- Man kan numrera platserna i ett binärt träd:



- Roten får index 0

  vänster barn till i = 2*i+1

  höger barn till i = 2*i+2

  förälder till i = (i-1) DIV 2

- Kan använda en ARRAY + ett speciellt värde i löven.

Problem med den nyss beskrivna metoden

Man måste bestämma en övre gräns

- För låg: strukturen får ej plats, programmet fungerar inte.
- För hög: slöseri med minnesutrymme
- Lösning?

  Pekare och dynamisk minnesallokering!

Vissa operationer kräver att hela eller stora delar av datastrukturer kopieras

  Konkatenering av listor.

  Att sätta in ett nytt element i en lista

  Att sätta ihop två delträd till ett nytt träd.

- Lösning?

  Länkad lagringsstruktur!

Länkad lagringsstruktur

Idé

- Det är "dumt" att ha ett förutbestämt sätt att räkna ut index till nästa element i en lista resp till barnnoderna i ett träd.
- Lagra index till efterföljare/barn i varje nod!
- Låt en lista/ett träd representeras av indexet till första elementet/roten.

Listor

```
CONST Max = 1000;

TYPE Elem = INTEGER;
     List = [0..Max];
     Node = RECORD
              hd:Elem; tl:List
            END;

VAR node : ARRAY [1..Max] OF Node;
    ledig : [1..Max];

      ...
```

Träd...

Dynamiska datastrukturer

Länkade strukturer

Notera

- Många listor kan lagras.
- Alla listelement lagras i en gemensam array.
- Variabler av typen `List` innehåller inte hela listor, utan bara ett index som säger var i arrayen listan början.
- Varje plats i arrayen innehåller ett listelement och indexet till nästa element i listan.

Problem

- Det behövs ett sätt att hålla reda på vilka platser i arrayen som är lediga/upptagna.
- Arrayen har en begränsad storlek.

Dynamiska datastrukturer

Modulen Lists1

```
DEFINITION MODULE lists1;

(* Länkade lagringsstrukturer:
   listor, utan pekare & dynamisk minnesallokering.
 *)

CONST Max = 1000;  (* max så här många listelement totalt *)

TYPE List = [0..Max]; (* en lista är ett index till en nod
                         (eller nil) *)

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

(* === Konstrurerare === *)

PROCEDURE nil():List; (* tomma listan *)

PROCEDURE cons(x:Info; xs:List):List;
(* cons lägger till x först i listan xs *)

(* === Predikat === *)

PROCEDURE null(l:List):BOOLEAN;
(* null testar om en lista är tom *)

(* Selektorer *)

PROCEDURE hd(l:List):Info;
(* hd ger första elementet i listan l *)

PROCEDURE tl(l:List):List;
(* tl ger resten av listan l utan första elementet *)

END lists1.
```


Dynamiska datastrukturer

Modulen Lists1

```
IMPLEMENTATION MODULE lists1;

(* Länkade lagringsstrukturer:
   listor,
   utan pekare & dynamisk minnesallokering.
 *)

TYPE NodIndex = [1..Max];

     ListNod = RECORD
                 head:Info;
                 tail:List
               END;

VAR nod : ARRAY NodIndex OF ListNod;

    ledig:NodIndex; (* index till den första lediga noden *)

PROCEDURE ny(VAR n:NodIndex);
(* ny ger indexet till en ny, hittills oanvänd nod *)
BEGIN
  (* tag den första lediga noden *)
  n:=ledig;
  INC(ledig);
END ny;

PROCEDURE null(l:List):BOOLEAN;
(* null testar om en lista är tom *)
BEGIN
  RETURN l=0
END null;

PROCEDURE nil():List;
BEGIN
```



```
  RETURN 0;
END nil;

PROCEDURE cons(x:Info; xs:List):List;
(* cons lägger till x först i listan xs *)
VAR i:NodIndex;
BEGIN
  (* tag en ny nod *)
  ny(i);
  (* fyll i den *)
  nod[i].head:=x;
  nod[i].tail:=xs;
  RETURN i
END cons;

PROCEDURE hd(l:List):Info;
(* hd ger första elementet i listan l *)
BEGIN
  RETURN nod[l].head
END hd;

PROCEDURE tl(l:List):List;
(* tl ger resten av listan l utan första elementet *)
BEGIN
  RETURN nod[l].tail
END tl;

BEGIN
  ledig:=1
END lists1.
```

Poster med varianter

Syntax för typer

```
Type = SimpleType | ArrayType | RecordType |
       SetType | PointerType | ProcedureType

SimpleType = ...

ArrayType = ARRAY SimpleType {"," SimpleType} OF Type.

RecordType = RECORD FieldListSequence END.

FieldListSequence = FieldList { ";" FieldList }.

FieldList = IdentList ":" Type
          |
           CASE Ident ":" Ident OF
             Variant { "|" Variant }
           [ELSE FieldListSequence]
           END

Variant = CaseLabelList ":" FieldListSequence.
```

CaseLabelList = *som i* **CASE**-*satser*

...