1.

a. Hur skulle man skriva

```
IF villkor THEN
  satsföljd1
ELSE
  satsföljd2
END
```

om det inte fanns `IF`-satser i Modula-2?

(3 poäng)

b. Ge exempel på följande tre typer av programmeringsfel:

- Typ 1: Fel som upptäcks av kompilatorn då man kompilerar programmet.
- Typ 2: Fel som upptäcks av datorn då man kör programmet.
- Typ 3: Fel som datorn inte upptäcker, men som en människa kan upptäcka då programmet körs.

(3 poäng)

2.

a. Vad skrivs ut då följande program körs?

```
MODULE Rek;

FROM InOut IMPORT WriteCard,WriteLn;

PROCEDURE p(n:CARDINAL);
BEGIN
  IF n>0 THEN
    DEC(n);
    WriteCard(n,2);
    p(n);
    WriteCard(n,2)
  END
END p;

VAR start:CARDINAL;

BEGIN
  start:=4;
  p(start);
  WriteLn
END Rek.
```

(5 poäng)

b. Vad skrivs ut om man ändrar raden

```
PROCEDURE p(n:CARDINAL);
```

i programmet ovan till

```
PROCEDURE p(VAR n:CARDINAL);
```

(4 poäng)

3. Antag att du har tagit jobb på det multinationella mjukvaruföretaget PuzzleSoft. Där håller man för tillfället på att utveckla ett program för att lösa korsord. Din uppgift är att skriva de två mest grundläggande och därmed viktigaste delarna av programmet.

a. Skriv en funktion som testar om ett ord passar mot ett mönster. Ett mönster är ett ord där det förutom bokstäver också kan förekomma frågetecken, "`?`". T ex passar alla ord med fyra bokstäver mot mönstret `????` och alla ord som består av 6 bokstäver och som börjar på `M` och slutar på `A` mot mönstret `M????A`. *(4 poäng)*

b. Skriv en procedur som i en ordlista letar upp alla ord som passar mot ett mönster. Proceduren ska ha tre argument:

- mönstret.
- en öppen `ARRAY`-parameter som är en variabel som ska fyllas i med de ord som passar mot mönstret. Om inte det inte finns plats för alla passande ord hoppar man bara över de som inte får plats.
- en heltalsvariabel där man ska lagra antalet passande ord.

Antag att det finns en global variabel som innehåller ordlistan, representerad som en länkad lista av ord:

```
TYPE Ord = ARRAY [1..20] OF CHAR;

TYPE Ordlista = POINTER TO Ordnod;
     Ordnod = RECORD
                ord:Ord;
                resten:Ordlista
              END;

VAR ordlista: Ordlista;
```

(4 poäng)

4. I ett visst program används följande typer för att lagra listor med hemliga telefonnummer[1]:

```
CONST Max = 1000;

TYPE Number =  [100000..999999];

     NumberList = RECORD
                    number : ARRAY [1..Max] OF Number;
                            (* numren i listan *)
                    count  : [0..Max]
                            (* nuvarande antal nummer i listan *)
                  END;
```

a. Skriv en procedur `Init` som skapar en tom nummerlista och en procedur `WriteList` som skriver ut en nummerlista på terminalen med ett nummer per rad. Procedurerna ska ha följande huvuden:

```
PROCEDURE Init(VAR l:NumberList);
PROCEDURE WriteList(l:NumberList);
```

(4 poäng)

1. Eftersom det rör sig om *hemliga* telefonnummer lagras inga namn, utan endast numren!

b. Skriv en procedur `Insert` för att sätta in ett nytt nummer på rätt plats i en sorterad nummerlista och en procedur `Delete` för att ta bort ett nummer ur en nummerlista. Procedurerna ska ha följande huvuden:

```
PROCEDURE Insert(n:Number; VAR l:NumberList);
PROCEDURE Delete(n:Number; VAR l:NumberList);
```

(9 poäng)

5. I ett visst sammanhang visade det sig att programmet i föregående uppgift var för långsamt. För att snabba upp det bestämde man sig för att använda binära sökträd för att lagra nummerlistor.

a. Skriv typdefinitioner för att lagra nummerlistor som binära sökträd. *(4 poäng)*

b. Skriv nya versioner av procedurerna `Init`, `WriteList` och `Insert` (du kan hoppa över `Delete`) för typerna i del a. De ska ha samma huvuden som tidigare och utifrån sett fungera likadant. *(8 poäng)*

6. Med Write-procedurerna i modulen `InOut` kan man skriva text från vänster till höger, uppifrån och ner. De flesta text-terminaler är dock mer avancerade än så. Genom att skriva ut styrkoder (teckenföljder med speciell betydelse) kan man sudda skärmen, flytta markören (och därmed välja var på skärmen nästa utskrift ska hamna), byta mellan olika teckenstilar (fetstil, understrykning, blinkande, ...), mm.

Denna uppgift går ut på att skriva en modul i Modula-2 för text-terminaler. Låt oss kalla modulen `TextTerminal`.

a. Skriv definitionsdelen till modulen `TextTerminal`. Modulen ska tillhandahålla följande:

- `Clear`, en procedur som suddar skärmen.
- `MoveTo`, en procedur för att flytta markören. Den ska ha rad- och kolumnnummer som parametrar.
- `Style`, en typ för teckenstilar. Det ska finnas fyra stilar: `Normal`, `Bold`, `Underline` och `Blinking`.
- `ChangeStyle`, en procedur som slår på den teckenstil som anges som argument.

(4 poäng)

b. Skriv en implementeringsdel till modulen `TextTerminal`. Hur styrkoderna ser ut skiljer sig förstås mellan olika terminaltyper. Här antar vi att terminalen i fråga följer ANSI-standarden för styrkoder, som säger bland annat följande:

- Styrkoderna börjar med en *startkod*, som består av tecknet Escape (ASCII-kod 27) följt av tecknet "`[`".
- För att sudda skärmen ska man skriva ut startkoden följt av tecknet "`2`" och tecknet "`J`".
- För att flytta markören ska man skriva ut startkoden, radnumret, ett "`;`", kolumnnumret och sist "`H`".
- För att ändra teckenstil ska man skriva ut startkoden, stilkoden och sist "`m`". Följande stilkoder används: "`0`"=normal stil, "`1`"=fetstil, "`4`"=understrykning, "5"=blinkande.

(8 poäng)

Biblioteksmoduler

```
DEFINITION MODULE InOut;

(*      @(#)InOut.def   1.10    91/01/24        SMI     *)

  CONST   EOL = 12C; (* Unix line separator *)

  VAR
    Done   : BOOLEAN; (* Indicates prev operation's Success   *)
    termCH : CHAR;    (* The Char which terminated ReadString *)

  PROCEDURE Read (VAR ch : CHAR );
  (* Done == NOT end-of-file on reading. If true, ch is a     *)
  (* valid character, or is EOL.  If false, ch is 0C and end  *)
  (* of file has been reached.                                *)

  PROCEDURE ReadString (VAR s : ARRAY OF CHAR);
  (* Ignoring leading white space, read a sequence of chars   *)
  (* not including blanks or control chars.  The char which   *)
  (* follows the string is assigned to termCH.  It will not   *)
  (* be seen in subsequent call to any Read routine.          *)

  PROCEDURE ReadInt (VAR x : INTEGER);
  (* Combination of ReadString and conversion to INTEGER.     *)
  (* Syntax : ["+"|"-"] digit { digit } .  Leading blanks are *)
  (* ignored.  Variable termCH is set as in ReadString.       *)
  (* Done == "an integer was successfully read"               *)

  PROCEDURE ReadCard (VAR x : CARDINAL);
  (* Combination of ReadString and conversion to CARDINAL.    *)
  (* Syntax : digit { digit } .  Leading blanks are ignored.  *)
  (* Variable termCH is set as in ReadString.                 *)
  (* Done == "a cardinal was successfully read"               *)

(*      Output Procedures                                     *)

  PROCEDURE Write (ch : CHAR);
  (* Put ch to the output stream.  Write (EOL) is equivalent  *)
  (* to WriteLn.  Other control characters have terminal-     *)
  (* dependent effects.                                       *)

  PROCEDURE WriteLn;
  (* Write a line separator character (EOL).  Under normal    *)
  (* conditions, this will send both carriage return and      *)
  (* line feed to tty device, or the EOL character to a file. *)

  PROCEDURE WriteString (s : ARRAY OF CHAR);
  (* Write the string s to the output stream.  Embedded EOL   *)
  (* characters have the same effect as in Write.  Other      *)
  (* control characters have terminal-dependent effects.      *)

  PROCEDURE WriteInt (x : INTEGER; n : CARDINAL);
  (* Write the integer x with at least n characters of width. *)
  (* Leading zeroes are suppressed.  If n is greater than the *)
  (* needed number of columns, the number is right-justified. *)

  PROCEDURE WriteCard (x : CARDINAL; n : CARDINAL);
```

```
  (* Write the cardinal x with at least n characters of width.*)
  (* Leading zeroes are suppressed.  If n is greater than the *)
  (* needed number of columns, the number is right-justified. *)

END InOut.
```

DEFINITION MODULE MathLib;

```
(*      @(#)MathLib.def 1.8     91/01/24        SMI     *)

PROCEDURE Sqrt    (real : REAL) : REAL;
PROCEDURE Exp     (real : REAL) : REAL;
PROCEDURE Ln      (real : REAL) : REAL;
PROCEDURE Sin     (real : REAL) : REAL;    (* radians *)
PROCEDURE Cos     (real : REAL) : REAL;
PROCEDURE Arctan  (real : REAL) : REAL;
PROCEDURE Entier  (real : REAL) : INTEGER; (* trunc toward neg inf *)
PROCEDURE Power   (real : REAL;
                   exp  : REAL) : REAL;    (* real to the power exp *)
END MathLib.
```

DEFINITION MODULE Storage;

```
(** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
* Module        : Storage
* Project       : Modula-2 standard library
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
* Author        : Jirka Hoppe
* Creation date : 1 Jun 1985
* Version       : 1.8
* Copyright     : Oce - Wissenschaftliches Forschungsinstitut AG
*                 Zurich Switzerland, 1985
* Description   : memory allocation
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Modified by   : Jirka Hoppe
* Mod. date     : 22. August 1985
* Reference     :
* Object        :
* Description   : Procedure 'Change' removed
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

  FROM SYSTEM IMPORT
    ADDRESS;

  EXPORT QUALIFIED
    Allocate, ALLOCATE, DEALLOCATE;

  PROCEDURE Allocate (VAR adr: ADDRESS; units: CARDINAL);

  PROCEDURE ALLOCATE(VAR adr: ADDRESS; units: CARDINAL);

  PROCEDURE DEALLOCATE (VAR adr: ADDRESS; units: CARDINAL);

END Storage.
```


DEFINITION MODULE String;

```
(*      @(#)String.def  1.8     91/01/24        SMI     *)

TYPE
    CompareResult = (less, equal, greater);

PROCEDURE Length    (    str       : ARRAY OF CHAR)  (* in this string     *)
                                   : CARDINAL;       (* count of data chars *)


PROCEDURE Assign    (    source    : ARRAY OF CHAR;  (* copy from this str  *)
                     VAR dest      : ARRAY OF CHAR;  (* to this string      *)
                     VAR success   : BOOLEAN);


PROCEDURE Insert    (    source    : ARRAY OF CHAR;  (* insert this string  *)
                     VAR dest      : ARRAY OF CHAR;  (* into this string    *)
                         index     : CARDINAL;       (* before this char pos*)
                     VAR success   : BOOLEAN);


PROCEDURE Delete    (VAR str       : ARRAY OF CHAR;  (* from this string    *)
                         index     : CARDINAL;       (* 1st char to delete  *)
                         len       : CARDINAL;       (* cnt of chars to del *)
                     VAR success   : BOOLEAN);


PROCEDURE Position  (    pattern   : ARRAY OF CHAR;  (* search for this one *)
                         source    : ARRAY OF CHAR;  (* within this one     *)
                     VAR index     : CARDINAL;       (* 1st char of match   *)
                     VAR found     : BOOLEAN);


PROCEDURE Substring (    source    : ARRAY OF CHAR;  (* copy from this str  *)
                         index     : CARDINAL;       (* at this char postn  *)
                         len       : CARDINAL;       (* for this char count *)
                     VAR dest      : ARRAY OF CHAR;  (* into this string    *)
                     VAR success   : BOOLEAN);


PROCEDURE Concat    (    source1   : ARRAY OF CHAR;  (* this string with    *)
                         source2   : ARRAY OF CHAR;  (* this str on the end *)
                     VAR dest      : ARRAY OF CHAR;  (* assigned to this str*)
                     VAR success   : BOOLEAN);


PROCEDURE Compare   (    string1   : ARRAY OF CHAR;  (* compare this string *)
                         string2   : ARRAY OF CHAR)  (* to this string      *)
                                   : CompareResult;

END String.
```