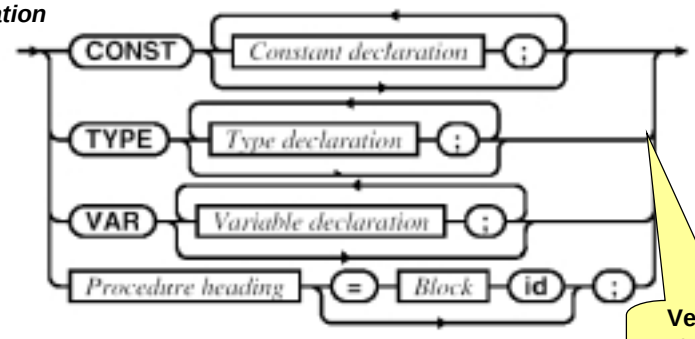
Datentypen I

- **Datentypen: Allgemeines**
- **Skalare benutzerdefinierte Datentypen**
  - Aufzählungstyp
  - Unterbereichstyp
- **Zusammengesetzte benutzerdefinierte Datentypen**
  - ARRAY-Type
  - RECORD-Type
  - SET-Type

Deklaration von Typen

- **Typdeklaration:**
  - Um das vorhandene ***Typkonzept*** erweitern zu können, sind in vielen imperativen Sprachen Typen selbst ***Programmobjekte***, die deklariert werden müssen.
  - Um Typen deklarieren zu können, benötigt man ***Datentypkonstruktoren***

Declaration



Verwendung von Datentyp-konstruktoren

Datentypen: Allgemeines

Einteilung von Typen - 1

- **In imperativen Programmiersprachen unterscheidet man einfache und zusammengesetzte Datentypen:**
  - ***Einfache Datentypen*** erlauben ***keinen*** Zugriff auf ihre innere Struktur. Ihre Werte können unmittelbar notiert werden. Die in einer Programmiersprache vorgegebenen einfachen Datentypen heißen elementar (skalar).
  - **Zusammengesetzte Datentypen** sind aus anderen Datentypen aufgebaut. Auf ihre einzelnen Elemente kann zugegriffen werden. Letztlich werden sie auf einfache Datentypen zurückgeführt.
- **Vorgegebene und benutzerdefinierte Datentypen:**
  - **Vordefinierter Datentypen** haben einen vordeklarierten Namen und können unmittelbar zur Deklaration von Variablen verwendet werden.
  - **Benutzerdefinierte** Datentypen haben einen selbst definierten Namen und müssen deklariert werden. Sie werden mit Hilfe bereits deklarierter vorgegebener oder benutzerdefinierter Datentypen gebildet.

Datentypen: Allgemeines

Einteilung von Typen - 2

- **Statische Datentypen**
  - Größe der Typobjekte ist von vornherein ***bekannt***
  - ***Statische*** Typkonstruktoren
    - ARRAY
    - RECORD
    - SET
- **Dynamische Datentypen**
  - Größe ist während der Laufzeit ***veränderbar***
  - Typkonstruktor für ***dynamische Datentypen***
    - Zeiger
    - Pointer

Skalare benutzerdefinierte Datentypen

Merkmale benutzerdefinierter Typen

- **Benutzerdefinierte Typen**
  - erlauben die Vergabe ***anwendungsbezogener*** Namen,
    - z.B. `Zeit` statt `REAL`,
  - sind ein wesentlicher ***Abstraktionsmechanismus***, da sie die programmiersprachliche Realisierung von Datenstrukturen ***verbergen*** können,
    - später werden wir das Konzept der Abstrakten Datentypen kennenlernen
  - sind "***Baumuster***" für die Erzeugung anwendungsbezogener Datenstrukturen,
    - z.B. Struktur

```
Zugverbindung =  Abfahrt: Zeit;
                 Ankunft: Zeit;
```

  - liefern "***Sprachelemente***" für die anwendungsbezogene Modellierung von Softwaresystemen.

Skalare benutzerdefinierte Datentypen

Benutzerdefinierte einfache Typen

- **Modula-3 erlaubt,**
  - benutzerdefinierte einfache Typen unter Verwendung einen vorgegebenen elementaren Typs zu deklarieren.
  -
```
TYPE     Zeit  = REAL;
         Alter = CARDINAL;
```

Basistyp muß ein Ordinaltyp sein

- **Unterbereichstyp (subrange type)**
  - benutzerdefinierte einfache Typen können als ***Einschränkung*** des Wertebereichs eines elementaren Typs deklariert werden
  -
```
TYPE     Index = [1..10];
         Alter = [1 .. 120];
```

- **Aufzählungstyp (emumeration type)**

Ordinaltyp

  - benutzerdefinierte einfache Typen können durch ***Aufzählung*** der zulässigen Werte deklariert werden
  -
```
TYPE Ampelfarbe   = {rot, gelb, gruen};
     Parteien     = {CDU, SPD, Gruene, FDP, PDS}
```

Skalare benutzerdefinierte Datentypen

Operationen auf Aufzählungstypen

- **Aufzählungstypen**
  - werden systemintern auf nichtnegative Zahlen abgebildet
  - (z.B.: rot ->1, blau -> 2, gruen -> 3 oder 0, 1, 2).
  - Dadurch sind die Werte von Aufzählungstypen dann ***vergleichbar*** – was aber selten Sinn macht (rot < gruen).
- **Bezeichner der Werte von Aufzählungstypen können bei *mehreren* Typen auftreten.**
  - `TYPE Ampelfarbe = {rot, gelb, gruen};`
  - `TYPE Parteifarbe = {rot, gelb, gruen, schwarz};`

```
VAR a : Ampelfarbe;
    p : Parteifarbe;
    a := Ampelfarbe.gruen;
    p := Parteifarbe.gruen
```

  - Gilt nicht für viele andere imperative Sprachen!

Skalare benutzerdefinierte Datentypen

Operationen auf Unterbereichstypen

- **Regel:**
  - Für einen Unterbreichstyp sind alle die Operationen definiert, die auch für seinen ***Basistyp*** definiert sind.
- **Es ist häufig unsinnig,**
  - ***arithmetische*** Operationen unmittelbar auf Unterbereichstypen anzuwenden, auch wenn dies die Sprache zuläßt (z.B. Addition zweier Jahreszahlen).
  -
```
TYPE AeraKohl = [1982 .. 1998];
VAR wahljahr1, wahljahr2, jahr : AeraKohl;

wahljahr1 := 1982; wahljahr2 := 1986;
jahr := wahljahr1 + wahljahr2;
```

    Laufzeitfehler

  - Vorsicht bei arithmetischen Operationen auf Unterbereichstypen, dies führt oft zu ***Laufzeitfehlern***.

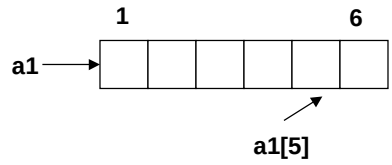
Zusammengesetzte benutzerdefinierte Datentypen

Feldtypen

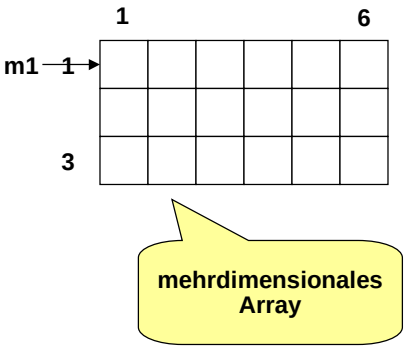
- **Definition:**
  - Nach Informatik-Duden: Feld (Reihung, engl. array): Aneinanderreihung von ***gleichartigen Elementen***, wobei auf die Komponenten mit Hilfe eines ***Indexausdrucks*** zugegriffen wird.
- **Eigenschaften von Arrays (Feldern) in Modula-3:**
  - Die Anzahl der Elemente ist ***fest*** und heißt ***Länge*** des Array.
  - Der ***Name*** einer Array-Variablen bezeichnet das gesamte Array.
  - Ein einzelnes Array-Element wird durch einen ***Index*** bzw. mehrere Indizes (im Fall mehrdimensionaler Arrays) identifiziert.
  - Zur Indizierung kann jeder ***Ordinaltyp*** verwendet werden.
    - INTEGER, CARDINAL, CHAR, BOOLEAN, Aufzählungs- und Unterbereichstypen

Zusammengesetzte benutzerdefinierte Datentypen

Beispiele: Array




```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;
VAR a1 : Vector
```




```
TYPE Spalte  = [1 .. 6];
     Zeile   = [1 .. 3];

TYPE
  Matrix = ARRAY Spalte, Zeile
           OF INTEGER;

VAR m1 : Matrix;
```

Zusammengesetzte benutzerdefinierte Datentypen

Felder und Zählschleifen

- **Beispiel:**
  - Initialisieren eines zweidimensionalen Arrays

```
TYPE Spalte  = [1 .. 6];
     Zeile   = [1 .. 3];

TYPE Matrix = ARRAY Spalte, Zeile OF INTEGER;

PROCEDURE Initialisieren (VAR m : Matrix) =
BEGIN
  FOR i := FIRST(Spalte) TO LAST(Spalte) DO
    FOR j := FIRST(Zeile) TO LAST(Zeile) DO
      m [i,j] := 0;
    END;
  END;
END Initialisieren;
```


Zusammengesetzte benutzerdefinierte Datentypen

Felder als zusammengesetzte Typen

- **Betrachten wir Arrays als zusammengesetzte Typen, dann stellen wir fest:**
  - Der ***Typkonstruktor***, der in Deklarationen benutzt wird, ist in Modula-3 (vereinfacht):

    ```
    <ArrayTyp> = ARRAY Index OF Komponenten;
    ```

  - Als ***Selektor*** für ein einzelnes Element wird die Indexangabe verwendet (indizierter Zugriff, Indizierung):

    ```
    val := a1[3] (* Wert des 3. Elements von a1 *)
    ```

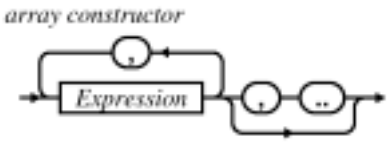
  - Üblicherweise wird die Indexangabe auch für die ***selektive Zuweisung*** (Feldkomponentenzuweisung) verwendet:

    ```
    a1[4]:= 42 (* 4. Elements von a1 wird 42 *)
    ```

Zusammengesetzte benutzerdefinierte Datentypen

Feldaggregate

- **Mithilfe eines sog. *Feldaggregats* können konstante ARRAY-Objekte erzeugt und Feldobjekte initialisiert werden:**
  - `VAR x := Array_Type { e1, .., en}`
    e1 bis en sind Ausdrücke; ihr Wert wird den Feldelementen initial zugewiesen



Feldvariableninitialisierung

```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;

VAR a1 := Vector {0, 0, 0, 0, 0, 0};
    a2 := Vector {0, ..}
```

```
TYPE Spalte = [1 .. 6];
     Reihe  = [1 .. 3];

TYPE Matrix = ARRAY Spalte , Reihe OF INTEGER;
VAR m1 := Matrix {ARRAY Reihe OF INTEGER {0, ..}, ..};
```


Zusammengesetzte benutzerdefinierte Datentypen

Operationen auf Feldobjekten

- **Zuweisung**
  - zwei ARRAY-Objekte sind ***zuweisungskompatibel***, wenn sie
    - den gleichen ***Komponententyp*** und die
    - gleiche ***Gestalt*** haben (gleiche Anzahl Elemente in jeder Dimension)
- **Vergleich**
  - zuweisungskompatible Arrays können auf ***Gleichheit*** und ***Ungleichheit*** geprüft werden.

```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;
CONST A = ARRAY [11 .. 16] OF INTEGER {1,2,3,4,5,6};
VAR v : Vector;
…
  v := A;

  IF v = A THEN
```

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Array - 1

```
MODULE StundenPlan EXPORTS Main;
IMPORT SIO;

TYPE
    Tage      = {Montag, Dienstag, Mittwoch, Donnerstag, Freitag};
    Stunden   = [7..20];
    Vormittag = [8..12];
    Fächer    = {Keine, Englisch, Software_1, Mathematik};
    Plan      = ARRAY Tage, Stunden OF Fächer;

CONST
   TagNamen  = ARRAY Tage OF TEXT {"Montag", "Dienstag","Mittwoch",
                                   "Donnerstag", "Freitag"};
   FachNamen = ARRAY Fächer OF TEXT {"Keine","Englisch","Software_1","Mathematik"};

VAR stundenPlan : Plan;          (*Speichert den Stundenplan*)
```

Typdeklarationen

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Array - 2

```
BEGIN
  FOR tag:= FIRST(Tage) TO LAST(Tage) DO
    FOR stunde:= FIRST(Stunden) TO LAST(Stunden) DO
      stundenPlan[tag, stunde]:= Fächer.Keine;    (*Initialisierung auf Keine*)
    END; (*FOR stunde*)
  END; (*FOR tag*)

  FOR stunde:= 8 TO 18 DO    (*Fast den ganzen Montag Englisch*)
    stundenPlan[Tage.Montag, stunde]:= Fächer.Englisch;
  END; (*FOR stunde*)

  FOR tag:= Tage.Dienstag TO Tage.Freitag DO
    stundenPlan[tag, 10]:= Fächer.Software_1;
  END; (*FOR tag*)

  stundenPlan[Tage.Dienstag, 8]:= Fächer.Mathematik;
  stundenPlan[Tage.Freitag, 9]:= Fächer.Mathematik;

  FOR tag:= FIRST(Tage) TO LAST(TAGE) DO
    SIO.PutText(TagNamen[tag]& "\ n");
    FOR stunde:= FIRST(Vormittag) TO LAST(Vormittag) DO
      SIO.PutInt(stunde);
      SIO.PutText(":" & FachNamen[stundenPlan[tag, stunde]]);
    END; (*FOR stunde*)
    SIO.NL();
  END; (*FOR tag*)
END StundenPlan.
```

Zusammengesetzte benutzerdefinierte Datentypen

Verbundtypen

Definitionen

- Nach Informatik-Duden:
  - Record *(Verbund, Struktur, Datensatz)*: ***Zusammenfassung*** von mehreren Datentypen zu einem Datentyp. Der neue Wertebereich ist das ***kartesische Produkt*** der Wertebereiche der einzelnen Datentypen, wobei die Anordnung keine Rolle spielt.
- Nach Sebesta:
  - A record is a ***possibly heterogeneous*** aggregation of data elements in which the individual elements are identified by **names**.
- Nach Ludewig:
  - Records (Verbunde) sind ***heterogene*** kartesische Produkte und dienen zur Darstellung ***inhomogener***, aber ***zusammengehöriger*** Informationen. Typische Beispiele sind
    - Personendaten (Name, Adresse, Jahrgang, Geschlecht)
    - Meßwerte (Zeit, Gerät, Wert)
    - Strings (tatsächliche Länge, Inhalt)

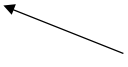
Zusammengesetzte benutzerdefinierte Datentypen

Verbunde in Modula-3

Record in Modula-3:

- Datentyp, der eine Sammlung von Elementen auch ***verschiedenen*** Typs (Elementtyp) repräsentiert.
- Der ***Name*** einer Record-Variablen bezeichnet den gesamten Record.
- Ein einzelnes Record-Element heißt auch ***Komponente*** (record field) und wird durch einen ***Namen*** (Selektorname, field identifier) bezeichnet.
- Von außen wird ein Feld über seinen Bezeichner mit der sog. ***Punktnotation*** (dot notation) angesprochen:
- `<RecordName>"."<FeldName>` z.B. `Person.Vorname`

Anrede	Vorname	Name	PersNr
Herr	Franz	Mustermann	4711
Dr.	Josef	Wanninger	4712
Frau	Susanne	Mitternacht	4713



ein Record

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: RECORD - 1

```
MODULE RecordDemo EXPORTS Main;

TYPE PersNr = [4700 .. 9999];
TYPE Anrede = {Frau, Herr, Dr};
TYPE Name   = TEXT;

TYPE Person = RECORD
                anrede   : Anrede;
                vorname  : Name;
                nachname : Name;
                persnr   : PersNr;
              END;

VAR person1 : Person;
BEGIN
  person1.anrede   := Anrede.Dr ;
  person1.vorname  := "Josef";
  person1.nachname := "Wanninger";
  person1.persnr   := 4712;
END RecordDemo.
```

Typdefinition

Typdeklaration

Faßt unterschiedliche Typen zu einer gemeinsamen Struktur zusammen.

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: RECORD - 2

```
TYPE PersNr = [4700 .. 9999];
TYPE Anrede = {Frau, Herr, Dr};

TYPE Name   = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : Pers;
              END;
VAR person1 : Person;

person1.anrede        := Anrede.Dr ;
person1.name.vorname  := "Josef";
person1.name.nachname := "Wanninger";
person1.persnr        := 4712;
```

- **Bemerkung:**
  - Ziel ist, Typen so zu konstruieren, daß sie möglichst sinnvoll ***aufeinander*** aufbauen.
  - Vorteile:
    - erleichterte Modifikation
    - Wiederverwendbarkeit
    - bessere Lesbarkeit
    - Begriffe der Anwendung können verwendet werden

Zusammengesetzte benutzerdefinierte Datentypen

Records als zusammengesetzte Typen

- **Betrachten wir Records als zusammengesetzte Typen, dann stellen wir fest:**
  - Der ***Typkonstruktor***, der in Deklarationen benutzt wird, ist in Modula-3 (vereinfacht):

    RECORD <Feldname> : <Basistyp> {;<Feldname> : <Basistyp>} END

  - Der ***Selektor*** für ein Feld eines Records, der bei der Verwendung benutzt wird, ist in Modula-3:

    <RecordBezeichner> . <FeldName>

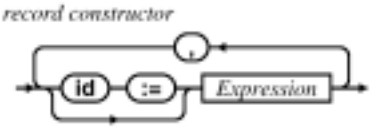
  - Eine rekursive Typdeklaration eines Records ist ***nicht möglich***:

```
Liste = RECORD Listenkopf : CHAR;
               Listenrest : Liste;
         END (* geht nicht *)
```


Zusammengesetzte benutzerdefinierte Datentypen

Verbundaggregate

- **Mit dem *Verbundaggregat* werden initialisierte RECORD-Objekte erzeugt:**
  - `VAR x := Record_Type { Bindings }`
    Bindings: analog zur Bindung der aktuellen Parameter an formale Parameter beim Prozeduraufruf




```
TYPE Name   = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
VAR n1 := Name { vorname := "Josef",  nachname := "Maier"};

TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : PersNr;
              END;

VAR p1 := Person {anrede := Anrede.Herr,
                  name   := Name { vorname := "Kai",  nachname := "Blau"},
                  persnr := 4700 };
```

Angabe der Werte per Name

Zusammengesetzte benutzerdefinierte Datentypen

Operationen auf RECORDs - 1

- **Zuweisung**
  - zwei RECORD-Objekte sind ***zuweisungskompatibel***, wenn
    - alle Komponenten den gleichen ***Namen*** und den gleichen Typ haben
    - alle Komponenten in der gleichen ***Reihenfolge*** deklariert sind
- **Vergleich**
  - zuweisungskompatible Records können auf ***Gleichheit*** und ***Ungleichheit*** geprüft werden.

```
TYPE Name1  = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
TYPE Name2  = RECORD
                nachname : TEXT;
                vorname  : TEXT;
              END;
VAR n1 : Name1; n2 : Name2;
n1 := n2  nicht zuweisungskompatibel
```


Zusammengesetzte benutzerdefinierte Datentypen

Die WITH-Anweisung

- **WITH-Anweisung**
  - dient dazu, komplexe Selektoren, die mehrmals verwendet werden müssen, mit einem ***ALIAS-Namen*** zu versehen.
  - Code wird ***kompakter***, lesbarer
- **Syntax:**

- **Semantik:**
  - der im Binding eingeführte Bezeichner ist im Block bis zum END gültig
  - der eingeführte Bezeichner wird als "***Abkürzung***" verwendet

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel WITH-Anweisung

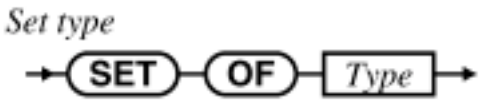
```
TYPE Name   = RECORD                    TYPE Person = RECORD
                vorname  : TEXT;                anrede : Anrede;
                nachname : TEXT;                name   : Name;
              END;                              persnr : PersNr;
                                              END;

  VAR bundeskanzler : Person;

  bundeskanzler.anrede        := Anrede.Herr;
  bundeskanzler.name.vorname  := "Gerhard";
  bundeskanzler.name.nachname := "Schroeder";
  bundeskanzler.persnr        := 4711;
```

```
WITH bk = bundeskanzler DO
  bk.anrede := Anrede.Herr;
  WITH bkn = bk.name DO
    bkn.vorname := "Gerhard";
    bkn.nachname := "Schroeder";
  END;
  bk.persnr := 4700;
END
```


Zusammengesetzte benutzerdefinierte Datentypen

Mengen

- **Modula-3 bietet einen eigenen vordefinierten Mengentyp**
- **Syntax: (Typkonstruktor)**



- **Bemerkungen:**
  - Mengen sind ***ungeordnete*** Sammlungen von Elementen
  - der Elementtyp (Universum) muß ein ***Ordinaltyp*** sein!
  - Elemente einer Menge können **nicht** indiziert werden
  - Wertebereich eines Mengentyps ist die ***Potenzmenge***
    - Menge aller Teilmengen über dem Elementtyp
    - Bsp.: `ET = {rot, gruen}`
      Werte **`SET OF`** `ET: {} {rot} {gruen} {rot, gruen}`
  - Aus Effizienzgründen sollen Mengen nur über Elementmengen mit ***kleiner Kardinalität*** gebildet werden.

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel : Mengendeklaration

```
TYPE Lottozahl = [1 .. 49];

TYPE Ziehung = SET OF Lottozahl;

CONST Leer := Ziehung {};

VAR   z1   := Ziehung {1 .. 7};
      z2   := Ziehung {4,7,34,20,44,23};
```

Mengenaggregat

- **Operationen:**
  - Zuweisung
    - zuweisungskompatibel: ***Elementtypen*** sind gleich
  - Vereinigung, Differenz, Durchschnitt, symmetrische Differenz
  - Gleichheit, Ungleichheit, Teilmenge, … , Enthalten

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Buchstabenzählen - 1

```
MODULE BuchstabenOrg EXPORTS Main;
IMPORT SIO, Text;

TYPE Buchstabe = ['A' .. 'Z'];
     BuchstabenMenge = SET OF Buchstabe;

CONST Alle = BuchstabenMenge {'A' .. 'Z'};
VAR einmal, mehrmals, nie := BuchstabenMenge {};
    eingabe : TEXT;
    z : CHAR;
BEGIN
  eingabe := SIO.GetLine();

  FOR i := 0 TO Text.Length(eingabe) - 1 DO
    z := Text.GetChar(eingabe, i);
    IF z IN Alle THEN
      IF z IN einmal THEN
        mehrmals := mehrmals + BuchstabenMenge{z};
      ELSE
        einmal := einmal + BuchstabenMenge{z};
      END;
    END;
  END;
  nie := Alle - einmal;
  einmal := einmal - mehrmals;
```

Set-Aggregat

Mengen-vereinigung

Mengen-differenz

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Buchstabenzählen - 2

```
  SIO.PutLine("NIE:");
  FOR z := 'A' TO 'Z' DO
    IF z IN nie THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

  SIO.PutLine("EINMAL:");
  FOR z := 'A' TO 'Z' DO
    IF z IN einmal THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

  SIO.PutLine("MEHRMALS:");
  FOR z := 'A' TO 'Z' DO
    IF z IN mehrmals THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

END BuchstabenOrg.
```


Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 1 des Beispiels

```
PROCEDURE GebeMengeAus (m : BuchstabenMenge) =
BEGIN
  FOR z := FIRST(Buchstabe) TO LAST(Buchstabe) DO
    IF z IN m THEN
      SIO.PutChar(z)
    END;
  END;
END Ausgabe;

BEGIN (* Buchstaben1 *)
  eingabe := SIO.GetLine();
  FOR i := 0 TO Text.Length(eingabe) - 1 DO
    z := Text.GetChar(eingabe, i);
    IF z IN Alle THEN
      IF z IN einmal THEN
        mehrmals := mehrmals + BuchstabenMenge{z};
      ELSE
        einmal := einmal + BuchstabenMenge{z};
      END;
    END;
  END;
  nie := Alle - einmal;
  einmal := einmal - mehrmals;
  SIO.PutLine("NIE:");      GebeMengeAus(nie); SIO.Nl();
  SIO.PutLine("EINMAL:");   GebeMengeAus(einmal); SIO.Nl();
  SIO.PutLine("MEHRMALS:"); GebeMengeAus(mehrmals); SIO.Nl();
END Buchstaben1.
```

Verwenden einer Prozedur für die Ausgabe von Mengen

Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 2 des Beispiels - a

```
TYPE Vorkommen = RECORD
                   einmal, mehrmals, nie := BuchstabenMenge {};
                 END;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
  CONST Alle     := BuchstabenMenge {'A' .. 'Z'};
  VAR   resultat : Vorkommen; z : CHAR; vorgekommen : BuchstabenMenge;
BEGIN
  WITH r = resultat DO
    FOR i := 0 TO Text.Length(t) - 1 DO
      z := Text.GetChar (t, i);
      IF z IN Alle THEN
        IF z IN vorgekommen THEN
          r.mehrmals := r.mehrmals + BuchstabenMenge{z};
        ELSE
          vorgekommen := vorgekommen + BuchstabenMenge{z};
        END;
      END;
    END;
    r.nie := Alle - vorgekommen ;
    r.einmal := vorgekommen - r.mehrmals;
  END;
  RETURN resultat;
END Vorkommenzaehlen;
```

Wäre ein Array eine Alternative?

Bezeichner werden nur für einen Zweck eingesetzt!

Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 2 des Beispiels - b

```
MODULE Buchstaben1 EXPORTS Main;
…
VAR vork : Vorkommen;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
...


BEGIN

  vork := Vorkommenzaehlen(SIO.GetLine());

  SIO.PutLine("NIE:");       GebeMengeAus(vork.nie); SIO.Nl();
  SIO.PutLine("EINMAL:");    GebeMengeAus(vork.einmal); SIO.Nl();
  SIO.PutLine("MEHRMALS:"); GebeMengeAus(vork.mehrmals); SIO.Nl();

END Buchstaben1.
```

Verwenden die Ausgabeoperation für Mengen

Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 2 des Beispiels - c

```
PROCEDURE Ausgabe (v: Vorkommen) =
CONST NIE      = "     NIE : ";
      EINMAL   = "  EINMAL : ";
      MEHRMALS = "MEHRMALS : ";
BEGIN
  SIO.PutText(NIE);      GebeMengeAus(v.nie); SIO.Nl();
  SIO.PutText(EINMAL);   GebeMengeAus(v.einmal); SIO.Nl();
  SIO.PutText(MEHRMALS); GebeMengeAus(v.mehrmals); SIO.Nl();
END Ausgabe;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
...

BEGIN (*Buchstaben2 *)

  Ausgabe(Vorkommenzaehlen(SIO.GetLine()));

END Buchstaben2.
```


Zusammengesetzte benutzerdefinierte Datentypen

Arrays, Records, Mengen

- **Vergleich der Typkonstruktoren für**
  - statische, zusammengesetzte Typen

Aspekt	Array	Record	Menge
Größe	fest (!)	fest	fest
Element-typen	homogen	heterogen	homogen, Ordinaltyp
Zugriff	dynamisch indiziert	statisch	kein direkter Zugriff
Ordnung	statisch festgelegt Index ist geordnet	statisch festgelegt	ungeordnet

Was haben wir gelernt!

- Datentypen: Zweck, Typisierung, Deklaration, Einteilung
- benutzerdefinierte Datentypen mittels Datentypkonstruktoren
- Aufzählungs- und Unterbereichstypen als benutzerdefinierte skalare Typen, Aufzählungsliterale
- Feldtypen: Indextyp, Elementtyp, eindimensionale, mehrdimensionale Feldverarbeitung: mittels Zählschleifen, Feldattributen, Feldindizierung, Feldaggregate, Feldinitialisierung, Feldzuweisung und -vergleich
- Verbundtypen: Komponententypen, Komponentennamen, Selektor(pfad)
- Verbundverarbeitung: Komponentenzugriff, Verbundaggregate, Verbundzuweisung und -vergleich, with-Anweisung
- Mengentypen: Elementtyp (Trägermenge), Teilmengenbildung, charakteristische Speicherung
- Mengenverarbeitung: Mengenaggregate, Vereinigung, Durchschnitt, Teilmenge, Enthalten, etc.
- Vergleich Datentypkonstruktoren für zusammengesetzte Datentypen

Glossar

- Datentypenklassifikation: vor- oder selbstdefiniert, skalar oder zusammengesetzt, statisch oder dynamisch
- Typ: Charakterisierung
- Aufzählungstypen, Unterbereichstypen
- Feldtypen, Verbundtypen, Mengentypen
- Typdefinitionen, Typdeklarationen, Objektdeklarationen mittels Typ, ggfs. mit Initialisierung, bequem durch Aggregate
- Datentypkonstruktoren für Felder, Verbunde, Mengen
- Aggregate für Felder, Verbunde, Mengen