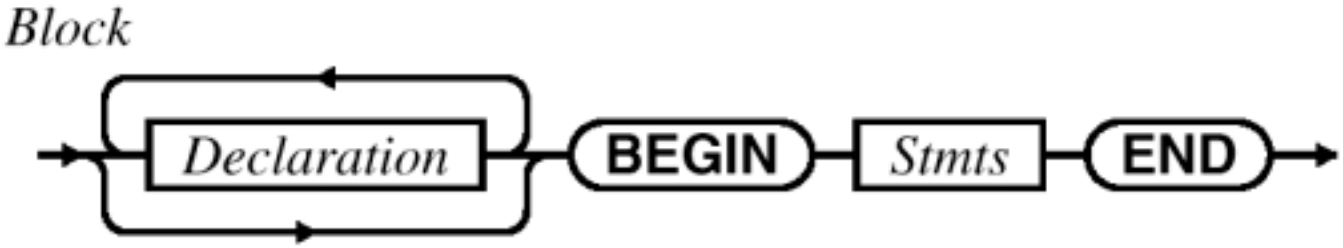
Erste Programmbeispiele

- **Vorschau: Deklarationen, Anweisungen, Ausdrücke, Datentypen**
- **Vordefinierte Datentypen**
- **Beispielprogramm aus dem Vorkurs**

Deklarationen (vorläufig)

- *Blöcke* **können** *Deklarationen* **enthalten**
- **und bestehen aus *Anweisungen* (Statements)**

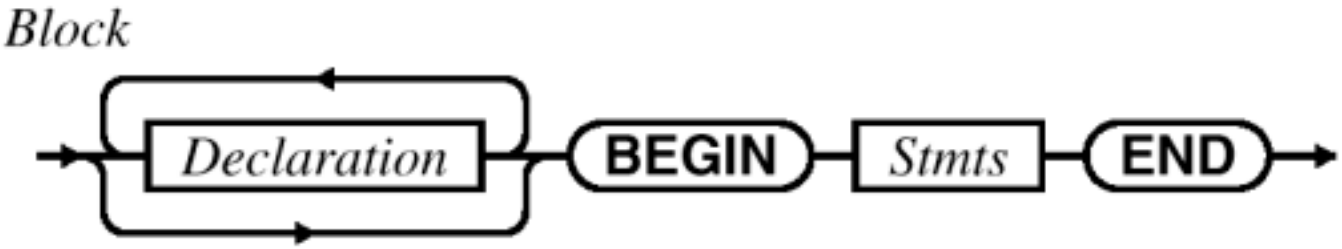


- **Deklarationen werden abgearbeitet**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***

Anweisungen (vorläufig)

Anweisungen werden ausgeführt

- sind in Blöcken enthalten, durch ";" getrennt
- atomare Bausteine für Modula-3 Programme



- Die Folge der Anweisungen eines Blocks werden ausgeführt. Unterscheidung statische Reihenfolge, Ausführungsreihenfolge (Programmpfad)

Unterscheidung

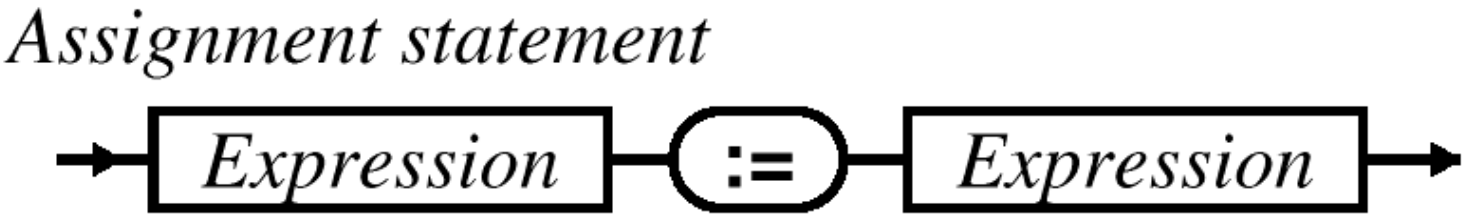
- einfache Anweisungen

zusammengesetzte Anweisungen

Konstruktionshilfsmittel Kontrollstrukturen

Ausdrücke

- **Ausdrücke werden ausgewertet, liefern einen Wert (Ergebnis)**
- **Beispiele**
  - arithmetische Ausdrücke
    - x * x
  - logische Ausdrücke, insb. relationale Ausdrücke
    - B_1 AND B_2 A <= B
- **Viele Anweisungen können Ausdrücke enthalten**



Datentypen (vorläufig)

Typbegriff

- im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung

Definition

- Unter einem ***Datentyp*** versteht man die ***Zusammenfassung*** von ***Wertebereich*** und ***Operationen*** zu einer ***Einheit.*** Ein Typ hat eine Struktur und Literale (Aggregate).

Man unterscheidet:

- ***einfache*** (skalare) Datentypen
- ***zusammengesetzte*** Datentypen
  Konstruktionsmittel: Datentypkonstruktoren
- ***vordefinierte*** Datentypen
- ***selbstdefinierte*** Datentypen

Vordefinierte Datentypen in Modula-3

Gruppierung

einfache, skalare:

- **Ganze Zahlen (diskret)**
  - darunter fallen die Typen **`INTEGER`** und **`CARDINAL`**
- **Zeichen (diskret)**
  - Werte eines bestimmten Zeichenvorrates; z.B. definiert der ASCII-Zeichenvorrat 128 Zeichen.

zusammengesetzte:

- **Texte**
  - sind eine Folge von Zeichen
- **Wahrheitswerte (diskret)**
  - Werte sind {wahr, falsch}, Literale **`TRUE`**, **`FALSE`**
- **Gleitkommazahlen**
  - reelle Zahlen, **`REAL`**, **`LONGREAL`**, **`EXTENDED`**

Ganzahlige Datentypen

- **Typen: `INTEGER` und `CARDINAL`**
  - **Integer-Zahlen** sind ***ganzzahlige*** Werte innerhalb der Unter- und Obergrenze des jeweiligen Rechners
  - **Cardinal-Zahlen** sind sind ***nicht-negative*** ganzzahlige Werte, d.h. zwischen 0 und der Obergrenze des jeweiligen Rechners
- **Wertebereich**
  - auf einen 32-Bit-Rechner:
    - **`INTEGER`** [-2147483648 .. 2147483647] oder [-2^{31} .. $2^{31}-1$]
    - 1 Bit für das Vorzeichen, 31Bit für die Zahlendarstellung
  - Zahlen sind geordnet (***Ordinaltyp***)
- **Operationen**
  - arithmetische Operationen (**`+`**, **`-`**, **`*`**, **`DIV`**, **`MOD`**)
  - Vergleichsoperationen (**`=`**, **`#`**, **`<`**, **`>`**, **`<=`**, **`>=`**)
  - vordefinierte Funktionen (**`FIRST(type)`**, **`LAST(type)`**, **`INC(z)`**, **`DEC(z)`**, **`ABS(z)`**)

Beispiel

```
MODULE Zahlen EXPORTS Main;
(* Dieses Programm zeigt dem Umgang mit ganzen Zahlen *)

IMPORT SIO;

PROCEDURE Modulo (x,y: INTEGER): INTEGER =
(* MOD ist def. : x MOD y = x - y*(x DIV y) *)
BEGIN
  RETURN ( x - y*(x DIV y) );
END Modulo;

BEGIN
  (* Ausgabe der Unter- und Obergrenze von INTEGER *)
  SIO.PutInt (FIRST(INTEGER)); SIO.Nl();
  SIO.PutInt (LAST(INTEGER));  SIO.Nl();

  SIO.PutInt (20 DIV 6); SIO.Nl();      (* = 3 *)
  SIO.PutInt (20 MOD 6); SIO.Nl();      (* = 2 *)
  SIO.PutInt (Modulo(20,6));            (* = 2 *)
END Zahlen.
```

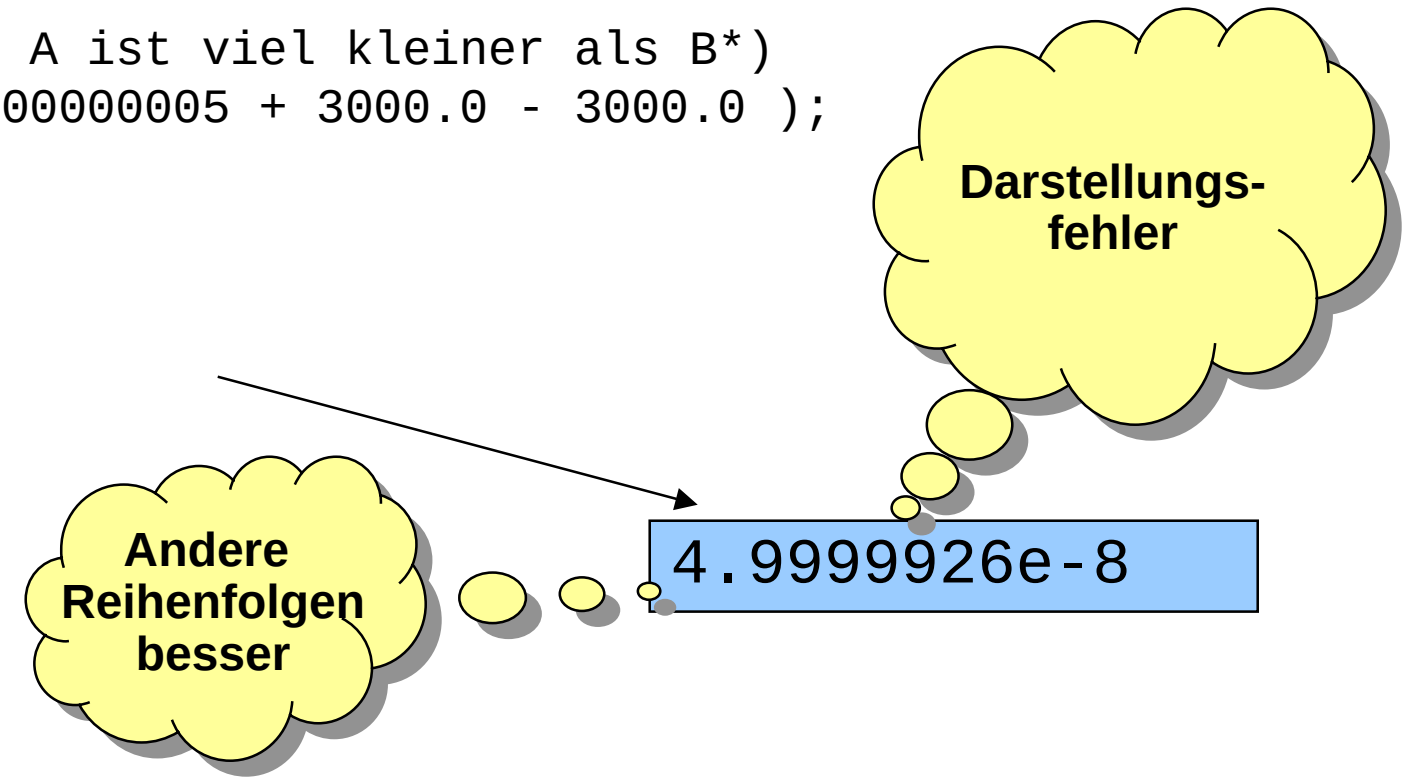
Gleitpunktdatentypen

- **Typen: `REAL`, `LONGREAL`, `EXTENDED`**
  - repräsentieren die darstellbaren reellen Zahlen

- **Darstellung**
  - Wertebereich ist ***beschränkt*** (im Unterschied zur Mathematik)
  - Genauigkeit der Darstellung ist ***beschränkt***
  - Rechnen mit Gleitkommazahlen ist immer **fehlerbehaftet**!
    - "Numerik" liefert Techniken und Algorithmen, um Fehler zu beherrschen

Beispiel für Gleitkommazahlen

```
MODULE Gleitkomma EXPORTS Main;
(* Dieses Programm zeigt Rundungsprobleme bei Gleitkommazahlen *)

IMPORT SIO;


BEGIN
  (* A + B - B mit A ist viel kleiner als B*)
  SIO.PutReal ( 0.00000005 + 3000.0 - 3000.0 );

END Gleitkomma.
```



Zeichentyp CHAR

Typ CHAR

- **CHAR** (character) bezeichnet eine ***endliche***, ***geordnete Menge*** von Zeichen
- **CHAR** ist ein ***Ordinaltyp***

Wertebereich

- Latin-1, viele Rechner benutzen den ASCII-Zeichensatz
- Zeichenliterale: `'A'` `'z'` `'1'`
- Spezialzeichen:

`\n`	Zeilenvorschub	`\t` Tabulator	`\\` Backslash
`\'`	Apostroph	`\f` Seitenumbruch	
`\r`	Wagenrücklauf	`\"` Anführungszeichen	

Operationen

- Vergleichsoperationen (**=**, #, **<**, **>**, **<=**, **>=**)
- Vordefinierte Funktionen **FIRST(CHAR), LAST(CHAR), INC(z), DEC(z), ORD(z), VAL(i)**

Beispiel - Zeichen

```
MODULE KleinGross EXPORTS Main;
(* Dieses Programm wandelt einen Klein- in einen Grossbuchstaben um *)

IMPORT SIO;

PROCEDURE Offset (): INTEGER =
BEGIN
  RETURN (ORD('A') - ORD('a'));
END Offset;


BEGIN
  (* Kleinbuchstaben einlesen und umwandeln *)
  SIO.PutChar ( VAL(ORD(SIO.GetChar()) + Offset(), CHAR) );

END KleinGross.
```

Texte (Zeichenketten)

- **Typ TEXT**
  - repräsentiert eine ***beliebig lange Folge*** von Zeichen (kann auch leer sein)
  - in vielen Sprachen nicht explizit vorhanden
- **Wertebereich**
  - Textliterale werden in " " notiert
  - z.B. "Das ist ein Text mit Zeilenvorschub\n"
"\"Dieser Text beginnt und endet mit einem Hochkomma\""
- **Operationen**
  - Konkatenation : &
    - Bspl: "Heute " & "ist " & "Freitag." ⇨ "Heute ist Freitag."
  - Schnittstelle des Moduls "Text"
    - Equal, Length, Empty, FindChar ….

Wahrheitswerte

- **Typ `BOOLEAN`**
  - repräsentiert die beiden vordefinierten Wahrheitswerte
- **Wertebereich**
  - wahr, Literal **`TRUE`**
  - falsch, Literal **`FALSE`**
- **Operationen**
  - Komplement (**`NOT`**), Oder (**`OR`**), Und (**`AND`**)

p	q	NOT q	p OR q	p AND q
1	1	0	1	1
1	0	1	1	0
0	1	0	1	0
0	0	1	0	0

1 wahr
0 falsch

Beispiel für Wahrheitswerte

```
MODULE BooleanM EXPORTS Main;
(* Dieses Programm berechnet die Wahrheitstabelle NOT, OR, AND *)
IMPORT SIO, Fmt;

PROCEDURE NotOrAnd ( a, b : BOOLEAN) : TEXT =
BEGIN
  RETURN (Fmt.Bool(a) & " " & Fmt.Bool(b) & " : " &
          Fmt.Bool( NOT(b))       & " " &
          Fmt.Bool( a OR b)       & " " &
          Fmt.Bool(a AND b) );
END NotOrAnd;

BEGIN
  (* Ausgabe der Wertetabelle *)
  SIO.PutLine(NotOrAnd(FALSE, FALSE));
  SIO.PutText(NotOrAnd(TRUE, FALSE));
  SIO.PutLine(NotOrAnd(FALSE, TRUE));
  SIO.PutLine(NotOrAnd(TRUE, TRUE));

END BooleanM.
```

produzierte Ausgabe

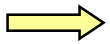
```
FALSE FALSE : TRUE  FALSE FALSE
TRUE  FALSE : TRUE  TRUE  FALSE
FALSE TRUE  : FALSE TRUE  FALSE
TRUE  TRUE  : FALSE TRUE  TRUE
```

Schrittweise Verfeinerung
Kommentierung, aussagekräftige Bezeichner

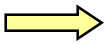
Eingabe

Berechnung

Ausgabe



```
(* Eingabe *)
   Überschrift
   Bestellung eingeben
   Bestätigung der Bestellung
...
```



```
(* Eingabe *)
  (* Überschrift *)
   PutText("----------"); Nl();
   PutText(“Kaffeebestellung"); Nl();
   PutText("----------"); Nl();
  (* Bestellung eingeben *)
   PutText("Wieviel Kaffee wollen Sie kaufen? "); Nl();
   AnzahlKaffee := GetInt();
  (* Bestätigung der Bestellung *)
   PutText("Sie haben bestellt:"); Nl();
   PutInt (AnzahlKaffee);
   PutText(" Säcke Kaffee zum Preis von je ");
   PutInt (KostenKaffee); PutText(" DM"); Nl();
   ...
```

Eingabe:

Übersicht
Was wird erwartet?
Eingabe war falsch?
Was war falsch?
Bestätigung der Eingabe

Ausgabe:

Übersicht
Ergebnis der Berechnung

Variable, Konstante

```
VAR
   AnzahlKaffee : CARDINAL;
   SummeKaffee  : CARDINAL;
   GesamtSumme  : REAL;
```

Typisierter Behälter für Werte
mit Speicherplatz abhängig vom Typ,
hier arithmetische Operationen
Literale 10, 100, etc.

```
CONST
   KostenKaffee      = 200;
   MWST : CARDINAL   = 16;
   MWSTAnteil        = FLOAT(MWST)/100.0;
```

Hier muß keine Typangabe stehen,
inferiert aus Anfangswert,
Literal, Compilezeitausdruck

Einfache Berechnung mit Kontrollstrukturen: Fallunterscheidung

```
(* Bedingte Anweisung *)
IF (AnzahlKaffee >= 0) AND (AnzahlKaffee < 5)THEN
     KaffeeRabatt:= 0;
ELSEIF (AnzahlKaffee >= 5) AND (AnzahlKaffee < 10) THEN
     KaffeeRabatt:= 5;
ELSEIF (AnzahlKaffee >= 10) AND (AnzahlKaffee < 50) THEN
     KaffeeRabatt:= 10;
ELSEIF (AnzahlKaffee >= 50) AND (AnzahlKaffee < 100) THEN
     KaffeeRabatt:= 20;
ELSE (* AnzahlKaffee >= 100 *)
     KaffeeRabatt:= 30;
END;
...
```

sequentielle Ausführung

Alternativen disjunkt

```
CASE AnzahlKaffee OF
   | 0..4 =>        KaffeeRabatt := 0;
   | 5..9 =>        KaffeeRabatt := 5;
   | 10..49 =>      KaffeeRabatt := 10;
   | 50..99 =>      KaffeeRabatt := 20
ELSE (* größer 100 *)
    KaffeeRabatt := 30;
END;
```

Einfache Berechnung mit Kontrollstrukturen: Schleifen

```
FOR i:= 1 TO 4 DO

    PutText("Bestellung der Sorte ");
    PutInt(i);
    Nl();
    PutText("Wieviele Säcke Kaffee dieser Sorte wollen Sie kaufen? ");
    AnzahlKaffeeProSorte[i] := GetInt();
```

FOR-Schleife
Anzahl der Schleifendurchläufe bekannt

keine Terminationsprobleme

WHILE-Schleife
Anzahl der Schleifendurchläufe nicht bekannt

Terminationsüberlegung

```
BEGIN
WHILE Taste # 'n' DO
        Programmtext
  PutText("Noch eine Bestellung aufgeben ? (j/n)  ");
  Taste := GetChar();
END; (* WHILE *)
```

Was haben wir gelernt

- Datentypen: einfache, zusammengesetzte, vordefinierte, selbstdefinierte
- einfache (skalare) vordefinierte Datentypen: `INTEGER`, `CARDINAL`, `REAL`, `LONGREAL`, `CHAR`, `BOOLEAN`
- zusammengesetzter vordefinierter Datentyp `TEXT`
- Blöcke: Deklarationen und Anweisungen
  Deklarationen abgearbeitet, Anweisungen ausgeführt, Ausdrücke ausgewertet
- zusammengesetzte Anweisungen mit Kontrollstrukturen (später)
- zusammengesetzte Typen mit Datentypkonstruktoren (später)
- Programmiertechnik: Bezeichnerwahl, Kommentierung, Bedienerschnittstellengestaltung
- Eingaben: Übersicht, was wird erwartet, Meldung: falsch, richtig, was war falsch, Bestätigung der Eingabe
- Ausgabe Übersicht, übersichtliche Darstellung der Ergebnisse
- I/O: Korrektheit, Layout, Bequemlichkeit (Masken, Menüs später)
  unterschiedliche Kenntnis (Novize, Experte später)
- Variable, Konstante
- Fallunterscheidungen: bedingte Anweisungen, Auswahlanweisungen
- Schleifen: Zählschleifen, bedingte Schleifen

Glossar

- Datentyp
- Block
- einfache/zusammengesetzte Anweisung
- einfacher/zusammengesetzter Datentyp
- Kontrollstrukturen
- Datentypkonstruktoren
- Fallunterscheidungen IF, CASE
- Schleifenformen FOR, WHILE; Schleifenkopf, Schleifenrumpf

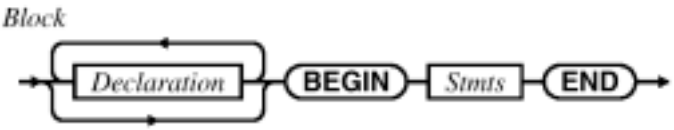
Erste Programmbeispiele

- **Vorschau: Deklarationen, Anweisungen, Ausdrücke, Datentypen**
- **Vordefinierte Datentypen**
- **Beispielprogramm aus dem Vorkurs**

Deklarationen (vorläufig)

Vorschau: ...

- *Blöcke* **können** *Deklarationen* **enthalten**
- **und bestehen aus *Anweisungen* (Statements)**

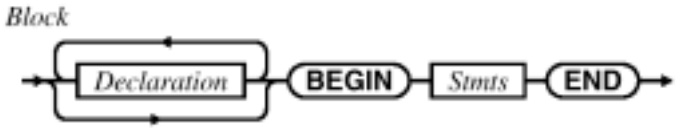


- **Deklarationen werden abgearbeitet**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***

Vorschau: ...

Anweisungen (vorläufig)

- **Anweisungen werden ausgeführt**
  - sind in Blöcken enthalten, durch ";" getrennt
  - atomare Bausteine für Modula-3 Programme



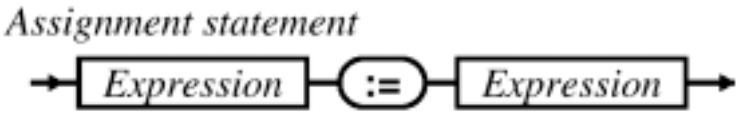
  - Die Folge der Anweisungen eines Blocks werden ausgeführt. Unterscheidung statische Reihenfolge, Ausführungsreihenfolge (Programmpfad)

- **Unterscheidung**
  - einfache Anweisungen
  - zusammengesetzte Anweisungen
  - Konstruktionshilfsmittel Kontrollstrukturen

Vorschau: ...

Ausdrücke

- **Ausdrücke werden ausgewertet, liefern einen Wert (Ergebnis)**
- **Beispiele**
  - arithmetische Ausdrücke
    - x * x
  - logische Ausdrücke, insb. relationale Ausdrücke
    - B_1 AND B_2 A <= B
- **Viele Anweisungen können Ausdrücke enthalten**



Vorschau: ...

Datentypen (vorläufig)

- **Typbegriff**
  - im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung
- **Definition**
  - Unter einem ***Datentyp*** versteht man die ***Zusammenfassung*** von ***Wertebereich*** und ***Operationen*** zu einer ***Einheit.*** Ein Typ hat eine Struktur und Literale (Aggregate).
- **Man unterscheidet:**
  - ***einfache*** (skalare) Datentypen
  - ***zusammengesetzte*** Datentypen
    Konstruktionsmittel: Datentypkonstruktoren
  - ***vordefinierte*** Datentypen
  - ***selbstdefinierte*** Datentypen

Vordefinierte Datentypen

Vordefinierte Datentypen in Modula-3

- **Gruppierung**

  einfache, skalare:
  - **Ganze Zahlen (diskret)**
    - darunter fallen die Typen **`INTEGER`** und **`CARDINAL`**
  - **Zeichen (diskret)**
    - Werte eines bestimmten Zeichenvorrates; z.B. definiert der ASCII-Zeichenvorrat 128 Zeichen.

  zusammengesetzte:
  - **Texte**
    - sind eine Folge von Zeichen
  - **Wahrheitswerte (diskret)**
    - Werte sind {wahr, falsch}, Literale **`TRUE`**, **`FALSE`**
  - **Gleitkommazahlen**
    - reelle Zahlen, **`REAL`**, **`LONGREAL`**, **`EXTENDED`**

Ganzahlige Datentypen

- **Typen: `INTEGER` und `CARDINAL`**
  - **Integer-Zahlen** sind ***ganzzahlige*** Werte innerhalb der Unter- und Obergrenze des jeweiligen Rechners
  - **Cardinal-Zahlen** sind sind ***nicht-negative*** ganzzahlige Werte, d.h. zwischen 0 und der Obergrenze des jeweiligen Rechners
- **Wertebereich**
  - auf einen 32-Bit-Rechner:
    - **`INTEGER`** [-2147483648 .. 2147483647] oder [-2^{31} .. $2^{31}-1$]
    - 1 Bit für das Vorzeichen, 31Bit für die Zahlendarstellung
  - Zahlen sind geordnet (***Ordinaltyp***)
- **Operationen**
  - arithmetische Operationen (**`+`**, **`-`**, **`*`**, **`DIV`**, **`MOD`**)
  - Vergleichsoperationen (**`=`**, #, **`<`**, **`>`**, **`<=`**, **`>=`**)
  - vordefinierte Funktionen (**`FIRST(type)`**, **`LAST(type)`**, **`INC(z)`**, **`DEC(z)`**, **`ABS(z)`**)

Beispiel

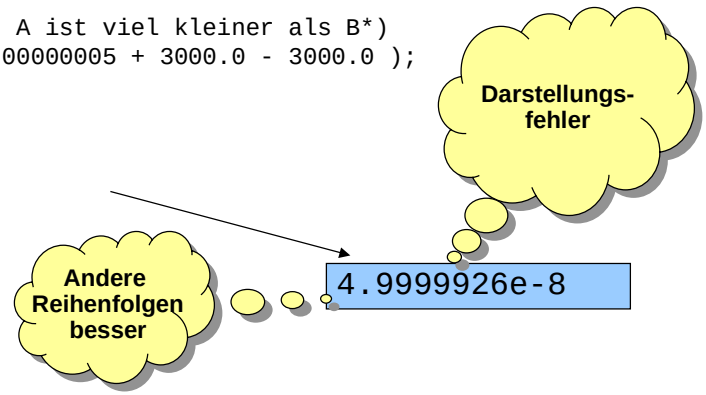
```
MODULE Zahlen EXPORTS Main;
(* Dieses Programm zeigt dem Umgang mit ganzen Zahlen *)

IMPORT SIO;

PROCEDURE Modulo (x,y: INTEGER): INTEGER =
(* MOD ist def. : x MOD y = x - y*(x DIV y) *)
BEGIN
  RETURN ( x - y*(x DIV y) );
END Modulo;

BEGIN
  (* Ausgabe der Unter- und Obergrenze von INTEGER *)
  SIO.PutInt (FIRST(INTEGER)); SIO.Nl();
  SIO.PutInt (LAST(INTEGER));  SIO.Nl();

  SIO.PutInt (20 DIV 6); SIO.Nl();     (* = 3 *)
  SIO.PutInt (20 MOD 6); SIO.Nl();     (* = 2 *)
  SIO.PutInt (Modulo(20,6));           (* = 2 *)
END Zahlen.
```

Gleitpunktdatentypen

- **Typen: REAL, LONGREAL, EXTENDED**
  - repräsentieren die darstellbaren reellen Zahlen
- **Darstellung**
  - Wertebereich ist ***beschränkt*** (im Unterschied zur Mathematik)
  - Genauigkeit der Darstellung ist ***beschränkt***
  - Rechnen mit Gleitkommazahlen ist immer **fehlerbehaftet**!
    - "Numerik" liefert Techniken und Algorithmen, um Fehler zu beherrschen

Beispiel für Gleitkommazahlen

```
MODULE Gleitkomma EXPORTS Main;
(* Dieses Programm zeigt Rundungsprobleme bei Gleitkommazahlen *)

IMPORT SIO;


BEGIN
  (* A + B - B mit A ist viel kleiner als B*)
  SIO.PutReal ( 0.00000005 + 3000.0 - 3000.0 );

END Gleitkomma.
```



Zeichentyp CHAR

Typ CHAR

- **CHAR** (character) bezeichnet eine ***endliche***, ***geordnete Menge*** von Zeichen
- **CHAR** ist ein ***Ordinaltyp***

Wertebereich

- Latin-1, viele Rechner benutzen den ASCII-Zeichensatz
- Zeichenliterale: 'A' 'z' '1'
- Spezialzeichen:

`\n`	Zeilenvorschub	`\t` Tabulator	`\\` Backslash
`\'`	Apostroph	`\f` Seitenumbruch	
`\r`	Wagenrücklauf	`\"` Anführungszeichen	

Operationen

- Vergleichsoperationen (**=**, **#**, **<**, **>**, **<=**, **>=**)
- Vordefinierte Funktionen **FIRST(CHAR), LAST(CHAR), INC(z), DEC(z), ORD(z), VAL(i)**

Beispiel - Zeichen

```
MODULE KleinGross EXPORTS Main;
(* Dieses Programm wandelt einen Klein- in einen Grossbuchstaben um *)

IMPORT SIO;

PROCEDURE Offset (): INTEGER =
BEGIN
  RETURN (ORD('A') - ORD('a'));
END Offset;


BEGIN
  (* Kleinbuchstaben einlesen und umwandeln *)
  SIO.PutChar ( VAL(ORD(SIO.GetChar()) + Offset(), CHAR) );

END KleinGross.
```

Texte (Zeichenketten)

- **Typ TEXT**
  - repräsentiert eine ***beliebig lange Folge*** von Zeichen (kann auch leer sein)
  - in vielen Sprachen nicht explizit vorhanden
- **Wertebereich**
  - Textliterale werden in " " notiert
  - z.B. "Das ist ein Text mit Zeilenvorschub\n"
"\"Dieser Text beginnt und endet mit einem Hochkomma\""
- **Operationen**
  - Konkatenation : &
    - Bspl: "Heute " & "ist " & "Freitag." ⇨ "Heute ist Freitag."
  - Schnittstelle des Moduls "Text"
    - Equal, Length, Empty, FindChar ….

Wahrheitswerte

- **Typ BOOLEAN**
  - repräsentiert die beiden vordefinierten Wahrheitswerte
- **Wertebereich**
  - wahr, Literal **TRUE**
  - falsch, Literal **FALSE**
- **Operationen**
  - Komplement (**NOT**), Oder (**OR**), Und (**AND**)

p	q	NOT q	p OR q	p AND q
1	1	0	1	1
1	0	1	1	0
0	1	0	1	0
0	0	1	0	0

1 wahr
0 falsch

Vordefinierte Datentypen

Beispiel für Wahrheitswerte

```
MODULE BooleanM EXPORTS Main;
(* Dieses Programm berechnet die Wahrheitstabelle NOT, OR, AND *)
IMPORT SIO, Fmt;

PROCEDURE NotOrAnd ( a, b : BOOLEAN) : TEXT =
BEGIN
  RETURN (Fmt.Bool(a) & " " & Fmt.Bool(b) & " : " &
          Fmt.Bool( NOT(b))       & " " &
          Fmt.Bool( a OR b)       & " " &
          Fmt.Bool(a AND b) );
END NotOrAnd;

BEGIN
  (* Ausgabe der Wertetabelle *)
  SIO.PutLine(NotOrAnd(FALSE, FALSE));
  SIO.PutText(NotOrAnd(TRUE, FALSE));
  SIO.PutLine(NotOrAnd(FALSE, TRUE));
  SIO.PutLine(NotOrAnd(TRUE, TRUE));

END BooleanM.
```


```
FALSE FALSE : TRUE  FALSE FALSE
TRUE  FALSE : TRUE  TRUE  FALSE
FALSE TRUE  : FALSE TRUE  FALSE
TRUE  TRUE  : FALSE TRUE  TRUE
```


Beispielprogramm aus Vorkurs

Schrittweise Verfeinerung
Kommentierung, aussagekräftige Bezeichner

```
Eingabe
Berechnung
Ausgabe
```

```
(* Eingabe *)
   Überschrift
   Bestellung eingeben
   Bestätigung der Bestellung
...
```

```
(* Eingabe *)
  (* Überschrift *)
   PutText("----------"); Nl();
   PutText("Kaffeebestellung"); Nl();
   PutText("----------"); Nl();
  (* Bestellung eingeben *)
   PutText("Wieviel Kaffee wollen Sie kaufen? "); Nl();
   AnzahlKaffee := GetInt();
  (* Bestätigung der Bestellung *)
   PutText("Sie haben bestellt:"); Nl();
   PutInt (AnzahlKaffee);
   PutText(" Säcke Kaffee zum Preis von je ");
   PutInt (KostenKaffee); PutText(" DM"); Nl();
   ...
```

Eingabe:
Übersicht
Was wird erwartet?
Eingabe war falsch?
Was war falsch?
Bestätigung der Eingabe

Ausgabe:
Übersicht
Ergebnis der Berechnung

Beispielprogramm aus Vorkurs

Variable, Konstante

```
VAR
  AnzahlKaffee : CARDINAL;
  SummeKaffee  : CARDINAL;
  GesamtSumme  : REAL;
```

Typisierter Behälter für Werte
mit Speicherplatz abhängig vom Typ,
hier arithmetische Operationen
Literale 10, 100, etc.

```
CONST
  KostenKaffee     = 200;
  MWST : CARDINAL  = 16;
  MWSTAnteil       = FLOAT(MWST)/100.0;
```

Hier muß keine Typangabe stehen,
inferiert aus Anfangswert,
Literal, Compilezeitausdruck

Beispielprogramm aus Vorkurs

Einfache Berechnung mit Kontrollstrukturen: Fallunterscheidung

```
(* Bedingte Anweisung *)
IF (AnzahlKaffee >= 0) AND (AnzahlKaffee < 5)THEN
     KaffeeRabatt:= 0;
ELSEIF (AnzahlKaffee >= 5) AND (AnzahlKaffee < 10) THEN
     KaffeeRabatt:= 5;
ELSEIF (AnzahlKaffee >= 10) AND (AnzahlKaffee < 50) THEN
     KaffeeRabatt:= 10;
ELSEIF (AnzahlKaffee >= 50) AND (AnzahlKaffee < 100) THEN
     KaffeeRabatt:= 20;
ELSE (* AnzahlKaffee >= 100 *)
     KaffeeRabatt:= 30;
END;
...
```

sequentielle Ausführung

Alternativen disjunkt

```
CASE AnzahlKaffee OF
  | 0..4 =>       KaffeeRabatt := 0;
  | 5..9 =>       KaffeeRabatt := 5;
  | 10..49 =>     KaffeeRabatt := 10;
  | 50..99 =>     KaffeeRabatt := 20
ELSE (* größer 100 *)
   KaffeeRabatt := 30;
END;
```

Beispielprogramm aus Vorkurs

Einfache Berechnung mit Kontrollstrukturen: Schleifen

```
FOR i:= 1 TO 4 DO

    PutText("Bestellung der Sorte ");
    PutInt(i);
    Nl();
    PutText("Wieviele Säcke Kaffee dieser Sorte wollen Sie kaufen? ");
    AnzahlKaffeeProSorte[i] := GetInt();
```

FOR-Schleife
Anzahl der Schleifendurchläufe bekannt

keine Terminationsprobleme

WHILE-Schleife
Anzahl der Schleifendurchläufe nicht bekannt

Terminationsüberlegung

```
BEGIN
WHILE Taste # 'n' DO
        Programmtext

  PutText("Noch eine Bestellung aufgeben ? (j/n)  ");
  Taste := GetChar();
END; (* WHILE *)
```


Was haben wir gelernt

- **Datentypen: einfache, zusammengesetzte, vordefinierte, selbstdefinierte**
- **einfache (skalare) vordefinierte Datentypen: `INTEGER`, `CARDINAL`, `REAL`, `LONGREAL`, `CHAR`, `BOOLEAN`**
- **zusammengesetzter vordefinierter Datentyp `TEXT`**
- **Blöcke: Deklarationen und Anweisungen**
  Deklarationen abgearbeitet, Anweisungen ausgeführt, Ausdrücke ausgewertet
- **zusammengesetzte Anweisungen mit Kontrollstrukturen (später)**
- **zusammengesetzte Typen mit Datentypkonstruktoren (später)**
- **Programmiertechnik: Bezeichnerwahl, Kommentierung, Bedienerschnittstellengestaltung**
- **Eingaben: Übersicht, was wird erwartet, Meldung: falsch, richtig, was war falsch, Bestätigung der Eingabe**
- **Ausgabe Übersicht, übersichtliche Darstellung der Ergebnisse**
- **I/O: Korrektheit, Layout, Bequemlichkeit (Masken, Menüs später) unterschiedliche Kenntnis (Novize, Experte später)**
- **Variable, Konstante**
- **Fallunterscheidungen: bedingte Anweisungen, Auswahlanweisungen**
- **Schleifen: Zählschleifen, bedingte Schleifen**

Glossar

- Datentyp
- Block
- einfache/zusammengesetzte Anweisung
- einfacher/zusammengesetzter Datentyp
- Kontrollstrukturen
- Datentypkonstruktoren
- Fallunterscheidungen IF, CASE
- Schleifenformen FOR, WHILE; Schleifenkopf, Schleifenrumpf

Informatik-Grundlagen

- Klärung „Informatik“
- Geschichte der Informatik
- Algorithmus
- Software, Programm, Programmentwicklung
- Von-Neumann-Rechner

Was ist Informatik?

Der Begriff "Informatik"

- ein ***Kunstwort*** , das zu Beginn der 60er Jahre zur Bezeichnung einer sich neu entwickelnden Disziplin geschaffen wurde. Es setzt sich aus Bestandteilen der beiden Worte ***Information*** und ***Mathematik*** zusammen. Darin kommt zum Ausdruck, daß Informatik die Wissenschaft von der Informationsverarbeitung ist, und eine große Nähe zur Mathematik hat. Heute wird der Begriff z.T. sehr anwendungsnah gebraucht und Synonym zur ‘Informationstechnik‘ verwendet.

Informatik ist die Wissenschaft

- von der ***systematischen*** Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von ***Computern*** (vgl. DUDEN Informatik, 1993)

Informatik versteht sich als Wissenschaft

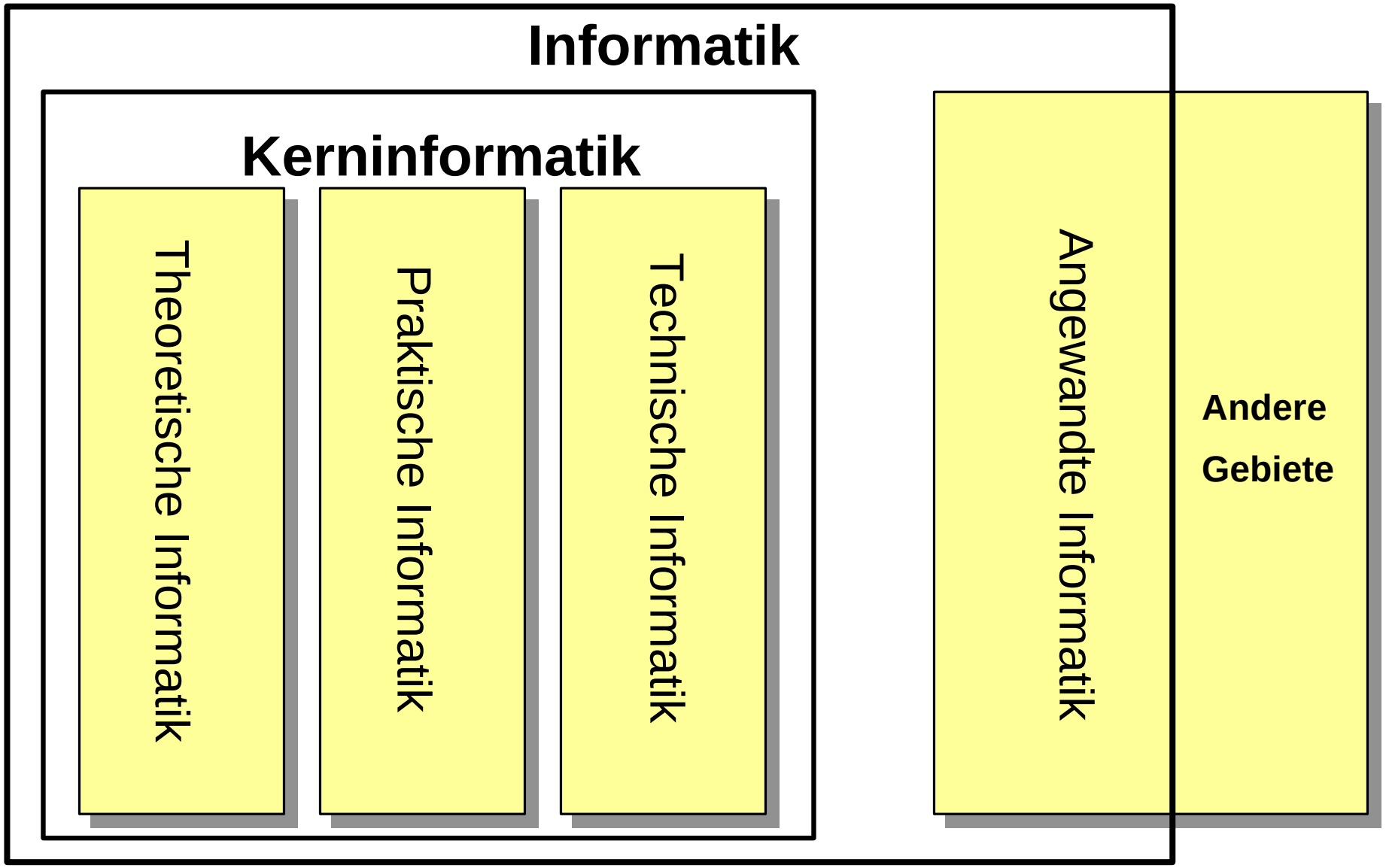
- der Analyse, Konzeption und Realisierung von Systemen, die aus miteinander und mit ihrer Umwelt kommunizierenden Akteuren bestehen (vgl. Studienführer Informatik, RWTH Aachen, 1998)

Hauptaufgaben der Informatik

- **Hauptaufgabe der Informatik ist die Entwicklung**
  - ***formaler*, *maschinell ausführbarer Verfahren*** zur Lösung von Informationsverarbeitungsproblemen, die häufig als Teilprobleme komplexer Kommunikations- oder Organisationsprobleme auftreten.

- **Die Forderung der Durchführbarkeit mittels einer Maschine (i.a. eines Digitalrechners) bedingt,**
  - daß die zu verarbeitenden Informationen als ***maschinell verarbeitbare Daten*** dargestellt werden, und
  - daß die Lösungsverfahren bis ***ins Detail*** formal beschrieben werden.

Gebiete der Informatik - 1



Gebiete der Informatik - 2

Theoretische Informatik

- ***Formale Modelle*** zur Beschreibung und Untersuchung von Algorithmen, Computern, etc.
- Teilgebiete: Formale Sprachen, Automatentheorie, Komplexitätstheorie etc.

Technische Informatik

- Funktioneller ***Aufbau von Computern***, Entwurf und Entwicklung von Rechnern, Geräten und Schaltungen
- Teilgebiete: Rechnerarchitektur, VLSI-Entwurf etc.

Praktische Informatik

- Prinzipien und Techniken der Fundierung und Realisierung in Software (großer Programmsysteme!)
- Teilgebiete: Softwaretechnik, Informationssysteme, Compilerbau, Betriebssysteme, Künstliche Intelligenz, Kommunikation/verteilte Systeme, Parallele Systeme, etc.

Gebiete der Informatik - 3

- **Angewandte Informatik**
  - ***Anwendung*** der Methoden der ***Kerninformatik*** in anderen Wissenschaften
    - Entwicklung ***spezieller*** Verfahren und Darstellungstechniken
  - Bindestrich-Informatik:
    - Wirtschafts-Informatik, Medizin-Informatik, Bio-Informatik, Rechts-Informatik
  - Grenzen zwischen ***Praktischer Informatik*** und ***Angewandter Informatik*** sind zum Teil fließend.

- **Das, was man unter Informatik versteht, kann man in solchen Definitionen jedoch nicht *endgültig* fassen. Schließlich ändert sich die Beschreibung der Definitionen einer Wissenschaft, wie in anderen Fällen auch, über die Zeit.**

Geschichte - 1

- **Altertum–Mittelalter:**
  - Verwendung des ***Abakus*** (Brett mit verschiebbaren Kugeln) als Hilfsmittel für die vier Grundrechenarten.
- **9. JH.:**
  - Der arabische Mathematiker und Astronom ***Ibn Musa Al-Chwarismi*** schreibt das Lehrbuch “Kitab al jabr w' almuqabala” (“Regeln der Wiedereinsetzung und Reduktion”). Das Wort “***Algorithmus***” geht auf seinen Namen zurück.
- **1547: Adam Riese (1492–1559)**
  - veröffentlicht ein Rechenbuch, in dem er die ***Rechengesetze*** des aus Indien stammenden ***Dezimalsystems*** (5. Jh.n. Chr.) beschreibt. Im 17. Jahrhundert setzt sich das Dezimalsystem in Europa durch.
- **Wilhelm Schickard (1592–1635)**
  - konstruiert für seinen Freund Kepler (1571–1630) eine ***Maschine***, die addieren, subtrahieren, multiplizieren und dividieren kann. Sie bleibt unbeachtet.
- **1641: Blaise Pascal (1623–1662)**
  - konstruiert eine Maschine, mit der man ***sechsstellige Zahlen*** addieren kann.

Geschichte - 2

- **1674: Gottfried Wilhelm Leibniz (1646–1716)**
  - konstruiert eine ***Rechenmaschine*** mit Staffelwalzen für die vier ***Grundrechenarten***. In diesem Zusammenhang befaßt er sich auch mit der binären Darstellung von Zahlen.
- **1774: Philipp Matthäus Hahn (1739–1790)**
  - entwickelte eine mechanische Rechenmaschine, die ***erstmals zuverlässig*** arbeitet.
- **Ab 1818:**
  - Rechenmaschinen nach dem Vorbild der Leibnizschen Maschine werden serienmäßig hergestellt und dabei ständig weiterentwickelt.
- **1838: Charles Babbage (1792–1871)**
  - plant eine Maschine, die "***Analytical Engine***", bei der die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene ***Lochkarten*** gesteuert wird.
- **1886: Hermann Hollerith (1860–1929)**
  - entwickelt in den USA elektrisch arbeitende ***Zählmaschinen für Lochkarten***, mit denen die statistischen Auswertungen der Volkszählungen vorgenommen werden.

Geschichte - 3

- **1934: Konrad Zuse (1910–1995)**
  - beginnt mit der Planung einer ***programmgesteuerten Rechenmaschine***. Sie verwendet das binäre Zahlensystem.
- **1937: Die mechanische Anlage *Z 1* von Zuse ist fertig.**
- **1941: Die elektromechanische Anlage Z 3 von Zuse ist fertig.**
  - Dies ist der ***erste funktionsfähige programmgesteuerte Rechenautomat***. Das Programm wurde mit ***Lochstreifen*** eingegeben. Die Anlage verfügt über 2000 Relais und eine Speicherkapazität von 64 Worten a 22 Bit. Multiplikationszeit: etwa 3 s.
- **1944: Howard H. Aiken (1900–1973)**
  - erstellt in Zusammenarbeit mit der Harvard-University und der Firma IBM die teilweise programmgesteuerte Rechenanlage MARK I. Additionszeit 1/3 s, Multiplikationszeit: 6 s.
- **1946: J. P. Eckert und J. W. Mauchly**
  - stellen die ENIAC (Electronic Numerical Integrator and Automatic Calculator) fertig. Dies ist der erste ***voll elektronische Rechner*** (18.000 Elektronenröhren). Multiplikationszeit: 3 ms.

Geschichte - 4

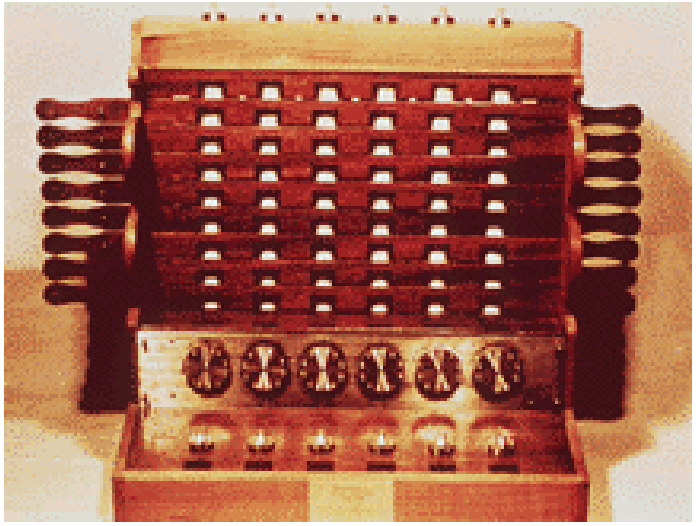
- **1946–1952:**
  - Auf der Grundlage der Ideen **John v. Neumanns** (1903–1957) (Einzelprozessor, Programm und Daten im gleichen Speicher; Von-Neumann-Rechner) und seiner Kollegen am Institute of Advanced Study at Princeton (H.H.Goldstine, A.W.Burks) werden weitere Computer in Universitätslabors entwickelt (“Pionierzeit”).
- **1949: M.V. Wilkes (University of Manchester)**
  - stellt mit der EDSAC (Electronic Delay Storage Automatic Calculator) den **ersten universellen Digitalrechner** (gespeichertes Programm) fertig.
- **Ab 1950:**
  - **Industrielle** Rechnerentwicklung und -produktion.
- **Bedeutung von Software: Softwaregeschichte**

  Algorithmen → Babylonier

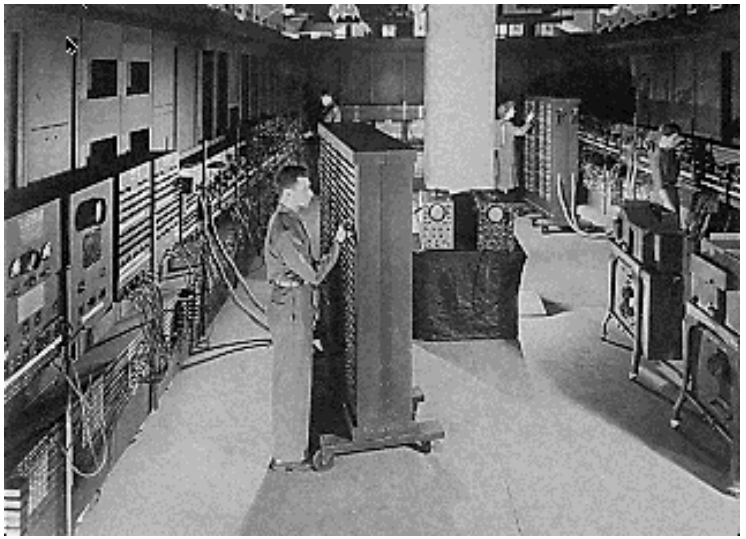
  Programmierung → Ada, Zuse

Historische Rechner

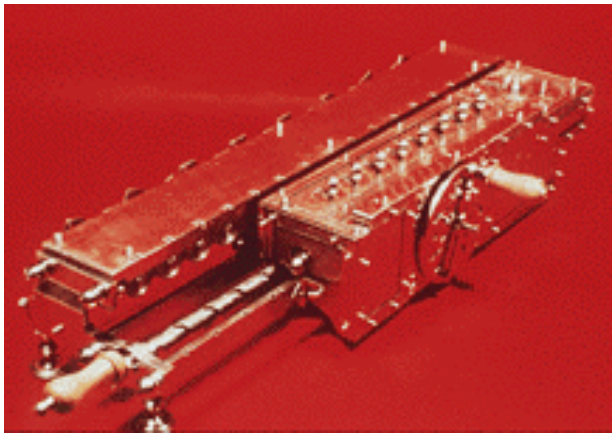
Schickard



ENIAC



Leibniz



MARK1

Informelle Definition

- **Ein Algorithmus ist ein Verfahren, welches**
  - in einem ***endlichen*** Text niedergelegt werden muß
  - ***effektiv*** ausführbar ist,
  - Elementaroperationen enthält, die durch die jeweilige Situation ***eindeutig*** bestimmt sind
  - ***Ein- und Ausgabe*** ermöglicht
  - durch eine (mechanisch oder elektronisch arbeitende) *Maschine* ***ausgeführt*** werden kann.
- **Anzahl und Ausführungszeit der Elementaroperationen sind beschränkt**
- **Ein Algorithmus (Programm) wird durch eine Maschine schrittweise ausgeführt**
  - die ausführende Instanz muß die Vorschrift ***interpretieren*** und ***korrekt*** ausführen
  - ein Algorithmus ***terminiert***, wenn er nach endlich vielen Schritten abbricht

Euklidischer Algorithmus - 1

Euklidischer Algorithmus

- Problem:
  - Man bestimme zu je zwei natürlichen Zahlen n und m den größten gemeinsamen Teiler ggt(n,m)

Algorithmus

Wiederhole:

❶ WENN n < m ist, DANN vertausche man n und m

❷ WENN m = 0 ist, DANN ist n der ggt(n,m) und
man beende den Algorithmus

❸ WENN m≠ 0 ist, DANN bilde man den Rest r, der bei der Division von n durch m bleibt,
dann ersetze man n durch m und m durch r
und beginne von vorn.

Euklidischer Algorithmus - 2

n = 6
m = 10

- **Erster Durchlauf**
  - da 6 < 10, so setzt man n = 10 und m = 6
  - da 6 ≠ 0 ist, gehen zu Schritt 3
  - r = 4. Man erhält also n = 6 und m = 4
- **Zweiter Durchlauf**
  - Da 6 ≥ 4 ist, gehe zu Schritt 2
  - Da 4 ≠ 0 ist, gehe zu Schritt 3
  - r = 2. Man erhält n = 4 und m = 2
- **Dritter Durchlauf**
  - Da 4 ≥ 2 ist, gehe zu Schritt 2
  - Da 2 ≠ 0 ist, gehe zu Schritt 3
  - r = 0. Man erhält n = 2 und m = 0
- **Abbruch**
  - Da 2 ≥ 0 ist, gehe zu Schritt 2
  - Da m = 0 ist, Programmende
- **Ergebnis: ggt(10, 6) =2**

n	m
6	**10**
10	6
10	6
6	4
6	4
6	4
4	2
4	2
4	2
2	0
2	0
2	0

Ablaufverfolgung

(Trace)

für Überprüfung von Programmen

Eigenschaften - 1

Abstraktion

- ein Algorithmus löst i. a. eine ***Klasse von Problemstellungen*** (z.B. Suchen eines Musters in einer Zeichenkette): versch. Ein-, Ausgabewerte

Finitheit

- statisch finit: ein Algorithmus besitzt eine ***endliche Länge***
- dynamisch finit: während der Abarbeitung darf nur ***endlich viel Speicherplatz*** belegt werden

Terminierung

- terminierend: nach ***endlich vielen Schritten*** liegt ein Resultat vor
- sonst ***nicht-terminierend*** (z.B. Steuerungsalgorithmen)

Determinismus

- deterministisch: zu jedem Zeitpunkt besteht ***höchstens eine*** Möglichkeit der Fortsetzung
- nicht-deterministisch: an mindestens einer Stelle gibt es eine ***Wahlmöglichkeit*** für die Fortsetzung

Eigenschaften - 2

Determiniertheit

- determiniert: bei ***gleichen Eingaben*** und Startbedingungen wird das ***gleiche Ergebnis*** erzielt
- nicht-determiniert: es werden ***unterschiedliche Ergebnisse*** erzielt (z.B. Anwendung von heuristischen Methoden, syst. Probieren)

Bemerkung

- ein terminierender, deterministischer Algorithmus ist immer determiniert
- ein terminierender, nicht-deterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Terminierung - Nichtterminierung

- **Sei**
  - $power : N^+ \times N \rightarrow N$

$$power(a, b) = \begin{cases} 1 & \text{falls } b = 0 \\ a * power(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert*** im Definitionsbereich.

  - Wir weiten den Definitionsbereich aus:

$$power2 : Z \times Z \rightarrow Z$$

$$power2(a, b) = \begin{cases} 0 & \text{falls } a = 0 \text{ und } b > 0 \\ 1 & \text{falls } a \neq 0 \text{ und } b = 0 \\ a * power2(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert nicht*** für power2 (0,0) und power2 (a,b) mit $b < 0$ und $a \neq 0$)

Anmerkung zur Nichtterminierung

Während wir die Nichtterminierung bei der ***manuellen Auswertung*** leicht feststellen können, fällt dies auf dem Rechner sehr viel schwerer. Solange der Rechner ***vor sich hin*** rechnet, können wir den Unterschied zwischen einem ***nicht-terminierenden*** und einem ***sehr langwierigen Algorithmus*** nicht feststellen.

Fragen

- **Wie kann man aus einer Lösungsidee einen Algorithmus konstruieren?**
  - "schrittweise Programmentwicklung"
- **Wie kann man Algorithmen darstellen?**
  - "Flußdiagramme", Programme
- **Wie beweist man, daß ein Algorithmus tatsächlich das tut, was er tun soll?**
  - Verifikation: partielle Korrektheit
  - Termination
- **Wie "gut" ist ein Algorithmus?**
  - Speicherverbrauch, benötigte Zeit
  - Aufwandsabschätzungen

Typische Problemklassen

- **Sortieralgorithmen**
  - Ordnen von Elementen
- **Suchalgorithmen**
  - Auffinden von Elementen
- **Algorithmen zur Verarbeitung von Zeichenfolgen**
  - Mustererkennung, Verschlüsselung, Komprimierung
- **Geometrische Algorithmen**
  - z.B. Schnittmenge geometrischer Objekte
- **Algorithmen für Graphen**
  - Suchen im Graph, kürzester Weg
- **Mathematische Algorithmen**
  - Rechnen mit Polynomen und Matrizen
- **Viele Anwendungsalgorithmen:** Kontrolle, Steuerung, Simulation

Standard-Algorithmen der Informatik

Überblick

- **Neben der Entwicklung der Hardware (Rechner)**
  - wurden seit 1955 ***Programmiersprachen*** entwickelt, um Algorithmen zu formulieren, damit sie von einem Rechner ausgeführt werden können.
- **Rechner "realisieren" Algorithmen,**
  - durch ***schrittweise Abarbeitung*** von ***Programmen***, die in einer Programmiersprache geschrieben sind.
- **Um einen Rechner zu programmieren,**
  - muß die ***Syntax*** und ***Semantik*** der verwendeten Programmiersprache bekannt sein
- **In diesem Zusammenhang spricht man häufig auch von Software**
  - "***Software***" und "***Programm***" sind aber nicht dasselbe

Definition: Software

1. Definition: Software

- Informatik-Duden: Gesamtheit ***aller Programme***, die auf einer Rechenanlage eingesetzt werden können
  - ***Systemsoftware***: Programme die für den korrekten Ablauf einer Rechenanlage notwendig sind
  - ***Anwendungssoftware***: dient zur Lösung von Benutzerproblemen

2. Definition: Software

- IEEE Standrad Glossary of Software Engineering Terminoloy
  - "Computer ***programs***, ***procedures***, and possibly associated ***documentation*** and ***data*** pertaining to the operation of a computer system."

Definition: Programm

1. Definition: Programm in einer Programmiersprache

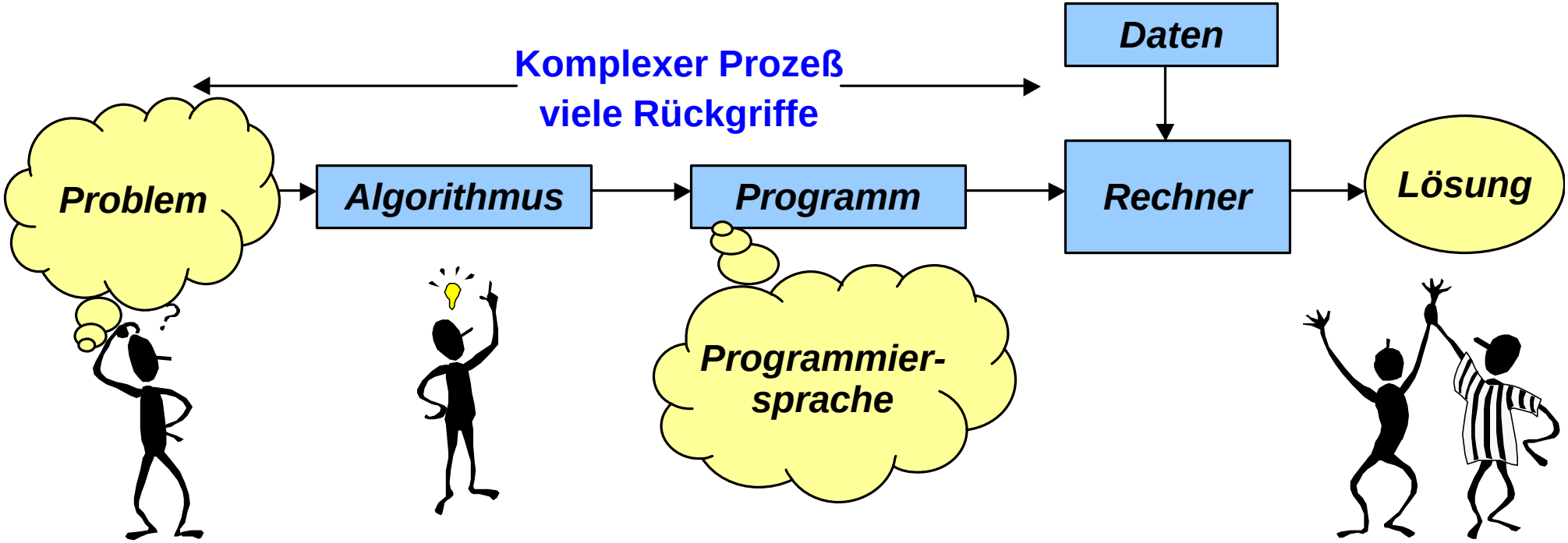
- Formulierung eines Algorithmus und der dazugehörigen Datenbereiche in einer Programmiersprache
  - ist ***exakt*** (formal) definiert
  - nimmt Bezug auf eine bestimmte Darstellung der ***Daten***
  - ist auf einer Rechenanlage ***ausführbar***

2. Definition: Software

- IEEE Standrad Glossary of Software Engineering Terminoloy
  - "A combination of ***computer instructions*** and ***data definitions*** that enable computer hardware to ***perform*** computational or control functions".

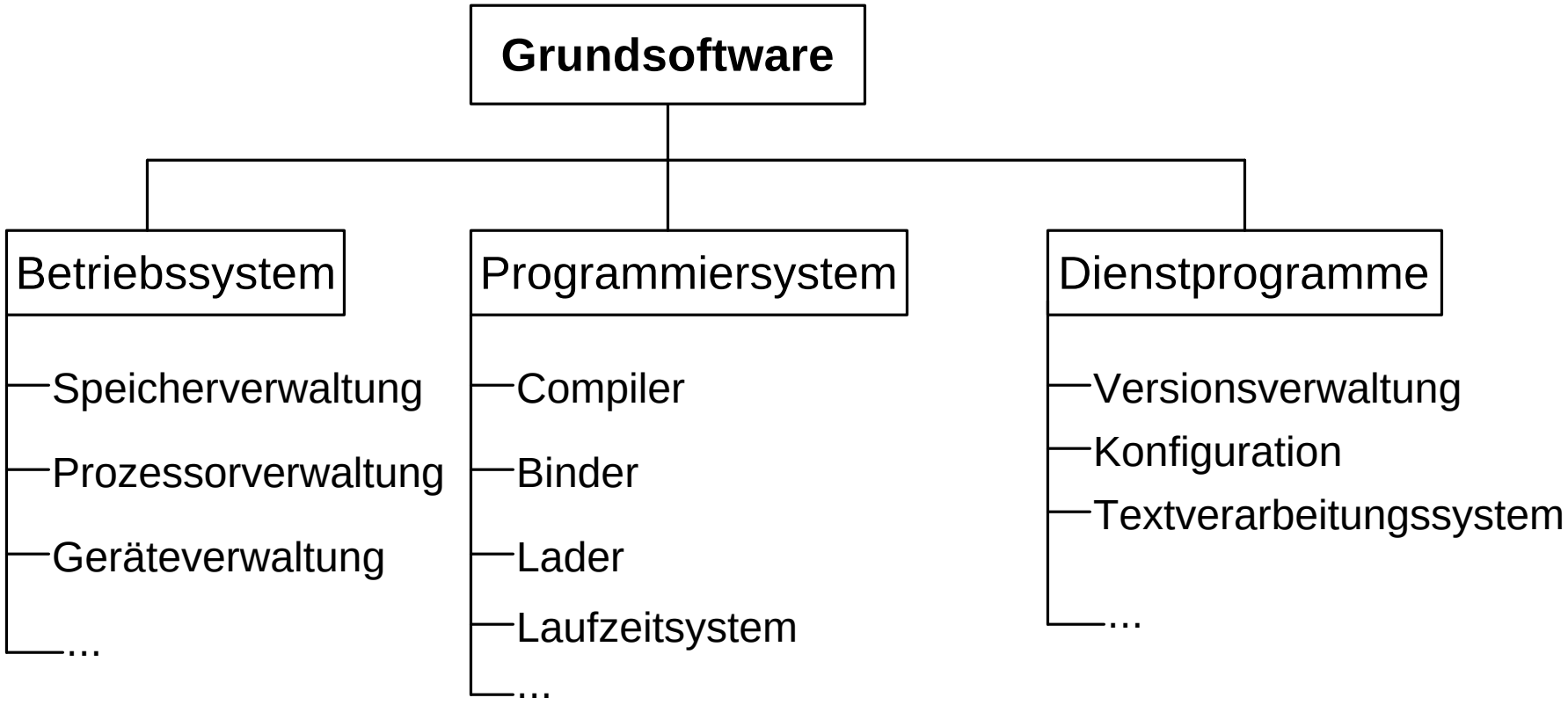
Programmentwicklung

- **Unter dem Begriff Programmieren versteht man**
  - das **Lösen von Problemen** unter Zuhilfenahme eines **Rechners**



- **Programmieren [1] Software-Entwicklung**
- **Programmieren ist *ein Teil* der Software-Entwicklung**

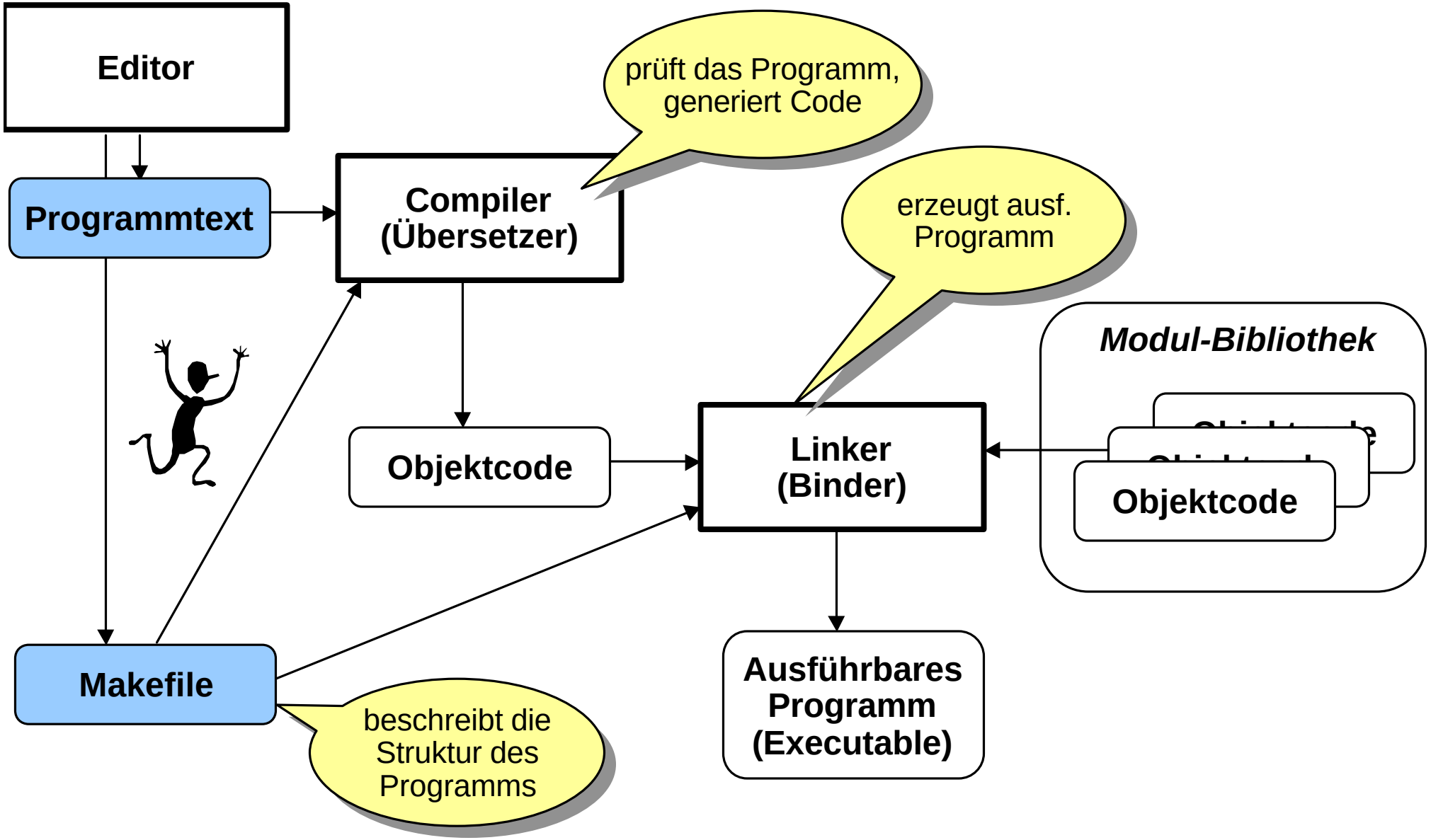
Grundsoftware(Systemsoftware)



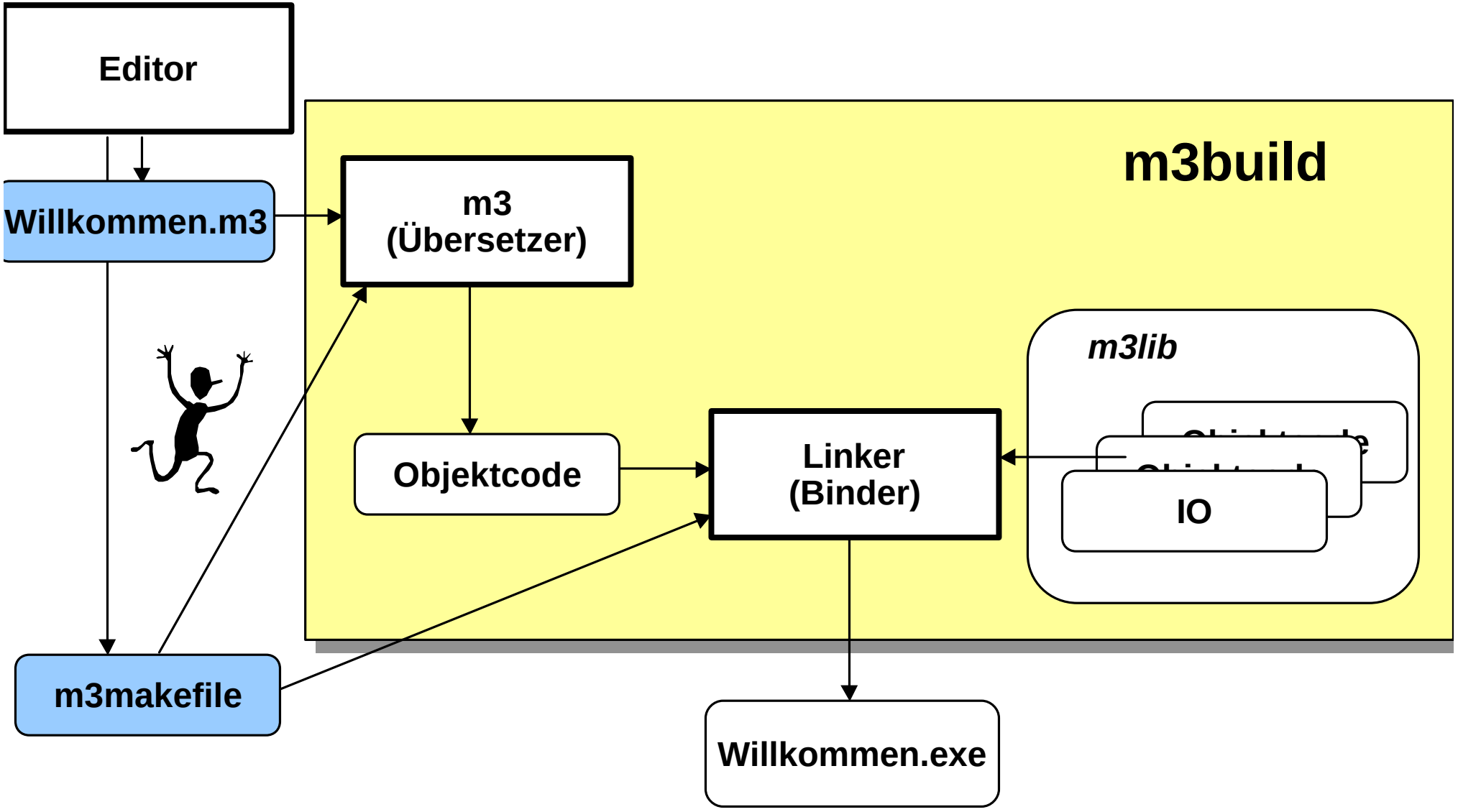
Das erste M3-Programm

```
MODULE Willkommen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Umgebung          : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt          : 16.08.98
   Letzte Aenderung: 20.08.98
*)

IMPORT SIO;
BEGIN
  SIO.PutText("Willkommen zum Studium in Aachen.");
END Willkommen.
```

Vom Programmtext zum ausführbaren Programm



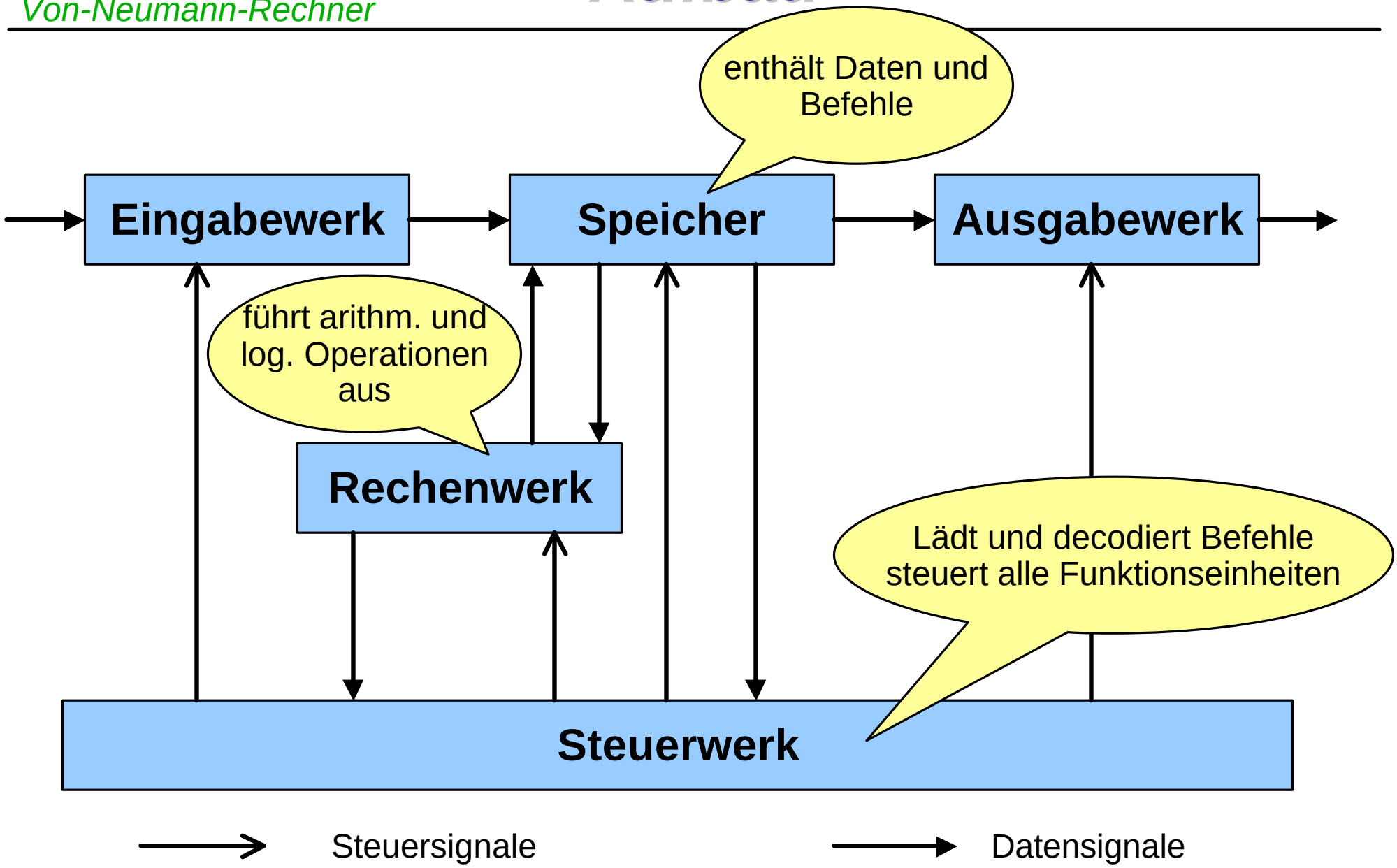
Edit-Compile-Link-Run bei Modula-3



Überblick

- **Definiert die wesentlichen Elemente eines Universalrechners**
  - Rechner soll nicht für eine ***bestimmte*** Problemklasse konstruiert sein
  - Zur Lösung des Problems muß ein ***Programm*** eingegeben und in den ***Speicher*** abgelegt werden
- **1946 von John von Neumann als Konzept für die EDVAC vorgeschlagen**
- **fast alle heutigen Rechner basieren darauf und sind Weiterentwicklungen davon**
  - Nicht-von-Neumann-Rechner sind Gegenstand der Forschung
- **diese Architektur prägt viele Programmiersprachen (imperative Programmiersprachen)**

Aufbau


enthält Daten und Befehle
Eingabewerk
Speicher
Ausgabewerk
führt arithm. und log. Operationen aus
Rechenwerk
Lädt und decodiert Befehle steuert alle Funktionseinheiten
Steuerwerk
Steuersignale
Datensignale

Steuerwerk

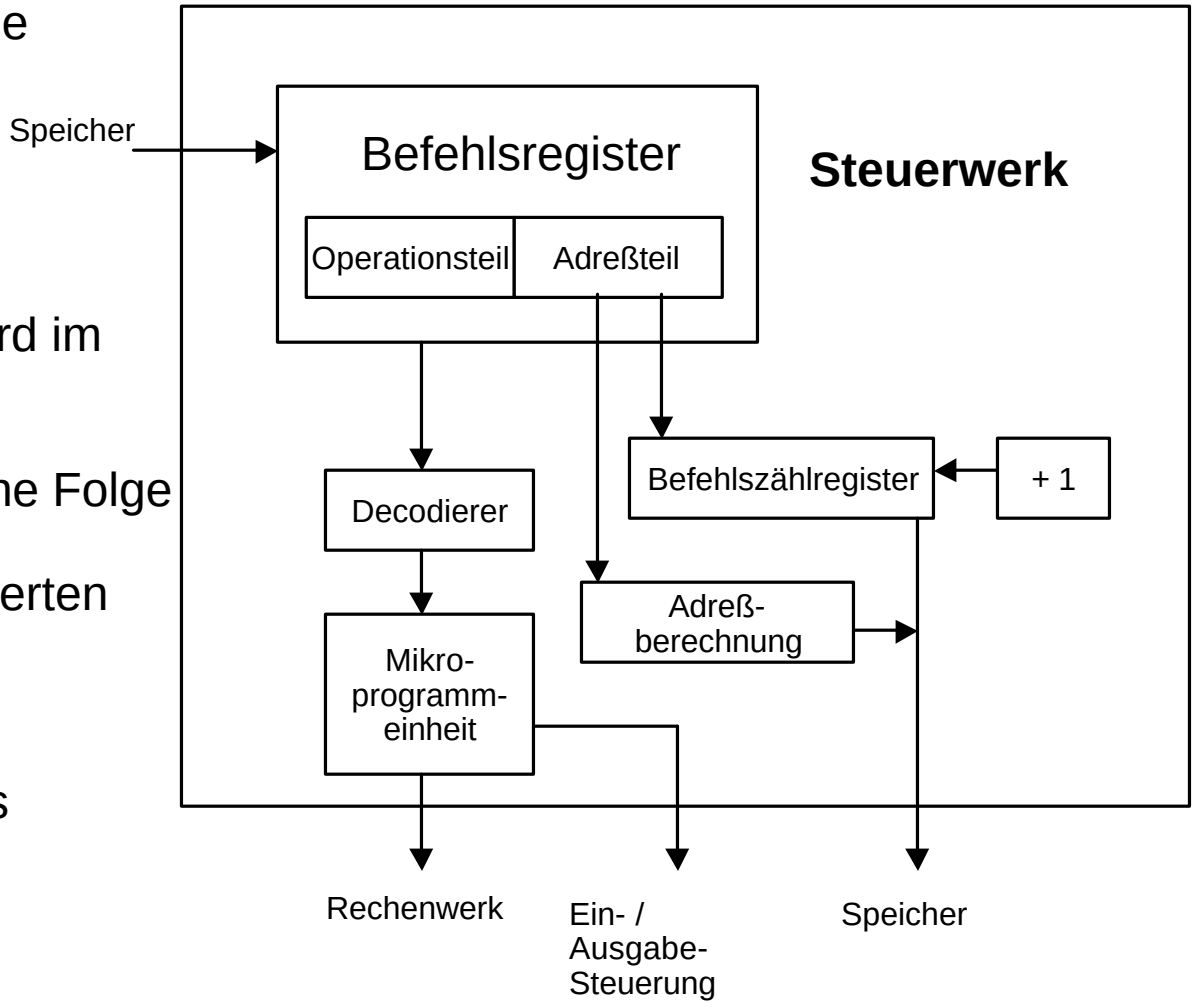
Lädt, decodiert und interpretiert die Befehle

Befehlsregister enthält den aktuellen Befehl

Der Operationsteil des Befehls wird im Decodierer entschlüsselt

Mikroprogrammeinheit erzeugt eine Folge von Signalen zur Ausführung des Befehls (abhängig von der decodierten Information)

Das Befehlzählregister speichert die Adresse des nächsten Befehls



Rechenwerk

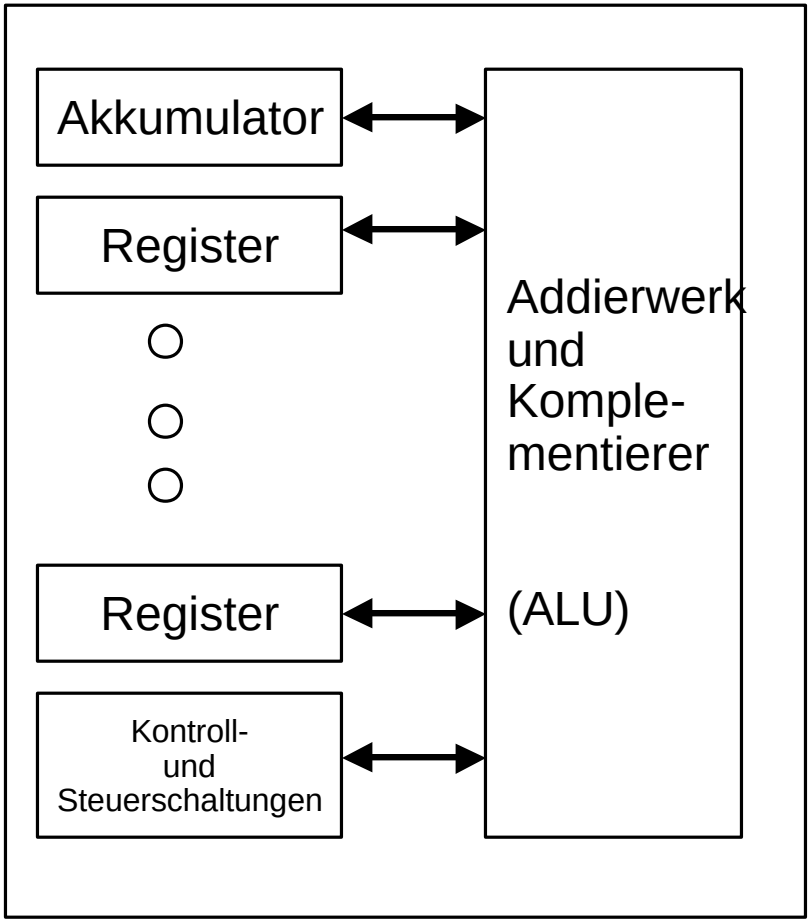
Führt arithmetische und logische Operationen durch (ALU)

erhält vom Steuerwerk die benötigten Operanden

wesentliche Einheiten sind Addierer und Komplementierer (damit können die Grundrechenarten durchgeführt werden)

implementiert Algorithmen für Multiplikation und Division

zusammen mit dem Steuerwerk nennt man es auch CPU



Prinzipien - 1

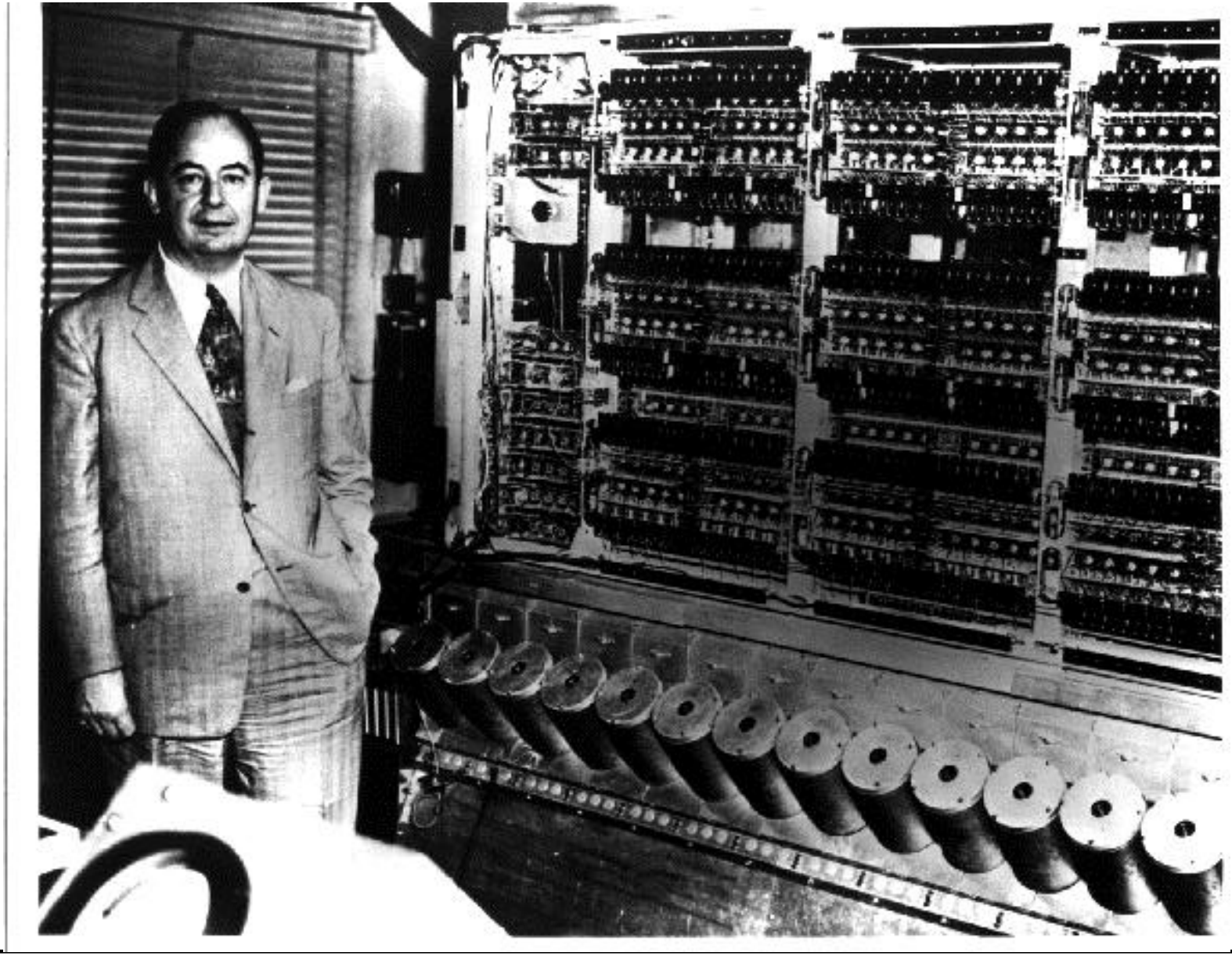
- Der Rechner besteht aus *Steuerwerk, Rechenwerk, Speicher, Eingabewerk* und Ausgabewerk.
- Die Struktur des von-Neumann-Rechners ist *unabhängig* von den zu bearbeitenden Problemen. Zur Lösung eines Problems muß von außen das *Programm* eingegeben und im Speicher abgelegt werden.
- Programme, Daten, Zwischen- und Endergebnisse werden in *demselben Speicher* abgelegt.
- Der Speicher ist in *gleichgroße Zellen* unterteilt, die fortlaufend durchnumeriert sind. Über die Nummer (*Adresse*) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden.
- Aufeinanderfolgende Befehle eines Programms werden in *aufeinanderfolgenden* Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
- Durch *Sprungbefehle* kann von der Bearbeitung der Befehle in der gespeicherten Reihenfolge abgewichen werden.

Prinzipien - 2

- Es gibt zumindest
  - *arithmetische* Befehle wie Addieren, Multiplizieren usw.;
  - *logische* Befehle wie Vergleiche, logisches Nicht, Und, Oder usw.;
  - *Transportbefehle*, z.B. vom Speicher zum Rechenwerk und für die Ein-/Ausgabe;
  - bedingte *Sprünge*.
  - Weitere Befehle wie Schieben, Unterbrechen, Warten usw. kommen hinzu.
- Alle Daten (Befehle, Adressen usw.) werden *binär codiert*. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (*Decodierung*).

Historische Rechner - J. v. Neumann

Von-Neumann-Rechner



Was haben wir gelernt

- **Einordnung der Informatik als Wissenschaft**
  - Aufgabe der Informatik
  - Einteilung der Informatik
- **Wo kommt die Informatik her**
  - Entwicklung der Rechenmaschinen
- **Was versteht man unter einem Algorithmus**
  - Eigenschaften von Algorithmen
- **Was versteht man unter den zentralen Begriffen**
  - Software
  - Programm
  - Programmieren
- **grober Aufbau eines von-Neumann-Rechners**

Glossar

- **Informatik:**
  - Begriff, Einteilung, Einordnung, Aufgaben, Geschichte
- **Algorithmus:**
  - Definition, Eigenschaften, Klassen von Algorithmen
- **Software**
- **Grundsoftware**
  - Betriebssystem, Programmiersystem, Dienst- und Hilfsprogramme (Utilities)
- **Programm**
- **Programmieren, Programmentwicklung**
- **Von-Neumann-Rechner:**
  - Speicher, Rechenwerk, Steuerwerk. Ein-/Ausgabeeinheit, Befehlsgruppen

Informatik-Grundlagen

- **Klärung „Informatik“**
- **Geschichte der Informatik**
- **Algorithmus**
- **Software, Programm, Programmentwicklung**
- **Von-Neumann-Rechner**

Was ist Informatik?

Klärung Informatik

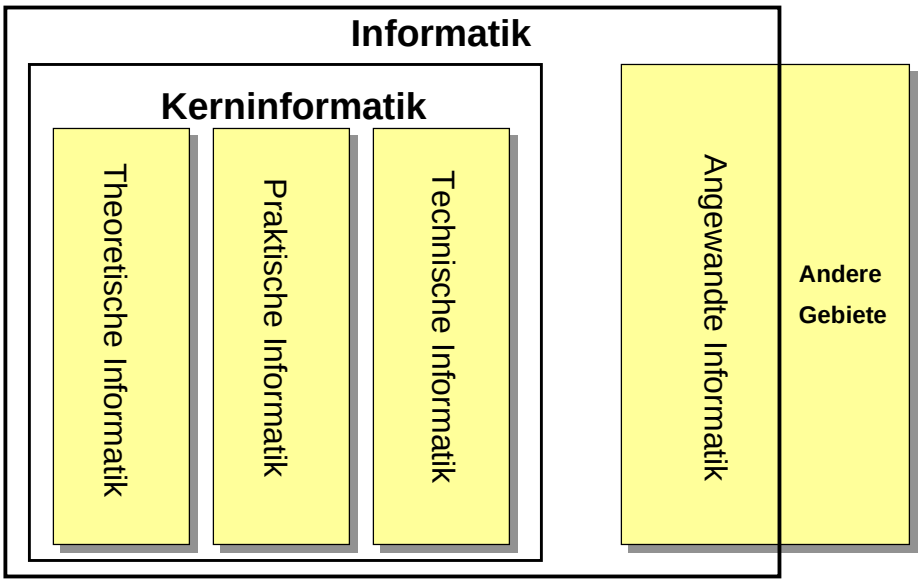
- **Der Begriff "Informatik"**
  - ein ***Kunstwort*** , das zu Beginn der 60er Jahre zur Bezeichnung einer sich neu entwickelnden Disziplin geschaffen wurde. Es setzt sich aus Bestandteilen der beiden Worte ***Information*** und ***Mathematik*** zusammen. Darin kommt zum Ausdruck, daß Informatik die Wissenschaft von der Informationsverarbeitung ist, und eine große Nähe zur Mathematik hat. Heute wird der Begriff z.T. sehr anwendungsnah gebraucht und Synonym zur ‘Informationstechnik‘ verwendet.
- **Informatik ist die Wissenschaft**
  - von der ***systematischen*** Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von ***Computern*** (vgl. DUDEN Informatik, 1993)
- **Informatik versteht sich als Wissenschaft**
  - der Analyse, Konzeption und Realisierung von Systemen, die aus miteinander und mit ihrer Umwelt kommunizierenden Akteuren bestehen (vgl. Studienführer Informatik, RWTH Aachen, 1998)

Hauptaufgaben der Informatik

- **Hauptaufgabe der Informatik ist die Entwicklung**
  - ***formaler, maschinell ausführbarer Verfahren*** zur Lösung von Informationsverarbeitungsproblemen, die häufig als Teilprobleme komplexer Kommunikations- oder Organisationsprobleme auftreten.

- **Die Forderung der Durchführbarkeit mittels einer Maschine (i.a. eines Digitalrechners) bedingt,**
  - daß die zu verarbeitenden Informationen als ***maschinell verarbeitbare Daten*** dargestellt werden, und
  - daß die Lösungsverfahren bis ***ins Detail*** formal beschrieben werden.

Gebiete der Informatik - 1



Gebiete der Informatik - 2

- **Theoretische Informatik**
  - ***Formale Modelle*** zur Beschreibung und Untersuchung von Algorithmen, Computern, etc.
  - Teilgebiete: Formale Sprachen, Automatentheorie, Komplexitätstheorie etc.
- **Technische Informatik**
  - Funktioneller ***Aufbau von Computern***, Entwurf und Entwicklung von Rechnern, Geräten und Schaltungen
  - Teilgebiete: Rechnerarchitektur, VLSI-Entwurf etc.
- **Praktische Informatik**
  - Prinzipien und Techniken der Fundierung und Realisierung in Software (großer Programmsysteme!)
  - Teilgebiete: Softwaretechnik, Informationssysteme, Compilerbau, Betriebssysteme, Künstliche Intelligenz, Kommunikation/verteilte Systeme, Parallele Systeme, etc.

Gebiete der Informatik - 3

- **Angewandte Informatik**
  - ***Anwendung*** der Methoden der ***Kerninformatik*** in anderen Wissenschaften
    - Entwicklung ***spezieller*** Verfahren und Darstellungstechniken
  - Bindestrich-Informatik:
    - Wirtschafts-Informatik, Medizin-Informatik, Bio-Informatik, Rechts-Informatik
  - Grenzen zwischen ***Praktischer Informatik*** und ***Angewandter Informatik*** sind zum Teil fließend.
- **Das, was man unter Informatik versteht, kann man in solchen Definitionen jedoch nicht *endgültig* fassen. Schließlich ändert sich die Beschreibung der Definitionen einer Wissenschaft, wie in anderen Fällen auch, über die Zeit.**

Geschichte - 1

- **Altertum–Mittelalter:**
  - Verwendung des ***Abakus*** (Brett mit verschiebbaren Kugeln) als Hilfsmittel für die vier Grundrechenarten.
- **9. JH.:**
  - Der arabische Mathematiker und Astronom ***Ibn Musa Al-Chwarismi*** schreibt das Lehrbuch "Kitab al jabr w' almuqabala" ("Regeln der Wiedereinsetzung und Reduktion"). Das Wort "***Algorithmus***" geht auf seinen Namen zurück.
- **1547: Adam Riese (1492–1559)**
  - veröffentlicht ein Rechenbuch, in dem er die ***Rechengesetze*** des aus Indien stammenden ***Dezimalsystems*** (5. Jh.n. Chr.) beschreibt. Im 17. Jahrhundert setzt sich das Dezimalsystem in Europa durch.
- **Wilhelm Schickard (1592–1635)**
  - konstruiert für seinen Freund Kepler (1571–1630) eine ***Maschine***, die addieren, subtrahieren, multiplizieren und dividieren kann. Sie bleibt unbeachtet.
- **1641: Blaise Pascal (1623–1662)**
  - konstruiert eine Maschine, mit der man ***sechsstellige Zahlen*** addieren kann.

Geschichte - 2

- **1674: Gottfried Wilhelm Leibniz (1646–1716)**
  - konstruiert eine ***Rechenmaschine*** mit Staffelwalzen für die vier ***Grundrechenarten***. In diesem Zusammenhang befaßt er sich auch mit der binären Darstellung von Zahlen.
- **1774: Philipp Matthäus Hahn (1739–1790)**
  - entwickelte eine mechanische Rechenmaschine, die ***erstmals zuverlässig*** arbeitet.
- **Ab 1818:**
  - Rechenmaschinen nach dem Vorbild der Leibnizschen Maschine werden serienmäßig hergestellt und dabei ständig weiterentwickelt.
- **1838: Charles Babbage (1792–1871)**
  - plant eine Maschine, die "***Analytical Engine***", bei der die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene ***Lochkarten*** gesteuert wird.
- **1886: Hermann Hollerith (1860–1929)**
  - entwickelt in den USA elektrisch arbeitende ***Zählmaschinen für Lochkarten***, mit denen die statistischen Auswertungen der Volkszählungen vorgenommen werden.

Geschichte - 3

- **1934: Konrad Zuse (1910–1995)**
  - beginnt mit der Planung einer ***programmgesteuerten Rechenmaschine***. Sie verwendet das binäre Zahlensystem.
- **1937: Die mechanische Anlage *Z 1* von Zuse ist fertig.**
- **1941: Die elektromechanische Anlage Z 3 von Zuse ist fertig.**
  - Dies ist der ***erste funktionsfähige programmgesteuerte Rechenautomat***. Das Programm wurde mit ***Lochstreifen*** eingegeben. Die Anlage verfügt über 2000 Relais und eine Speicherkapazität von 64 Worten a 22 Bit. Multiplikationszeit: etwa 3 s.
- **1944: Howard H. Aiken (1900–1973)**
  - erstellt in Zusammenarbeit mit der Harvard-University und der Firma IBM die teilweise programmgesteuerte Rechenanlage MARK I. Additionszeit 1/3 s, Multiplikationszeit: 6 s.
- **1946: J. P. Eckert und J. W. Mauchly**
  - stellen die ENIAC (Electronic Numerical Integrator and Automatic Calculator) fertig. Dies ist der erste ***voll elektronische Rechner*** (18.000 Elektronenröhren). Multiplikationszeit: 3 ms.

Geschichte - 4

- **1946–1952:**
  - Auf der Grundlage der Ideen **John v. Neumanns** (1903–1957) (Einzelprozessor, Programm und Daten im gleichen Speicher; Von-Neumann-Rechner) und seiner Kollegen am Institute of Advanced Study at Princeton (H.H.Goldstine, A.W.Burks) werden weitere Computer in Universitätslabors entwickelt (“Pionierzeit”).
- **1949: M.V. Wilkes (University of Manchester)**
  - stellt mit der EDSAC (Electronic Delay Storage Automatic Calculator) den **ersten universellen Digitalrechner** (gespeichertes Programm) fertig.
- **Ab 1950:**
  - **Industrielle** Rechnerentwicklung und -produktion.
- **Bedeutung von Software: Softwaregeschichte**

  Algorithmen → Babylonier

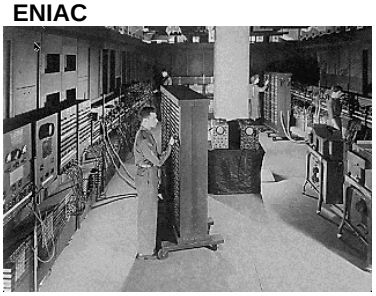
  Programmierung → Ada, Zuse

Historische Rechner

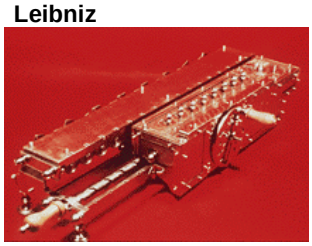
Geschichte der Informatik

Schickard

ENIAC



Leibniz



MARK1

Informelle Definition

Algorithmus

- **Ein Algorithmus ist ein Verfahren, welches**
  - in einem ***endlichen*** Text niedergelegt werden muß
  - ***effektiv*** ausführbar ist,
  - Elementaroperationen enthält, die durch die jeweilige Situation ***eindeutig*** bestimmt sind
  - ***Ein- und Ausgabe*** ermöglicht
  - durch eine (mechanisch oder elektronisch arbeitende) *Maschine* ***ausgeführt*** werden kann.
- **Anzahl und Ausführungszeit der Elementaroperationen sind beschränkt**
- **Ein Algorithmus (Programm) wird durch eine Maschine schrittweise ausgeführt**
  - die ausführende Instanz muß die Vorschrift ***interpretieren*** und ***korrekt*** ausführen
  - ein Algorithmus ***terminiert***, wenn er nach endlich vielen Schritten abbricht

Euklidischer Algorithmus - 1

Euklidischer Algorithmus

- Problem:
  - Man bestimme zu je zwei natürlichen Zahlen n und m den größten gemeinsamen Teiler ggt(n,m)

Algorithmus

Wiederhole:

❶ WENN n < m ist, DANN vertausche man n und m

❷ WENN m = 0 ist, DANN ist n der ggt(n,m) und man beende den Algorithmus

❸ WENN m≠ 0 ist, DANN bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

Euklidischer Algorithmus - 2

n = 6
m = 10

Ablaufverfolgung (Trace)

für Überprüfung von Programmen

	n	m
	6	10
Erster Durchlauf		
da 6 < 10, so setzt man n = 10 und m = 6	10	6
da 6 ≠ 0 ist, gehen zu Schritt 3	10	6
r = 4. Man erhält also n = 6 und m = 4	6	4
Zweiter Durchlauf		
Da 6 ≥ 4 ist, gehe zu Schritt 2	6	4
Da 4 ≠ 0 ist, gehe zu Schritt 3	6	4
r = 2. Man erhält n = 4 und m = 2	4	2
Dritter Durchlauf		
Da 4 ≥ 2 ist, gehe zu Schritt 2	4	2
Da 2 ≠ 0 ist, gehe zu Schritt 3	4	2
r = 0. Man erhält n = 2 und m = 0	2	0
Abbruch		
Da 2 ≥ 0 ist, gehe zu Schritt 2	2	0
Da m = 0 ist, Programmende	2	0

Ergebnis: ggt(10, 6) =2

Eigenschaften - 1

- **Abstraktion**
  - ein Algorithmus löst i. a. eine ***Klasse von Problemstellungen*** (z.B. Suchen eines Musters in einer Zeichenkette): versch. Ein-, Ausgabewerte
- **Finitheit**
  - statisch finit: ein Algorithmus besitzt eine ***endliche Länge***
  - dynamisch finit: während der Abarbeitung darf nur ***endlich viel Speicherplatz*** belegt werden
- **Terminierung**
  - terminierend: nach ***endlich vielen Schritten*** liegt ein Resultat vor
  - sonst ***nicht-terminierend*** (z.B. Steuerungsalgorithmen)
- **Determinismus**
  - deterministisch: zu jedem Zeitpunkt besteht ***höchstens eine*** Möglichkeit der Fortsetzung
  - nicht-deterministisch: an mindestens einer Stelle gibt es eine ***Wahlmöglichkeit*** für die Fortsetzung

Eigenschaften - 2

- **Determiniertheit**
  - determiniert: bei ***gleichen Eingaben*** und Startbedingungen wird das ***gleiche Ergebnis*** erzielt
  - nicht-determiniert: es werden ***unterschiedliche Ergebnisse*** erzielt (z.B. Anwendung von heuristischen Methoden, syst. Probieren)
- **Bemerkung**
  - ein terminierender, deterministischer Algorithmus ist immer determiniert
  - ein terminierender, nicht-deterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Terminierung - Nichtterminierung

- **Sei**
  - $power : N^+ \times N \rightarrow N$

$$power(a, b) = \begin{cases} 1 & \text{falls } b = 0 \\ a * power(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert*** im Definitionsbereich.

  - Wir weiten den Definitionsbereich aus:

$$power2 : Z \times Z \rightarrow Z$$

$$power2(a, b) = \begin{cases} 0 & \text{falls } a = 0 \text{ und } b > 0 \\ 1 & \text{falls } a \neq 0 \text{ und } b = 0 \\ a * power2(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert nicht*** für power2 (0,0) und power2 (a,b) mit $b < 0$ und $a \neq 0$)

Anmerkung zur Nichtterminierung

Während wir die Nichtterminierung bei der ***manuellen Auswertung*** leicht feststellen können, fällt dies auf dem Rechner sehr viel schwerer. Solange der Rechner ***vor sich hin*** rechnet, können wir den Unterschied zwischen einem ***nicht-terminierenden*** und einem ***sehr langwierigen Algorithmus*** nicht feststellen.

Fragen

- **Wie kann man aus einer Lösungsidee einen Algorithmus konstruieren?**
  - "schrittweise Programmentwicklung"
- **Wie kann man Algorithmen darstellen?**
  - "Flußdiagramme", Programme
- **Wie beweist man, daß ein Algorithmus tatsächlich das tut, was er tun soll?**
  - Verifikation: partielle Korrektheit
  - Termination
- **Wie "gut" ist ein Algorithmus?**
  - Speicherverbrauch, benötigte Zeit
  - Aufwandsabschätzungen

Typische Problemklassen

- **Sortieralgorithmen**
  - Ordnen von Elementen
- **Suchalgorithmen**
  - Auffinden von Elementen
- **Algorithmen zur Verarbeitung von Zeichenfolgen**
  - Mustererkennung, Verschlüsselung, Komprimierung
- **Geometrische Algorithmen**
  - z.B. Schnittmenge geometrischer Objekte
- **Algorithmen für Graphen**
  - Suchen im Graph, kürzester Weg
- **Mathematische Algorithmen**
  - Rechnen mit Polynomen und Matrizen
- **Viele Anwendungsalgorithmen:** Kontrolle, Steuerung, Simulation

Standard-Algorithmen der Informatik

Überblick

- **Neben der Entwicklung der Hardware (Rechner)**
  - wurden seit 1955 ***Programmiersprachen*** entwickelt, um Algorithmen zu formulieren, damit sie von einem Rechner ausgeführt werden können.
- **Rechner "realisieren" Algorithmen,**
  - durch ***schrittweise Abarbeitung*** von ***Programmen***, die in einer Programmiersprache geschrieben sind.
- **Um einen Rechner zu programmieren,**
  - muß die ***Syntax*** und ***Semantik*** der verwendeten Programmiersprache bekannt sein
- **In diesem Zusammenhang spricht man häufig auch von Software**
  - "***Software***" und "***Programm***" sind aber nicht dasselbe

Definition: Software

- **1. Definition: Software**
  - Informatik-Duden: Gesamtheit ***aller Programme***, die auf einer Rechenanlage eingesetzt werden können
    - ***Systemsoftware***: Programme die für den korrekten Ablauf einer Rechenanlage notwendig sind
    - ***Anwendungssoftware***: dient zur Lösung von Benutzerproblemen
- **2. Definition: Software**
  - IEEE Standrad Glossary of Software Engineering Terminoloy
    - "Computer ***programs***, ***procedures***, and possibly associated ***documentation*** and ***data*** pertaining to the operation of a computer system."

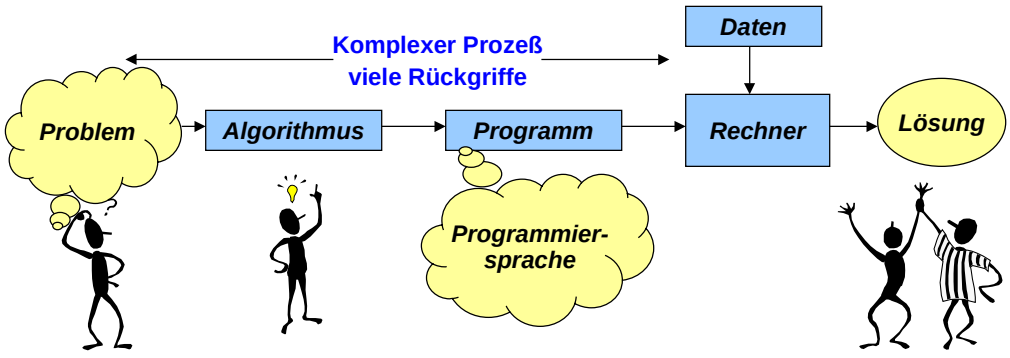
Definition: Programm

- **1. Definition: Programm in einer Programmiersprache**
  - Formulierung eines Algorithmus und der dazugehörigen Datenbereiche in einer Programmiersprache
    - ist ***exakt*** (formal) definiert
    - nimmt Bezug auf eine bestimmte Darstellung der ***Daten***
    - ist auf einer Rechenanlage ***ausführbar***

- **2. Definition: Software**
  - IEEE Standrad Glossary of Software Engineering Terminoloy
    - "A combination of ***computer instructions*** and ***data definitions*** that enable computer hardware to ***perform*** computational or control functions".

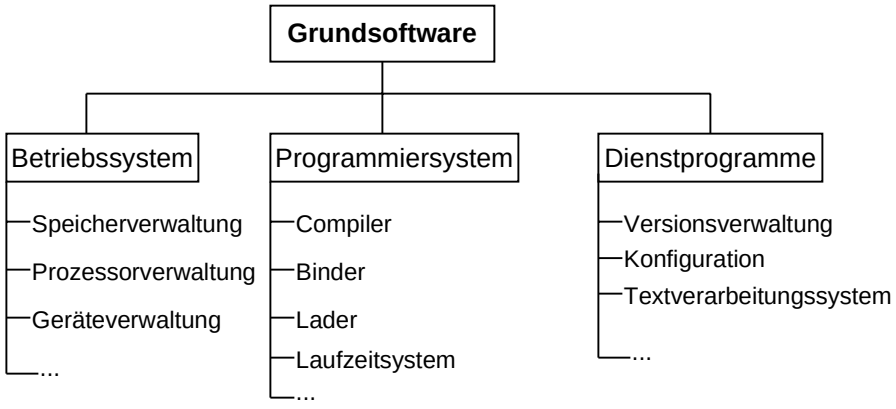
Programmentwicklung

- **Unter dem Begriff Programmieren versteht man**
  - das **Lösen von Problemen** unter Zuhilfenahme eines **Rechners**



- **Programmieren ≠ Software-Entwicklung**
- **Programmieren ist *ein Teil* der Software-Entwicklung**

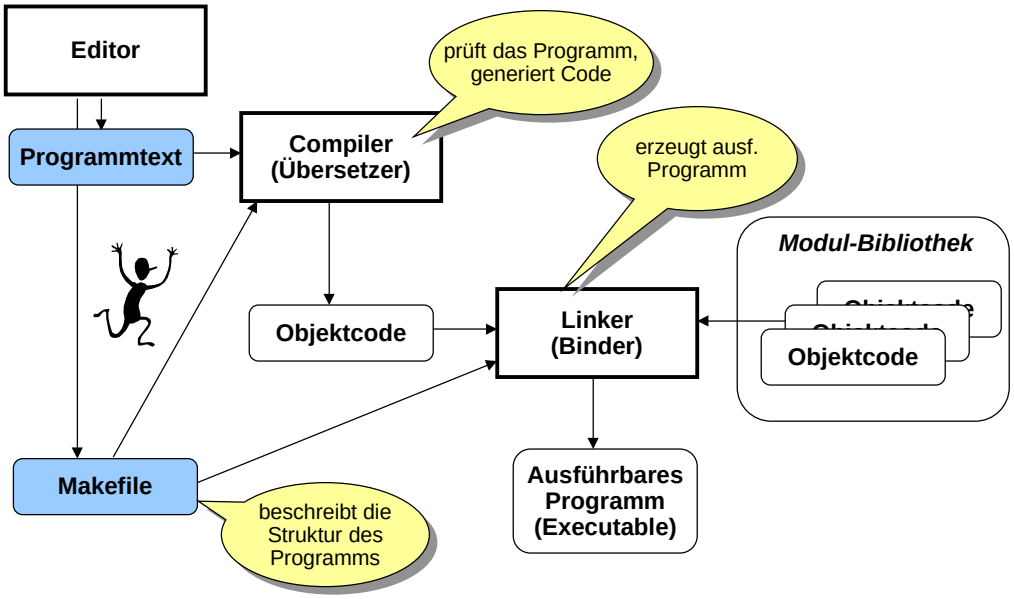
Grundsoftware(Systemsoftware)



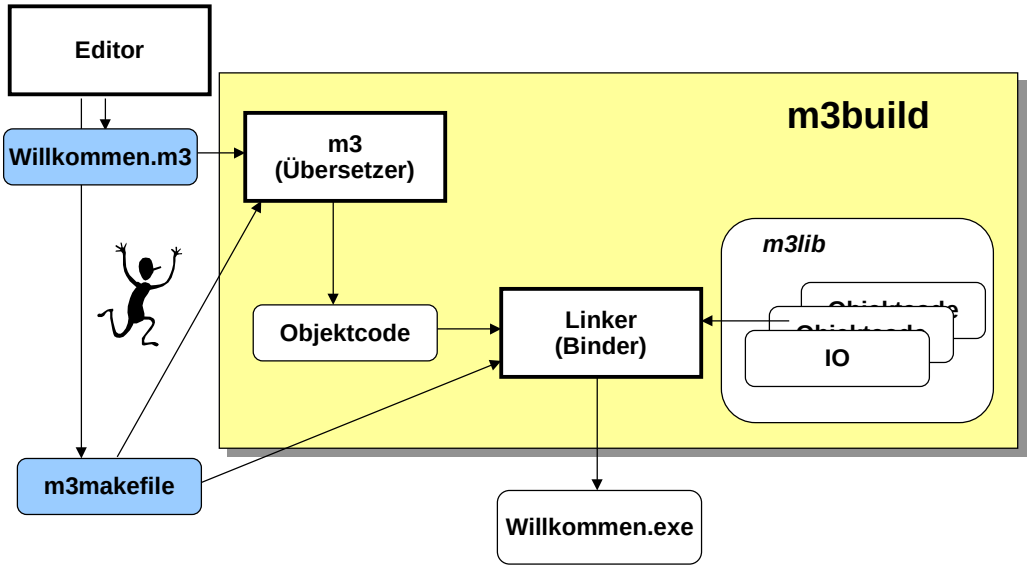
Das erste M3-Programm

```
MODULE Willkommen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 16.08.98
   Letzte Aenderung: 20.08.98
*)

IMPORT SIO;
BEGIN
  SIO.PutText("Willkommen zum Studium in Aachen.");
END Willkommen.
```

Vom Programmtext zum ausführbaren Programm



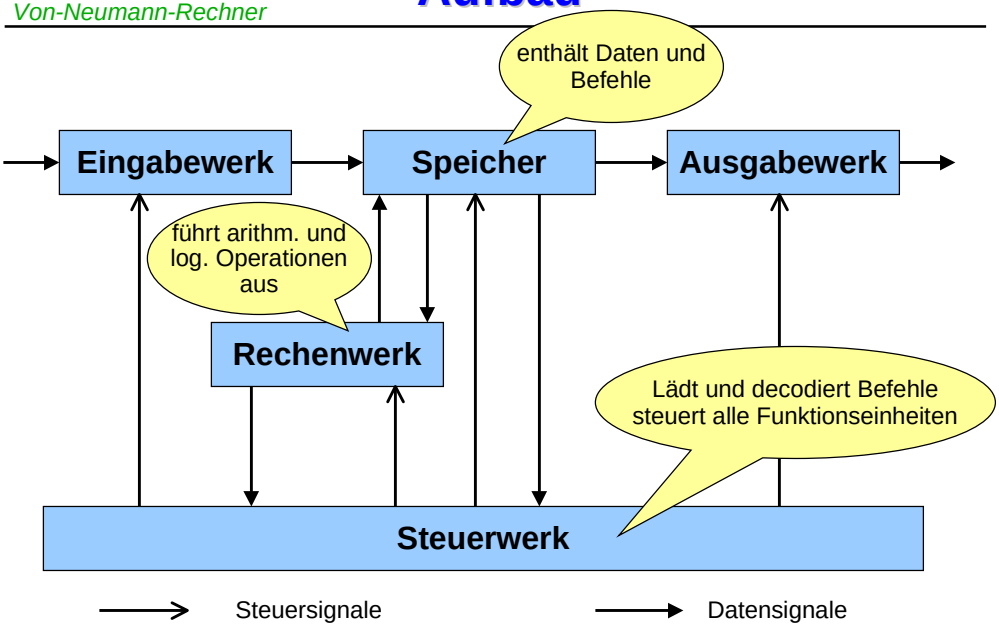
Edit-Compile-Link-Run bei Modula-3



Überblick

- **Definiert die wesentlichen Elemente eines Universalrechners**
  - Rechner soll nicht für eine ***bestimmte*** Problemklasse konstruiert sein
  - Zur Lösung des Problems muß ein ***Programm*** eingegeben und in den ***Speicher*** abgelegt werden
- **1946 von John von Neumann als Konzept für die EDVAC vorgeschlagen**
- **fast alle heutigen Rechner basieren darauf und sind Weiterentwicklungen davon**
  - Nicht-von-Neumann-Rechner sind Gegenstand der Forschung
- **diese Architektur prägt viele Programmiersprachen (imperative Programmiersprachen)**

Aufbau



Steuerwerk

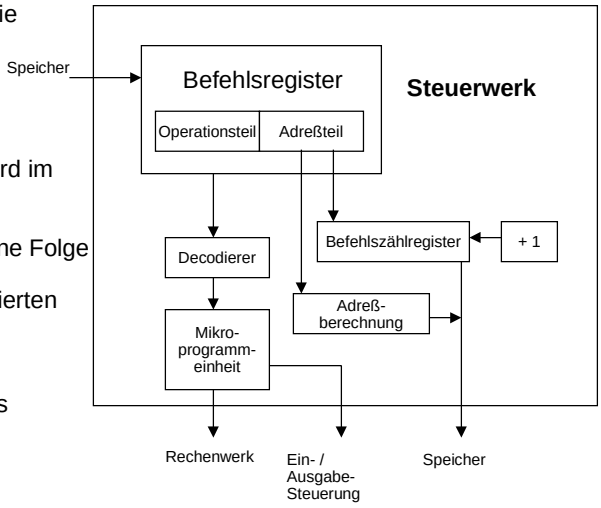
Lädt, decodiert und interpretiert die Befehle

Befehlsregister enthält den aktuellen Befehl

Der Operationsteil des Befehls wird im Decodierer entschlüsselt

Mikroprogrammeinheit erzeugt eine Folge von Signalen zur Ausführung des Befehls (abhängig von der decodierten Information)

Das Befehlzählregister speichert die Adresse des nächsten Befehls



Rechenwerk

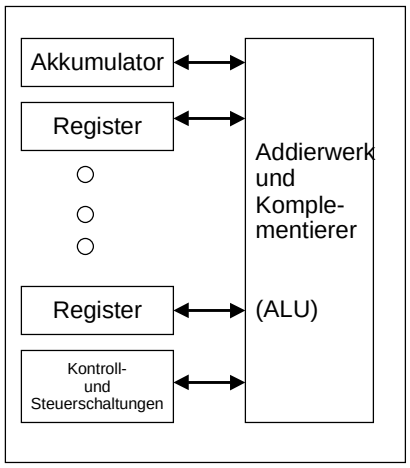
Führt arithmetische und logische Operationen durch (ALU)

erhält vom Steuerwerk die benötigten Operanden

wesentliche Einheiten sind Addierer und Komplementierer (damit können die Grundrechenarten durchgeführt werden)

implementiert Algorithmen für Multiplikation und Division

zusammen mit dem Steuerwerk nennt man es auch CPU



Prinzipien - 1

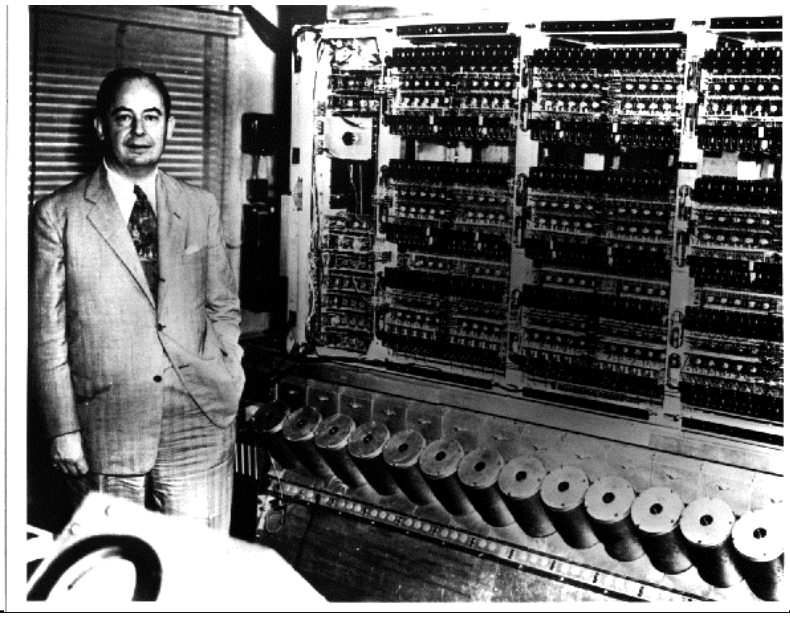
- Der Rechner besteht aus *Steuerwerk, Rechenwerk, Speicher, Eingabewerk* und Ausgabewerk.
- Die Struktur des von-Neumann-Rechners ist *unabhängig* von den zu bearbeitenden Problemen. Zur Lösung eines Problems muß von außen das *Programm* eingegeben und im Speicher abgelegt werden.
- Programme, Daten, Zwischen- und Endergebnisse werden in *demselben Speicher* abgelegt.
- Der Speicher ist in *gleichgroße Zellen* unterteilt, die fortlaufend durchnumeriert sind. Über die Nummer (*Adresse*) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden.
- Aufeinanderfolgende Befehle eines Programms werden in *aufeinanderfolgenden* Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
- Durch *Sprungbefehle* kann von der Bearbeitung der Befehle in der gespeicherten Reihenfolge abgewichen werden.

Prinzipien - 2

- Es gibt zumindest
  - *arithmetische* Befehle wie Addieren, Multiplizieren usw.;
  - *logische* Befehle wie Vergleiche, logisches Nicht, Und, Oder usw.;
  - *Transportbefehle*, z.B. vom Speicher zum Rechenwerk und für die Ein-/Ausgabe;
  - bedingte *Sprünge*.
  - Weitere Befehle wie Schieben, Unterbrechen, Warten usw. kommen hinzu.
- Alle Daten (Befehle, Adressen usw.) werden *binär codiert*. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (*Decodierung*).

Historische Rechner - J. v. Neumann

Von-Neumann-Rechner



Was haben wir gelernt

- **Einordnung der Informatik als Wissenschaft**
  - Aufgabe der Informatik
  - Einteilung der Informatik
- **Wo kommt die Informatik her**
  - Entwicklung der Rechenmaschinen
- **Was versteht man unter einem Algorithmus**
  - Eigenschaften von Algorithmen
- **Was versteht man unter den zentralen Begriffen**
  - Software
  - Programm
  - Programmieren
- **grober Aufbau eines von-Neumann-Rechners**

Glossar

- **Informatik:**
  - Begriff, Einteilung, Einordnung, Aufgaben, Geschichte
- **Algorithmus:**
  - Definition, Eigenschaften, Klassen von Algorithmen
- **Software**
- **Grundsoftware**
  - Betriebssystem, Programmiersystem, Dienst- und Hilfsprogramme (Utilities)
- **Programm**
- **Programmieren, Programmentwicklung**
- **Von-Neumann-Rechner:**
  - Speicher, Rechenwerk, Steuerwerk. Ein-/Ausgabeeinheit, Befehlsgruppen

Programmiersprachen-Grundlagen

- Syntax (und Semantik)
- Grammatik-Notationen
- Programmiersprachen: Allgemeines
- Warum arbeiten wir mit Modula-3

Programmiersprachen - Definition

- **Programmiersprachen sind *künstliche Sprachen* (keine natürlichen Sprachen), deren Syntax und Semantik genau festgelegt ist.**

- **Syntax:**
  - Die ***Syntax*** einer Programmiersprache S ist die Definition aller in S ***zulässigen Aussagen***, die in einer Sprache formuliert werden können (Wörter, hier Programme).

- **Semantik:**
  - Die ***Semantik*** einer Sprache S ist die Definition der den zulässigen Aussagen zugeordneten ***Bedeutungen***.
  - Syntaktische ***falsche*** Aussagen haben ***keine*** Semantik
  - Aber auch syntaktisch korrekte Aussagen haben nicht immer eine Semantik (z.B. ein Programm, in dem durch 0 dividiert wird)
  - statische, dynamische Semantik

- **Pragmatik**
  - menschliche, ökonomische

Beispiel: Syntax, Semantik

Natürliche Zahlen

- (ohne die Null) dargestellt im Dezimalsystem in arabischen Ziffern bilden eine einfache künstliche Sprache
- ***Syntax***:
  - jede Zahl ist eine Sequenz von Ziffern (0,1, .. , 9), wobei die erste Ziffer nicht 0 ist
- ***Semantik***:
  - der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist (***rekursive*** Definition)

Beispiel

- syntaktisch ***korrekt*** ist: 367 Semantik: 7+10* (36)
7+10*(6+10*(3))
7+10*(6+10*3)
- syntaktisch ***falsch*** ist: 007 keine Semantik

Alphabet

Alphabet

- Ein Alphabet (Zeichensatz) ist eine ***nichtleere endliche*** Menge von ***unterscheidbaren*** Zeichen ("Buchstaben“, Symbolen)

  $A = \{a_1, a_2, a_3, \ldots\}$ mit einer ***Ordnungsrelation*** $\leq$ $(a_1 \leq a_2 \leq a_3 \ldots.)$

- **Beispiel**:
  - das lateinische Alphabet (a, b, c,.. z)
  - der ASCII-Code (bestehend aus 128 Zeichen)
  - hier Latin-1-Zeichensatz ISO

- **Wort über einem Alphabet**
  - ***endliche Folge*** von Buchstaben, die auch ***leer*** sein kann (ε leere Wort)
  - A* bezeichnet die ***Menge aller Wörter*** über dem Alphabet A (inkl. dem leeren Wort)

Formale Sprache

- **Definition:**
  - Sei A ein Alphabet. Eine (formale) Sprache (über A) ist ***eine beliebige Teilmenge von A****.
- **Beispiele:**
  - $A_1 = \{ 0,1\}$, $A_1^* = \{\varepsilon , 0, 1, 01, 10, 10, 000, 100 , \ldots \}$
  - $L = \{0, 1, 10, 11, 100, 101, .. \} \subset A_1^*$, die Menge der Binärdarstellungen natürlicher Zahlen (mit Null, ohne führende Nullen)
  - $A_2 = \{(,), +, -, *, /, a\}$, $A_2^* = \{\varepsilon, (), (+-a), (a*a), \ldots \}$
  - die Sprache der korrekt geklammerten Ausdrücke $EXPR \subset A_2^*$: EXPR = { (((a))), (a+ a), (a -a)*a+ a /(a+ a) -a, .. }
- **Da solche Sprachen i.d.R. unendlich sind, benötigt man eine endliche Beschreibungsvorschrift**
  - ***Grammatik***, die die Sprache erzeugt
  - ***Automat***, der die Sprache erkennt

Grammatik - Definition - 1

- **Definition:**
  - Eine Grammatik G für eine Sprache L ist definiert durch
  - ein ***Viertupel (N, T, P, S)***
- **N: Menge der *Nichtterminalsymbole***
  - sind Zeichen oder Zeichenfolgen für syntaktische ***Abstraktionen***
  - Beispiel: <Statement>, <Expression>
  - kommen nicht in den Wörtern der Sprache vor
  - werden durch Anwendung der ***Produktionsregeln*** solange ersetzt, bis nur noch Terminalsymbole übrig sind
- **T: Menge der *Terminalsymbole***
  - sind Zeichen oder Zeichenfolgen des Alphabets, aus denen die Wörter der Sprache bestehen
  - Beispiel: ; (.. :=

Grammatik - Definition - 2

- **P: Menge von *Produktionsregeln***
  - definieren, wie aus bekannten Konstrukten ***neue Konstrukte*** geschaffen werden
  - Die Anwendung einer Regel bedeutet, daß in der bereits erzeugten Satzform der Teil, der der linken Seite der Regel entspricht, durch die rechte Seite ***ersetzt*** wird
  - **Beispiel**:
    - ... <Declaration> BEGIN <Statement>; <Ident> := <Number> END ...
    - ... <Declaration> BEGIN <Statement>; A := <Number> END ...

- **S: das *Startsymbol***
  - ist ein spezielles Nichtterminalsymbol, aus dem ***alle Wörter*** der Sprache mit Hilfe der Grammatik erzeugt werden
  - **Beispiel**: <Compilation>
- **Sprache:** Jedes durch Anwendung der Regeln erzeugbare Wort, ***das nur aus Terminalsymbolen besteht***, gehört zu der von der Grammatik erzeugten Sprache L(G)

Grammatik - Definition - 3

- **Es gilt:**
  - $N \cap T = \varnothing$
  - $V = N \cup T$ (Gesamtalphabet, Vokabular)
  - sei $p \in P$: $(\alpha \rightarrow \beta)$, $\alpha \in V^* N V^*$, $\beta \in V^*$

Terminale und Nichterminale sind verschieden

Auf der linken Seite einer Produktion steht wenigstens ein Nichtterminal

- **Ableitung**
  - Ableitungsprozeß ist eine Relation "$\Rightarrow$" auf V^*
  - Für $u,v, \beta \in V^*$ und $\alpha \in V^* N V^*$ gilt
    - $u\alpha v \Rightarrow u\beta v$ genau dann, wenn $(\alpha \rightarrow \beta) \in P$

- **Die von einer Grammatik *erzeugte Sprache* ist definiert als:**
  - $L(G) = \{ w \mid w \in T^*, S \overset{*}{\Rightarrow} w\}$

w ist herleitbar aus dem Startsymbol

zwei Grammatiken heißen ***äquivalent***, wenn sie dieselbe Sprache erzeugen

Typen von Produktionen

- **Je nach Gestalt der in P *zugelassenen Produktionen* definiert Chomsky 4 Typen von Grammatiken**

Produktion	Typ	Eigenschaften	CH-Typ
$(\alpha \rightarrow \beta)$	allgemein	$\alpha, \beta \in V^*$ beliebig	Typ-0
$(\alpha \rightarrow \varepsilon)$	ε -Produktion	$\alpha \in V^*, \; r = \varepsilon$	
$(\alpha \rightarrow \beta)$	beschränkt	$\alpha, \beta \in V^*, \quad 1 <= \lvert\alpha\rvert <= \lvert\beta\rvert$	Typ-1
$(uAv \rightarrow u\beta v)$	kontextsensitiv	$A \in N, \; u, v, \beta \in V^*, \; \beta \neq \varepsilon$	Typ-1
$(A \rightarrow \beta)$	kontextfrei	$A \in N, \quad \beta \in V^*$	Typ-2
$(A \rightarrow Bx)$	linkslinear	$A, B \in N, \quad x \in T$	Typ-3
$(A \rightarrow xB)$	rechtslinear		Typ-3
$(A \rightarrow x)$	terminierend	$A \in N, \quad x \in T$	

Chomsky-Grammatiken

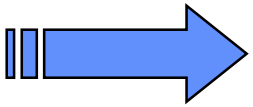
- **Die Chomsky-Grammatiken bilden eine Hierarchie**
  - d.h. die Menge der von Typ-n-Grammatiken erzeugten Sprachen umfaßt die Menge der Sprachen, die von Typ n+1 Grammatiken erzeugt werden
- **Typ-0-Grammatik**
  - Gestalt der Produktionen ist nicht eingeschränkt
  - Alle Sprachen, die überhaupt mit endlichen Regelsystemen erzeugt werden können
- **Typ-1 oder kontextsensitive Grammatik**
  - Produktionen sind beschränkt oder kontextsensitiv
- **Typ-2 oder *kontextfreie* Grammatik**
  - Produktionen sind kontextfrei (d.h. die linke Seite einer Produktion ist immer ein Nichtterminal)
- **Typ-3 oder reguläre Grammatik**
  - Produktionen sind terminierend, links- , rechtslinear

Kontextfreie Grammatik

- **Wichtigste Klasse zur formalen Beschreibung der Syntax von Programmiersprachen.**
- **Es ist möglich, Automaten zu bauen, die Wörter einer kontextfreien Sprache erkennen**
  - ***Wortproblem***
    - Kann für eine kf. Grammatik G und ein Wort $w \in T^*$ festgestellt werden, ob w von G erzeugt wird oder nicht.
  - ***Analyseproblem***
    - Gibt es einen Algorithmus, der zu einer kf. Grammatik G und einem Wort $w \in T^*$ die syntaktische Struktur von w bestimmt, oder aber feststellt, daß w nicht in L(G) liegt
    - ***Parser*** (Zerteilungsalgorithmus): Syntaxanalyse
- **Notationen zur Darstellung kontextfreier Grammatiken**
  - ***Syntaxdiagramme***
  - ***Extended Backus-Naur-Form* (EBNF)**

Grammatik - Beispiel

- **kf. Grammatik, die korrekt geklammerte arithmetische Ausdrücke mit Operatoren *, + erzeugt**
- G1 = ({E,T,F}, {(,), a, +, *}, P, E) mit
  - P = { ❶ E -> T,
    ❷ E -> E + T,
    ❸ T -> F,
    ❹ T -> T * F,
    ❺ F -> a,
    ❻ F -> (E) }
- **a * (a + a) ∈ L(G1)**
  - denn a*(a+a) läßt sich aus E folgendermaßen ableiten
  - dabei wurde immer das am weitesten links stehende Nichtterminal ersetzt (***Linksableitung***)



		Angewandte Regel
E	=> T	❶
	=> T * F	❹
	=> F * F	❸
	=> a * F	❺
	=> a * (E)	❻
	=> a * (E + T)	❷
	=> a * (T + T)	❶
	=> a * (F + T)	❸
	=> a * (a + T)	❺
	=> a * (a + F)	❸
	=> a * (a + a)	❺

Diskussion Beispiel - 1

- **Betrachtet man die Erzeugung von arithmetischen Ausdrücken genauer:**
  - Erzeugungsprozeß ist rückwärts betrachtet ein Prozeß der ***sukzessiven Zusammenfassung*** von Teilausdrücken: Reduktion
  - schließlich wird der gesamte Ausdruck auf das Startsymbol ***zurückgeführt***
- **Dabei wird z.B. folgendes beachtet**
  - Vorrang der Klammerstruktur
  - Vorrang der Multiplikation von der Addition
  - Zusammenfassen von links nach rechts bei gleichrangigen Operatoren
- **Beispiel zeigt**
  - daß die Syntax bereits auf grundlegende Eigenschaften der Semantik ***abgestimmt*** sein kann!

Diskussion Beispiel - 2

- **Folgende kf. Grammatik erzeugt dieselbe Sprache wie G1**

G2 = ({E}, {(,), a, +, *}, P, E) mit

P = { ❶ E -> E + E,
❷ E -> E * E,
❸ E -> (E),
❹ E -> a }

- **Bemerkung**
  - bei G2 fehlt die bei G1 festgestellte Abstimmung der Syntax, d.h. des Erzeugungsprozesses auf die Regeln der Auswertung arithmetischer Ausdrücke
  - G1 kann als Modell für die Definition von Ausdrücken in höheren Programmiersprachen angesehen werden (keine Auswertungs-reihenfolge, Prioritätsfestlegung).

Grammatik - Beispiel

- **Sei G = (N, T, P, S) mit**
  - N = {A, B}
  - T = {a, b, c, d}
  - P = { (A -> aBbc),
    (B -> aBb),
    (aBb -> d) }
  - S = A
  - G ist keine kontextfreie Grammatik, da die dritte Produktionsregel auf der linken Seite mehr als nur das Nichtterminalsymbol enthält.

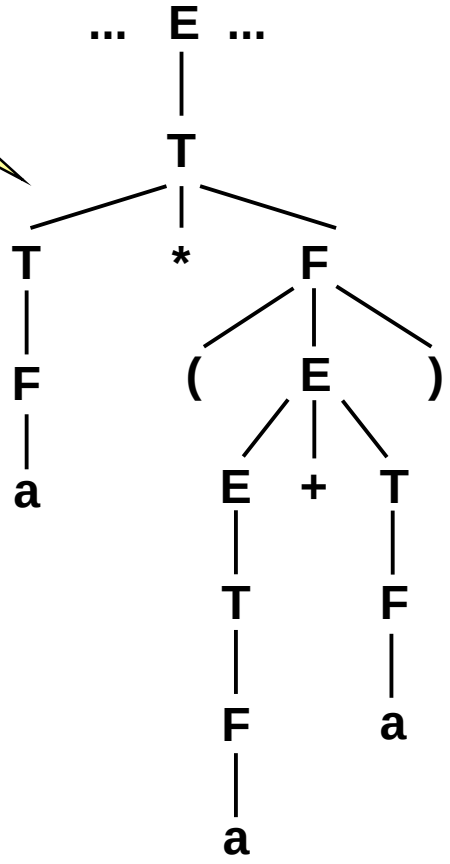
- **Ersetzt man in G die Produktionen P durch P', dann ist G' kontextfrei. Es gilt L(G) = L(G')**
  - P' = { (A -> Bc),
    (B -> aBb),
    (B -> d) }

Ableitungsbaum

- **Vaterknoten = linke Seite einer Regel**
- **Söhne = rechte Seite einer Regel**

**Mittel,
um die syntaktische
Struktur eines Wortes
darzustellen**

**Für Grammatik
von oben**



Syntaxdarstellung - EBNF - 1

EBNF

- **E**xtended **B**ackus-**N**aur-**F**orm
- ***Meta-Sprache*** zur Beschreibung der Syntax formaler Sprachen
- BNF erstmals benutzt zur Definition der Sprache ***Algol-60***

- ***Metasymbole*** von EBNF sind
  - = „definiert als“
  - (...|...) genau eine Alternative aus der Klammer muß stehen
  - [...] Inhalt der Klammer kann stehen oder nicht
  - { ... } Inhalt der Klammer kann n-fach stehen, n >= 0
  - . Ende der Produktion
  - Terminalsymbole werden in " " eingeschlossen
- verschiedene Varianten für EBNF
  “Programmieren“ von Sprachdefinitionen } ⇨ Übung

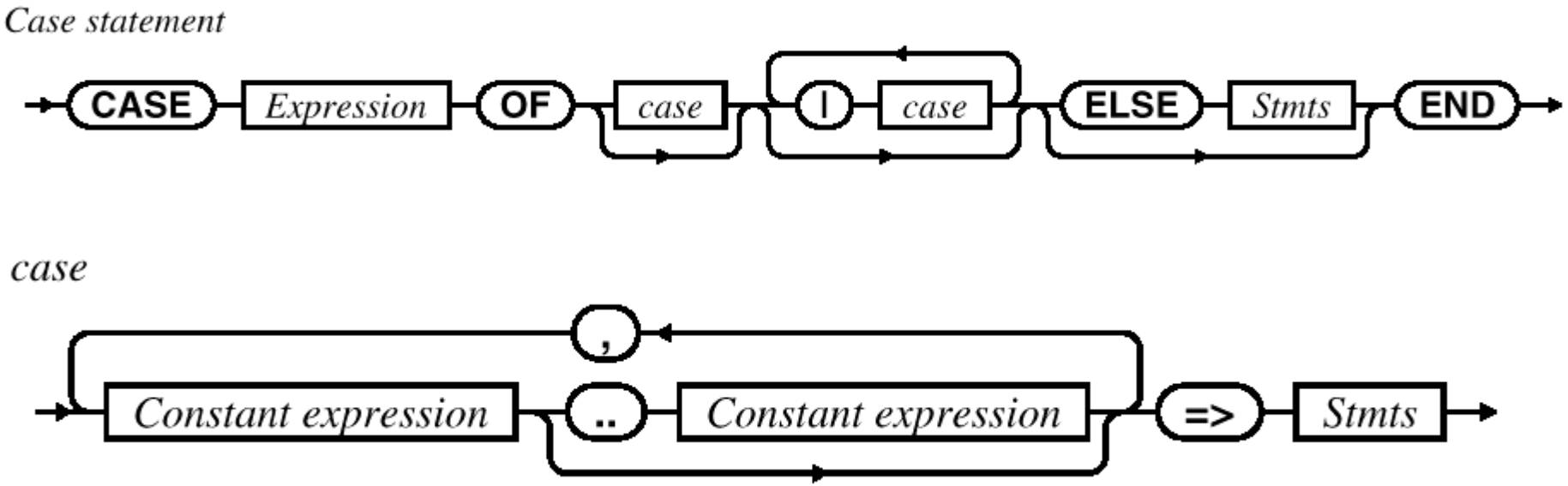
Syntaxdarstellung - EBNF - 2

CaseStatement = „**CASE**“ Expression „**OF**“
[Case] { „**|**“ Case }
[„**ELSE**“ Stmts] „**END**“ .

```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 |'/'  => resultat := a / b;
 ELSE (* Fehler *)
END;
```

Syntaxdiagramme - Beispiel

- **Syntaxdiagramme**
  - beschreiben Produktionen ***grafisch***
  - Nichterminalsymbole sind Rechtecke
  - Terminalsymbole sind Langrunde



Syntaxebenen

- **Lexikalische Syntax**
- **kontextfreie Syntax**

} **Vgl. Modula-Syntax im Anhang**

- **kontextsensitive Syntax (umgangssprachlich)**

```
MODULE WillkommenInAachen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss *)
IMPORT SIO;
VAR satz :TEXT;
BEGIN
  satz := "Willkommen zum Studium in Aachen!";
  SIO.Nl();
  SIO.PutText("---------------------------------");
  SIO.Nl();
  SIO.PutText(satz);
  SIO.Nl();
  SIO.PutText("---------------------------------");
  SIO.Nl();
END WillkommenInAachen.
```

Lexikalische Einheiten

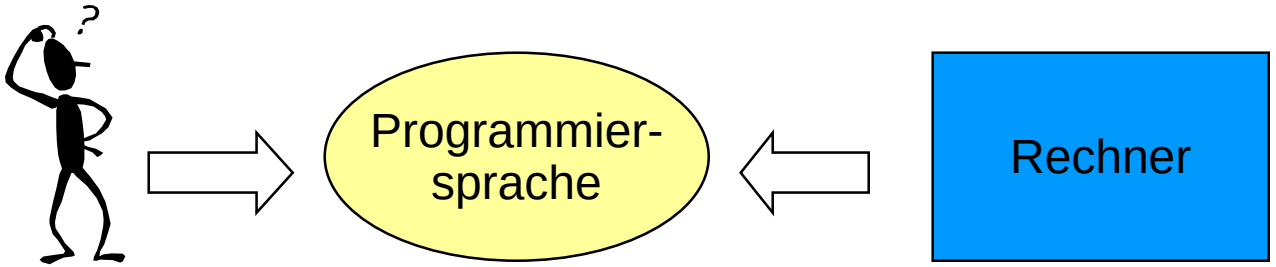
- Bezeichner
- Begrenzer
- Wortsymbole
- Literale
- Kommentare

} Eventl. Trennzeichen zwischen lexikalischen Einheiten

```
MODULE WillkommenInAachen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss *)
IMPORT SIO;
VAR satz :TEXT;
BEGIN
  satz := "Willkommen zum Studium in Aachen!";
  SIO.Nl();
  SIO.PutText("--------------------------------");
  SIO.Nl();
  SIO.PutText(satz);
  SIO.Nl();
  SIO.PutText("--------------------------------");
  SIO.Nl();
END WillkommenInAachen.
```

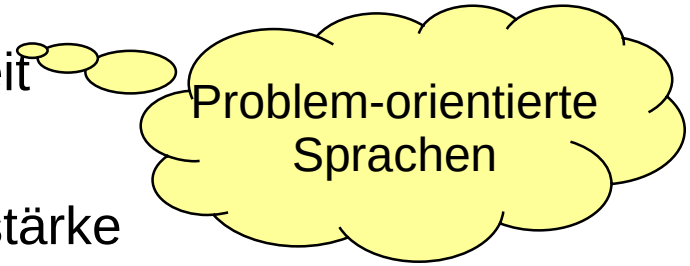
Rolle

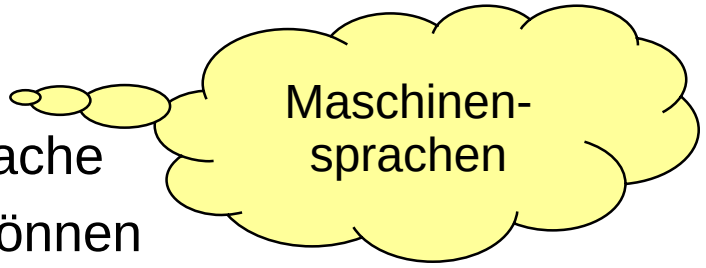
- **Die Programmiersprache bildet die *Schnittstelle* zwischen Mensch und Rechner**



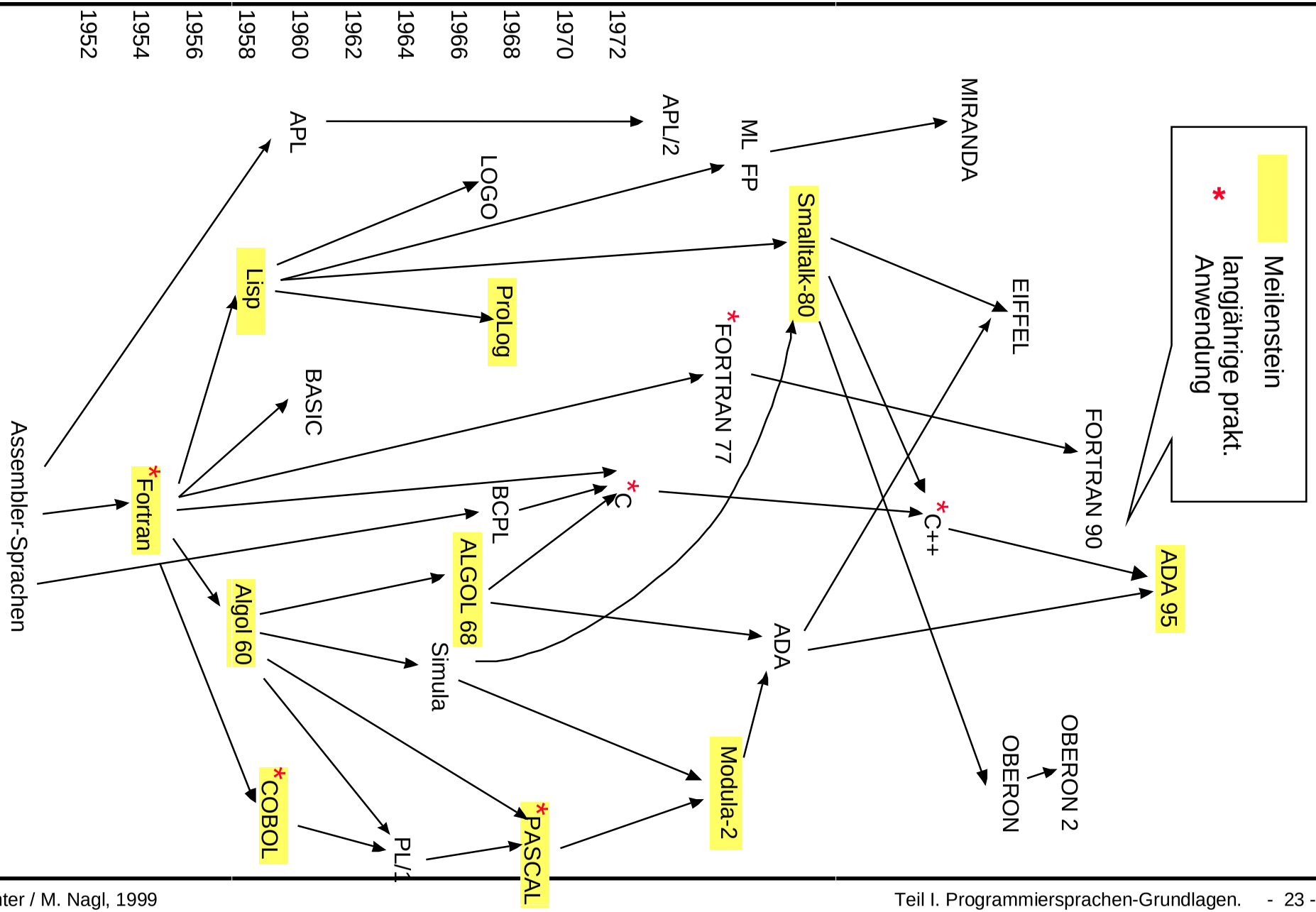
Beide haben unterschiedliche Anforderungen

- **Mensch**
  - Erlernbarkeit
  - Lesbarkeit
  - Ausdrucksstärke
- **Rechner**
  - einfaches Übersetzen in Maschinensprache
  - effizienter Code soll generiert werden können





Entwicklung



Werkstoff

- **Programmiersprachen sind der *Werkstoff* des Informatikers**
- **Das, was wir erzeugen (Programme)**
  - ist ***immateriell***
  - wir bauen es aus dem Werkstoff "Programmiersprache"
- **Ein Informatiker**
  - sollte die ***Qualität*** und ***Eignung*** verschiedener Werkstoffe (Programmiersprachen) kennen
  - Welche Sprache ist wofür geeignet?
- **Analogie!**
  - Ein Bauingenieur verwendet andere Werkstoffe, wenn ein
    - Hochhaus (Stahl, Beton etc.)
    - oder ein Einfamilienhaus (Ziegelsteine, Holz, Beton, etc.)
  - gebaut werden soll

Die Programmiersprache Modula-3

- **Modula-3**
  - erlaubt Programmierkonzepte ***elegant*** zu formulieren
  - zeigt die zentralen Konzepte der ***imperativen*** und ***objektorientierten*** Programmierung
  - ist ***leicht*** erlernbar
  - ist im Sinne der ***software-technischen Qualität*** von Programmen entwickelt worden
  - das fördert einen "***guten***" Programmierstil
  - die erlernten Konzepte werden Sie später in anderen Sprachen in anderer Form wiederfinden
- **Wichtig ist (im Sinne der VL)**
  - nicht die Programmiersprache Modula-3 selbst
  - sondern die ***Programmierkonzepte***

Modula-3

As Sam Harbison writes in his book on Modula-3,

Modula-3 is a member of the ***Pascal*** family of languages. Designed in the late 1980s at Digital Equipment Corporation and Olivetti, Modula-3 corrects many of the ***deficiencies*** of Pascal and Modula-2 for ***practical software engineering***. In particular, Modula-3 keeps the ***simplicity*** of type safety of the earlier languages, while providing ***new facilities*** for exception handling, concurrency, object-oriented programming, and automatic garbage collection. Modula-3 is both a practical implementation language for large software projects and an ***excellent teaching language***.

- **Modula = Modular Language**
  - Modul ist ein wichtiges Sprach- und Programmierkonzept
- **Modula-3 ist**
  - eine ***imperative*** (prozedurale) Programmiersprache
  - eine ***strukturierte*** Programmiersprache
  - eine objektorientierte Programmiersprache

Aufbau von Modula-3 Programmen

Modula-3 Programm

- besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***

Was ist ein Modul?

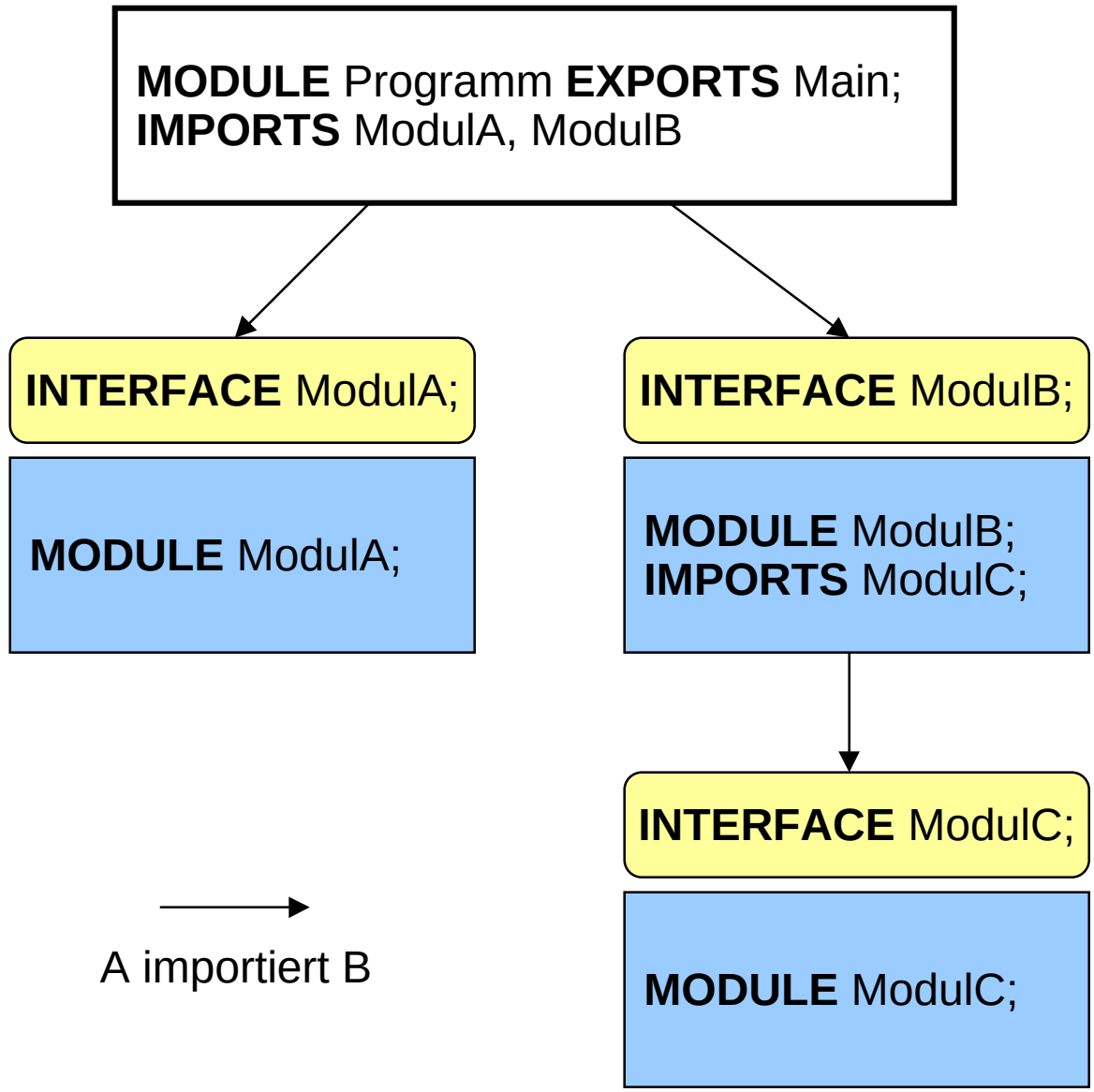
- ein Modul ist ein Programmteil, das ***sinnvoll*** zusammengehörende Elemente enthält
- ein Modul besteht (bis auf das Hauptmodul) aus
  - ***Exportschnittstelle*** (Interface)
    - definiert, was ein Modul an Diensten anbietet
  - ***Implementierung*** (Rumpf)
    - enthält die Implementierung der exportierten Elemente
    - versteckt die Implementierung

Hauptmodul

- exportiert die vordefinierte ***leere*** Schnittstelle `Main`

Modul-Hierarchie

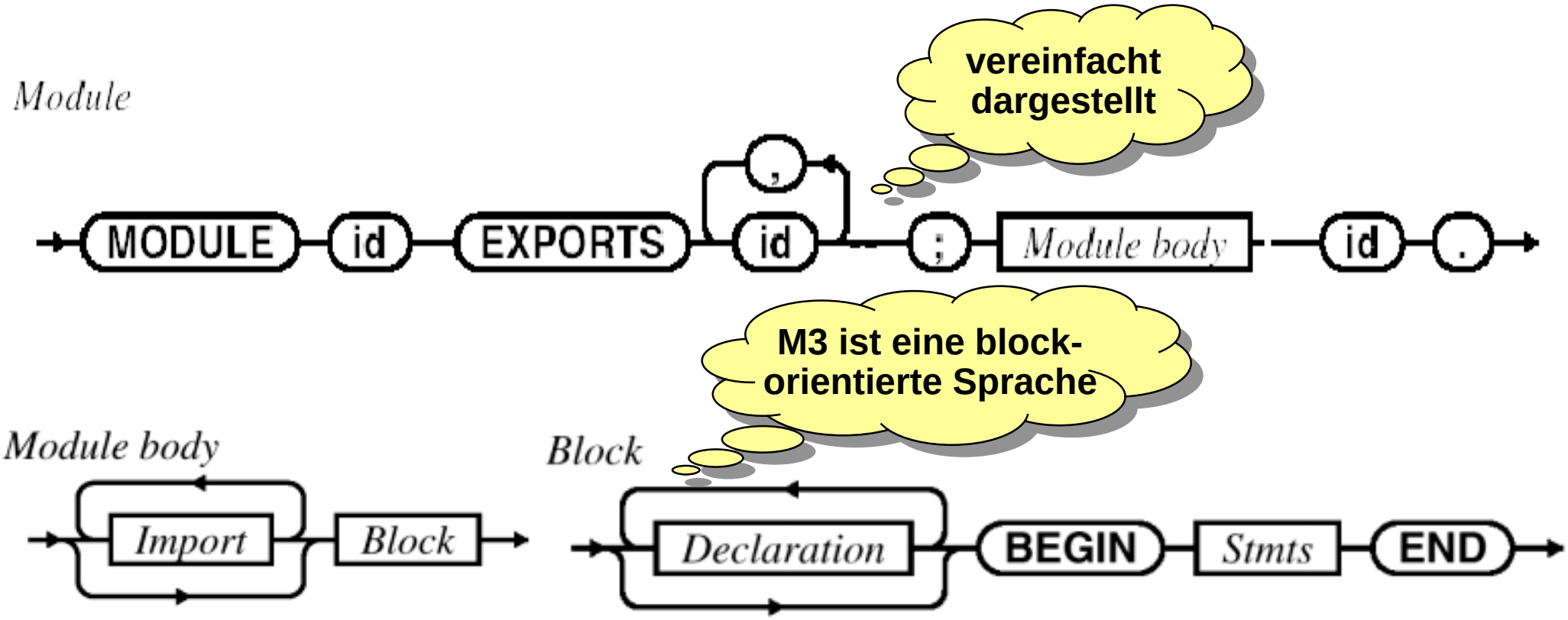
- ein Modul kann Elemente anderer Module ***benutzen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar



Aufbau eines Moduls

- **Ein Modul**
  - ***exportiert*** Elemente und ***importiert*** andere Module
  - enthält einen ***Block***

- **Ein Block**
  - enthält ***Deklarationen*** und ***Anweisungen***



Was haben wir gelernt

- Wie wissen was Programmiersprachen sind
- Wir kennen den Begriff "Formale Sprache"
- Wir haben gesehen, daß eine Grammatik eine Sprache erzeugt
- Wir kennen EBNF und Syntaxdiagramme
- Wir sehen Programmiersprachen als Werkstoff des Informatikers

Glossar

- **Programmiersprache**
- **Syntax, Syntaxebenen**
- **Semantik**
- **Alphabet**
- **Formale Sprache**
- **Grammatik**
  - Chomsky-Grammatik
  - Kontextfreie Grammatik
- **Syntaxdarstellung: EBNF, Syntaxdiagramm**
- **Ableitungsbaum, Syntaxbaum**
- **Lexikalische Einheiten**
- **Modul: Export, Import, Rumpf**

Programmiersprachen-Grundlagen

- Syntax (und Semantik)
- Grammatik-Notationen
- Programmiersprachen: Allgemeines
- Warum arbeiten wir mit Modula-3

Syntax und Semantik

Programmiersprachen - Definition

- **Programmiersprachen sind *künstliche Sprachen* (keine natürlichen Sprachen), deren Syntax und Semantik genau festgelegt ist.**
- **Syntax:**
  - Die ***Syntax*** einer Programmiersprache S ist die Definition aller in S ***zulässigen Aussagen***, die in einer Sprache formuliert werden können (Wörter, hier Programme).
- **Semantik:**
  - Die ***Semantik*** einer Sprache S ist die Definition der den zulässigen Aussagen zugeordneten ***Bedeutungen***.
  - Syntaktische ***falsche*** Aussagen haben ***keine*** Semantik
  - Aber auch syntaktisch korrekte Aussagen haben nicht immer eine Semantik (z.B. ein Programm, in dem durch 0 dividiert wird)
  - statische, dynamische Semantik
- **Pragmatik**
  - menschliche, ökonomische

Beispiel: Syntax, Semantik

- **Natürliche Zahlen**
  - (ohne die Null) dargestellt im Dezimalsystem in arabischen Ziffern bilden eine einfache künstliche Sprache
  - ***Syntax***:
    - jede Zahl ist eine Sequenz von Ziffern (0,1, .. , 9), wobei die erste Ziffer nicht 0 ist
  - ***Semantik***:
    - der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist (***rekursive*** Definition)
- **Beispiel**
  - syntaktisch ***korrekt*** ist: 367 Semantik: 7+10* (36)
7+10*(6+10*(3))
7+10*(6+10*3)
  - syntaktisch ***falsch*** ist: 007 keine Semantik

Alphabet

- **Alphabet**
  - Ein Alphabet (Zeichensatz) ist eine ***nichtleere endliche*** Menge von ***unterscheidbaren*** Zeichen ("Buchstaben", Symbolen)

    A = {a_1, a_2, a_3, …} mit einer ***Ordnungsrelation*** ≤ ($a_1 \le a_2 \le a_3$ ….)
  - **Beispiel**:
    - das lateinische Alphabet (a, b, c,.. z)
    - der ASCII-Code (bestehend aus 128 Zeichen)
    - hier Latin-1-Zeichensatz ISO
  - **Wort über einem Alphabet**
    - ***endliche Folge*** von Buchstaben, die auch ***leer*** sein kann (ε leere Wort)
    - A* bezeichnet die ***Menge aller Wörter*** über dem Alphabet A (inkl. dem leeren Wort)

Formale Sprache

- **Definition:**
  - Sei A ein Alphabet. Eine (formale) Sprache (über A) ist ***eine beliebige Teilmenge von A****.
- **Beispiele:**
  - $A_1 = \{ 0,1\}$, $A_1^* = \{\varepsilon , 0, 1, 01, 10, 10, 000, 100 , \dots \}$
  - $L = \{0, 1, 10, 11, 100, 101, .. \} \subset A_1^*$, die Menge der Binärdarstellungen natürlicher Zahlen (mit Null, ohne führende Nullen)
  - $A_2 = \{(,), +, -, *, /, a\}$, $A_2^* = \{\varepsilon, (), (+-a), (a*a), \dots \}$
  - die Sprache der korrekt geklammerten Ausdrücke $EXPR \subset A_2^*$: EXPR = { (((a))), (a+ a), (a -a)*a+ a /(a+ a) -a, .. }
- **Da solche Sprachen i.d.R. unendlich sind, benötigt man eine endliche Beschreibungsvorschrift**
  - ***Grammatik***, die die Sprache erzeugt
  - ***Automat***, der die Sprache erkennt

Grammatik - Definition - 1

- **Definition:**
  - Eine Grammatik G für eine Sprache L ist definiert durch
  - ein ***Viertupel (N, T, P, S)***
- **N: Menge der *Nichtterminalsymbole***
  - sind Zeichen oder Zeichenfolgen für syntaktische ***Abstraktionen***
  - Beispiel: <Statement>, <Expression>
  - kommen nicht in den Wörtern der Sprache vor
  - werden durch Anwendung der ***Produktionsregeln*** solange ersetzt, bis nur noch Terminalsymbole übrig sind
- **T: Menge der *Terminalsymbole***
  - sind Zeichen oder Zeichenfolgen des Alphabets, aus denen die Wörter der Sprache bestehen
  - Beispiel: ; (.. :=

Grammatik - Definition - 2

- **P: Menge von *Produktionsregeln***
  - definieren, wie aus bekannten Konstrukten ***neue Konstrukte*** geschaffen werden
  - Die Anwendung einer Regel bedeutet, daß in der bereits erzeugten Satzform der Teil, der der linken Seite der Regel entspricht, durch die rechte Seite ***ersetzt*** wird
  - **Beispiel**:
    - ... <Declaration> BEGIN <Statement>; <Ident> := <Number> END ...
    - ... <Declaration> BEGIN <Statement>; A := <Number> END ...

⇩

- **S: das *Startsymbol***
  - ist ein spezielles Nichtterminalsymbol, aus dem ***alle Wörter*** der Sprache mit Hilfe der Grammatik erzeugt werden
  - **Beispiel**: <Compilation>
- **Sprache:** Jedes durch Anwendung der Regeln erzeugbare Wort, ***das nur aus Terminalsymbolen besteht***, gehört zu der von der Grammatik erzeugten Sprache L(G)

Grammatik - Definition - 3

- **Es gilt:**
  - $N \cap T = \varnothing$
  - $V = N \cup T$ (Gesamtalphabet, Vokabular)
  - sei $p \in P$: $(\alpha \rightarrow \beta)$, $\alpha \in V^* N V^*$, $\beta \in V^*$

Terminale und Nichterminale sind verschieden

Auf der linken Seite einer Produktion steht wenigstens ein Nichtterminal

- **Ableitung**
  - Ableitungsprozeß ist eine Relation "$\Rightarrow$" auf V^*
  - Für $u,v, \beta \in V^*$ und $\alpha \in V^* N V^*$ gilt
    - $u\alpha v \Rightarrow u\beta v$ genau dann, wenn $(\alpha \rightarrow \beta) \in P$
- **Die von einer Grammatik *erzeugte Sprache* ist definiert als:**
  - $L(G) = \{ w \mid w \in T^*, S \overset{*}{\Rightarrow} w\}$

w ist herleitbar aus dem Startsymbol

zwei Grammatiken heißen ***äquivalent***, wenn sie dieselbe Sprache erzeugen

Typen von Produktionen

- **Je nach Gestalt der in P *zugelassenen Produktionen* definiert Chomsky 4 Typen von Grammatiken**

Produktion	Typ	Eigenschaften	CH-Typ
(α -> β)	allgemein	$\alpha, \beta \in V^*$ beliebig	Typ-0
(α -> ε)	ε -Produktion	$\alpha \in V^*$, r = ε	
(α -> β)	beschränkt	$\alpha, \beta \in V^*$, 1 <= $\lvert\alpha\rvert$ <= $\lvert\beta\rvert$	Typ-1
(uAv -> uβv)	kontextsensitiv	A $\in$ N, u, v, $\beta \in V^*$, $\beta \neq \varepsilon$	Typ-1
(A -> β)	kontextfrei	A $\in$ N, $\beta \in V^*$	Typ-2
(A -> Bx)	linkslinear	A, B $\in$ N, x $\in$ T	Typ-3
(A -> xB)	rechtslinear		Typ-3
(A -> x)	terminierend	A $\in$ N, x $\in$ T	

Chomsky-Grammatiken

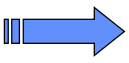
- **Die Chomsky-Grammatiken bilden eine Hierarchie**
  - d.h. die Menge der von Typ-n-Grammatiken erzeugten Sprachen umfaßt die Menge der Sprachen, die von Typ n+1 Grammatiken erzeugt werden
- **Typ-0-Grammatik**
  - Gestalt der Produktionen ist nicht eingeschränkt
  - Alle Sprachen, die überhaupt mit endlichen Regelsystemen erzeugt werden können
- **Typ-1 oder kontextsensitive Grammatik**
  - Produktionen sind beschränkt oder kontextsensitiv
- **Typ-2 oder *kontextfreie* Grammatik**
  - Produktionen sind kontextfrei (d.h. die linke Seite einer Produktion ist immer ein Nichtterminal)
- **Typ-3 oder reguläre Grammatik**
  - Produktionen sind terminierend, links- , rechtslinear

Kontextfreie Grammatik

- **Wichtigste Klasse zur formalen Beschreibung der Syntax von Programmiersprachen.**
- **Es ist möglich, Automaten zu bauen, die Wörter einer kontextfreien Sprache erkennen**
  - ***Wortproblem***
    - Kann für eine kf. Grammatik G und ein Wort w ∈ T* festgestellt werden, ob w von G erzeugt wird oder nicht.
  - ***Analyseproblem***
    - Gibt es einen Algorithmus, der zu einer kf. Grammatik G und einem Wort w ∈ T* die syntaktische Struktur von w bestimmt, oder aber feststellt, daß w nicht in L(G) liegt
    - ***Parser*** (Zerteilungsalgorithmus): Syntaxanalyse
- **Notationen zur Darstellung kontextfreier Grammatiken**
  - ***Syntaxdiagramme***
  - ***Extended Backus-Naur-Form* (EBNF)**

Grammatik - Beispiel

- **kf. Grammatik, die korrekt geklammerte arithmetische Ausdrücke mit Operatoren *, + erzeugt**
- G1 = ({E,T,F}, {(,), a, +, *}, P, E) mit
  - P = { ❶ E -> T,
    ❷ E -> E + T,
    ❸ T -> F,
    ❹ T -> T * F,
    ❺ F -> a,
    ❻ F -> (E) }
- **a * (a + a) ∈ L(G1)**
  - denn a*(a+a) läßt sich aus E folgendermaßen ableiten
  - dabei wurde immer das am weitesten links stehende Nichtterminal ersetzt (***Linksableitung***)



		Angewandte Regel
E	=> T	❶
	=> T * F	❹
	=> F * F	❸
	=> a * F	❺
	=> a * (E)	❻
	=> a * (E + T)	❷
	=> a * (T + T)	❶
	=> a * (F + T)	❸
	=> a * (a + T)	❺
	=> a * (a + F)	❸
	=> a * (a + a)	❺

Diskussion Beispiel - 1

- **Betrachtet man die Erzeugung von arithmetischen Ausdrücken genauer:**
  - Erzeugungsprozeß ist rückwärts betrachtet ein Prozeß der ***sukzessiven Zusammenfassung*** von Teilausdrücken: Reduktion
  - schließlich wird der gesamte Ausdruck auf das Startsymbol ***zurückgeführt***
- **Dabei wird z.B. folgendes beachtet**
  - Vorrang der Klammerstruktur
  - Vorrang der Multiplikation von der Addition
  - Zusammenfassen von links nach rechts bei gleichrangigen Operatoren
- **Beispiel zeigt**
  - daß die Syntax bereits auf grundlegende Eigenschaften der Semantik ***abgestimmt*** sein kann!

Diskussion Beispiel - 2

- **Folgende kf. Grammatik erzeugt dieselbe Sprache wie G1**

G2 = ({E}, {(,), a, +, *}, P, E) mit

P = { ❶ E -> E + E,
❷ E -> E * E,
❸ E -> (E),
❹ E -> a }

- **Bemerkung**
  - bei G2 fehlt die bei G1 festgestellte Abstimmung der Syntax, d.h. des Erzeugungsprozesses auf die Regeln der Auswertung arithmetischer Ausdrücke
  - G1 kann als Modell für die Definition von Ausdrücken in höheren Programmiersprachen angesehen werden (keine Auswertungs-reihenfolge, Prioritätsfestlegung).

Grammatik - Beispiel

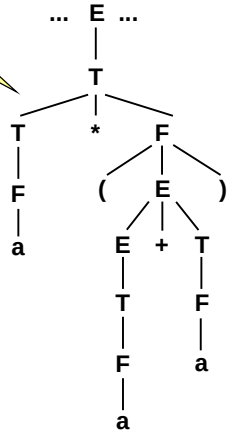
- **Sei G = (N, T, P, S) mit**
  - N = {A, B}
  - T = {a, b, c, d}
  - P = { (A -> aBbc),
    (B -> aBb),
    (aBb -> d) }
  - S = A
  - G ist keine kontextfreie Grammatik, da die dritte Produktionsregel auf der linken Seite mehr als nur das Nichtterminalsymbol enthält.
- **Ersetzt man in G die Produktionen P durch P', dann ist G' kontextfrei. Es gilt L(G) = L(G')**
  - P' = { (A -> Bc),
    (B -> aBb),
    (B -> d) }

Ableitungsbaum

- **Vaterknoten = linke Seite einer Regel**
- **Söhne = rechte Seite einer Regel**

Mittel, um die syntaktische Struktur eines Wortes darzustellen

Für Grammatik von oben



Syntaxdarstellung - EBNF - 1

- **EBNF**
  - **E**xtended **B**ackus-**N**aur-**F**orm
  - ***Meta-Sprache*** zur Beschreibung der Syntax formaler Sprachen
  - BNF erstmals benutzt zur Definition der Sprache ***Algol-60***
  - ***Metasymbole*** von EBNF sind
    - = „definiert als“
    - (...|...) genau eine Alternative aus der Klammer muß stehen
    - [...] Inhalt der Klammer kann stehen oder nicht
    - { ... } Inhalt der Klammer kann n-fach stehen, n >= 0
    - . Ende der Produktion
    - Terminalsymbole werden in " " eingeschlossen
  - verschiedene Varianten für EBNF
“Programmieren“ von Sprachdefinitionen } ⇨ Übung

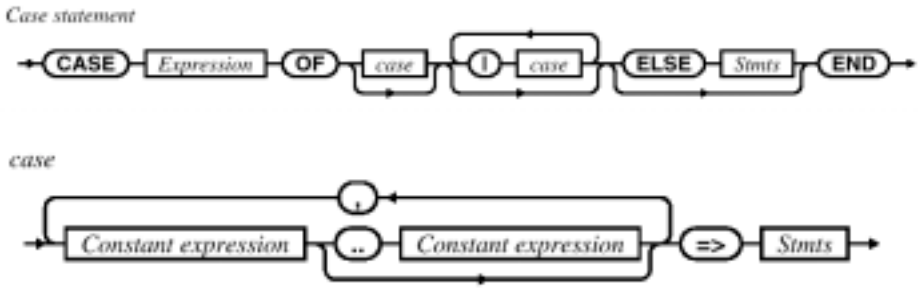
Syntaxdarstellung - EBNF - 2

CaseStatement = „**CASE**“ Expression „**OF**“
[Case] { „**|**“ Case }
[„**ELSE**“ Stmts] „**END**“ .

```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 |'/'  => resultat := a / b;
 ELSE (* Fehler *)
END;
```

Syntaxdiagramme - Beispiel

- **Syntaxdiagramme**
  - beschreiben Produktionen ***grafisch***
  - Nichterminalsymbole sind Rechtecke
  - Terminalsymbole sind Langrunde



Syntaxebenen

- **Lexikalische Syntax**
- **kontextfreie Syntax**

} **Vgl. Modula-Syntax im Anhang**

- **kontextsensitive Syntax (umgangssprachlich)**

```
MODULE WillkommenInAachen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss *)
IMPORT SIO;
VAR satz :TEXT;
BEGIN
  satz := "Willkommen zum Studium in Aachen!";
  SIO.Nl();
  SIO.PutText("--------------------------------");
  SIO.Nl();
  SIO.PutText(satz);
  SIO.Nl();
  SIO.PutText("--------------------------------");
  SIO.Nl();
END WillkommenInAachen.
```

Lexikalische Einheiten

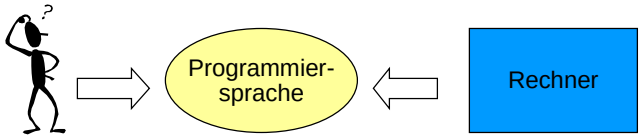
- Bezeichner
- Begrenzer
- Wortsymbole
- Literale
- Kommentare

Eventl. Trennzeichen zwischen lexikalischen Einheiten

```
MODULE WillkommenInAachen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss *)
IMPORT SIO;
VAR satz :TEXT;
BEGIN
  satz := "Willkommen zum Studium in Aachen!";
  SIO.Nl();
  SIO.PutText("---------------------------------");
  SIO.Nl();
  SIO.PutText(satz);
  SIO.Nl();
  SIO.PutText("---------------------------------");
  SIO.Nl();
END WillkommenInAachen.
```


Rolle

- **Die Programmiersprache bildet die *Schnittstelle* zwischen Mensch und Rechner**



Beide haben unterschiedliche Anforderungen

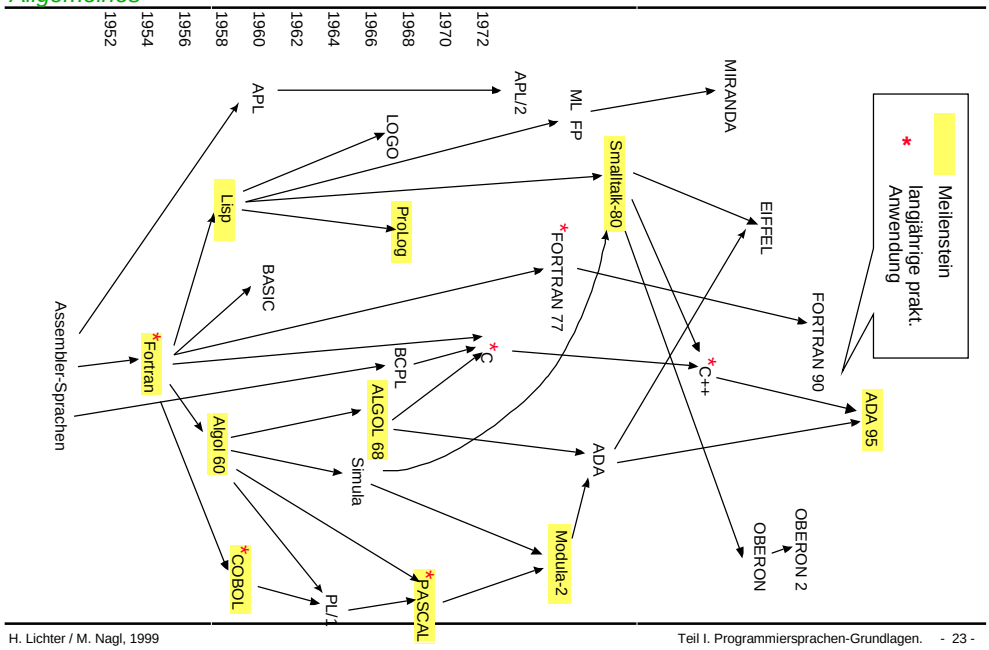
- **Mensch**
  - Erlernbarkeit
  - Lesbarkeit
  - Ausdrucksstärke

Problem-orientierte Sprachen

- **Rechner**
  - einfaches Übersetzen in Maschinensprache
  - effizienter Code soll generiert werden können

Maschinen-sprachen

Entwicklung



Werkstoff

- **Programmiersprachen sind der *Werkstoff* des Informatikers**
- **Das, was wir erzeugen (Programme)**
  - ist ***immateriell***
  - wir bauen es aus dem Werkstoff "Programmiersprache"
- **Ein Informatiker**
  - sollte die ***Qualität*** und ***Eignung*** verschiedener Werkstoffe (Programmiersprachen) kennen
  - Welche Sprache ist wofür geeignet?
- **Analogie!**
  - Ein Bauingenieur verwendet andere Werkstoffe, wenn ein
    - Hochhaus (Stahl, Beton etc.)
    - oder ein Einfamilienhaus (Ziegelsteine, Holz, Beton, etc.)
  - gebaut werden soll

Die Programmiersprache Modula-3

- **Modula-3**
  - erlaubt Programmierkonzepte ***elegant*** zu formulieren
  - zeigt die zentralen Konzepte der ***imperativen*** und ***objektorientierten*** Programmierung
  - ist ***leicht*** erlernbar
  - ist im Sinne der ***software-technischen Qualität*** von Programmen entwickelt worden
  - das fördert einen "***guten***" Programmierstil
  - die erlernten Konzepte werden Sie später in anderen Sprachen in anderer Form wiederfinden
- **Wichtig ist (im Sinne der VL)**
  - nicht die Programmiersprache Modula-3 selbst
  - sondern die ***Programmierkonzepte***

Modula-3

As Sam Harbison writes in his book on Modula-3,

Modula-3 is a member of the ***Pascal*** family of languages. Designed in the late 1980s at Digital Equipment Corporation and Olivetti, Modula-3 corrects many of the ***deficiencies*** of Pascal and Modula-2 for ***practical software engineering***. In particular, Modula-3 keeps the ***simplicity*** of type safety of the earlier languages, while providing ***new facilities*** for exception handling, concurrency, object-oriented programming, and automatic garbage collection. Modula-3 is both a practical implementation language for large software projects and an ***excellent teaching language***.

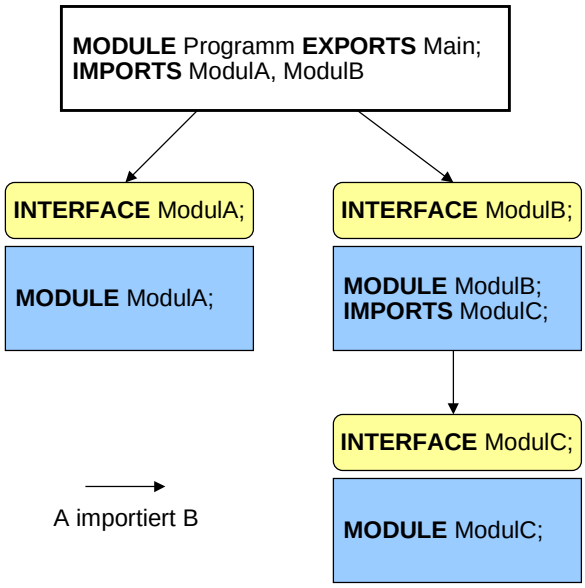
- **Modula = Modular Language**
  - Modul ist ein wichtiges Sprach- und Programmierkonzept
- **Modula-3 ist**
  - eine ***imperative*** (prozedurale) Programmiersprache
  - eine ***strukturierte*** Programmiersprache
  - eine objektorientierte Programmiersprache

Aufbau von Modula-3 Programmen

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***
- **Was ist ein Modul?**
  - ein Modul ist ein Programmteil, das ***sinnvoll*** zusammengehörende Elemente enthält
  - ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Exportschnittstelle*** (Interface)
      - definiert, was ein Modul an Diensten anbietet
    - ***Implementierung*** (Rumpf)
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Hauptmodul**
  - exportiert die vordefinierte ***leere*** Schnittstelle `Main`

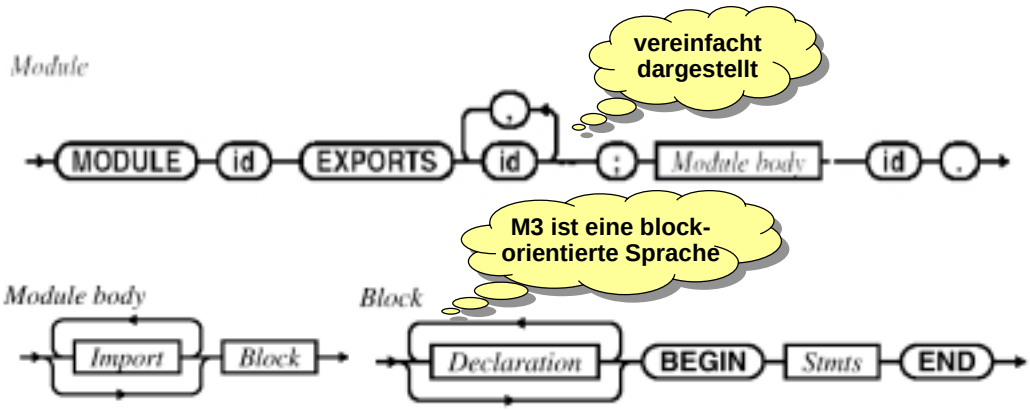
Modul-Hierarchie

- ein Modul kann Elemente anderer Module ***benutzen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar



Aufbau eines Moduls

- **Ein Modul**
  - ***exportiert*** Elemente und ***importiert*** andere Module
  - enthält einen ***Block***
- **Ein Block**
  - enthält ***Deklarationen*** und ***Anweisungen***



Was haben wir gelernt

- **Wie wissen was Programmiersprachen sind**
- **Wir kennen den Begriff "Formale Sprache"**
- **Wir haben gesehen, daß eine Grammatik eine Sprache erzeugt**
- **Wir kennen EBNF und Syntaxdiagramme**
- **Wir sehen Programmiersprachen als Werkstoff des Informatikers**

Glossar

- **Programmiersprache**
- **Syntax, Syntaxebenen**
- **Semantik**
- **Alphabet**
- **Formale Sprache**
- **Grammatik**
  - Chomsky-Grammatik
  - Kontextfreie Grammatik
- **Syntaxdarstellung: EBNF, Syntaxdiagramm**
- **Ableitungsbaum, Syntaxbaum**
- **Lexikalische Einheiten**
- **Modul: Export, Import, Rumpf**

Herzlich willkommen

zum Informatik-Studium

an der RWTH AACHEN !

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE

Vorlesung Programmierung WS 99/2000

Prof. Dr.-Ing. Manfred Nagl

Dr. Katja Cremer, Dirk Jäger

Unterlagen von H. Lichter, M. Nagl

Unter Mitarbeit von K. Cremer, D. Jäger und M. Schnizler

Teil I: Einführung

- Übersicht
- Informatik-Grundlagen
- Programmiersprachen-Grundlagen
- Erste Beispiele in Modula-3

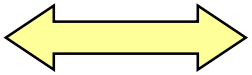
Übersicht

- Inhalt, Ziele
- Orientierung
- Literatur

Was ist "Informatik I"

- **In der Informatik unterscheiden wir zwei komplementäre Aspekte:**
  - Beschäftigung mit dem Computer als einer Maschine, die aus Hardware-Teilen zusammengesetzt ist,
  - Beschäftigung mit dem Computer als einer Maschine, auf der Programme (Software) ablaufen.
- **Diese Veranstaltung behandelt den *Softwareaspekt*:**
  - Was ist ein Programm?
  - Wie konstruiert (entwickelt) man ein Programm?
  - Wie benutzt man ein Programm?
  - Was kann man theoretisch über Programme sagen?
- **Weitere Veranstaltungen zu Software im Grundstudium**
  - Datenstrukturen und Algorithmen (2. Semester)
  - Systemprogrammierung (3. Semester)
  - Programmierpraktikum im Grundstudium (3. oder 4. Semester)

Ziel der LV "Programmierung"

- **Entwicklung von Programmen:**
  - Was ist ein ***Programm***?
  - Was ist eine ***Programmiersprache***?
  - Mit welchen Mitteln beschreibt oder konstruiert man ein Programm?
  - Wie können wir es schrittweise aus ***Komponenten*** entwickeln?
  - Wie bringen wir es auf den ***Rechner***?
  - Welche unterschiedlichen Arten der ***Programmstrukturierung*** gibt es?
- **Erlernen grundlegender Programmiertechniken**
  - Software-Ingenieur  Künstler
- **Dazu bedienen wir uns der klassischen Programmiersprache Modula-3**
- **Exkurse Prolog, LISP**

Weitere Ziele

- **Verstehen, was Informatik ist**
  - Begriffsklärung
  - Was sind die ***Hauptaufgaben*** der Informatik
- **Fähigkeit erwerben, in Gruppen zu arbeiten**
  - ***Teamfähigkeit*** und ***Kommunikation*** sind wichtig
  - An der Hochschule und besonders später im Arbeitsleben
- **Sich an die Arbeitsweise an der Hochschule zu gewöhnen**
  - Lehrveranstaltungen sind ***Angebote***
  - Sie entscheiden, was Sie von diesen Angeboten wahrnehmen wollen
  - Lernen, ***selbständig*** zu Lernen

Wie wollen wir das erreichen?

- **"Programmierung" ist eine dreiteilige Lehrveranstaltung**
  - Vorlesung
  - Diskussion
  - Gruppenübung
- **Weshalb Vorlesung?**
  - kompakte Vermittlung von Wissen und Erfahrung
  - Möglichkeit zur Abstimmung und Rückkopplung
- **Weshalb Übung?**
  - das Gelernte überprüfen und vertiefen
  - in der Gruppe arbeiten
  - etwas und sich selbst darstellen lernen
  - Handwerkszeug handhaben lernen
  - Hilfe zur Selbsthilfe

Nutzen Sie diese Angebote!
Es ist ihre Lehrveranstaltung

Was läuft wo?

- **In der Vorlesung:**
  - Einführung in die Programmierung
  - Begriffe, Konzepte und Beispiele
  - Ergänzende und vertiefende Themen
- **In der Diskussion:**
  - Vorstellen von Themen, die für alle relevant sind (z.B. Arbeiten mit einer Programmierumgebung)
  - Fragen zur Vorlesung
- **In der Gruppenübung**
  - Diskussion von speziellen Aspekten der Übungsaufgaben
  - Präsentation von Lösungen

Spezialist und Novize

- **Viele haben bereits Erfahrungen ("Spezialisten")**
  - im Umgang mit Rechnern
  - im Programmieren mit einer Programmiersprache
  - aus eigenem Interesse am Thema oder durch Kurse am Gymnasium
- **Andere haben keine oder nur wenig Vorkenntnisse ("Novizen")**
  - alles ist neu und ungewohnt
  - Angst, schlechter als die "Spezialisten" zu sein

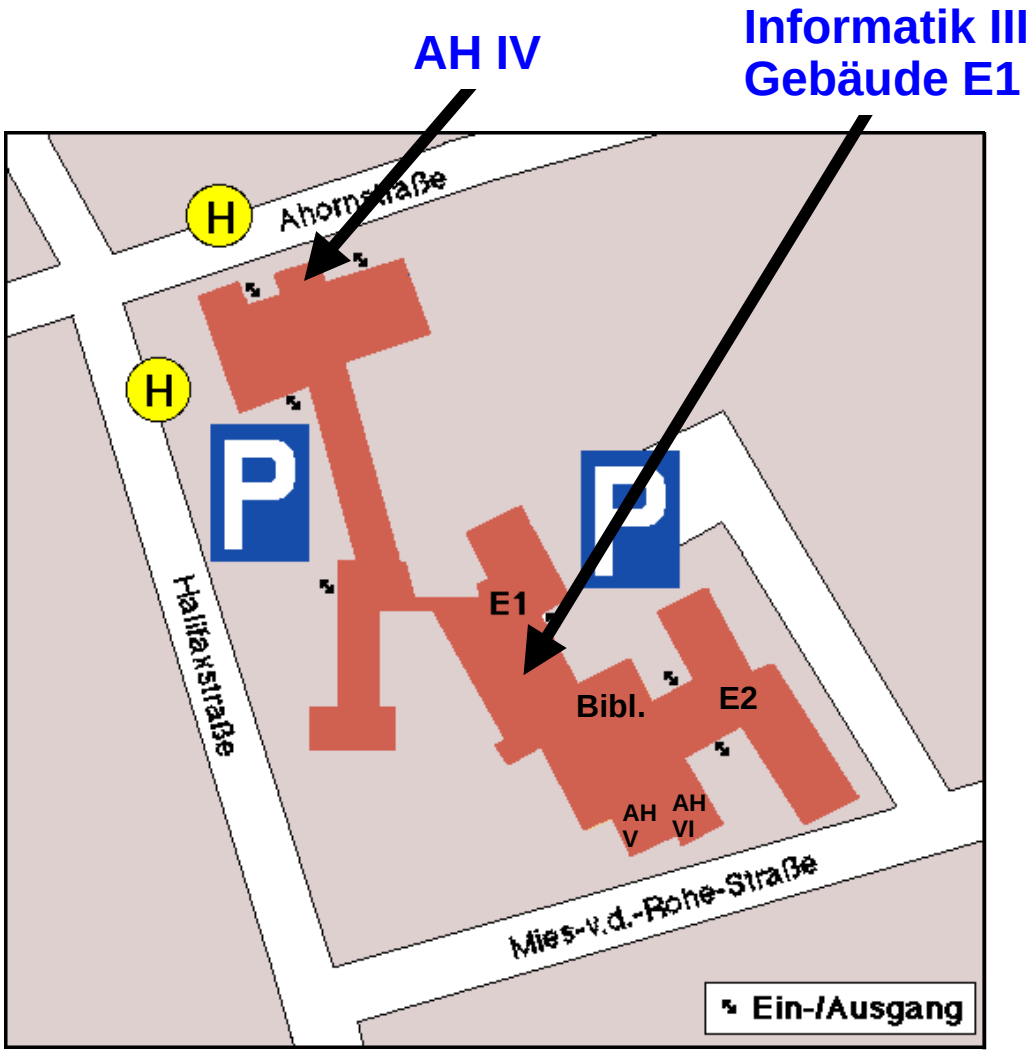
→ Vorkurs Informatik

Diese Angst ist unbegründet !

Wir erarbeiten systematisch das Gebiet der Programmierung!

Wer macht was?

- **Vorlesung**
  - Manfred Nagl
  - Raum: E1 - 4309a
  - Sprechstunde:
    - Fr, 11:00 - 12:00
- **Diskussion**
  - K. Cremer, D. Jäger
  - Raum: AH IV
  - Sprechstunde:
    - Mo, 8:30 - 10:00
- **Gruppenübung**
  - viele studentische Hilfskräfte als Tutoren



Termine

Vorlesung

- Montag: 13:30 - 15:00 Ro
- Donnerstag: 10:50 - 12:20 Ro

Diskussion

- Dienstag: 17:30 - 19:00 AH IV , Beginn: 19. Oktober

Gruppenübungen

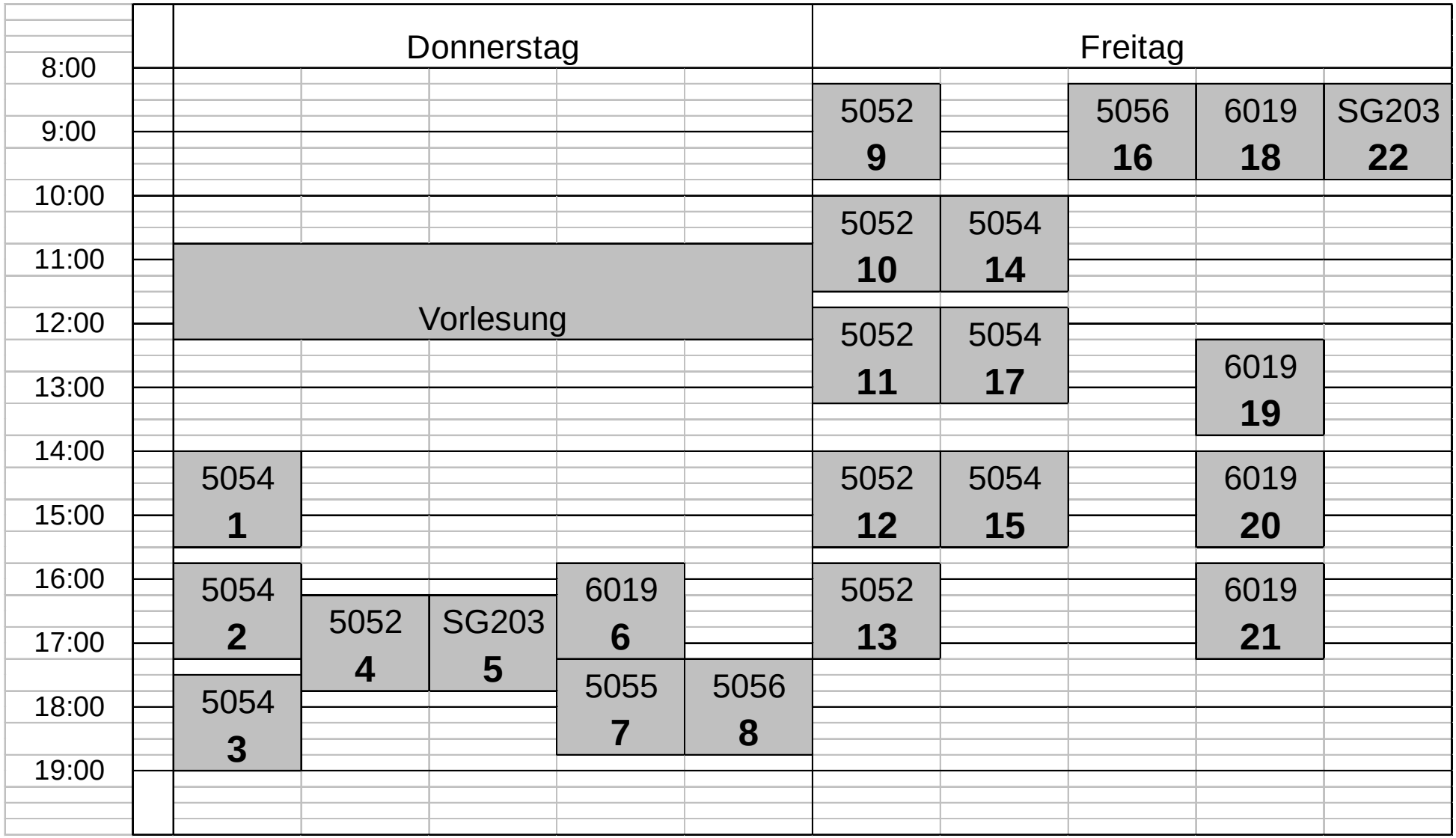
- Donnerstag:

14:00 - 15:30	(1)	Beginn: 21. Oktober
15:45 - 17:15	(2)	
16:15 - 17:45	(2)	
17:15 - 18:45	(2)	
17:30 - 19:00	(1)	

- Freitag:

08:15 - 09:45	(4)	Beginn: 22. Oktober
10:00 - 11:30	(2)	
11:45 - 13:15	(2)	
12:15 - 13:45	(1)	
14:00 - 15:30	(3)	
15:45 - 17:15	(2)	

Termine


Donnerstag
Freitag
8:00
9:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
Vorlesung
5052 9
5056 16
6019 18
SG203 22
5052 10
5054 14
5052 11
5054 17
6019 19
5052 12
5054 15
6019 20
5052 13
6019 21
5054 1
5054 2
5052 4
SG203 5
6019 6
5054 3
5055 7
5056 8

Wie zu was anmelden?

- **Anmeldung zu den Gruppenübungen**
- **Listen für die einzelnen Übungsgruppen hängen aus:**
- **Nachzügler**
  - Gebäude E1, 3. Etage
  - ab heute
  - bis ***Freitag 15.10.99, 17:00***
- **Bitte die Listen gleichmäßig ausfüllen !**
  - Bitte nur den vorgesehenen Platz verwenden
- **Übungen werden in Kleingruppen bearbeitet und abgegeben:**
  - mindestens zwei, maximal drei Personen pro Kleingruppe, "Einpersonengruppen" werden nicht akzeptiert!
  - Keine Ausländergruppen
  - Frauengruppen

Literatur zur Vorlesung

- **Diese Vorlesung stützt sich in großen Teilen auf die folgenden Materialien:**
  - Hans-Jürgen Appelrath, Jochen Ludewig: "***Skriptum Informatik - eine konventionelle Einführung***", B.G. Teubner Stuttgart,1995 .
  - László Böszörményi, Carsten Weich: "***Programmieren mit Modula-3 - Eine Einführung in stilvolle Programmierung*** ", Springer Verlag,1995.
  - Robert W. Sebesta: "***Concepts of Programming Languages***". Benjamin Cummings, 2. Auflage, 1993.
- **PROLOG, LISP**
  - L. Sterling, E. Shapiro: "***The Art of PROLOG***", MIT Press, 1994.
  - Berthold K. Horn, Patrick H. Winston, "***LISP***", Addison-Wesley, 1989.
- **Arbeiten an der Hochschule**
  - selbständig Inhalte bestimmen (besonders im Hauptstudium)
  - selbständig Inhalte erarbeiten, Umgang mit ***Literatur***

Unterlagen zur Vorlesung

- **Stehen im "world wide web" zur Verfügung**
  - ***http://www-i3.informatik.rwth-aachen.de/Programmierung***
    - Neuigkeiten,
    - Folien der Vorlesungen,
    - Übungsblätter,
    - Lösungen

- **Folien der Vorlesungen**
  - werden - nach Wunsch - in Teilen vervielfältigt und verteilt
    - je nach Fortschritt
    - in der Diskussion, in den Kleingruppen

- **Literatur, s.u.**

Was erwarten wir von Ihnen?

- **Dies ist *Ihre* Lehrveranstaltung!**
- **Mitarbeit**
  - die LV ist keine ***Beschäftigungstherapie*** und kein Kabelprogramm
  - ohne Arbeit keine ***bleibenden Ergebnisse***
  - ohne ***aktive Mitarbeit*** keine Erkenntnis
- **Gruppenarbeit**
  - ohne Gruppenarbeit keine ***Qualifikation*** zur Softwareentwicklung
  - ohne Gruppenarbeit wird das Studium ***zäh bis erfolglos***
- **Rückkopplung:**
  - über Inhalte
  - über Formen
  - über Probleme

Prüfung !

- **Die Diplom-Prüfungsordnung (DPO) regelt, welche Prüfungen Sie ablegen müssen.**
- **Vordiplomprüfung**
  - Fachprüfung "Informatik I"
    - LV "Programmierung“ 1. Semester
    - LV "Datenstrukturen“ 2. Semester
- **Zulassung:**
  - Übungsschein "Programmierung"
    - Voraussetzung für Vordiplomsprüfung
  - Leistungsnachweis "Programmierung“
    - Voraussetzung für Übungsschein
    - aktive und erfolgreiche Teilnahme

Diesen Übungsschein sollten Sie in dieser Veranstaltung erwerben!

Prüfungsmodalitäten

- **Wer erhält einen Übungsschein?**
  - Alle, die die am Semesterende angebotene ***Übungsscheinklausur*** bestanden haben.
  - Die Klausur geht über den ganzen Stoff

- **Scheinklausur**
  - Dauer: 2 Stunden
  - Termin: Februar 1999
  - Ort: Roter Hörsaal, AH IV, u.a.
- **Wer darf an der Scheinklausur teilnehmen?**
  - Alle, die ***50*** % der gestellten selbst ***Übungsaufgaben*** gelöst haben!
    Von beiden Hälften des Semesters => Leistungsnachweis Programmierung
- **Wieviel Übungsaufgaben gibt es?**
  - 14 ***Übungsblätter*** (mit unterschiedlicher Anzahl von Aufgaben)
  - Pro Übungsblatt werden ***20 Punkte*** vergeben!
  - Die Übungsaufgabe werden schwieriger

Übungsbetrieb

- **Semesterwoche n+1**
  - **Montag**: ***Ausgabe*** des Übungsblatts ***n***
Abgabe der Lösungen zu Übungsblatt ***n-1***
  - **Dienstag**: Fragen zum Vorlesungsstoff der Woche ***n***
  - **Donnerstag**: Diskussion in den Gruppenübungen zu ***n-1***
  - **Freitag**: Diskussion in den Gruppenübungen zu ***n-1***

- **Ausgabe der Übungsblätter**
  - Montag, nach der Vorlesung

- **Abgabe der Übungen**
  - Montag **vor** der Vorlesung ***in der Folgewoche***
  - oder Gebäude E1, 3. Etage **bis 13:00**

- **Erstes Übungsblatt: *Montag, 18. Oktober***

Informationen zum Rechnerbetrieb - 1

- **Sie benötigen Zugang zu den Rechnern, um**
  - auf die "online" zur Verfügung gestellten Informationen zugreifen zu können
  - Programmieraufgaben lösen zu können
- **Rechner werden im sogenannten "Rechnerpool Informatik" zur Verfügung gestellt**
  - 19 Workstations unter Windows NT
  - 33 SUN-Workstations unter Solaris (Unix)
  - 13 HP-Workstations unter HP-UX (Unix)

  } Gebäude E1
- **Öffnungszeiten**
  - Mo - Do : 9:00 - 19:00, Fr 9:00 -17:00
  - In der ersten Vorlesung werden Anträge für Benutzerkennungen ausgegeben

    Einige haben diese bereits aus dem Vorkurs

Informationen zum Rechnerbetrieb - 2

- **Folgende Zeiten sind für Hörer der Veranstaltung "Programmierung" reserviert:**

- **Montag: 17.00 - 19.00**
- **Dienstag: 09.00 - 12.00**
- **Donnerstag: 09.00 - 15.00**
- **Freitag: 09.00 - 13.00**

Herzlich willkommen

zum Informatik-Studium

an der RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE RWTH AACHEN **!**

Vorlesung Programmierung WS 99/2000

Prof. Dr.-Ing. Manfred Nagl

Dr. Katja Cremer, Dirk Jäger

Unterlagen von H. Lichter, M. Nagl

Unter Mitarbeit von K. Cremer, D. Jäger und M. Schnizler

Teil I: Einführung

- Übersicht
- Informatik-Grundlagen
- Programmiersprachen-Grundlagen
- Erste Beispiele in Modula-3

Übersicht

- Inhalt, Ziele
- Orientierung
- Literatur

Inhalt, Ziele

Was ist "Informatik I"

- **In der Informatik unterscheiden wir zwei komplementäre Aspekte:**
  - Beschäftigung mit dem Computer als einer Maschine, die aus Hardware-Teilen zusammengesetzt ist,
  - Beschäftigung mit dem Computer als einer Maschine, auf der Programme (Software) ablaufen.
- **Diese Veranstaltung behandelt den *Softwareaspekt*:**
  - Was ist ein Programm?
  - Wie konstruiert (entwickelt) man ein Programm?
  - Wie benutzt man ein Programm?
  - Was kann man theoretisch über Programme sagen?
- **Weitere Veranstaltungen zu Software im Grundstudium**
  - Datenstrukturen und Algorithmen (2. Semester)
  - Systemprogrammierung (3. Semester)
  - Programmierpraktikum im Grundstudium (3. oder 4. Semester)

Inhalt, Ziele

Ziel der LV "Programmierung"

- **Entwicklung von Programmen:**
  - Was ist ein ***Programm***?
  - Was ist eine ***Programmiersprache***?
  - Mit welchen Mitteln beschreibt oder konstruiert man ein Programm?
  - Wie können wir es schrittweise aus ***Komponenten*** entwickeln?
  - Wie bringen wir es auf den ***Rechner***?
  - Welche unterschiedlichen Arten der ***Programmstrukturierung*** gibt es?
- **Erlernen grundlegender Programmiertechniken**
  - Software-Ingenieur ⟺ Künstler
- **Dazu bedienen wir uns der klassischen Programmiersprache Modula-3**
- **Exkurse Prolog, LISP**

Weitere Ziele

- **Verstehen, was Informatik ist**
  - Begriffsklärung
  - Was sind die ***Hauptaufgaben*** der Informatik
- **Fähigkeit erwerben, in Gruppen zu arbeiten**
  - ***Teamfähigkeit*** und ***Kommunikation*** sind wichtig
  - An der Hochschule und besonders später im Arbeitsleben
- **Sich an die Arbeitsweise an der Hochschule zu gewöhnen**
  - Lehrveranstaltungen sind ***Angebote***
  - Sie entscheiden, was Sie von diesen Angeboten wahrnehmen wollen
  - Lernen, ***selbständig*** zu Lernen

Wie wollen wir das erreichen?

- **"Programmierung" ist eine dreiteilige Lehrveranstaltung**
  - Vorlesung
  - Diskussion
  - Gruppenübung
- **Weshalb Vorlesung?**
  - kompakte Vermittlung von Wissen und Erfahrung
  - Möglichkeit zur Abstimmung und Rückkopplung
- **Weshalb Übung?**
  - das Gelernte überprüfen und vertiefen
  - in der Gruppe arbeiten
  - etwas und sich selbst darstellen lernen
  - Handwerkszeug handhaben lernen
  - Hilfe zur Selbsthilfe

Nutzen Sie diese Angebote!
Es ist ihre Lehrveranstaltung

Was läuft wo?

- **In der Vorlesung:**
  - Einführung in die Programmierung
  - Begriffe, Konzepte und Beispiele
  - Ergänzende und vertiefende Themen
- **In der Diskussion:**
  - Vorstellen von Themen, die für alle relevant sind (z.B. Arbeiten mit einer Programmierumgebung)
  - Fragen zur Vorlesung
- **In der Gruppenübung**
  - Diskussion von speziellen Aspekten der Übungsaufgaben
  - Präsentation von Lösungen

Spezialist und Novize

- **Viele haben bereits Erfahrungen ("Spezialisten")**
  - im Umgang mit Rechnern
  - im Programmieren mit einer Programmiersprache
  - aus eigenem Interesse am Thema oder durch Kurse am Gymnasium
- **Andere haben keine oder nur wenig Vorkenntnisse ("Novizen")**
  - alles ist neu und ungewohnt
  - Angst, schlechter als die "Spezialisten" zu sein

} → Vorkurs Informatik

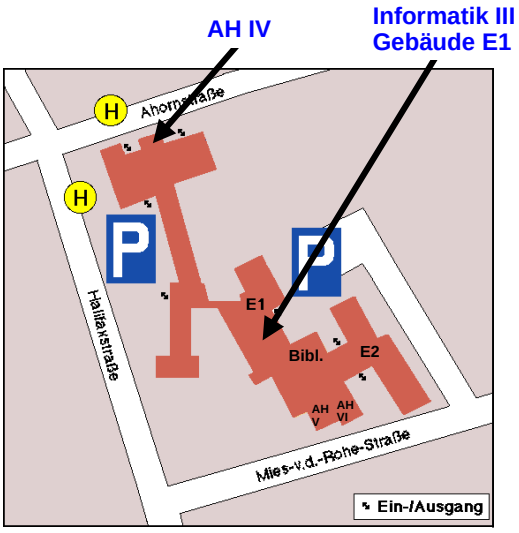
Diese Angst ist unbegründet !

Wir erarbeiten systematisch das Gebiet der Programmierung!

Orientierung

Wer macht was?

- **Vorlesung**
  - Manfred Nagl
  - Raum: E1 - 4309a
  - Sprechstunde:
    - Fr, 11:00 - 12:00
- **Diskussion**
  - K. Cremer, D. Jäger
  - Raum: AH IV
  - Sprechstunde:
    - Mo, 8:30 - 10:00
- **Gruppenübung**
  - viele studentische Hilfskräfte als Tutoren



Orientierung

Termine

- **Vorlesung**
  - Montag: 13:30 - 15:00 Ro
  - Donnerstag: 10:50 - 12:20 Ro
- **Diskussion**
  - Dienstag: 17:30 - 19:00 AH IV , Beginn: 19. Oktober
- **Gruppenübungen**
  - Donnerstag:

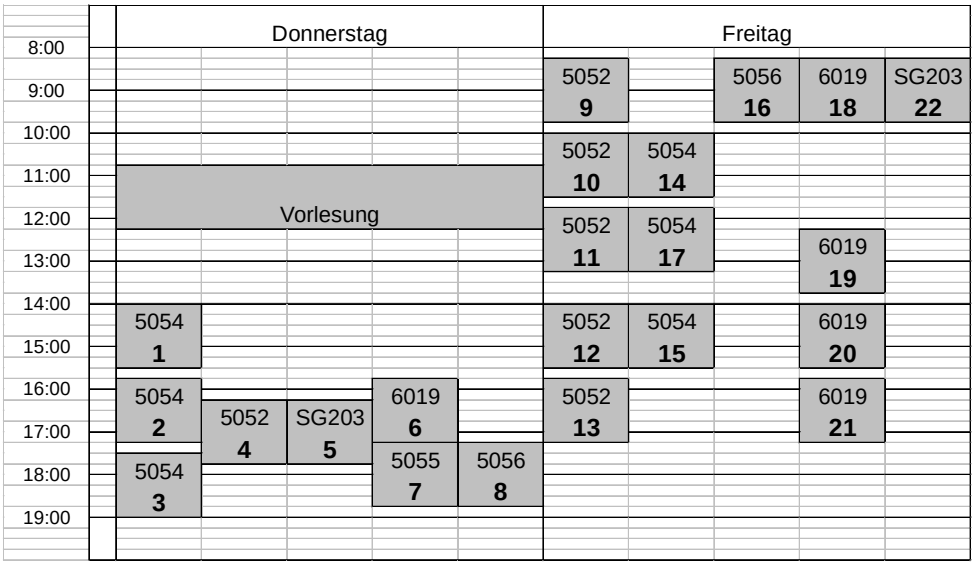
14:00 - 15:30	(1)	Beginn: 21. Oktober
15:45 - 17:15	(2)	
16:15 - 17:45	(2)	
17:15 - 18:45	(2)	
17:30 - 19:00	(1)	

  - Freitag:

08:15 - 09:45	(4)	Beginn: 22. Oktober
10:00 - 11:30	(2)	
11:45 - 13:15	(2)	
12:15 - 13:45	(1)	
14:00 - 15:30	(3)	
15:45 - 17:15	(2)	

Termine

Orientierung



Wie zu was anmelden?

Orientierung

- **Anmeldung zu den Gruppenübungen**
- **Listen für die einzelnen Übungsgruppen hängen aus:**
- **Nachzügler**
  - Gebäude E1, 3. Etage
  - ab heute
  - bis ***Freitag 15.10.99, 17:00***
- **Bitte die Listen gleichmäßig ausfüllen !**
  - Bitte nur den vorgesehenen Platz verwenden
- **Übungen werden in Kleingruppen bearbeitet und abgegeben:**
  - mindestens zwei, maximal drei Personen pro Kleingruppe, "Einpersonengruppen" werden nicht akzeptiert!
  - Keine Ausländergruppen
  - Frauengruppen

Literatur zur Vorlesung

- **Diese Vorlesung stützt sich in großen Teilen auf die folgenden Materialien:**
  - Hans-Jürgen Appelrath, Jochen Ludewig: "***Skriptum Informatik - eine konventionelle Einführung***", B.G. Teubner Stuttgart,1995 .
  - László Böszörményi, Carsten Weich: "***Programmieren mit Modula-3 - Eine Einführung in stilvolle Programmierung*** ", Springer Verlag,1995.
  - Robert W. Sebesta: "***Concepts of Programming Languages***". Benjamin Cummings, 2. Auflage, 1993.
- **PROLOG, LISP**
  - L. Sterling, E. Shapiro: "***The Art of PROLOG***", MIT Press, 1994.
  - Berthold K. Horn, Patrick H. Winston, "***LISP***", Addison-Wesley, 1989.
- **Arbeiten an der Hochschule**
  - selbständig Inhalte bestimmen (besonders im Hauptstudium)
  - selbständig Inhalte erarbeiten, Umgang mit ***Literatur***

Unterlagen zur Vorlesung

- **Stehen im "world wide web" zur Verfügung**
  - ***http://www-i3.informatik.rwth-aachen.de/Programmierung***
    - Neuigkeiten,
    - Folien der Vorlesungen,
    - Übungsblätter,
    - Lösungen
- **Folien der Vorlesungen**
  - werden - nach Wunsch - in Teilen vervielfältigt und verteilt
    - je nach Fortschritt
    - in der Diskussion, in den Kleingruppen
- **Literatur, s.u.**

Was erwarten wir von Ihnen?

- **Dies ist *Ihre* Lehrveranstaltung!**
- **Mitarbeit**
  - die LV ist keine ***Beschäftigungstherapie*** und kein Kabelprogramm
  - ohne Arbeit keine ***bleibenden Ergebnisse***
  - ohne ***aktive Mitarbeit*** keine Erkenntnis
- **Gruppenarbeit**
  - ohne Gruppenarbeit keine ***Qualifikation*** zur Softwareentwicklung
  - ohne Gruppenarbeit wird das Studium ***zäh bis erfolglos***
- **Rückkopplung:**
  - über Inhalte
  - über Formen
  - über Probleme

Prüfung !

- **Die Diplom-Prüfungsordnung (DPO) regelt, welche Prüfungen Sie ablegen müssen.**
- **Vordiplomprüfung**
  - Fachprüfung "Informatik I"
    - LV "Programmierung“ 1. Semester
    - LV "Datenstrukturen“ 2. Semester
- **Zulassung:**
  - Übungsschein "Programmierung"
    - Voraussetzung für Vordiplomsprüfung
  - Leistungsnachweis "Programmierung“
    - Voraussetzung für Übungsschein
    - aktive und erfolgreiche Teilnahme

Diesen Übungsschein sollten Sie in dieser Veranstaltung erwerben!

Orientierung

Prüfungsmodalitäten

- **Wer erhält einen Übungsschein?**
  - Alle, die die am Semesterende angebotene ***Übungsscheinklausur*** bestanden haben.
  - Die Klausur geht über den ganzen Stoff
- **Scheinklausur**
  - Dauer: 2 Stunden
  - Termin: Februar 1999
  - Ort: Roter Hörsaal, AH IV, u.a.

- **Wer darf an der Scheinklausur teilnehmen?**
  - Alle, die ***50*** % der gestellten selbst ***Übungsaufgaben*** gelöst haben!
    Von beiden Hälften des Semesters => Leistungsnachweis Programmierung
- **Wieviel Übungsaufgaben gibt es?**
  - 14 ***Übungsblätter*** (mit unterschiedlicher Anzahl von Aufgaben)
  - Pro Übungsblatt werden ***20 Punkte*** vergeben!
  - Die Übungsaufgabe werden schwieriger

Orientierung

Übungsbetrieb

- **Semesterwoche n+1**
  - **Montag**: ***Ausgabe*** des Übungsblatts ***n***
    Abgabe der Lösungen zu Übungsblatt ***n-1***
  - **Dienstag**: Fragen zum Vorlesungsstoff der Woche ***n***
  - **Donnerstag**: Diskussion in den Gruppenübungen zu ***n-1***
  - **Freitag**: Diskussion in den Gruppenübungen zu ***n-1***
- **Ausgabe der Übungsblätter**
  - Montag, nach der Vorlesung
- **Abgabe der Übungen**
  - Montag **vor** der Vorlesung ***in der Folgewoche***
  - oder Gebäude E1, 3. Etage **bis 13:00**
- **Erstes Übungsblatt: *Montag, 18. Oktober***

Informationen zum Rechnerbetrieb - 1

- **Sie benötigen Zugang zu den Rechnern, um**
  - auf die "online" zur Verfügung gestellten Informationen zugreifen zu können
  - Programmieraufgaben lösen zu können
- **Rechner werden im sogenannten "Rechnerpool Informatik" zur Verfügung gestellt**
  - 19 Workstations unter Windows NT
  - 33 SUN-Workstations unter Solaris (Unix)
  - 13 HP-Workstations unter HP-UX (Unix)

  } Gebäude E1
- **Öffnungszeiten**
  - Mo - Do : 9:00 - 19:00, Fr 9:00 -17:00
  - In der ersten Vorlesung werden Anträge für Benutzerkennungen ausgegeben

    Einige haben diese bereits aus dem Vorkurs

Informationen zum Rechnerbetrieb - 2

- **Folgende Zeiten sind für Hörer der Veranstaltung "Programmierung" reserviert:**

Montag:	**17.00 - 19.00**
Dienstag:	**09.00 - 12.00**
Donnerstag:	**09.00 - 15.00**
Freitag:	**09.00 - 13.00**

Datentypen I

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

Datentypen - Grundidee

- **Software dient zur Verarbeitung von *Anwendungsdaten*.**
  - Frage: Wie gut passen die verfügbaren Datentypen der verwendeten Programmiersprache zu den zu modellierenden Größen des *Anwendungsbereichs*.
- **Daraus resultiert die Anforderung:**
  - Programmiersprachen müssen einen ***sinnvollen*** Satz an Datentypen anbieten.
- **Ziel:**
  - Auf der Basis vordefinierter Datentypen sollen anwendungsbezogene Datenstrukturen konstruiert werden können.
- **Zwei Lösungsansätze:**
  - Eine große Vielfalt von ***vordeklarierten*** Datentypen (wie in PL/I) soll möglichst viele Anwendungsfälle abdecken.
  - Ein kleiner Satz von elementaren Typen und flexiblen ***Konstruktions-mechanismen*** (wie in Algol 68) soll die anwendungsbezogene Definition neuer Datentypen erlauben.

Bedeutung von Typen in imp. Sprachen

- **In imperativen Programmiersprachen hat ein Typ folgende Bedeutung (Wiederholung):**
  - Festlegung der/des gültigen ***Struktur/Wertebereichs*** für Variablen,
  - damit verbunden die ***Kardinalität*** (Anzahl der verschiedenen Werte),
  - Festlegen der ***Operationen*** und Literale/Aggregate.
  - Statische ***Prüfung*** der Zulässigkeit von Operationen:
    - Zuweisung, Parameterübergabe etc.
- **Festlegung von Maschineneigenschaften**
  - z.B. Reservierung von Speicherplatz

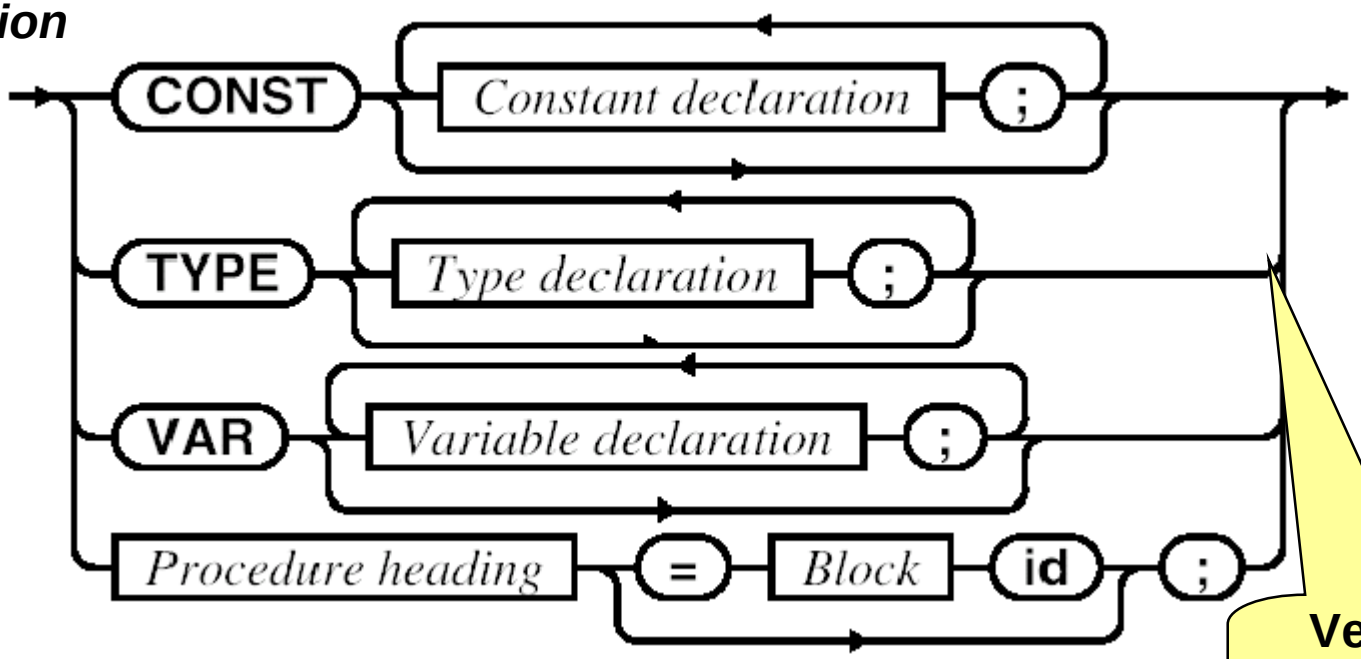
Erinnerung: Deklaration

- **In imperativen Programmiersprachen ist die explizite Deklaration von Bezeichnern notwendig.**
- **Grundidee der Deklaration:**
  - Eine Deklaration verbindet einen ***Bezeichner*** an Eigenschaften eines Programmobjekts, z.B. Typ und Sichtbarkeit.
  - Während sich z.B. bei Variablen der Wert des Objektes verändern kann, bleiben seine deklarierten Eigenschaften für die Lebensdauer erhalten.
- **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***

Deklaration (erweitert)

- **Die Syntax für Deklarationen wird erweitert, damit *benutzerdefinierte* Typen deklariert werden können.**

Declaration



Verwendung von Datentyp-konstruktoren

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.

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

Typen in Modula-3

- **Modula-3 kennt *vordefinierte einfache* Typen :**
  - SHORTINT, INTEGER, LONGINT, REAL, LONGREAL, BOOLEAN, SET, CHAR.
- **Modula-3 unterstützt wie viele imperative Sprachen *benutzerdefinierte* Datentypen.**
- **Häufig verwenden wir in der Definition die *Typkonstruktoren* für die zusammengesetzten Datentypen**
  - Array, Record, Set.
- **Eine wesentliche Erweiterung bringt der *Zeigertyp* (Pointer).**
  - Er ermöglicht den Aufbau ***dynamischer*** Datenstrukturen.

Benutzerdefinierte einfache Typen

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

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];
```

Ordinaltyp

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

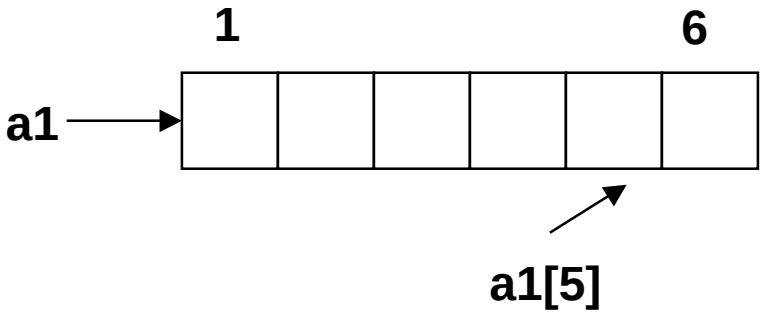
Operationen auf Aufzählungstypen

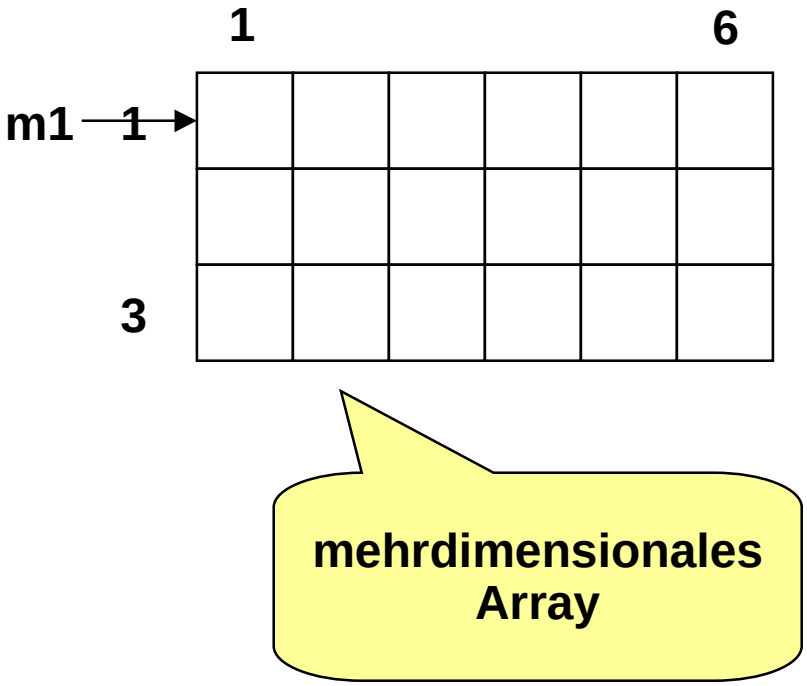
- **Aufzählungstypen**
  - werden systemintern auf nichtnegative Zahlen abgebildet
  - (z.B.: rot ->1, blau -> 2, gruen -> 3 oder 0, 1, 2).
  - Dadurch sind die Werte von Aufzählungstypen dann ***vergleichbar*** – was aber selten Sinn macht (rot < gruen).
- **Bezeichner der Werte von Aufzählungstypen 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!

Operationen auf Unterbereichstypen

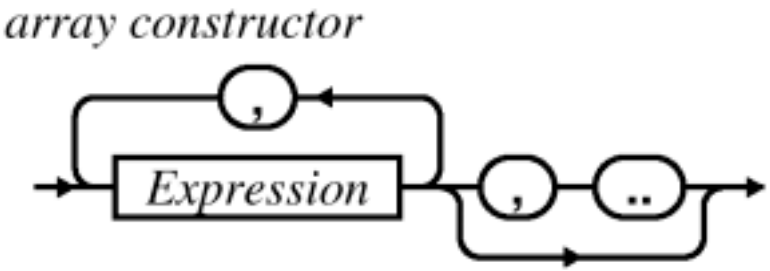
- **Regel:**
  - Für einen Unterbreichstyp sind alle die Operationen definiert, die auch für seinen ***Basistyp*** definiert sind.
- **Es ist häufig unsinnig,**
  - ***arithmethische*** Operationen unmittelbar auf Unterbereichstypen anzuwenden, auch wenn dies die Sprache zuläßt (z.B. Addition zweier 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***.
    ```

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.

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
- Dem Array als Datenstruktur entspricht die ***FOR-Anweisung*** als Kontrollstruktur.
- Bei mehrfach indizierten Arrays gibt es entsprechend ***geschachtelte*** Schleifen.

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;
```

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;
```

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 *)
    ```

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, ..}  Feldaggregate
```

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

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

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
```

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;

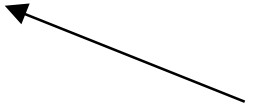
CONST
   TagNamen  = ARRAY Tage OF TEXT {"Montag", "Dienstag","Mittwoch",
                                   "Donnerstag", "Freitag"};
   FachNamen = ARRAY Fächer OF TEXT {"Keine","Englisch","Software_1","Mathematik"};

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

Typdeklarationen

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;

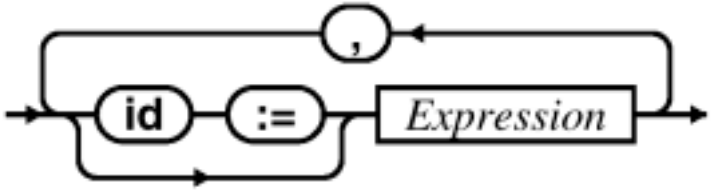
  FOR tag:= FIRST(Tage) TO LAST(TAGE) DO
    SIO.PutText(TagNamen[tag]& "\ n");
    FOR stunde:= FIRST(Vormittag) TO LAST(Vormittag) DO
      SIO.PutInt(stunde);
      SIO.PutText(":" & FachNamen[stundenPlan[tag, stunde]]);
    END; (*FOR stunde*)
    SIO.NL();
  END; (*FOR tag*)
END StundenPlan.
```

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)

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

Beispiel: RECORD - 1

```
MODULE RecordDemo EXPORTS Main;

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

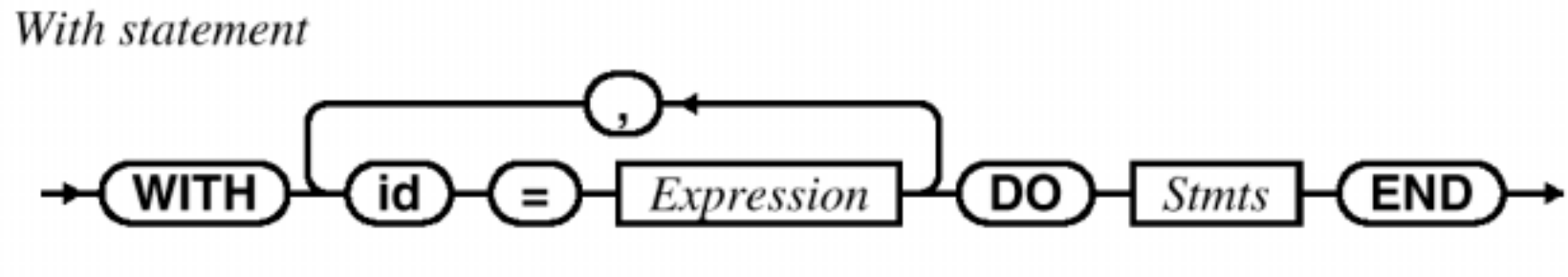
TYPE Person = RECORD
                anrede   : Anrede;
                vorname  : Name;
                nachname : Name;
                persnr   : PersNr;
              END;

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.

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

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 *)
```

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

record constructor




```
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

Operationen auf RECORDs - 1

Zuweisung

- zwei RECORD-Objekte sind ***zuweisungskompatibel***, wenn
  - alle Felder den gleichen ***Namen*** und den gleichen Typ haben
  - alle Felder in der gleichen ***Reihenfolge*** deklariert sind

Vergleich

- zuweisungskompatible Arrays 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
```

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

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
```

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.
  - Nicht Elemente, sondern Zugehörigkeitsinformationen abgelegt: charakteristische Speicherung

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

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

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.
```

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

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!

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

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 *)

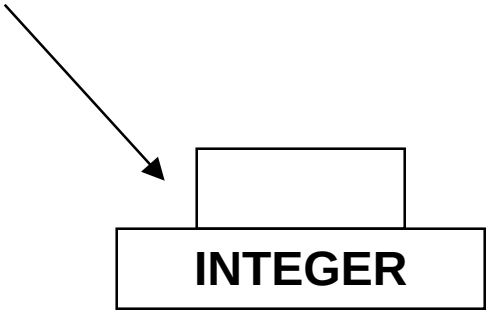
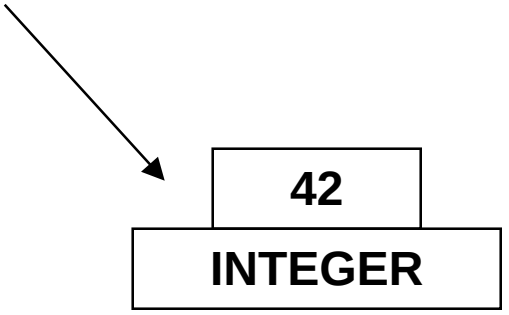
  Ausgabe(Vorkommenzaehlen(SIO.GetLine()));

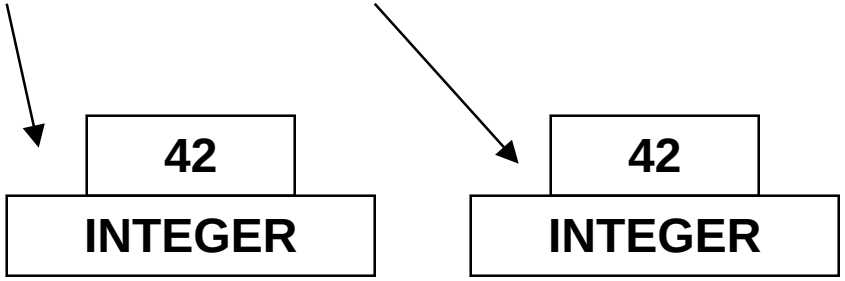
END Buchstaben2.
```

Verbesserung 2 des Beispiels - d

```
PROCEDURE LiesEingabeBis (stop : CHAR) : TEXT =
VAR eingabe : TEXT := "";
    zeichen : CHAR;
BEGIN
  zeichen := SIO.GetChar();
  WHILE zeichen # stop DO
    eingabe := eingabe & Text.FromChar(zeichen);
    zeichen := SIO.GetChar();
  END;
  RETURN eingabe;
END LiesEingabeBis;

…

BEGIN (*Buchstaben3*)

  Ausgabe(Vorkommenzaehlen(LiesEingabeBis(':')));

END Buchstaben3.
```

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 (siehe auch Begriffe von vorangegangener Seite)

- **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**

Datentypen II

- **Dynamische Datentypen**
  - Zeigertypen
  - dynamische Datenstrukturen
- **Anwendungsbeispiele**
  - lineare Liste
  - sortierte Liste
- **Prozedurtyp**

Bezeichner und Objekte

- **Wir haben als elementares imperatives Konzept die *Variable* kennengelernt.**
- **Abarbeitung der Deklaration:**
  - Ein Name dient als ***Bezeichner***, der mit einem ***getypten Wert*** verbunden werden kann (Struktur, Speicherplatz).
- **Bei den bisher betrachteten Datentypen**
  - wurde der Zusammenhang von Bezeichner und Wert implizit in der Sprache durch den *Zuweisungsmechanismus* hergestellt.

```
VAR Antwort: INTEGER;
```




```
Antwort := 42;
```



Wertsemantik

- **Die bisher betrachteten Variablen und die Zuweisung basieren auf der *Wertsemantik*:**
  - Eine Variable hat einen ***definierten*** oder ***undefinierten*** Wert.
  - Bei der ***Zuweisung*** wird der Wert des Ausdrucks der rechten Seite (rhv) an die Variable der linken Seite (lhv) zugewiesen.
  - Eine ***Identität*** von Objekten wird nicht hergestellt.
  - ```
    VAR Antwort1, Antwort2: INTEGER;
    …
    Antwort1 := 42; (* 1 *)
    Antwort2 := Antwort1; (* 2 *)
    Antwort1 := 24; (* 3 *)

    Antwort1 := Antwort2;
    ```


    ```

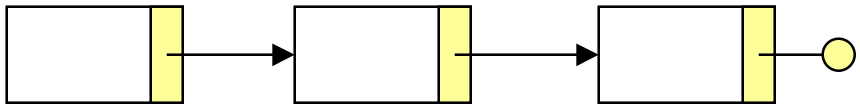
Statische Datentypen

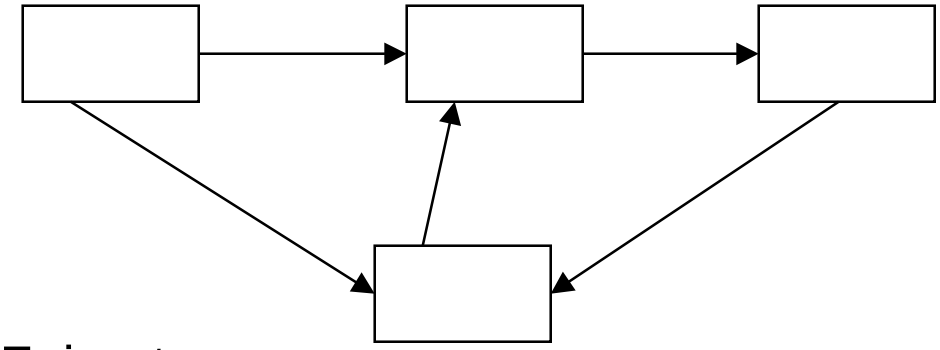
- **Kennzeichnend für imperative Sprachen ist, daß *Wertsemantik* und *statische Datentypen* zusammenfallen:**
  - Die Zuordnung von Bezeichner und Wert geschieht über die Zuweisung.
  - Die Variablen eines statischen Typs ***behalten ihre Struktur*** während ihrer Lebensdauer bei.
  - Der ***Speicher*** einer statisch getypten Variablen wird bei der Übersetzung anhand der Deklaration ***festgelegt*** und bei der "ersten Verwendung implizit angelegt ".
- **Speicherbereiche für statisch getypte Variablen wird im sog. *Kellerspeicher* reserviert**
  - zusammenhängender Bereich pro Objekt
  - Bereich wird ***angelegt***, beim Eintritt in entsprechende Prozedur
  - Bereich wird ***freigegeben***, beim Verlassen der Prozedur
  - oder auf statischem Speicherbereich (Teil des Kellers)

Statische Datentypen und Listen

Probleme mit statischen Datentypen

- Es werden Datenstrukturen benötigt, deren ***Größen Schwankungen*** unterworfen sind.
- Es sollen ***komplizierte*** Datenstrukturen gebildet werden
  - Listen
  - Bäume
  - Graphen





Lösung

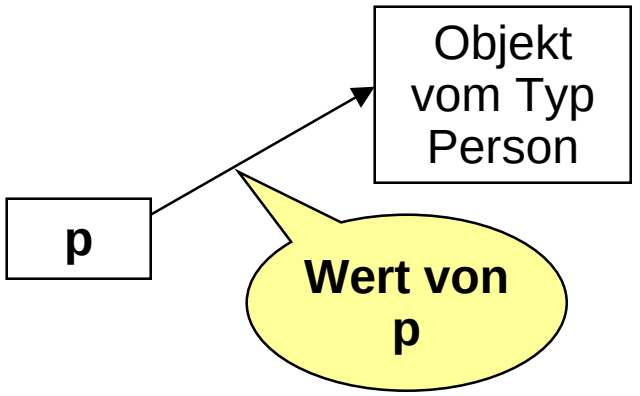
- Dynamische Datentypen
- oder dynamische Variable und Zeigertypen

Zeigertyp - Referenztyp

Zeigertyp

- um einen Zeigertyp zu deklarieren, bietet Modula-3 den ***Typkonstruktor***
- **`<ZeigerTyp> = REF <ReferenzierterTyp>`** an
- damit kann aus ***jedem Typ*** ein Zeigertyp abgeleitet werden

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

TYPE PersonRef = REF Person;

VAR p : PersonRef;
```



- p kann als Werte ***Zeiger auf Objekte*** vom Typ `Person` annehmen
- zeigt p auf kein Objekt, dann wird dies durch den Wert ***`NIL`*** angezeigt
- `NIL` ist Objekt jedes Zeigertyps

Eigenschaften dynamischer Datentypen

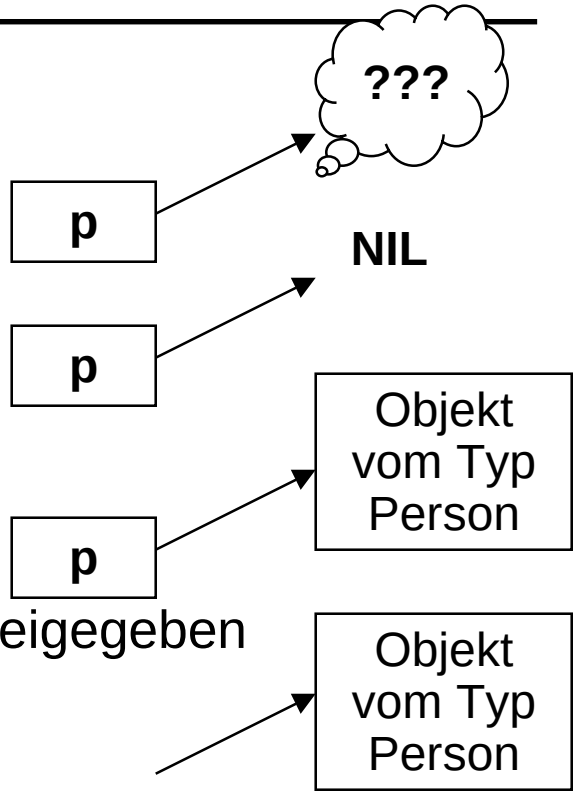
Eigenschaften von Objekten dynamischer Datentypen

- ***Lebensdauer*** ist nicht an die Ausführung einer Prozedur oder Moduls gebunden
- sie werden zur Laufzeit ***explizit (vom Programmierer) erzeugt*** und eventuell auch wieder beseitigt
- sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
- können in prinzipiell ***beliebiger*** Menge geschaffen werden
- haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
- sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
- Zeiger können im Keller oder auf der Halde liegen
- die referenzierten Objekte liegen ***immer*** auf der Halde

Erzeugen von Haldenobjekten

- Beispiel:

```
PROCEDURE PROC …
  TYPE PersonRef = REF Person;
  VAR p : PersonRef;
BEGIN
  p := NIL;
  p := NEW (PersonRef);
END
```



- beim Betreten der Prozedur wird Speicher für p zur Verfügung gestellt und beim Verlassen wieder freigegeben
- durch den Aufruf von `NEW(PersonRef)`
  - wird auf der Halde ***Speicher*** für ein Objekt von Typ Person angelegt
  - die ***Adresse*** dieses Speicherplatzes wird zurückgeliefert und der Variablen p zugewiesen
  - Das Objekt selbst erhält keinen eigenen Bezeichner – man spricht von einer ***dynamischen*** oder auch von einer ***anonymen*** Variablen.
- dieser Speicher wird nicht freigegeben, wenn die Proz. verlassen wird

Dereferenzierung - 1

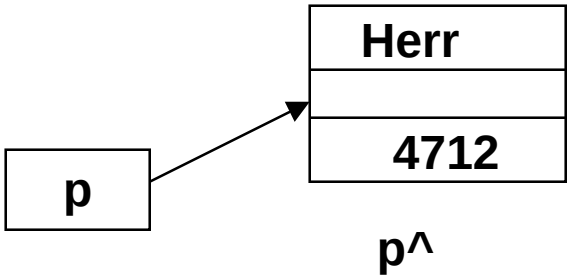
- **Eine gesetzte Referenzvariable verweist auf ein Objekt.**
- **Um das Objekt selbst zu erhalten, müssen wir dem Verweis "nachgehen".**
  - Diese Operation, bei der auf das referenzierte Objekt einer Referenzvariablen zugegriffen wird, heißt ***dereferenzieren***.
- **Dereferenzieren ist, neben dem Erzeugen der referenzierten Objekte, die**
  - zweite charakteristische Operation von Referenztypen.
  - Nur eine ***gesetzte Referenzvariable*** kann dereferenziert werden.
  - Der Versuch einer Dereferenzierung der Referenz `NIL` führt in den meisten Programmiersprachen zum ***Programmabbruch***.

Dereferenzierung - 2

Dereferenzierung

- Zugriff auf ***Werte*** von Zeigerobjekten

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p^.persnr := 4732;

p := NEW (PersonRef);
p^.persnr := 4712;
p^.anrede := AnredeTyp.Herr;

nr := p^.persnr;
```



Laufzeitfehler: Zeigervariable nicht gesetzt

Dereferenzierungsoperator

- In Modula-3 kann der ^-Operator entfallen, wenn ein weiterer Operator folgt (z.B."[ ]", ".")
- Das trägt ***nicht zur Klarheit*** bei !!!
- Darum: Verwenden Sie ***immer*** den ^-Operator !!!

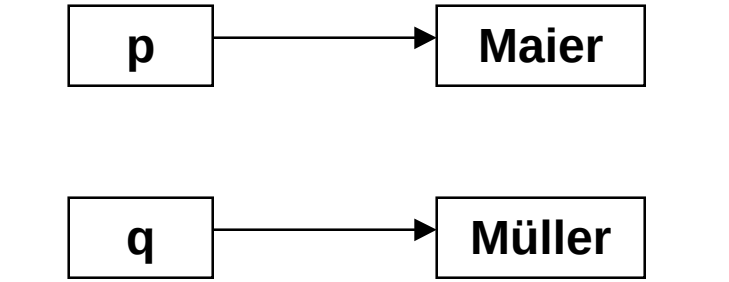
Operationen auf Referenztypen

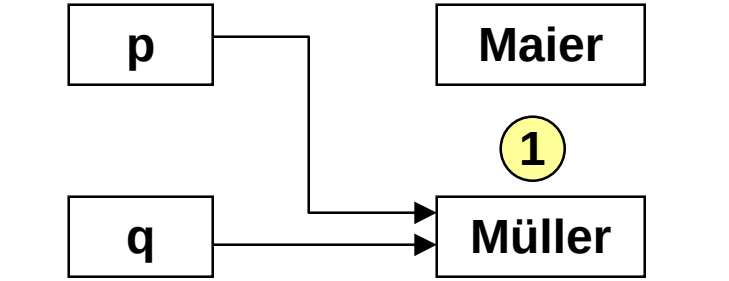
- **Zulässige Werte von Referenztypen**
  - sind Referenzen oder der Wert "***keine Referenz***" (`NIL`).
- **Gesetzte Referenzen haben keine externe Repräsentation.**
  - Entsprechend können sie ***nicht*** ausgegeben oder z.B. in ***Rechenoperationen*** verwendet werden.
- **Die einzigen zulässigen Operationen auf Referenzen sind:**
  - Test auf ***Gleichheit*** oder ***Ungleichheit***,
  - ***Zuweisung*** auf Variablen von kompatiblem Typ.
- **Beispiel in Modula-3:**

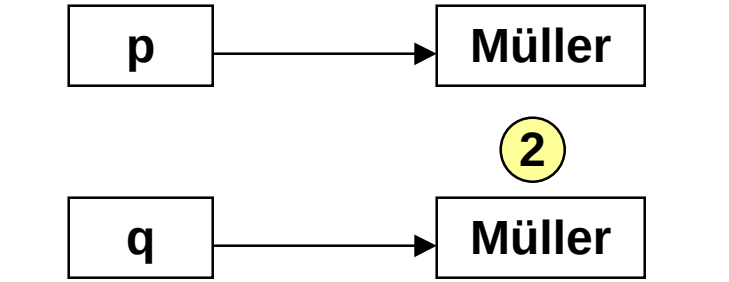
```
VAR p1, p2 : PersonRef;
…
IF p1 = NIL THEN
  p1 := NEW (PersonRef)
END;
p2 := p1;
```

Zuweisung

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
p := NEW(PersonRef);
p^.name.nachname := "Maier";
q := NEW(PersonRef);
q^.name.nachname := "Mueller";

p := q;     (1)

p^ := q^;   (2)
```







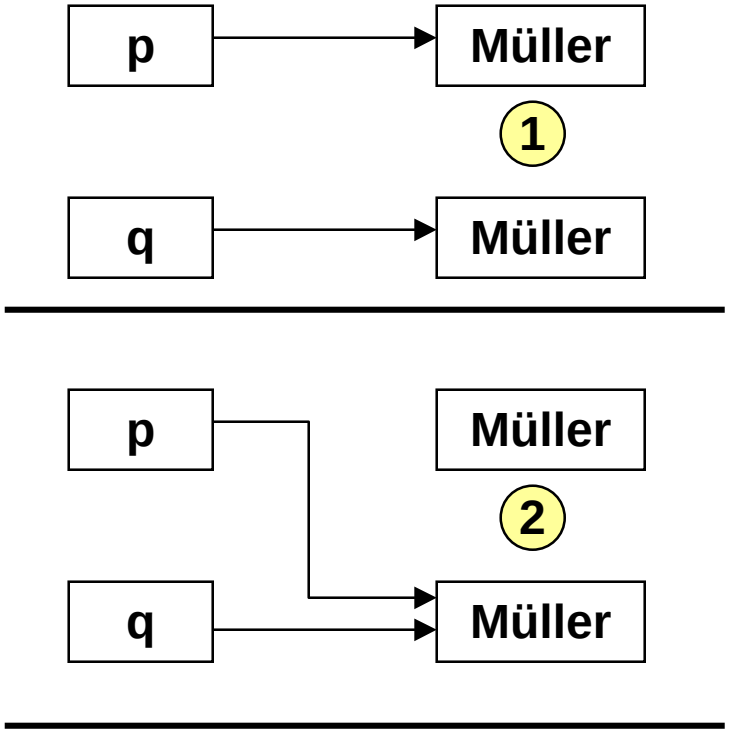
- Der Effekt der Anweisung (1) ist streng zu unterscheiden von der Wirkung der Anweisung (2)

(1) ist eine ***Referenzzuweisung***

(2) ist eine ***Wertzuweisung***

Vergleich

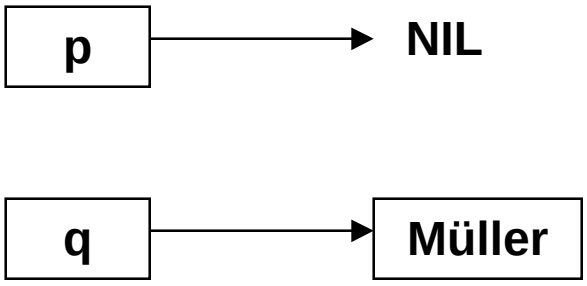
```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;

... p = q ...                    1

... p^ = q^ ...


... p = q ...                    2

... p^ = q^ ...
```

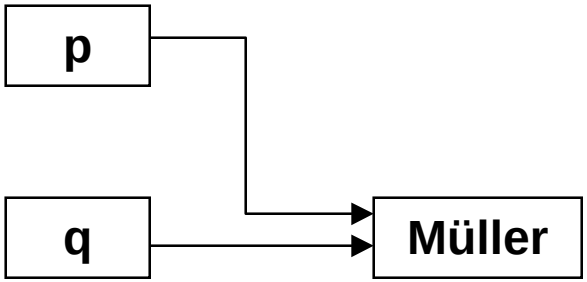


- Zwei Zeiger sind gleich, wenn sie auf ***dasselbe*** referenzierte Objekt (oder: auf die gleiche Adresse) zeigen!

Das Alias-Problem

- **Die Zuweisung bei Referenztypen basiert auf der sog. *Referenzsemantik*:**
  - Der ***Verweis*** auf ein Objekt wird zugewiesen; nicht etwa der Wert des referenzierten Objekts.
  - Dies ist vielfach erwünscht, schafft aber folgendes Problem
- **Mehrere Referenzvariablen können auf dasselbe Objekt verweisen.**
  - Damit ist lokal oft nicht ***entscheidbar***, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben oder nicht.
  - Dies ist das ***Alias-Problem***, das bei allen Referenztypen auftritt.

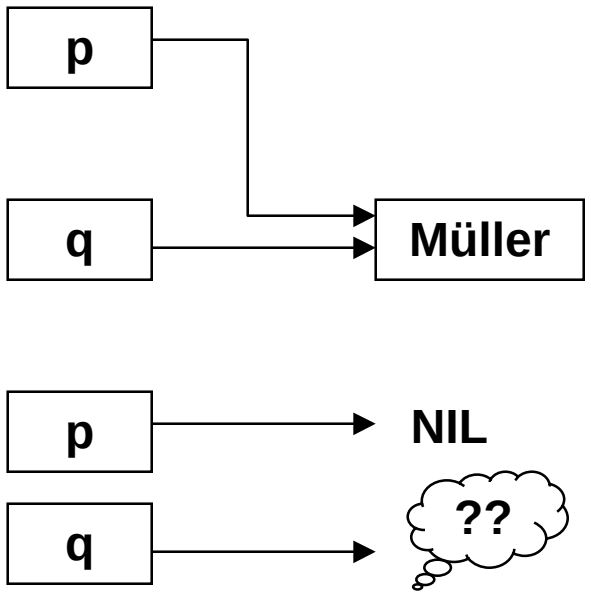




Freigeben von Zeigerobjekten

- **In der Regel müssen Zeigerobjekte vom Programmierer kontrolliert werden:**
  - Er muß sie explizit ***anlegen*** und ***freigeben***
- **Beim Freigeben können folgende Fehlersituationen auftreten**
  - Nicht mehr benötigte Zeigerobjekte wurden ***nicht frei*** gegeben
    - Es wird Speicher verschwendet
  - Es werden Zeigerobjekte freigegeben, die noch ***benötigt*** werden
    - Problem der "***dangling references***"

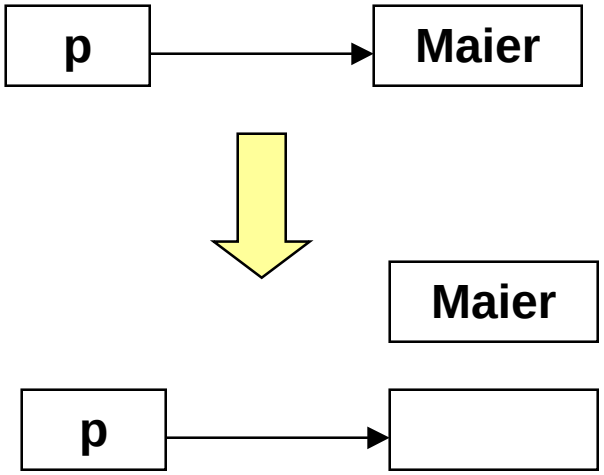
```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
...
p := q;
DISPOSE(p);
q^.anrede := ...
```



Probleme mit Referenzvariablen

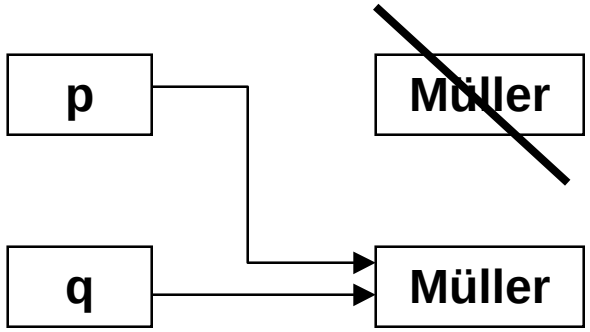
- **"Lost Object":**
  - Es kann dynamisch angelegte Objekte geben, auf die keine ***Referenzvariable*** mehr verweist.
  - Dieses Objekt kann nicht mehr erreicht werden und wird als ***Garbage*** bezeichnet.
- **Häufige Fehlersituation:**

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p := NEW (PersonRef);
p^.name := "Maier";
...
p := NEW (PersonRef);
```



Automatische Speicherbereinigung

- **Einige Laufzeitumgebungen von Sprachen können Zeigerobjekte kontrollieren**
  - sie geben Zeigerobjekte, die ***nicht mehr erreicht*** werden können, automatisch frei
  - "***Garbage Collector***" (Müllsammler)



- **Diskussion Garbage Collector**
  - ***Vorteile*** (wenn keine explizite Freigabe vorhanden oder genutzt)
    - Fehlerklasse "dangling reference" ausgeschaltet
  - ***Nachteile***
    - Müllsammeln kostet Zeit

Dynamische Datenstrukturen

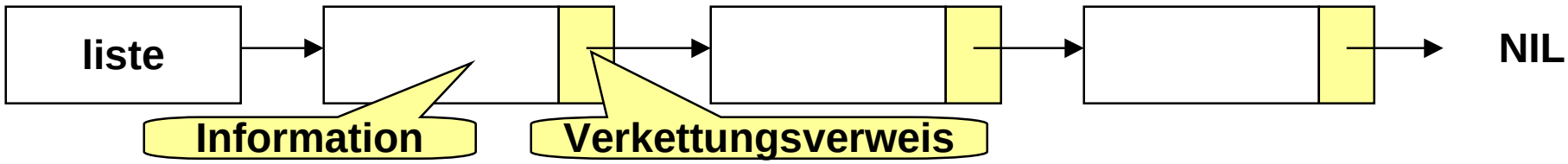
- **Ziel:**
  - Definition dynamischer Datenstrukturen, die ***einfach*** vom Programmierer erstellt, verwaltet und kontrolliert werden können
- **Referenzvariablen**
  - können auf zusammengesetzte Datenstrukturen verweisen, die selbst wieder ***Referenzobjekte*** enthalten. Wenn diese Strukturen rekursiv sind, sprechen wir von ***Verweisketten***, die den Aufbau dynamischer Datenstrukturen ermöglichen.
- **Beispiel: Einfach verkettete lineare Liste**
  - Elemente der Liste sind ***Zeigerobjekte***
  - Jedes Listenobjekt ist so konstruiert, das es ***selbst*** wieder auf ein Listenobjekt ***verweisen*** kann
  - zusätzlich enthält es die ***eigentlichen*** Informationen

Lineare Liste

```
TYPE Jahr = [1890 .. 1998];

TYPE Person = RECORD
  name, vorname : TEXT;
  geburtsjahr : Jahr;
END;

TYPE ListenElement =
  RECORD
    person : Person;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : RListenElement;
```

Typ, für die in der Liste verwaltete Information

"Anker", um auf die Liste zugreifen zu können



Operationen auf linearen Listen - 1

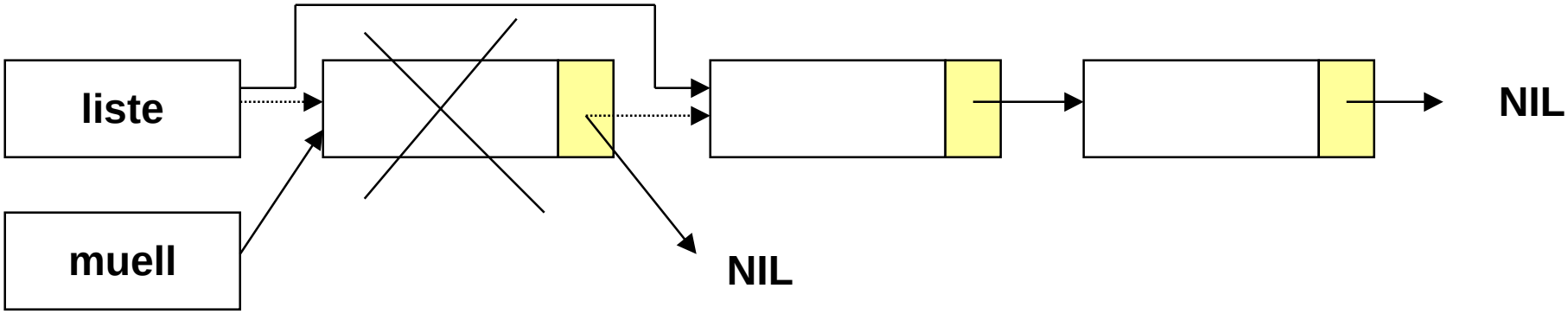
```
PROCEDURE EinfuegeVorne (VAR liste : RListenElement;
                             pers  : Person) =
VAR neu : RListenElement;
BEGIN
  neu := NEW(RListenElement);
  neu^.person := pers;
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegeVorne;
```

Operationen auf linearen Listen - 2

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```

Operationen auf linearen Listen - 3

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
(* Mit expliziter Speicherfreigabe *)

VAR muell : RListenElement;
BEGIN
  IF liste # NIL THEN
    muell := liste;
    liste := liste^.nachfolger;
    muell^.nachfolger := NIL;
    DISPOSE(muell);
  END;
END LoescheVorne;
```



Suchen in linearen Listen - 1

```
PROCEDURE Suche (liste : RListenElement;
                 was : Person): RListenElement =
VAR position : RListenElement;
    gefunden : BOOLEAN := FALSE;

BEGIN
  position := liste;
  WHILE (position # NIL AND NOT gefunden) DO
    IF was = position^.person THEN
      gefunden := TRUE;
    ELSE
      position := position^.nachfolger;
    END;
  END;
  RETURN position;
END Suche;
```

Sequentielles Suchen

Ergebnis : Zeiger auf das gefundene Element sonst NIL

Suchen in linearen Listen - 2

- **Listen sind rekursive Datenstrukturen**
  - ***Rekursive Datenstrukturen*** können *elegant* mit ***rekursiven Prozeduren*** bearbeitet werden!

```
PROCEDURE SucheRek (liste : RListenElement;
                    was : Person): RListenElement =
BEGIN
  IF liste # NIL THEN
    IF was = liste^.person THEN
        RETURN liste;
    ELSE
      RETURN SucheRek (liste^.nachfolger, was);
    END;
  ELSE
    RETURN NIL;
  END;
END SucheRek;
```

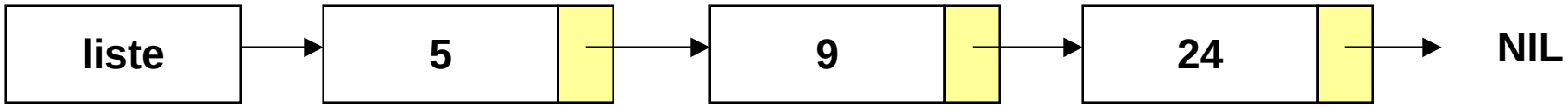
Sortierte lineare Liste

```
TYPE ListenElement =
  RECORD
    wert : INTEGER;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : ListenElement;
```

Für alle Elemente x der Liste gilt:

`x^.wert <= x^.nachfolger^.wert`

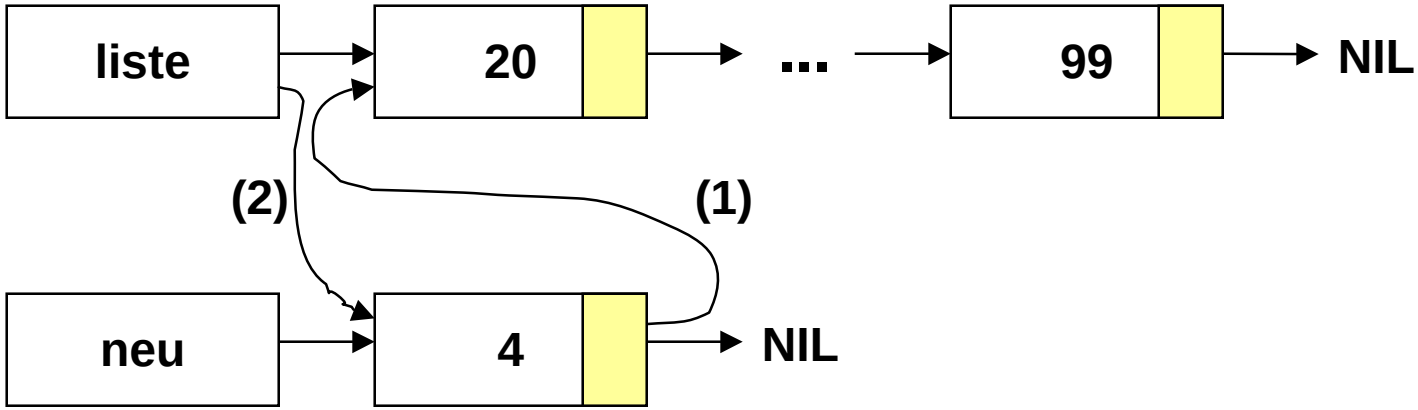
Liste soll immer im Zustand "aufsteigend" sortiert sein



Einfügen in eine sortierte Liste - 1

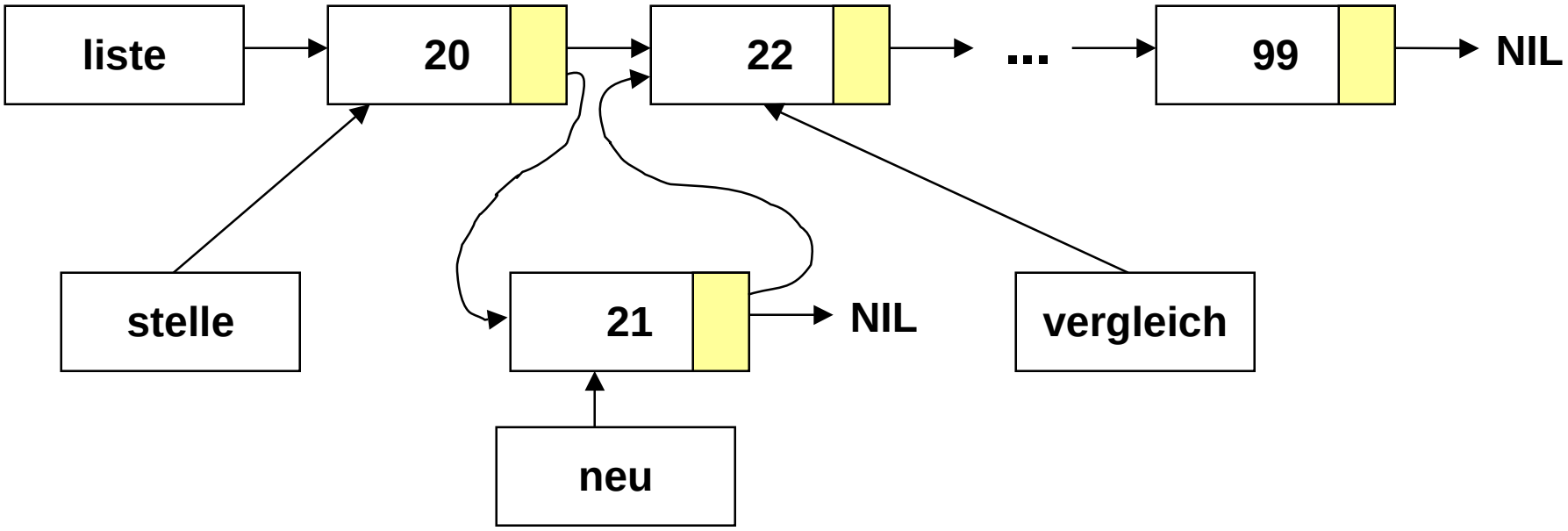
- **Fall 1: Die Liste ist leer.**

- **Fall 2: Element muß vorne eingefügt werden**



Einfügen in eine sortierte Liste - 2

- **Fall 3: Element muß sonst irgendwo eingefügt werden.**
  - Es muß nach der ***Einfügestelle*** gesucht werden
  - Damit man auf die Elemente vor und hinter der Einfügestelle zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das Element hinten angefügt werden muß !!

Einfügen: Lösung A -1

```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, einfuegestelle : RListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                    (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN           (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);          (* neu vorne einhaengen *)
  ELSE
    einfuegestelle := SucheEinfuegestelle(liste, w);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Einfügen: Lösung A -2

```
PROCEDURE ErzeugeNeuesListenelement(w : INTEGER): RListenElement =
VAR elem : RListenElement;
BEGIN
  elem := NEW(RListenElement);
  elem^.wert := w;
  elem^.nachfolger := NIL;
  RETURN elem
END ErzeugeNeuesListenelement;
```

```
PROCEDURE EinfuegenVorne(VAR liste : RListenElement;
                         VAR neu   : RListenElement)=
BEGIN
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegenVorne;
```

Einfügen: Lösung A -3

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                                    w : INTEGER): RListenElement =
VAR stelle, vergleich: ListenElement;
BEGIN
  vergleich := liste;
  stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

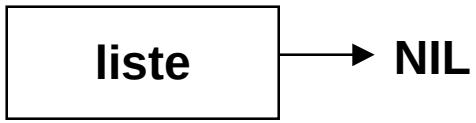
```
PROCEDURE FuegeAnEinfuegestelleEin(VAR neu   : RListenElement;
                                   VAR stelle: RListenElement)=
BEGIN
  neu^.nachfolger := stelle^.nachfolger;
  stelle^.nachfolger := neu;
END FuegeAnEinfuegestelleEin;
```

Einfügen: Lösung B

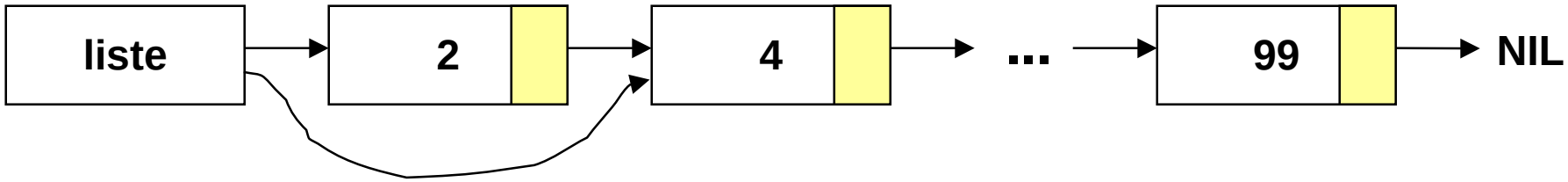
```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, position, vorgaenger : RListenElement;
BEGIN
  neu := NEW(RListenElement);
  neu^.wert := w;
  neu^.nachfolger := NIL;

  IF liste = NIL THEN              (* liste ist leer *)
    liste := neu;
  ELSIF (w <= liste^.wert) THEN    (* w ist kleinstes Element*)
    neu^.nachfolger := liste;      (* w vorne einhaengen *)
    liste := neu;
  ELSE                             (* Einfuegestelle suchen *)
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert <= w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                           (* vorgaenger = Einfuegestelle *)
    neu^.nachfolger := position;   (* w einhaengen *)
    vorgaenger^.nachfolger := neu;
  END;
END Einfuegen;
```

Löschen in einer sortierten Liste - 1

- **Fall 1: Die Liste ist leer.**

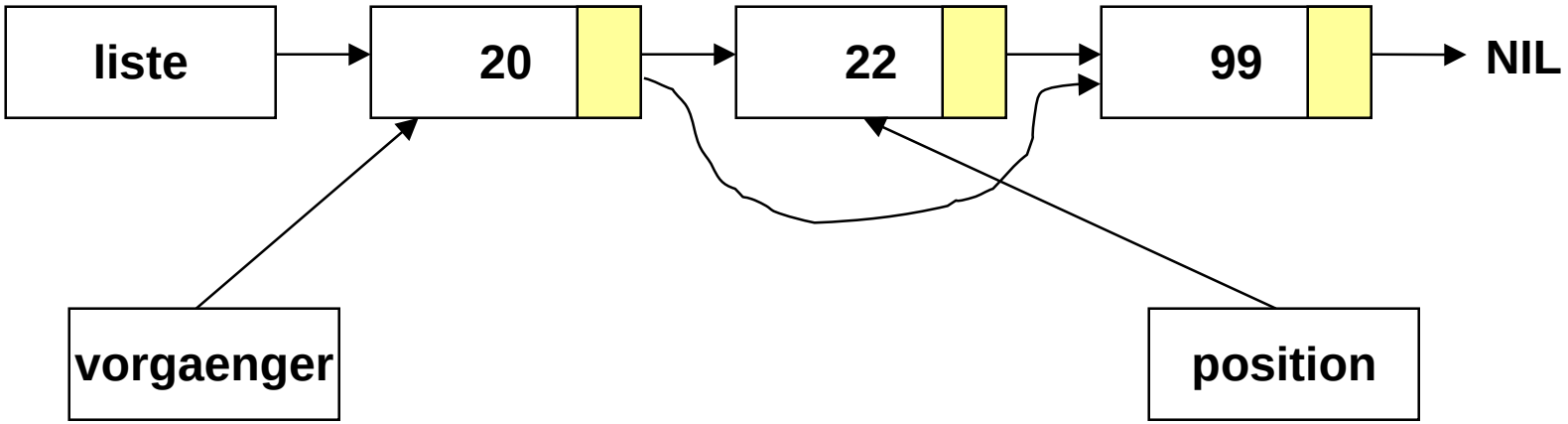


- **Fall 2: Das erste Element muß gelöscht werden**



Löschen in einer sortierten Liste - 2

- **Fall 3: Element muß sonst irgendwo gelöscht werden.**
  - Damit man auf die Elemente vor und hinter dem zu löschenden Element zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das letzte Element gelöscht werden muß !!

Löschen: Lösung A

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert # w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* position = NIL v position^.wert = w *)
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        liste := position^.nachfolger;
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Löschen: Lösung B -1

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position);
    gefunden := (position # NIL);  (* position= NIL v position^.wert = w *)
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Löschen: Lösung B -2

```
PROCEDURE Suche (liste : RListenElement;
                 w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```

Beobachtung !

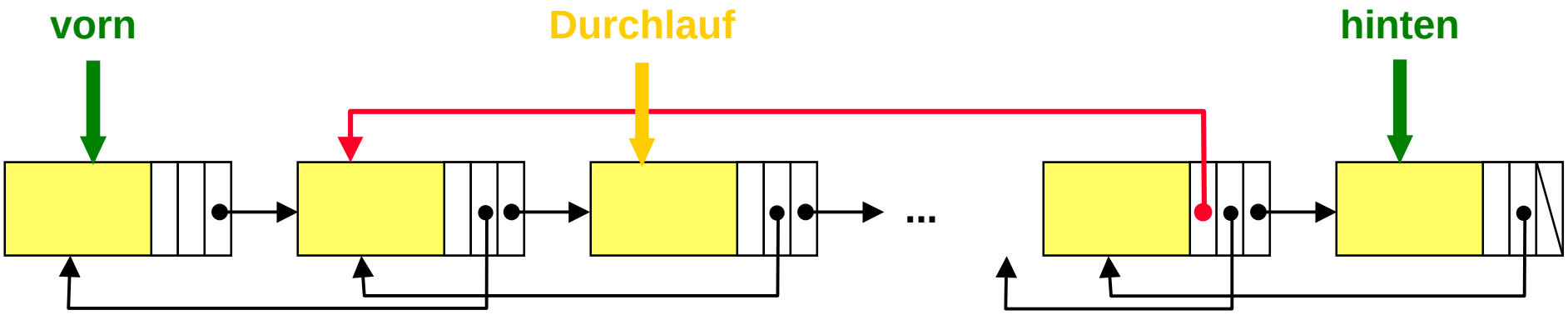
- **Einfügen und Loeschen benutzen ähnliche Such-Prozeduren**

```
PROCEDURE Suche (liste : RListenElement; w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste; vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                                  w : INTEGER): RListenElement =
VAR stelle, vergleich: RListenElement;
BEGIN
  vergleich := liste; stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

Zweck der Verwendung von Zeigern

- Anker
- Vernetzung
- Durchlauf
- Abkürzung für Durchlauf
- semantische Beziehung



Prozedurtyp und Prozeduren dieses Typs

- **Frage:**
  - Gibt es eine Möglichkeit, Prozeduren so zu ***parametrisieren***, daß als Parameter ein Algorithmus (Prozedur) übergeben werden kann?
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur.***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.
  - Prozedurvariablen können ***passende*** Prozeduren zugewiesen werden.
  - Prozedurvariablen können als ***Parameter*** übergeben werden.
  - Gesetzte Prozedurvariablen können in Anweisungen mit ***aktuellen Parametern*** aufgerufen werden.

Beispiel: ProzedurTyp - 1

```
TYPE  Bereich      = [-10 .. 10]; Index   = [1 .. 10];
      Zahlenmenge  = SET OF Bereich;
      Zahlenfeld   = ARRAY Index OF Bereich;
      TestProzedur = PROCEDURE (a : INTEGER) : BOOLEAN;

PROCEDURE IstPositiv (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i > 0);
END IstPositiv;

PROCEDURE IstNegativ (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i < 0);
END IstNegativ;

PROCEDURE Filtern (        feld : FeldTyp;
                   VAR resultat : Zahlenmenge;  p : TestProzedur) =
BEGIN
  FOR i := FIRST(Index) TO LAST(Index) DO
    IF p(feld[i]) THEN
      resultat := resultat + Zahlenmenge{feld[i]};
    END;
  END;
END Filtern;
```

Gegeben ist ein Feld mit Zahlen. Bestimme die darin enthaltenen positiven und negativen Zahlen

Signatur-konforme Prozeduren

Prozedur-parameter

Beispiel: ProzedurTyp - 2

```
VAR proc : TestProzedur;

    werte := Zahlenfeld{3, -5, 9, 8, -6, 9, -6, -1, -1, 5};
    positiv, negativ := Zahlenmenge{};

BEGIN
  proc := IstPositiv;
  Filtern (werte, positiv, proc);
  proc := IstNegativ;
  Filtern (werte, negativ, proc);

  SIO.PutLine ("Positive Zahlen:");
  FOR e := FIRST(Bereich) TO LAST(Bereich) DO
    IF e IN positiv THEN SIO.PutInt(e); SIO.PutText(", "); END;
  END;
  SIO.Nl();
```

Prozedurvariable

Zuweisung einer Prozedur an die Prozedurvariable

Zuweisung an Prozedurvariable

- **Zuweisung**
  - `TYPE PType = PROCEDURE …`
    `VAR proc : Ptype`

    `proc := PT; (* Ausdruck der vom Typ PT ist *)`
  - Diese Zuweisung ist korrekt, wenn
    - PT = ***NIL*** ist (NIL ist mit jedem Wert eines Prozedurtyps kompatibel)
    - ***Anzahl*** der Parameter und deren ***Typen*** sind gleich für PT und PType
    - Beide haben den gleichen ***Ergebnistyp*** oder keinen

```
TYPE TestProzedur = PROCEDURE (a : INTEGER)  : BOOLEAN;
     PT1          = PROCEDURE (in : INTEGER) : BOOLEAN;

VAR test : TestProzedur;
      p1 : PT1;
…
p1 := test;
test := p1;
```

Typen sind signatur-konform

Verwendung Prozedurtyp - 1

```
TYPE Vergleichsoperation = PROCEDURE (a, b : INTEGER) : BOOLEAN;

PROCEDURE Suche (liste : ListenElement;
                     w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement;
                    op : Vergleichsoperation) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND  NOT op(stelle^.wert, w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

signaturkonform

```
PROCEDURE IstGleich
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a = b;
END IstGleich;
```

```
PROCEDURE Groesser
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a > b;
END Groesser;
```

Verwendung Prozedurtyp - 2

```
PROCEDURE Loeschen (VAR liste : ListenElement; w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position, IstGleich);
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Verwendung Prozedurtyp - 3

```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, einfuegestelle, groessererWert : ListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                    (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN           (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);          (* neu vorne einhaengen *)
  ELSE
    Suche(liste, w, einfuegestelle, groessererWert , Groesser);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Diskussion: Dynamische Datentypen

- **Je deutlicher die Realisierung dynamischer Datenstrukturen durch Zeiger in der Sprache sichtbar ist,**
  - desto leichter fällt die softwaretechnisch ***unsaubere*** Verwendung.
- **Die explizite Speicherverwaltung führt zu einigen Problemen.**
  - Moderne imperative Sprachen sollten daher auf explizites Löschen verzichten und hierfür einen ***Garbage Collector*** besitzen.
- **Erst in objektorientierten Sprachen,**
  - die vorrangig mit ***Referenzsemantik*** arbeiten,
  - können grundlegende Probleme der Referenzierung gelöst werden.
- **Prozedurtypen sind nicht dynamisch, aber Prozedurvariablen können dynamisch unterschiedlichen Prozedurobjekten zugewiesen werden**

Was haben wir gelernt?

- Charakterisierung statischer Datentypen (Wiederholung) und Defizit für Listenverarbeitung
- Typen für Listenelemente, Zeigertypen, dynamische Objekte (Haldenobjekte)
- dynamische Datenobjekte: Explizite Erzeugung, implizite Bezeichnung, Lebensdauer, Halde (Heap), Freigabe
- Derefenzieren, Lesen und Setzen von Haldenobjekten, Setzen von Zeigerobjekten, Vergleich
- Aliasing, nicht mehr greifbare Haldenobjekte, hängende Zeiger, Garbage Collection
- Lineare Liste: Einfügen/Löschen (vorn und in der Mitte), Suchen, geordnete lineare Liste
- Arten von Zeigern (Zweck ihrer Verwendung)
- Prozedurobjekt, Prozedurtyp, Prozedurvariable, Einsatz für Flexibilität

Glossar

- Wertsemantik, Verweis (Zeiger) semantik
- statische Datentypen, dynamische Datentypen, rekursive Datentypen
- Zeigertyp, Zeigerobjekt (-konstante oder -variable)
- Haldenobjekttyp, dynamische (anonyme) Datenobjekte: Haldenobjekte, Erzeugung mit Einrichtung eines Zeigerwerts, Löschen mit Löschen des Zeigerwerts
- statische Datenobjekte, Datenobjekte auf dem Kellerspeicher, Datenobjekte auf der Halde
- Zeiger: Dereferenzieren, Setzen, Lesen, Vergleichen, Setzen/Lesen von Komponenten von Haldenobjekten
- Aliasing, inaccessible objects, dangling references
- Ankerzeiger, Verkettungszeiger, Durchlaufzeiger, „semantische“ Zeiger, Abkürzung von Zugriffswegen
- einfach verkettete lineare Liste, sortierte Liste, doppelt verkettete Liste
- Listenoperationen: Einfügen vorn, in der Mitte, Suchen, Löschen vorn, in der Mitte, Suche als allgemeine Prozedur für Listenoperationen
- Prozedurtyp und Parameterprofil, signaturkonforme Prozedurobjekte, Prozedurvariable und Prozedurobjektzuweisung, Lebensdauer von Zeigerobjekten, von über Zeiger zugegriffenen Haldenobjekten

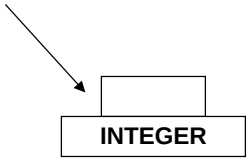
Datentypen II

- **Dynamische Datentypen**
  - Zeigertypen
  - dynamische Datenstrukturen
- **Anwendungsbeispiele**
  - lineare Liste
  - sortierte Liste
- **Prozedurtyp**

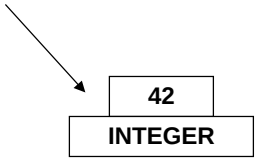
Bezeichner und Objekte

- **Wir haben als elementares imperatives Konzept die *Variable* kennengelernt.**
- **Abarbeitung der Deklaration:**
  - Ein Name dient als ***Bezeichner***, der mit einem ***getypten Wert*** verbunden werden kann (Struktur, Speicherplatz).
- **Bei den bisher betrachteten Datentypen**
  - wurde der Zusammenhang von Bezeichner und Wert implizit in der Sprache durch den *Zuweisungsmechanismus* hergestellt.

```
VAR Antwort: INTEGER;
```

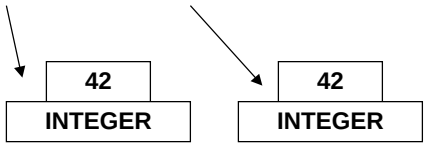



```
Antwort := 42;
```

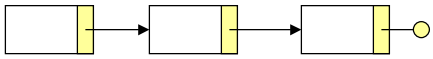
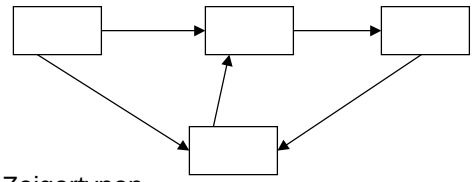


Wertsemantik

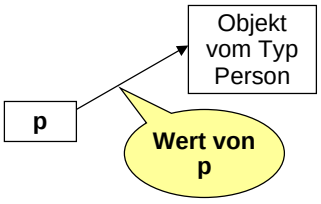
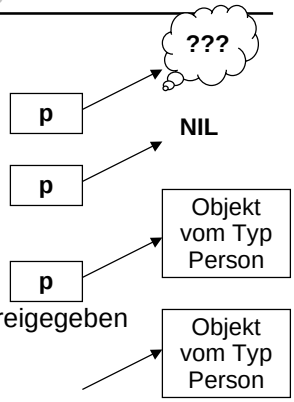
- **Die bisher betrachteten Variablen und die Zuweisung basieren auf der *Wertsemantik*:**
  - Eine Variable hat einen ***definierten*** oder ***undefinierten*** Wert.
  - Bei der ***Zuweisung*** wird der Wert des Ausdrucks der rechten Seite (rhv) an die Variable der linken Seite (lhv) zugewiesen.
  - Eine ***Identität*** von Objekten wird nicht hergestellt.
  - ```
    VAR Antwort1, Antwort2: INTEGER;
    …
    Antwort1 := 42; (* 1 *)
    Antwort2 := Antwort1; (* 2 *)
    Antwort1 := 24; (* 3 *)

    Antwort1 := Antwort2;
    ```



Statische Datentypen

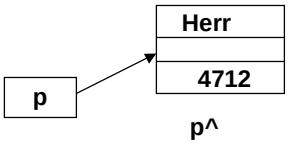
- **Kennzeichnend für imperative Sprachen ist, daß *Wertsemantik* und *statische Datentypen* zusammenfallen:**
  - Die Zuordnung von Bezeichner und Wert geschieht über die Zuweisung.
  - Die Variablen eines statischen Typs ***behalten ihre Struktur*** während ihrer Lebensdauer bei.
  - Der ***Speicher*** einer statisch getypten Variablen wird bei der Übersetzung anhand der Deklaration ***festgelegt*** und bei der "ersten Verwendung implizit angelegt ".
- **Speicherbereiche für statisch getypte Variablen wird im sog. *Kellerspeicher* reserviert**
  - zusammenhängender Bereich pro Objekt
  - Bereich wird ***angelegt***, beim Eintritt in entsprechende Prozedur
  - Bereich wird ***freigegeben***, beim Verlassen der Prozedur
  - oder auf statischem Speicherbereich (Teil des Kellers)
    ```

Statische Datentypen und Listen

- **Probleme mit statischen Datentypen**
  - Es werden Datenstrukturen benötigt, deren ***Größen Schwankungen*** unterworfen sind.
  - Es sollen ***komplizierte*** Datenstrukturen gebildet werden
    - Listen
    - Bäume
    - Graphen





- **Lösung**
  - Dynamische Datentypen
  - oder dynamische Variable und Zeigertypen

Zeigertyp - Referenztyp

- **Zeigertyp**
  - um einen Zeigertyp zu deklarieren, bietet Modula-3 den ***Typkonstruktor***
  - **`<ZeigerTyp> = REF <ReferenzierterTyp>`** an
  - damit kann aus ***jedem Typ*** ein Zeigertyp abgeleitet werden

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

TYPE PersonRef = REF Person;

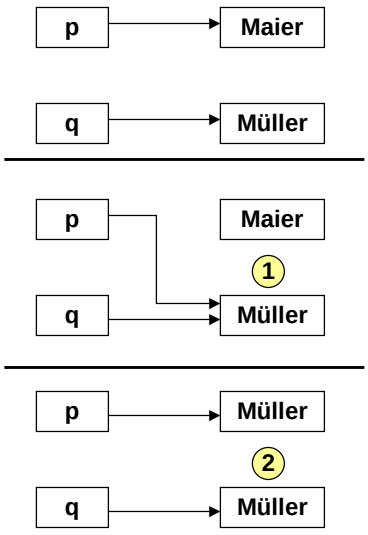
VAR p : PersonRef;
```



  - p kann als Werte ***Zeiger auf Objekte*** vom Typ `Person` annehmen
  - zeigt p auf kein Objekt, dann wird dies durch den Wert ***`NIL`*** angezeigt
  - `NIL` ist Objekt jedes Zeigertyps

Eigenschaften dynamischer Datentypen

Eigenschaften von Objekten dynamischer Datentypen

- ***Lebensdauer*** ist nicht an die Ausführung einer Prozedur oder Moduls gebunden
- sie werden zur Laufzeit ***explizit (vom Programmierer) erzeugt*** und eventuell auch wieder beseitigt
- sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
- können in prinzipiell ***beliebiger*** Menge geschaffen werden
- haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
- sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
- Zeiger können im Keller oder auf der Halde liegen
- die referenzierten Objekte liegen ***immer*** auf der Halde

Erzeugen von Haldenobjekten

- Beispiel:

```
PROCEDURE PROC …
  TYPE PersonRef = REF Person;
  VAR p : PersonRef;
BEGIN
  p := NIL;
  p := NEW (PersonRef);
END
```



- beim Betreten der Prozedur wird Speicher für p zur Verfügung gestellt und beim Verlassen wieder freigegeben
- durch den Aufruf von `NEW(PersonRef)`
  - wird auf der Halde ***Speicher*** für ein Objekt von Typ Person angelegt
  - die ***Adresse*** dieses Speicherplatzes wird zurückgeliefert und der Variablen p zugewiesen
  - Das Objekt selbst erhält keinen eigenen Bezeichner – man spricht von einer ***dynamischen*** oder auch von einer ***anonymen*** Variablen.
- dieser Speicher wird nicht freigegeben, wenn die Proz. verlassen wird

Dereferenzierung - 1

- **Eine gesetzte Referenzvariable verweist auf ein Objekt.**
- **Um das Objekt selbst zu erhalten, müssen wir dem Verweis "nachgehen".**
  - Diese Operation, bei der auf das referenzierte Objekt einer Referenzvariablen zugegriffen wird, heißt ***dereferenzieren***.
- **Dereferenzieren ist, neben dem Erzeugen der referenzierten Objekte, die**
  - zweite charakteristische Operation von Referenztypen.
  - Nur eine ***gesetzte Referenzvariable*** kann dereferenziert werden.
  - Der Versuch einer Dereferenzierung der Referenz `NIL` führt in den meisten Programmiersprachen zum ***Programmabbruch***.

Dereferenzierung - 2

- **Dereferenzierung**
  - Zugriff auf ***Werte*** von Zeigerobjekten

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p^.persnr := 4732;

p := NEW (PersonRef);
p^.persnr := 4712;
p^.anrede := AnredeTyp.Herr;

nr := p^.persnr;
```



Laufzeitfehler: Zeigervariable nicht gesetzt

Dereferenzierungsoperator

  - In Modula-3 kann der ^-Operator entfallen, wenn ein weiterer Operator folgt (z.B."[ ]", ".")
  - Das trägt ***nicht zur Klarheit*** bei !!!
  - Darum: Verwenden Sie ***immer*** den ^-Operator !!!

Operationen auf Referenztypen

- **Zulässige Werte von Referenztypen**
  - sind Referenzen oder der Wert "***keine Referenz***" (`NIL`).
- **Gesetzte Referenzen haben keine externe Repräsentation.**
  - Entsprechend können sie ***nicht*** ausgegeben oder z.B. in ***Rechenoperationen*** verwendet werden.
- **Die einzigen zulässigen Operationen auf Referenzen sind:**
  - Test auf ***Gleichheit*** oder ***Ungleichheit***,
  - ***Zuweisung*** auf Variablen von kompatiblem Typ.
- **Beispiel in Modula-3:**

```
VAR p1, p2 : PersonRef;
…
IF p1 = NIL THEN
  p1 := NEW (PersonRef)
END;
p2 := p1;
```


Zuweisung

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
p := NEW(PersonRef);
p^.name.nachname := "Maier";
q := NEW(PersonRef);
q^.name.nachname := "Mueller";

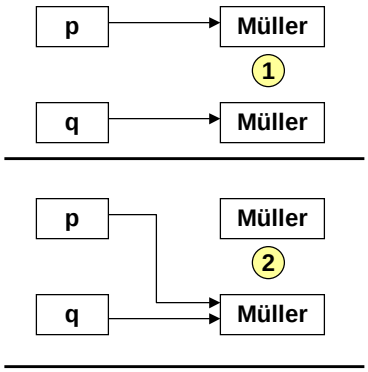
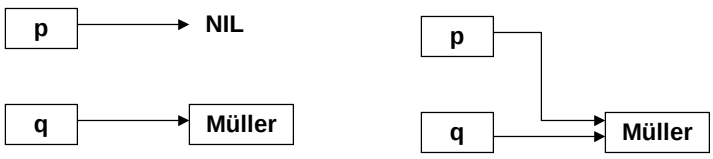
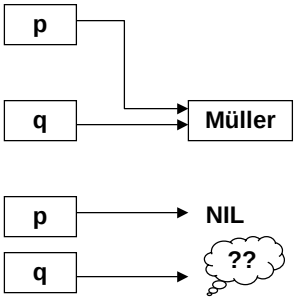
p := q;     (1)

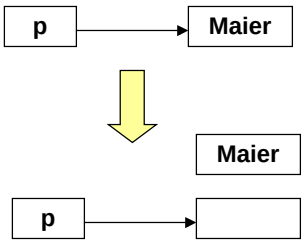
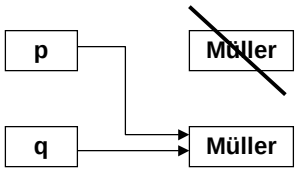
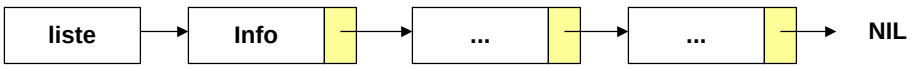
p^ := q^;   (2)
```



- Der Effekt der Anweisung (1) ist streng zu unterscheiden von der Wirkung der Anweisung (2)

(1) ist eine ***Referenzzuweisung***

(2) ist eine ***Wertzuweisung***

Vergleich

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;

... p = q ...                    1

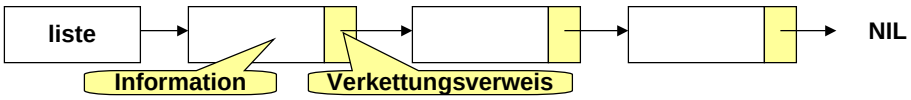
... p^ = q^ ...

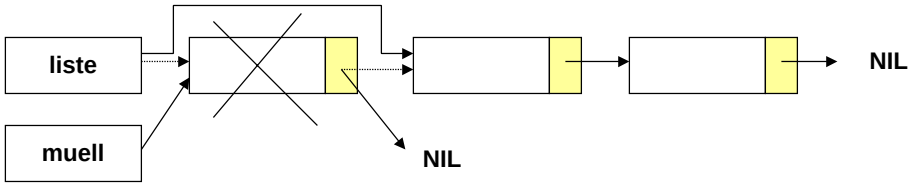
... p = q ...                    2

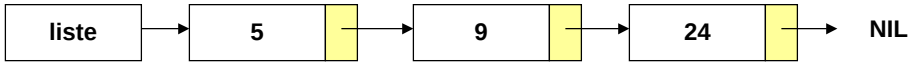
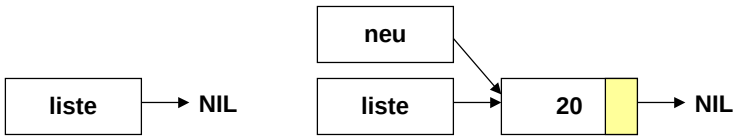
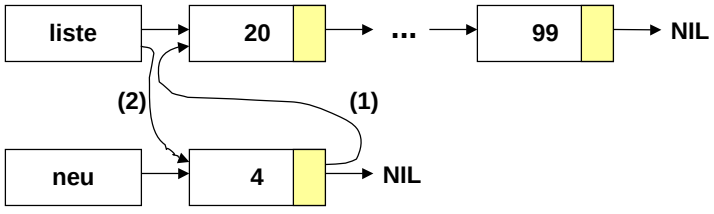
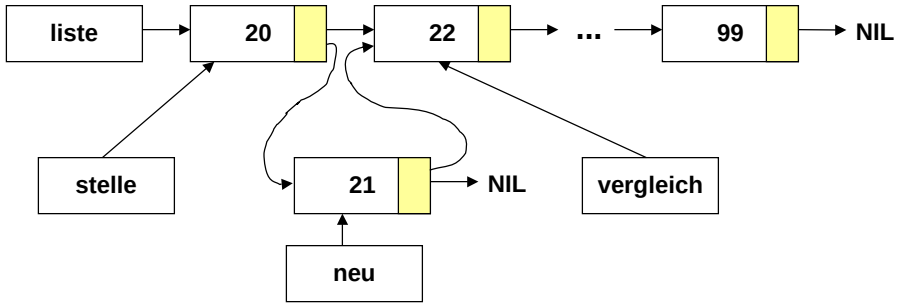
... p^ = q^ ...
```



- Zwei Zeiger sind gleich, wenn sie auf ***dasselbe*** referenzierte Objekt (oder: auf die gleiche Adresse) zeigen!

Das Alias-Problem

- **Die Zuweisung bei Referenztypen basiert auf der sog. *Referenzsemantik*:**
  - Der ***Verweis*** auf ein Objekt wird zugewiesen; nicht etwa der Wert des referenzierten Objekts.
  - Dies ist vielfach erwünscht, schafft aber folgendes Problem
- **Mehrere Referenzvariablen können auf dasselbe Objekt verweisen.**
  - Damit ist lokal oft nicht ***entscheidbar***, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben oder nicht.
  - Dies ist das ***Alias-Problem***, das bei allen Referenztypen auftritt.



Freigeben von Zeigerobjekten

- **In der Regel müssen Zeigerobjekte vom Programmierer kontrolliert werden:**
  - Er muß sie explizit ***anlegen*** und ***freigeben***
- **Beim Freigeben können folgende Fehlersituationen auftreten**
  - Nicht mehr benötigte Zeigerobjekte wurden ***nicht frei*** gegeben
    - Es wird Speicher verschwendet
  - Es werden Zeigerobjekte freigegeben, die noch ***benötigt*** werden
    - Problem der "***dangling references***"

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
...
p := q;
DISPOSE(p);
q^.anrede := ...
```



Probleme mit Referenzvariablen

- **"Lost Object":**
  - Es kann dynamisch angelegte Objekte geben, auf die keine ***Referenzvariable*** mehr verweist.
  - Dieses Objekt kann nicht mehr erreicht werden und wird als ***Garbage*** bezeichnet.
- **Häufige Fehlersituation:**

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

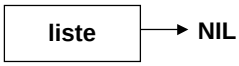
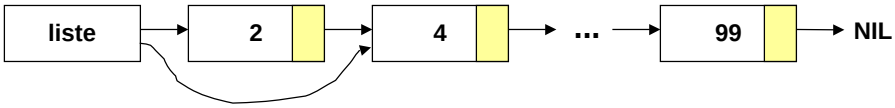
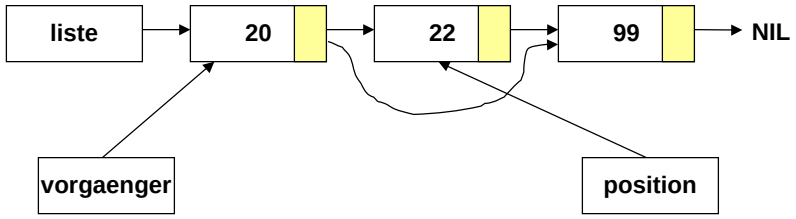
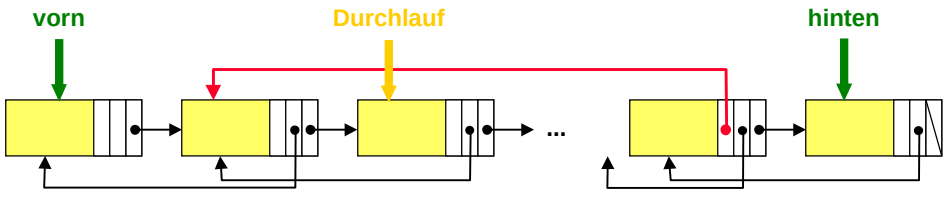
p := NEW (PersonRef);
p^.name := "Maier";
...
p := NEW (PersonRef);
```



Automatische Speicherbereinigung

- **Einige Laufzeitumgebungen von Sprachen können Zeigerobjekte kontrollieren**
  - sie geben Zeigerobjekte, die ***nicht mehr erreicht*** werden können, automatisch frei
  - "***Garbage Collector***" (Müllsammler)



- **Diskussion Garbage Collector**
  - ***Vorteile*** (wenn keine explizite Freigabe vorhanden oder genutzt)
    - Fehlerklasse "dangling reference" ausgeschaltet
  - ***Nachteile***
    - Müllsammeln kostet Zeit

Dynamische Datenstrukturen

- **Ziel:**
  - Definition dynamischer Datenstrukturen, die ***einfach*** vom Programmierer erstellt, verwaltet und kontrolliert werden können
- **Referenzvariablen**
  - können auf zusammengesetzte Datenstrukturen verweisen, die selbst wieder ***Referenzobjekte*** enthalten. Wenn diese Strukturen rekursiv sind, sprechen wir von ***Verweisketten***, die den Aufbau dynamischer Datenstrukturen ermöglichen.
- **Beispiel: Einfach verkettete lineare Liste**
  - Elemente der Liste sind ***Zeigerobjekte***
  - Jedes Listenobjekt ist so konstruiert, das es ***selbst*** wieder auf ein Listenobjekt ***verweisen*** kann
  - zusätzlich enthält es die ***eigentlichen*** Informationen



Lineare Liste

```
TYPE Jahr = [1890 .. 1998];

TYPE Person = RECORD
  name, vorname : TEXT;
  geburtsjahr : Jahr;
END;

TYPE ListenElement =
  RECORD
    person : Person;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : RListenElement;
```

Typ, für die in der Liste verwaltete Information

"Anker", um auf die Liste zugreifen zu können



Operationen auf linearen Listen - 1

```
PROCEDURE EinfuegeVorne (VAR liste : RListenElement;
                             pers  : Person) =
VAR neu : RListenElement;
BEGIN
  neu := NEW(RListenElement);
  neu^.person := pers;
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegeVorne;
```

Anwendungs-beispiele

Operationen auf linearen Listen - 2

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```


Anwendungs-beispiele

Operationen auf linearen Listen - 3

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
(* Mit expliziter Speicherfreigabe *)

VAR muell : RListenElement;
BEGIN
  IF liste # NIL THEN
    muell := liste;
    liste := liste^.nachfolger;
    muell^.nachfolger := NIL;
    DISPOSE(muell);
  END;
END LoescheVorne;
```



Anwendungs-beispiele

Suchen in linearen Listen - 1

```
PROCEDURE Suche (liste : RListenElement;
                 was : Person): RListenElement =
VAR position : RListenElement;
    gefunden : BOOLEAN := FALSE;

BEGIN
  position := liste;
  WHILE (position # NIL AND NOT gefunden) DO
    IF was = position^.person THEN
      gefunden := TRUE;
    ELSE
      position := position^.nachfolger;
    END;
  END;
  RETURN position;
END Suche;
```

Sequentielles Suchen

Ergebnis : Zeiger auf das gefundene Element sonst NIL

Anwendungs-beispiele

Suchen in linearen Listen - 2

- **Listen sind rekursive Datenstrukturen**
  - ***Rekursive Datenstrukturen*** können *elegant* mit ***rekursiven Prozeduren*** bearbeitet werden!

```
PROCEDURE SucheRek (liste : RListenElement;
                    was : Person): RListenElement =
BEGIN
  IF liste # NIL THEN
    IF was = liste^.person THEN
       RETURN liste;
    ELSE
      RETURN SucheRek (liste^.nachfolger, was);
    END;
  ELSE
    RETURN NIL;
  END;
END SucheRek;
```

rekursiver Aufruf

Sortierte lineare Liste

```
TYPE ListenElement =
  RECORD
    wert : INTEGER;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : ListenElement;
```

Für alle Elemente x der Liste gilt:

`x^.wert <= x^.nachfolger^.wert`

Liste soll immer im Zustand "aufsteigend" sortiert sein



Einfügen in eine sortierte Liste - 1

- **Fall 1: Die Liste ist leer.**



- **Fall 2: Element muß vorne eingefügt werden**



Einfügen in eine sortierte Liste - 2

- **Fall 3: Element muß sonst irgendwo eingefügt werden.**
  - Es muß nach der ***Einfügestelle*** gesucht werden
  - Damit man auf die Elemente vor und hinter der Einfügestelle zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das Element hinten angefügt werden muß !!

Einfügen: Lösung A -1

```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, einfuegestelle : RListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                  (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN         (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);        (* neu vorne einhaengen *)
  ELSE
    einfuegestelle := SucheEinfuegestelle(liste, w);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Einfügen: Lösung A -2

```
PROCEDURE ErzeugeNeuesListenelement(w : INTEGER): RListenElement =
VAR elem : RListenElement;
BEGIN
  elem := NEW(RListenElement);
  elem^.wert := w;
  elem^.nachfolger := NIL;
  RETURN elem
END ErzeugeNeuesListenelement;
```

```
PROCEDURE EinfuegenVorne(VAR liste : RListenElement;
                         VAR neu   : RListenElement)=
BEGIN
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegenVorne;
```


Einfügen: Lösung A -3

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                              w : INTEGER): RListenElement =
VAR stelle, vergleich: ListenElement;
BEGIN
  vergleich := liste;
  stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

```
PROCEDURE FuegeAnEinfuegestelleEin(VAR neu   : RListenElement;
                                   VAR stelle: RListenElement)=
BEGIN
  neu^.nachfolger := stelle^.nachfolger;
  stelle^.nachfolger := neu;
END FuegeAnEinfuegestelleEin;
```

Einfügen: Lösung B

```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, position, vorgaenger : RListenElement;
BEGIN
  neu := NEW(RListenElement);
  neu^.wert := w;
  neu^.nachfolger := NIL;

  IF liste = NIL THEN             (* liste ist leer *)
    liste := neu;
  ELSIF (w <= liste^.wert) THEN   (* w ist kleinstes Element*)
    neu^.nachfolger := liste;     (* w vorne einhaengen *)
    liste := neu;
  ELSE                            (* Einfuegestelle suchen *)
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert <= w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* vorgaenger = Einfuegestelle *)
    neu^.nachfolger := position;  (* w einhaengen *)
    vorgaenger^.nachfolger := neu;
  END;
END Einfuegen;
```


Löschen in einer sortierten Liste - 1

- **Fall 1: Die Liste ist leer.**



- **Fall 2: Das erste Element muß gelöscht werden**



Löschen in einer sortierten Liste - 2

- **Fall 3: Element muß sonst irgendwo gelöscht werden.**
  - Damit man auf die Elemente vor und hinter dem zu löschenden Element zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das letzte Element gelöscht werden muß !!

Löschen: Lösung A

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert # w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* position = NIL v position^.wert = w *)
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        liste := position^.nachfolger;
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Löschen: Lösung B -1

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position);
    gefunden := (position # NIL);  (* position= NIL v position^.wert = w *)
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```


Löschen: Lösung B -2

```
PROCEDURE Suche (liste : RListenElement;
                 w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```

Beobachtung !

- **Einfügen und Loeschen benutzen ähnliche Such-Prozeduren**

```
PROCEDURE Suche (liste : RListenElement; w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste; vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                                  w : INTEGER): RListenElement =
VAR stelle, vergleich: RListenElement;
BEGIN
  vergleich := liste; stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```


Zweck der Verwendung von Zeigern

- Anker
- Vernetzung
- Durchlauf
- Abkürzung für Durchlauf
- semantische Beziehung



Prozedurtyp und Prozeduren dieses Typs

- **Frage:**
  - Gibt es eine Möglichkeit, Prozeduren so zu ***parametrisieren***, daß als Parameter ein Algorithmus (Prozedur) übergeben werden kann?
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur.***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.
  - Prozedurvariablen können ***passende*** Prozeduren zugewiesen werden.
  - Prozedurvariablen können als ***Parameter*** übergeben werden.
  - Gesetzte Prozedurvariablen können in Anweisungen mit ***aktuellen Parametern*** aufgerufen werden.

Beispiel: ProzedurTyp - 1

```
TYPE  Bereich      = [-10 .. 10]; Index   = [1 .. 10];
      Zahlenmenge  = SET OF Bereich;
      Zahlenfeld   = ARRAY Index OF Bereich;
      TestProzedur = PROCEDURE (a : INTEGER) : BOOLEAN;

PROCEDURE IstPositiv (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i > 0);
END IstPositiv;

PROCEDURE IstNegativ (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i < 0);
END IstNegativ;

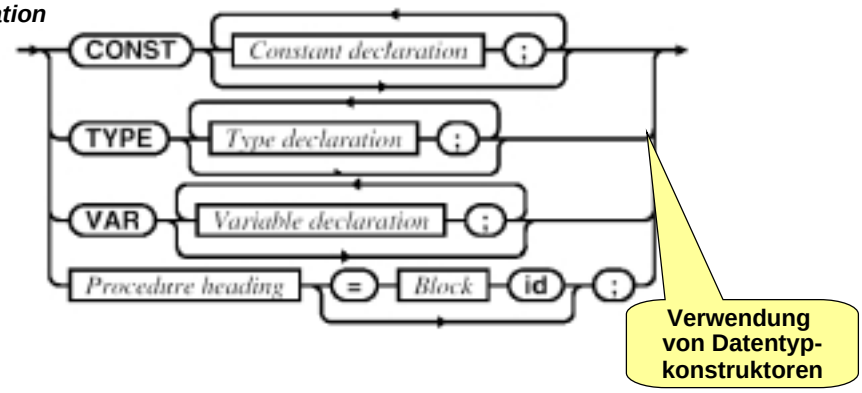
PROCEDURE Filtern (        feld : FeldTyp;
                   VAR resultat : Zahlenmenge;  p : TestProzedur) =
BEGIN
  FOR i := FIRST(Index) TO LAST(Index) DO
    IF p(feld[i]) THEN
      resultat := resultat + Zahlenmenge{feld[i]};
    END;
  END;
END Filtern;
```

Gegeben ist ein Feld mit Zahlen. Bestimme die darin enthaltenen positiven und negativen Zahlen

Signatur-konforme Prozeduren

Prozedur-parameter

Beispiel: ProzedurTyp - 2

```
VAR proc : TestProzedur;

    werte := Zahlenfeld{3, -5, 9, 8, -6, 9, -6, -1, -1, 5};
    positiv, negativ := Zahlenmenge{};

BEGIN
  proc := IstPositiv;
  Filtern (werte, positiv, proc);
  proc := IstNegativ;
  Filtern (werte, negativ, proc);

  SIO.PutLine ("Positive Zahlen:");
  FOR e := FIRST(Bereich) TO LAST(Bereich) DO
    IF e IN positiv THEN SIO.PutInt(e); SIO.PutText(", "); END;
  END;
  SIO.Nl();
```

Prozedurvariable

Zuweisung einer Prozedur an die Prozedurvariable

Zuweisung an Prozedurvariable

- **Zuweisung**
  - `TYPE PType = PROCEDURE …`
    `VAR proc : Ptype`

    `proc := PT; (* Ausdruck der vom Typ PT ist *)`
  - Diese Zuweisung ist korrekt, wenn
    - PT = ***NIL*** ist (NIL ist mit jedem Wert eines Prozedurtyps kompatibel)
    - ***Anzahl*** der Parameter und deren ***Typen*** sind gleich für PT und PType
    - Beide haben den gleichen ***Ergebnistyp*** oder keinen

```
TYPE TestProzedur = PROCEDURE (a : INTEGER)  : BOOLEAN;
     PT1          = PROCEDURE (in : INTEGER) : BOOLEAN;

VAR test : TestProzedur;
      p1 : PT1;
…
p1 := test;
test := p1;
```

Typen sind signatur-konform

Verwendung Prozedurtyp - 1

```
TYPE Vergleichsoperation = PROCEDURE (a, b : INTEGER) : BOOLEAN;

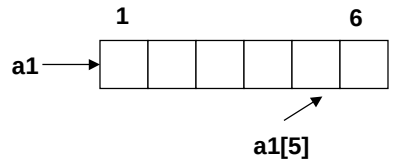
PROCEDURE Suche (liste : ListenElement;
                     w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement;
                    op : Vergleichsoperation) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND  NOT op(stelle^.wert, w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

signaturkonform

```
PROCEDURE IstGleich
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a = b;
END IstGleich;
```

```
PROCEDURE Groesser
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a > b;
END Groesser;
```


Verwendung Prozedurtyp - 2

```
PROCEDURE Loeschen (VAR liste : ListenElement; w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position, IstGleich);
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Verwendung Prozedurtyp - 3

```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, einfuegestelle, groessererWert : ListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN       (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);      (* neu vorne einhaengen *)
  ELSE
    Suche(liste, w, einfuegestelle, groessererWert , Groesser);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```


Diskussion: Dynamische Datentypen

- **Je deutlicher die Realisierung dynamischer Datenstrukturen durch Zeiger in der Sprache sichtbar ist,**
  - desto leichter fällt die softwaretechnisch ***unsaubere*** Verwendung.
- **Die explizite Speicherverwaltung führt zu einigen Problemen.**
  - Moderne imperative Sprachen sollten daher auf explizites Löschen verzichten und hierfür einen ***Garbage Collector*** besitzen.
- **Erst in objektorientierten Sprachen,**
  - die vorrangig mit ***Referenzsemantik*** arbeiten,
  - können grundlegende Probleme der Referenzierung gelöst werden.
- **Prozedurtypen sind nicht dynamisch, aber Prozedurvariablen können dynamisch unterschiedlichen Prozedurobjekten zugewiesen werden**

Was haben wir gelernt?

- Charakterisierung statischer Datentypen (Wiederholung) und Defizit für Listenverarbeitung
- Typen für Listenelemente, Zeigertypen, dynamische Objekte (Haldenobjekte)
- dynamische Datenobjekte: Explizite Erzeugung, implizite Bezeichnung, Lebensdauer, Halde (Heap), Freigabe
- Derefenzieren, Lesen und Setzen von Haldenobjekten, Setzen von Zeigerobjekten, Vergleich
- Aliasing, nicht mehr greifbare Haldenobjekte, hängende Zeiger, Garbage Collection
- Lineare Liste: Einfügen/Löschen (vorn und in der Mitte), Suchen, geordnete lineare Liste
- Arten von Zeigern (Zweck ihrer Verwendung)
- Prozedurobjekt, Prozedurtyp, Prozedurvariable, Einsatz für Flexibilität

Glossar

- Wertsemantik, Verweis (Zeiger) semantik
- statische Datentypen, dynamische Datentypen, rekursive Datentypen
- Zeigertyp, Zeigerobjekt (-konstante oder -variable)
- Haldenobjekttyp, dynamische (anonyme) Datenobjekte: Haldenobjekte, Erzeugung mit Einrichtung eines Zeigerwerts, Löschen mit Löschen des Zeigerwerts
- statische Datenobjekte, Datenobjekte auf dem Kellerspeicher, Datenobjekte auf der Halde
- Zeiger: Dereferenzieren, Setzen, Lesen, Vergleichen, Setzen/Lesen von Komponenten von Haldenobjekten
- Aliasing, inaccessible objects, dangling references
- Ankerzeiger, Verkettungszeiger, Durchlaufzeiger, „semantische“ Zeiger, Abkürzung von Zugriffswegen
- einfach verkettete lineare Liste, sortierte Liste, doppelt verkettete Liste
- Listenoperationen: Einfügen vorn, in der Mitte, Suchen, Löschen vorn, in der Mitte, Suche als allgemeine Prozedur für Listenoperationen
- Prozedurtyp und Parameterprofil, signaturkonforme Prozedurobjekte, Prozedurvariable und Prozedurobjektzuweisung, Lebensdauer von Zeigerobjekten, von über Zeiger zugegriffenen Haldenobjekten

Datentypen I

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

Datentypen - Grundidee

- **Software dient zur Verarbeitung von *Anwendungsdaten*.**
  - Frage: Wie gut passen die verfügbaren Datentypen der verwendeten Programmiersprache zu den zu modellierenden Größen des ***Anwendungsbereichs***.
- **Daraus resultiert die Anforderung:**
  - Programmiersprachen müssen einen ***sinnvollen*** Satz an Datentypen anbieten.
- **Ziel:**
  - Auf der Basis vordefinierter Datentypen sollen anwendungsbezogene Datenstrukturen konstruiert werden können.
- **Zwei Lösungsansätze:**
  - Eine große Vielfalt von ***vordeklarierten*** Datentypen (wie in PL/I) soll möglichst viele Anwendungsfälle abdecken.
  - Ein kleiner Satz von elementaren Typen und flexiblen ***Konstruktions-mechanismen*** (wie in Algol 68) soll die anwendungsbezogene Definition neuer Datentypen erlauben.

Bedeutung von Typen in imp. Sprachen

- **In imperativen Programmiersprachen hat ein Typ folgende Bedeutung (Wiederholung):**
  - Festlegung der/des gültigen ***Struktur/Wertebereichs*** für Variablen,
  - damit verbunden die ***Kardinalität*** (Anzahl der verschiedenen Werte),
  - Festlegen der ***Operationen*** und Literale/Aggregate.
  - Statische ***Prüfung*** der Zulässigkeit von Operationen:
    - Zuweisung, Parameterübergabe etc.
- **Festlegung von Maschineneigenschaften**
  - z.B. Reservierung von Speicherplatz

Erinnerung: Deklaration

- **In imperativen Programmiersprachen ist die explizite Deklaration von Bezeichnern notwendig.**
- **Grundidee der Deklaration:**
  - Eine Deklaration verbindet einen ***Bezeichner*** an Eigenschaften eines Programmobjekts, z.B. Typ und Sichtbarkeit.
  - Während sich z.B. bei Variablen der Wert des Objektes verändern kann, bleiben seine deklarierten Eigenschaften für die Lebensdauer erhalten.
- **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***

Deklaration (erweitert)

- **Die Syntax für Deklarationen wird erweitert, damit *benutzerdefinierte* Typen deklariert werden können.**

Declaration



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.

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

Typen in Modula-3

- **Modula-3 kennt *vordefinierte einfache* Typen :**
  - SHORTINT, INTEGER, LONGINT, REAL, LONGREAL, BOOLEAN, SET, CHAR.
- **Modula-3 unterstützt wie viele imperative Sprachen *benutzerdefinierte* Datentypen.**
- **Häufig verwenden wir in der Definition die *Typkonstruktoren* für die zusammengesetzten Datentypen**
  - Array, Record, Set.
- **Eine wesentliche Erweiterung bringt der *Zeigertyp* (Pointer).**
  - Er ermöglicht den Aufbau ***dynamischer*** Datenstrukturen.

Benutzerdefinierte einfache Typen

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

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];
```

Ordinaltyp

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


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!

Operationen auf Unterbereichstypen

- **Regel:**
  - Für einen Unterbreichstyp sind alle die Operationen definiert, die auch für seinen ***Basistyp*** definiert sind.
- **Es ist häufig unsinnig,**
  - ***arithmethische*** 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***.

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.
      ```

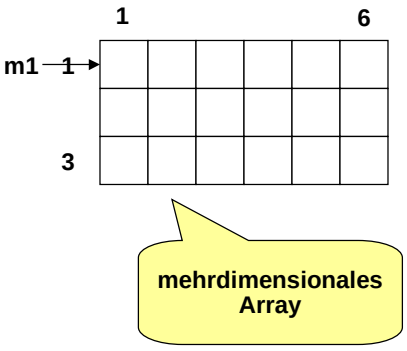
Feldtypen

- **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
  - Dem Array als Datenstruktur entspricht die ***FOR-Anweisung*** als Kontrollstruktur.
  - Bei mehrfach indizierten Arrays gibt es entsprechend ***geschachtelte*** Schleifen.

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;
```

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;
```


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):

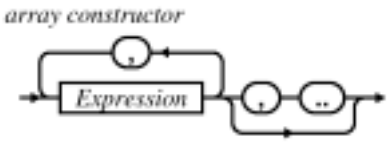
    ```
    val := a1[3] (* Wert des 3. Elements von a1 *)
    ```

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

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

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, ..}  Feldaggregate
```

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

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


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
```

Beispiel: Array - 1

```
MODULE StundenPlan EXPORTS Main;
IMPORT SIO;

TYPE                                          Typdeklarationen
    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*)
```


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.
```

Verbundtypen

Definitionen

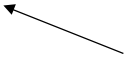
- Nach Informatik-Duden:
  - Record *(Verbund, Struktur, Datensatz)*: ***Zusammenfassung*** von mehreren Datentypen zu einem Datentyp. Der neue Wertebereich ist das ***kartesische Produkt*** der Wertebereiche der einzelnen Datentypen, wobei die Anordnung keine Rolle spielt.
- Nach Sebesta:
  - A record 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)

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

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.

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

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:

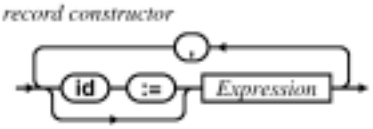
    <RecordBezeichner> . <FeldName>

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

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


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

Operationen auf RECORDs - 1

- **Zuweisung**
  - zwei RECORD-Objekte sind ***zuweisungskompatibel***, wenn
    - alle Felder den gleichen ***Namen*** und den gleichen Typ haben
    - alle Felder in der gleichen ***Reihenfolge*** deklariert sind
- **Vergleich**
  - zuweisungskompatible Arrays 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
```


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

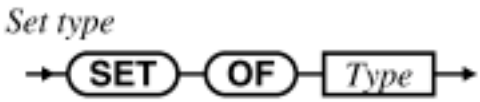
Beispiel WITH-Anweisung

```
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
```


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.
  - Nicht Elemente, sondern Zugehörigkeitsinformationen abgelegt: charakteristische Speicherung

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

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

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.
```


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

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!

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

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.
```


Verbesserung 2 des Beispiels - d

```
PROCEDURE LiesEingabeBis (stop : CHAR) : TEXT =
VAR eingabe : TEXT := "";
    zeichen : CHAR;
BEGIN
  zeichen := SIO.GetChar();
  WHILE zeichen # stop DO
    eingabe := eingabe & Text.FromChar(zeichen);
    zeichen := SIO.GetChar();
  END;
  RETURN eingabe;
END LiesEingabeBis;

…

BEGIN (*Buchstaben3*)

  Ausgabe(Vorkommenzaehlen(LiesEingabeBis(':')));

END Buchstaben3.
```

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 (siehe auch Begriffe von vorangegangener Seite)

- 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

Teil II

Programmiersprachenkonstrukte und Programmierstile

- "Funktionale" Programmierung
- Imperative Programmierung
- Kontrollstrukturen
- Datentypen I (statisch)
- Datentypen II (dynamisch)
- Zusammenfassung

"Funktionale" Programmierung

- **Funktionen, Parameter**
- **Vernetzung von Funktionen**
- **Bedingungen in funktionalen Programmen**
- **Rekursion**

Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
- **Was benötigt man dazu**
  - Daten / ***Datentypen***
  - ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Anmerkung:**
  - es gibt rein funktionale Programmiersprachen MIRANDA
  - viele sog. Hochsprachen erlauben eine gewisse Art der funktionalen Programmierung (wie Modula-3)

Bereits bekannte Hilfsmittel

- **Datentypen**
  - Argumentbereich und Ergebnisbereich bilden die Wertebereiche, die zu den Typen der Ein- und Ausgabewerte gehören. Auf diesem sind jeweis Operationen zulässig

- **Anweisungen**
  - **`RETURN`**-Anweisung
  - Auruf-Anweisung
  - Anweisungsfolgen
  - Kontrollstrukturen

Funktionsdeklaration

- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereiche)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift***
  - ist frei von ***Seiteneffekten***

Kopf, Signatur

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;
```

Berechnungsvorschrift

Name

formale Eingabeparameter

Ergebnistyp

Aufruf einer Funktion

```
MODULE QuadratM EXPORTS Main;
(* Dieses Programm berechnet das Quadrat einer Zahl *)

IMPORT SIO;

PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (SIO.GetReal()));
END QuadratM.
```

Deklaration und ***Definition*** der Funktion

Aufruf der Funktion mit einem ***aktuellen*** Parameter;

Das ***Ergebnis*** der Funktion wird an die Prozedur PutReal ***übergeben***

Formale & aktuelle Parameter

- **Parameter**
  - haben eine ***Typ***
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - dadurch werden Funktionen ***flexibel einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden
  - diese werden dann im Rumpf ***verwendet***.

Syntax von M3-Funktionen - 1

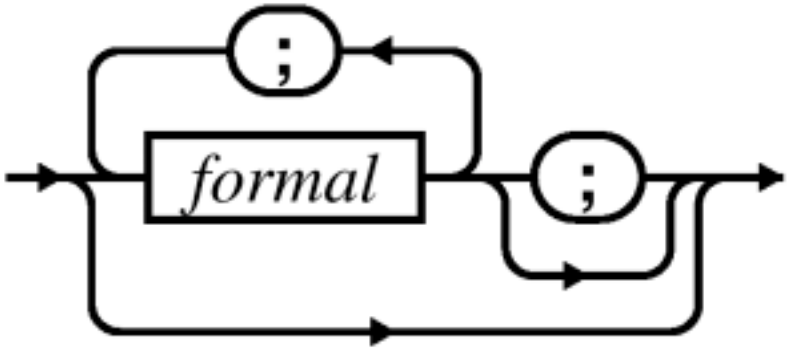
Declaration

Procedure heading

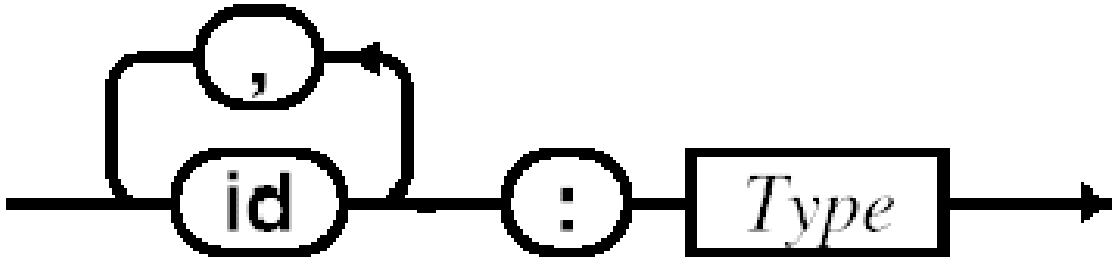
Signature

Syntax von M3-Funktionen - 2

formals



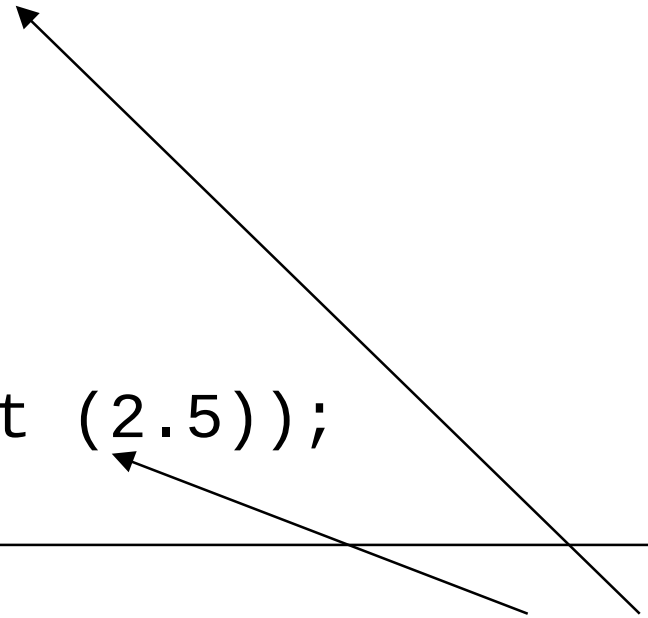
formal



Vereinfachte Darstellung !

Formale & aktuelle Parameter

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (2.5));
END QuadratM.
```



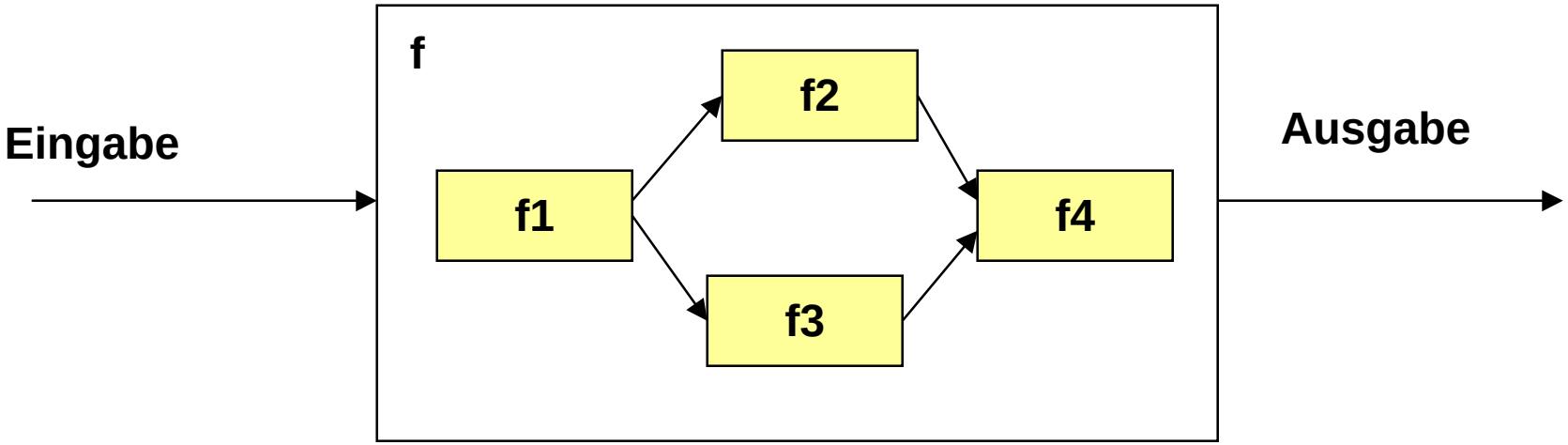
Binden der aktuellen Werte (Parameter) an die formalen Parameter (Stellvertreter)

Funktionskomposition

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, …] : x = (f : x, g : x, h : x, …)
  - **Bedingung**
    - if t then f else g : x =

$$\begin{cases} f : x & \text{, falls } t : x = \text{TRUE} \\ g : x & \text{, falls } t : x = \text{FALSE} \\ ? & \text{, sonst} \end{cases}$$

schematisches Beispiel

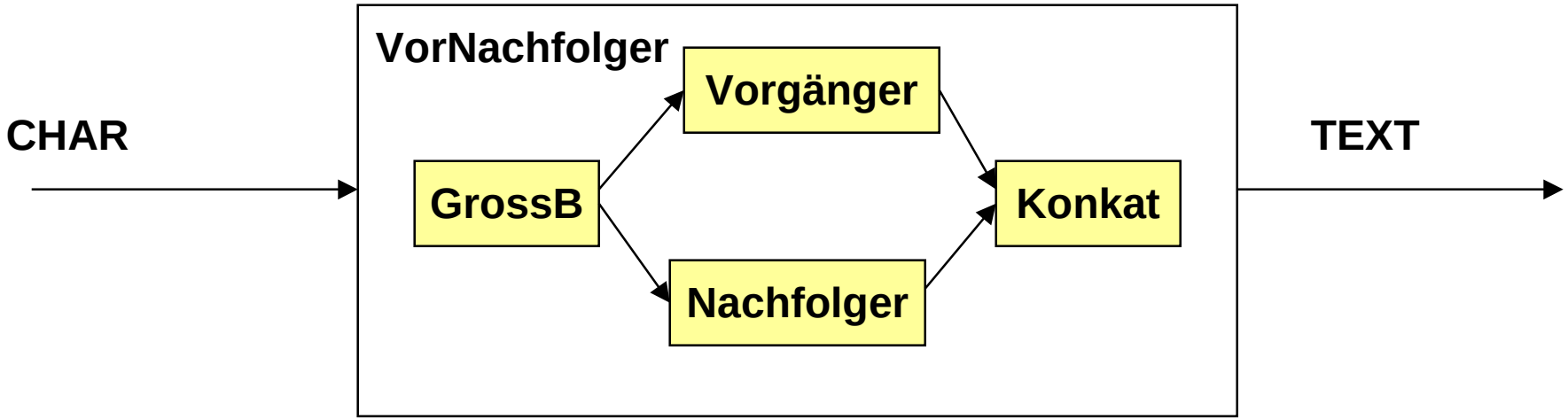


$$f = f1 \;\; o \;\; [\, f2, f3 \,] \;\; o \;\; f4$$

Komposition

Konstruktion

Beispielnetz



VorNachfolger = GrossB o [Vorgänger, Nachfolger] o Konkat

Beispiel:

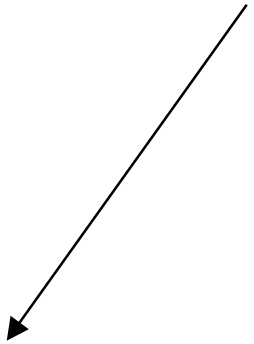
Eingabe: 'h'

Ausgabe: "GI"

Beispiel: Detaillierug

```
MODULE VorNachfolger EXPORTS Main;
IMPORT SIO;

PROCEDURE GrossB(c : CHAR) : CHAR =
...
PROCEDURE Vorgaenger (c : CHAR) : CHAR =
...
PROCEDURE Nachfolger (c : CHAR) : CHAR =
...
PROCEDURE AddOrd (c, d : CHAR) : CARDINAL =
. . .
PROCEDURE Konkat (c, d : CHAR) : TEXT =
...

BEGIN
  (* VorNachfolger = GrossB o   [ Vorgänger, Nachfolger ]   o   Konkat *)
  SIO.PutText ( Konkat ( Vorgaenger( GrossB( SIO.GetChar() )) ,
                         Nachfolger( GrossB( SIO.GetChar() ))
                       )
              );

END VorNachfolger.
```

Da **keine Variablen** (Zwischenspeicher) verwendet werden, muß der Wert **zweimal** eingelesen werden !

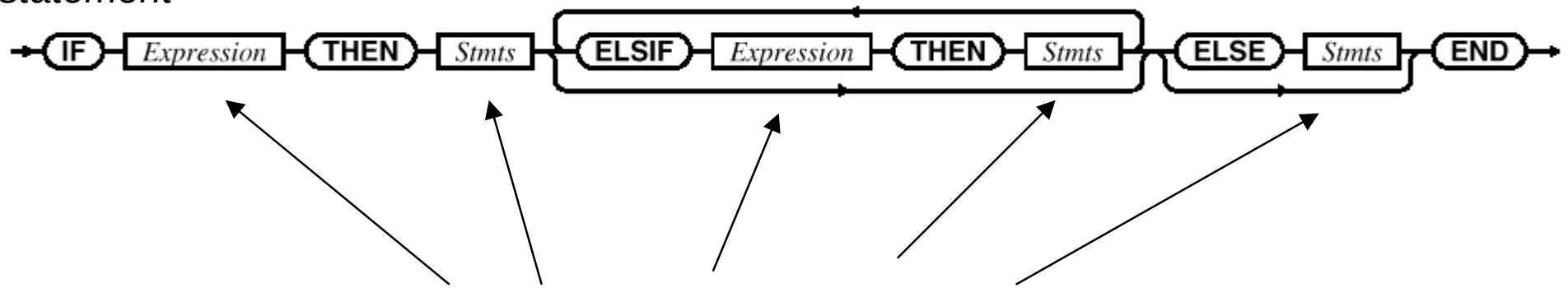


Funktion GrossB muß **zweimal** ausgeführt werden, da Modula-3 die **Konstruktion** von Funktionen nicht unterstützt.

Funktionale Programme

- **Idee:**
  - In Abhängigkeit von ***Bedingungen*** soll eine ***alternative*** Programmkomponente ausgeführt werden
  - Bedingungen müssen einen Wert vom Typ ***BOOLEAN*** liefern
  - Modula-3 bietet dazu u.a. die ***IF-Anweisung*** an

If statement



Bei funktionalen Programmen stehen hier ***ausschließlich*** Funktionsaufrufe!

Bedingung - IF-THEN-ELSE

- **Typische Formen von Bedingungen:**

```
IF bf THEN
  f1;
ELSE
  f2;
END;
```

Alternative "Entweder-Oder“ (zweiseitig bedingte Anweisung)

```
IF bf THEN
  f1;
END;
```

einseitig bedingte Anweisung

```
IF    bf1 THEN
  f1;
ELSIF bf2 THEN
  f2;
ELSIF bf3 THEN
  f3;
…
ELSE
  fn;
END;
```

mehrseitig bedingte Anweisung

Die ***Bedingungen*** (bfi) werden der Reihe nach ausgewertet, bis eine ***WAHR*** ist. Dann wird die entsprechende Anweisung (Funktionsaufruf) ausgeführt.

Ist ***keine*** Bedingung wahr, dann wird der ***ELSE-Zweig*** ausgeführt

Beispiel - 1

- **Aufgabe:**
  - Eingabe ist eine ganze Zahl
  - Stelle fest, ob diese 1-, 2-, 3-, mindestens 4-stellig oder negativ ist!

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF IstEinstellig(i)     THEN RETURN ("einstellig") END;
  IF IstZweistellig(i)    THEN RETURN ("zweistellig") END;
  IF IstDreistellig(i)    THEN RETURN ("dreistellig") END;

  IF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```

Beispiel - 2

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF    IstEinstellig(i)     THEN RETURN ("einstellig")
  ELSIF IstZweistellig(i)    THEN RETURN ("zweistellig")
  ELSIF IstDreistellig(i)    THEN RETURN ("dreistellig")
  ELSIF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE                            RETURN ("negativ")
  END;
END ErmittleStelligkeit;



BEGIN
  (* Aufruf der Hauptfunktion *)
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```

ELSIF-Konstruktionen machen Bedingungen semantisch klarer

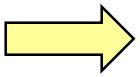
Beispiel - 3

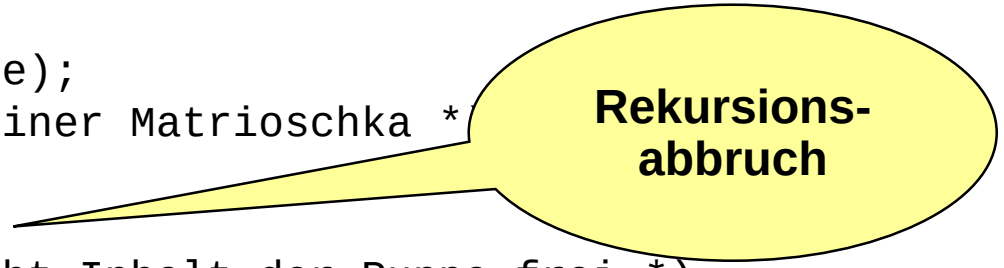
```
MODULE Stelligkeit EXPORTS Main;
(* Dieses Programm berechnet die Stelligkeit von Zahlen *)
IMPORT IO;

PROCEDURE IstEinstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=0) AND (i<10) );
END IstEinstellig;

PROCEDURE IstZweistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=10) AND (i<100) );
END IstZweistellig;

PROCEDURE IstDreistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=100) AND (i<1000) );
END IstDreistellig;

PROCEDURE IstMinVierstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=1000) );
END IstMinVierstellig;
```

Idee und Anwendung

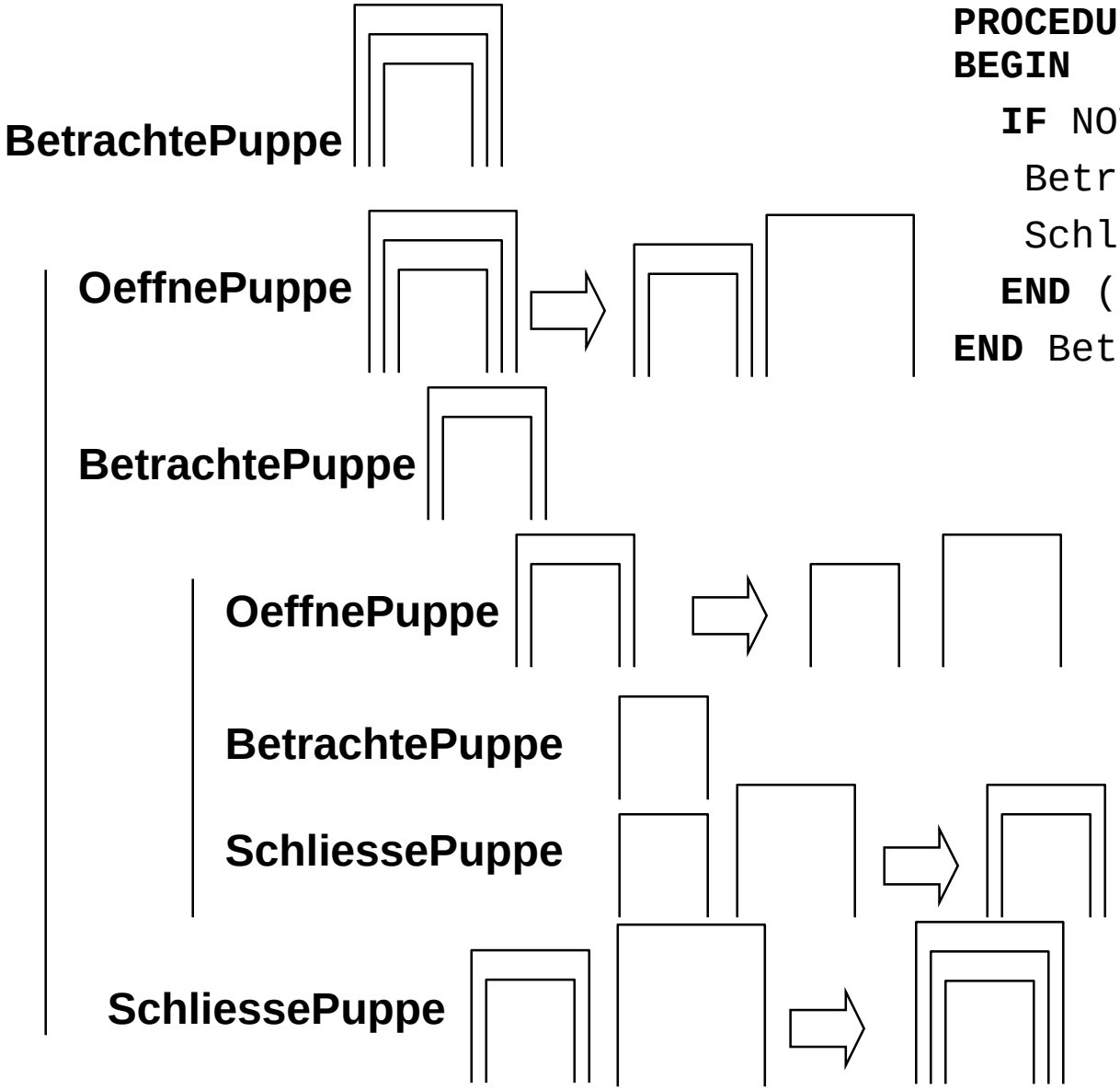
- **Idee:**
  - allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens ***durch Rückführung auf sich selbst***
- **Rekursive Funktion**
  - darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen***
- **Beispiel: Matrioschka-Puppen**

```
PROCEDURE BetrachtePuppe (puppe);
(* Pseudocode zum Betrachten einer Matrioschka *)
BEGIN (* BetrachtePuppe *)
  IF NOT Massiv (puppe) THEN
    OeffnePuppe (puppe); (* gibt Inhalt der Puppe frei *)
    BetrachtePuppe (inhaltDerPuppe); (* Rekursion *)
    SchliessePuppe (puppe);
  END (* IF *);
END BetrachtePuppe;
```





Beispiel



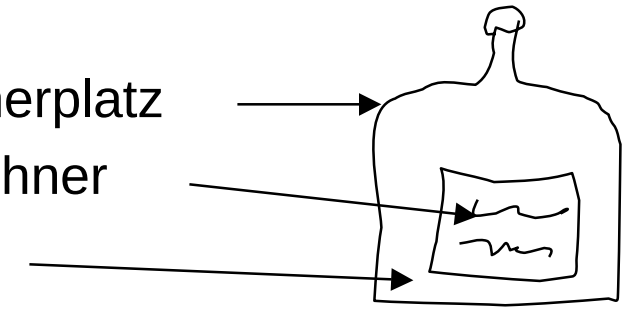
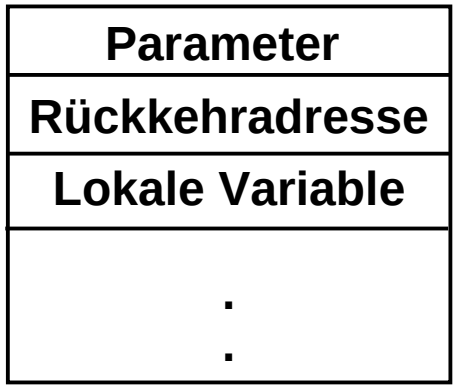
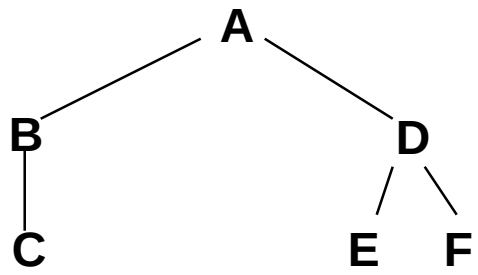

```
PROCEDURE BetrachtePuppe (puppe);
BEGIN
  IF NOT Massiv (puppe) THEN
   BetrachtePuppe(OeffnePuppe (puppe));
   SchliessePuppe (puppe);
  END (* IF *);
END BetrachtePuppe;
```

Jeder rekursive Aufruf erzeugt eine neue **Inkarnation** der Funktion.

Es wird solange rekursiv aufgerufen, bis der **Rekursions-abbruch** erreicht wird.

Anschließend werden die erzeugten Inkarnationen in **umgekehrter** Reihenfolge ausgeführt

Anmerkungen

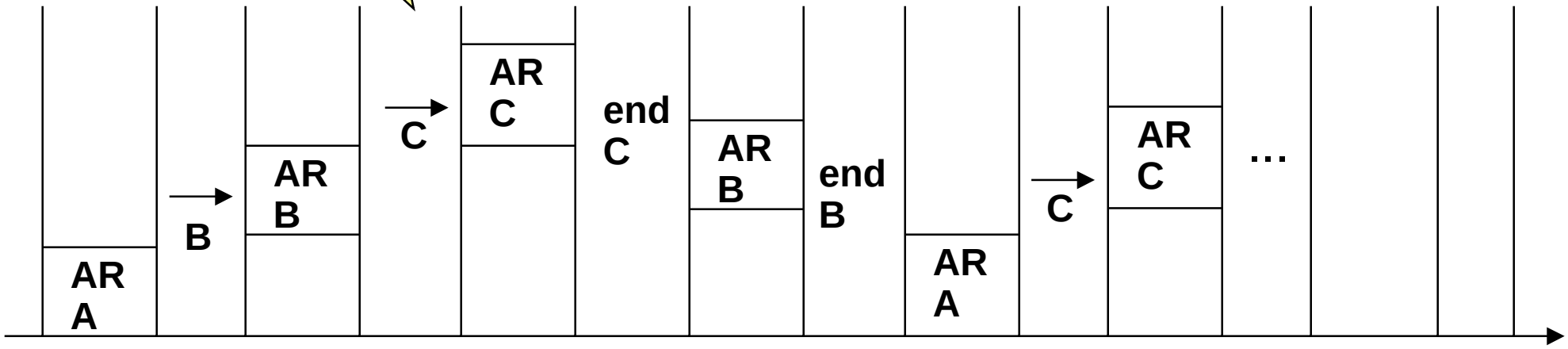
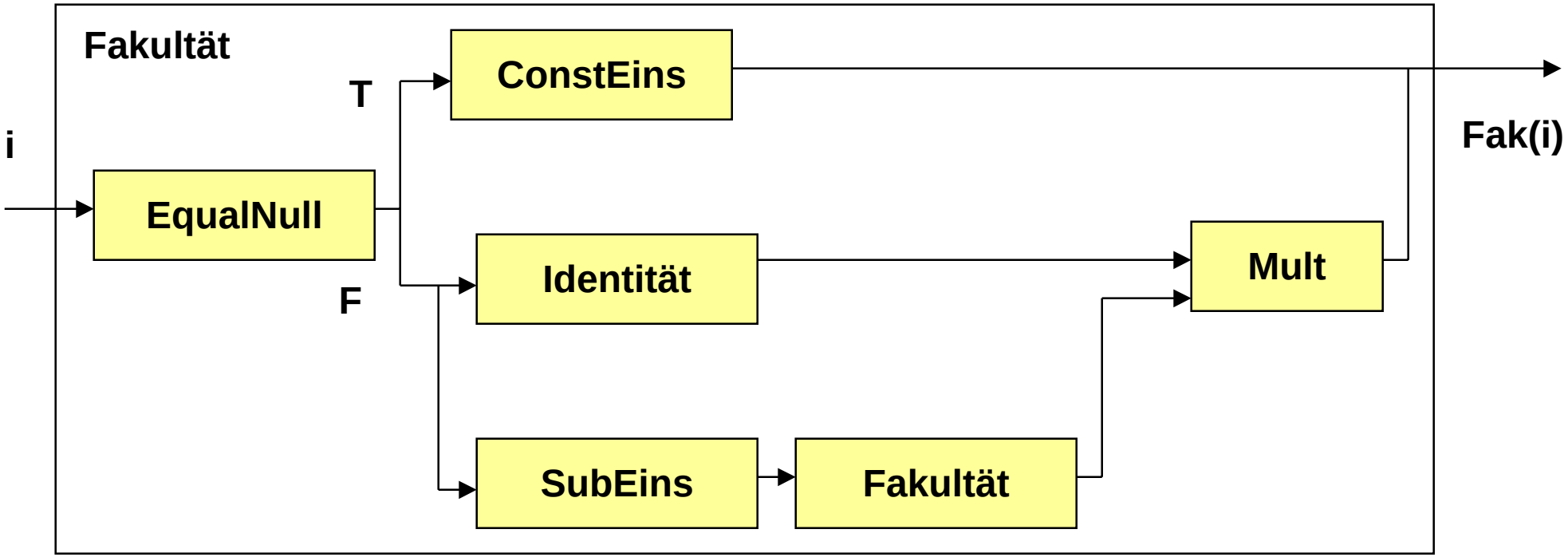
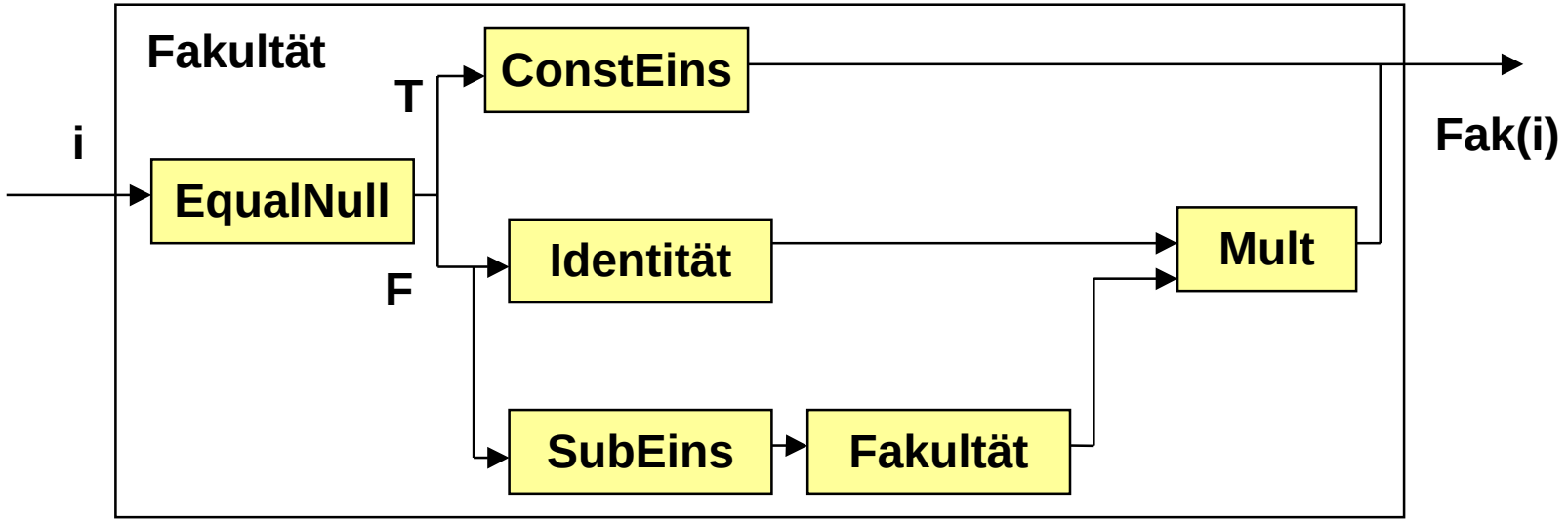
- **Ziel:**
  - es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen
  - dies führt immer zu einem ***Laufzeitfehler***
- **Konsequenz**
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung***
    - Rekursionsabbruch: in definierten Fällen wird der rekursive Aufruf nicht ausgeführt
  - Muster:
    - ```
      IF … THEN
        rekursiver Aufruf
      ELSE
        Rekusionsabbruch
      ```
- **Bemerkung**
  - Rekursion führt zu ***prägnaten, knappen*** und ***eleganten*** Algorithmen für bestimmte Problemstellungen oder -lösungen
      ```

Exkurs: Speicherverwaltung

- **In traditionellen Programmiersprachen (z.B. FORTRAN)**
  - Zuordnung Variable ⟺ Speicher (statisch, zur Übersetzung)
  - Speicherplatz
  - Bezeichner
  - Wert



  - Vorteile: einfach, statisch zu überprüfen
  - Nachteile: - Speicherverschwendung
    - eine Variable kann nicht mehrfach existieren (Rekursion?)
    - keine Rekursion

- **Ansatz in block-orientierten Sprachen**
  - keine feste Zuordnung Variable ⟺ Speicher
  - Variablen werden dann auf den Speicher abgebildet, wenn der entsprechende Block ***aktiv ist*** (block-orientiert)

Programm-Kellerspeicher - 1

- **Blöcke sind baumartig angeordnet**
  - Vater-Block endet niemals vor dem Sohn-Block!
- **Wird ein Block aktiv, dann wird ein Aktivierungsblock (Inkarnation) erzeugt**



  - Inkarnationen werden nach dem LIFO-Prinzip (Keller-Prinzip) verwaltet
  - automatisch, unter Kontrolle des Laufzeitsystems (Laufzeitkeller)
  - nur die "jüngste" Inkarnation ist aktiv
  - Lebensdauer der Objekte ist direkt gekoppelt an die Inkarnation
  - es können mehrere Inkarnationen eines Blocks auf dem Stack liegen
  - es wird kein Speicher verschwendet

Programm-Kellerspeicher - 2




```
proc A
    Aufruf B;
    Aufruf D;
end A

proc B
    Aufruf C;
end B

proc C
end C

proc D
    Aufruf E;
    Aufruf F;
end D

proc E
end E

proc F
end F
```

dynamischer Speicher; pulsierend, LIFO



Beispiel Fakultät - 1

- **Fakultät is definiert:**
  - fak : n 1 , falls n = 0
    n * fak(n-1) n ∈ N
- **Funktionnetz für Fakultät**



Beispiel Fakultät - 2




```
PROCEDURE Fakultaet (i : INTEGER) : INTEGER =
BEGIN
  IF EqualNull(i)
  THEN RETURN(ConstEins(i))
  ELSE RETURN (Mult(Identitaet(i), Fakultaet(SubEins(i))));
  END;
END Fakultaet ;
...
  SIO.PutInt(Fakultaet(SIO.GetInt()));
```

Beispiel Fakultät - 3

```
PROCEDURE EqualNull ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( i = 0 );
END EqualNull;

PROCEDURE ConstEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( 1 );
END ConstEins;

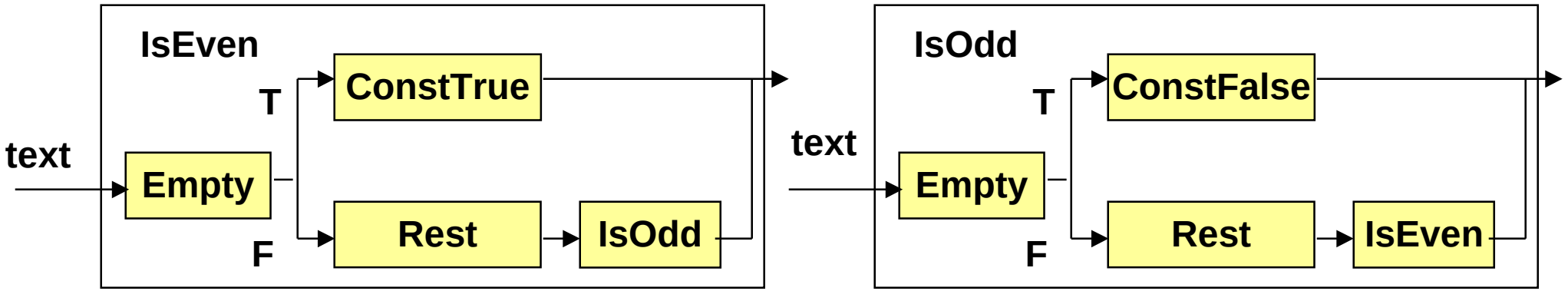
PROCEDURE Mult ( i, j : INTEGER) : INTEGER =
BEGIN
  RETURN ( i * j );
END Mult;

PROCEDURE Identitaet ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i );
END Identitaet;

PROCEDURE SubEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i - 1 );
END SubEins ;
```

Diese Funktionen sind nur im Sinne der **reinen funktionalen** Programmierung notwendig.

Sie können in der Definition der Funktion "Fakultaet" durch entsprechende **Ausdrücke** ersetzt werden!

Beispiel Fakultät - 4

```
MODULE FakultaetM1 EXPORTS Main;
(* Dieses Programm berechnet die Fakultaetsfunktion *)

IMPORT SIO;


PROCEDURE Fakultaet ( i : INTEGER) : INTEGER =
BEGIN
  IF i = 0
  THEN RETURN 1
  ELSE RETURN (i * Fakultaet(i-1));
  END;
END Fakultaet ;


BEGIN
  SIO.PutInt(Fakultaet(SIO.GetInt()));
END FakultaetM1.
```

Fibonacci-Funktion (rekursiv)

Wachstum einer Kaninchen-Population

- Wieviele Kaninchen-Pärchen gibt es nach n Jahren
  - Jahr 1 : 1 Pärchen
  - Jedes Pärchen hat ab dem zweiten Jahr je ein Pärchen Nachwuchs
-

Jahr	1	2	3	4	5	6	7	8	9
Anzahl	1	1	2	3	5	8	13	21	34

```
PROCEDURE Fibonacci (arg: INTEGER): INTEGER =
BEGIN
  IF arg <= 2 THEN
    RETURN 1
  ELSE
    RETURN (Fibonacci (arg -1) + Fibonacci (arg - 2))
  END;
END Fibonacci ;
```

Indirekte Rekursion

- **Definition:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Beispiel:**
  - Die beiden Funktionen (IsEven, IsOdd) sind ***indirekt*** rekursiv
  - können festzustellen, ob eine Zeichenfolge eine gerade oder ungerade Anzahl von Zeichen enthält



Beispiel - Indirekte Rekursion

```
MODULE EvenOdd EXPORTS Main;
(* Beispiel fuer die indirekte Rekursion *)
IMPORT SIO, Text;

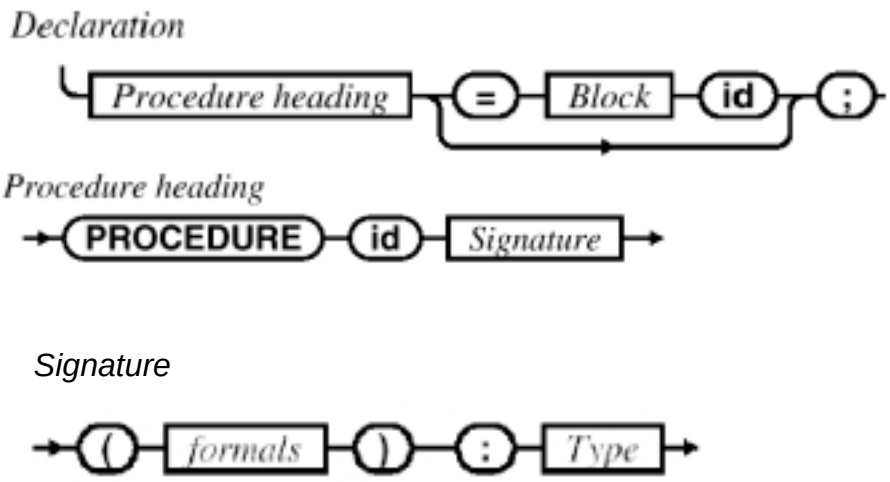
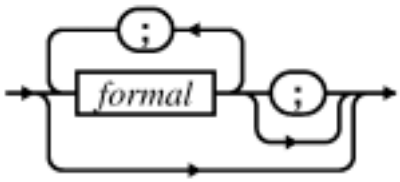
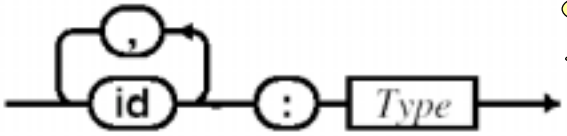
PROCEDURE IsEven (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN TRUE;
  ELSE RETURN (IsOdd (Text.Sub(str,1)));
  END;
END IsEven ;

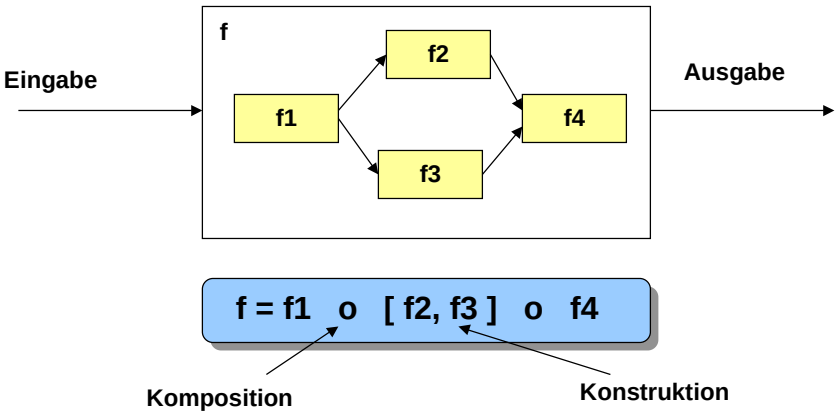
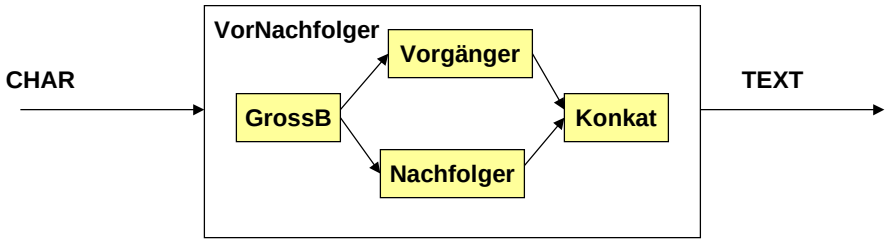
PROCEDURE IsOdd (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN FALSE;
  ELSE RETURN (IsEven (Text.Sub(str, 1)));
  END;
END IsOdd ;

BEGIN
  SIO.PutBool ( IsEven (SIO.GetWord()));     SIO.Nl();
  SIO.PutBool ( IsOdd  (SIO.GetWord()));
END EvenOdd.
```

Funktional, Rekursion und Iteration

- **Funktionale Algorithmen**
  - kennen keine ***Variablen*** zur Zwischenspeicherung von Ergebnissen
- **Nichtfunktionale Algorithmen**
  - speichern Zwischenergebnisse in ***Variablen*** ab
- **Rekursive Algorithmen**
  - lösen ein Problem, in dem sie sich ***selbst wieder aufrufen*** (direkt oder indirekt)
- **Iterative Algorithmen**
  - besitzen Abschnitte, die bei der Ausführung ***mehrmals*** durchlaufen werden
  - Jeder rekursive Algorithmus kann in einen iterativen umgewandelt werden
    - allgemein (siehe Compiler)
    - problemspezifisch

Was haben wir gelernt

- **Funktionale Programmierung**
  - Funktion
  - Parameter
  - Vernetzung von Funktionen
- **Verwendung von Anweisungen und Ausdrücken**
- **Verwendung elementarer Datentypen**
- **Konzept der Rekursion**
  - direkt rekursive Funktionen
  - indirekt rekursive Funktionen
  - Realisierung durch Laufzeitkeller

Glossar

- **Funktionale Programmierung**
- **Funktion, Funktionskopf, -rumpf**
- **Funktionalform**
- **Parameter**
  - Formal- , Aktual-
- **Seiteneffektfreiheit**
- **Rekursion**
  - direkt, indirekt
- **Datentyp (Wiederholung)**
  - einfacher elementarer Typ
  - Ordinaltyp
- **Speicherverwaltung nach Kellerprinzip (Laufzeitkeller), statische Speicherverwaltung**
- **Aktivierungsblock**

Teil II

Programmiersprachenkonstrukte und Programmierstile

- "Funktionale" Programmierung
- Imperative Programmierung
- Kontrollstrukturen
- Datentypen I (statisch)
- Datentypen II (dynamisch)
- Zusammenfassung

"Funktionale" Programmierung

- Funktionen, Parameter
- Vernetzung von Funktionen
- Bedingungen in funktionalen Programmen
- Rekursion

Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
- **Was benötigt man dazu**
  - Daten / ***Datentypen***
  - ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Anmerkung:**
  - es gibt rein funktionale Programmiersprachen MIRANDA
  - viele sog. Hochsprachen erlauben eine gewisse Art der funktionalen Programmierung (wie Modula-3)

Bereits bekannte Hilfsmittel

- **Datentypen**
  - Argumentbereich und Ergebnisbereich bilden die Wertebereiche, die zu den Typen der Ein- und Ausgabewerte gehören. Auf diesem sind jeweis Operationen zulässig
- **Anweisungen**
  - **`RETURN`**-Anweisung
  - Auruf-Anweisung
  - Anweisungsfolgen
  - Kontrollstrukturen

Funktionsdeklaration

- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereiche)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift***
  - ist frei von ***Seiteneffekten***

Kopf, Signatur

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;
```

Berechnungsvorschrift — **Name** — **formale Eingabeparameter** — **Ergebnistyp**

Aufruf einer Funktion

```
MODULE QuadratM EXPORTS Main;
(* Dieses Programm berechnet das Quadrat einer Zahl *)

IMPORT SIO;

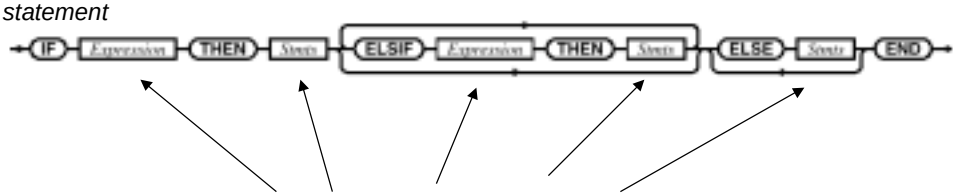
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (SIO.GetReal()));
END QuadratM.
```

Deklaration und ***Definition*** der Funktion

Aufruf der Funktion mit einem ***aktuellen*** Parameter;

Das ***Ergebnis*** der Funktion wird an die Prozedur PutReal ***übergeben***

Formale & aktuelle Parameter

- **Parameter**
  - haben eine ***Typ***
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - dadurch werden Funktionen ***flexibel einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden
  - diese werden dann im Rumpf ***verwendet***.

Syntax von M3-Funktionen - 1



Syntax von M3-Funktionen - 2

formals



formal



Vereinfachte Darstellung !

Formale & aktuelle Parameter

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (2.5));
END QuadratM.
```

Binden der aktuellen Werte (Parameter) an die formalen Parameter (Stellvertreter)

Funktionskomposition

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, ...] : x = (f : x, g : x, h : x, ...)
  - **Bedingung**
    - if t then f else g : x =

$$\begin{cases} f : x & \text{, falls } t : x = \text{TRUE} \\ g : x & \text{, falls } t : x = \text{FALSE} \\ ? & \text{, sonst} \end{cases}$$

schematisches Beispiel



Beispielnetz



VorNachfolger = GrossB o [Vorgänger, Nachfolger] o Konkat

Beispiel:

Eingabe: 'h'

Ausgabe: "GI"

Beispiel: Detaillierug

```
MODULE VorNachfolger EXPORTS Main;
IMPORT SIO;

PROCEDURE GrossB(c : CHAR) : CHAR =
...
PROCEDURE Vorgaenger (c : CHAR) : CHAR =
...
PROCEDURE Nachfolger (c : CHAR) : CHAR =
...
PROCEDURE AddOrd (c, d : CHAR) : CARDINAL =
. . .
PROCEDURE Konkat (c, d : CHAR) : TEXT =
...

BEGIN
  (* VorNachfolger = GrossB o   [ Vorgänger, Nachfolger ]   o   Konkat *)
  SIO.PutText ( Konkat ( Vorgaenger( GrossB( SIO.GetChar() )) ,
                         Nachfolger( GrossB( SIO.GetChar() ))
                       )
              );

END VorNachfolger.
```

Da **keine Variablen** (Zwischenspeicher) verwendet werden, muß der Wert **zweimal** eingelesen werden !

Funktion GrossB muß **zweimal** ausgeführt werden, da Modula-3 die **Konstruktion** von Funktionen nicht unterstützt.

Funktionale Programme

- **Idee:**
  - In Abhängigkeit von ***Bedingungen*** soll eine ***alternative*** Programmkomponente ausgeführt werden
  - Bedingungen müssen einen Wert vom Typ ***BOOLEAN*** liefern
  - Modula-3 bietet dazu u.a. die ***IF-Anweisung*** an

If statement



Bei funktionalen Programmen stehen hier ***ausschließlich*** Funktionsaufrufe!

Bedingung - IF-THEN-ELSE

- **Typische Formen von Bedingungen:**

```
IF bf THEN
  f1;
ELSE
  f2;
END;
```

Alternative "Entweder-Oder" (zweiseitig bedingte Anweisung)

```
IF bf THEN
  f1;
END;
```

einseitig bedingte Anweisung

```
IF    bf1 THEN
  f1;
ELSIF bf2 THEN
  f2;
ELSIF bf3 THEN
  f3;
...
ELSE
  fn;
END;
```

mehrseitig bedingte Anweisung

Die ***Bedingungen*** (bfi) werden der Reihe nach ausgewertet, bis eine ***WAHR*** ist. Dann wird die entsprechende Anweisung (Funktionsaufruf) ausgeführt.

Ist ***keine*** Bedingung wahr, dann wird der ***ELSE-Zweig*** ausgeführt

Beispiel - 1

- **Aufgabe:**
  - Eingabe ist eine ganze Zahl
  - Stelle fest, ob diese 1-, 2-, 3-, mindestens 4-stellig oder negativ ist!

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF IstEinstellig(i)     THEN RETURN ("einstellig") END;
  IF IstZweistellig(i)    THEN RETURN ("zweistellig") END;
  IF IstDreistellig(i)    THEN RETURN ("dreistellig") END;

  IF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE RETURN ("negativ")
  END;
END ErmittleStelligkeit;

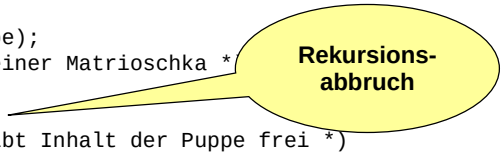
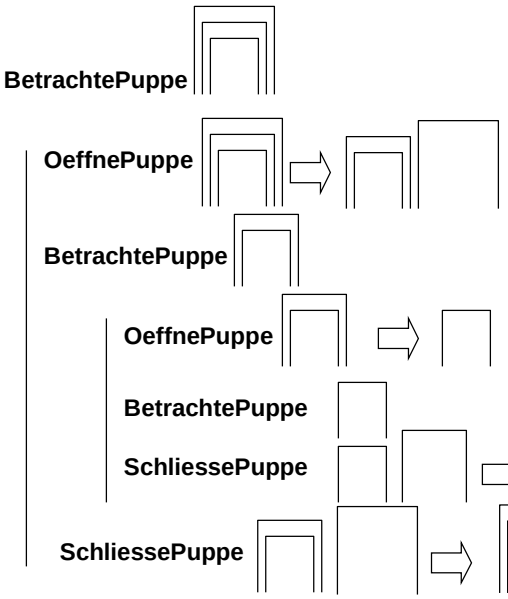
BEGIN
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```


Beispiel - 2

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF    IstEinstellig(i)     THEN RETURN ("einstellig")
  ELSIF IstZweistellig(i)    THEN RETURN ("zweistellig")
  ELSIF IstDreistellig(i)    THEN RETURN ("dreistellig")
  ELSIF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE                            RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  (* Aufruf der Hauptfunktion *)
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```

ELSIF-Konstruktionen machen Bedingungen semantisch klarer

Beispiel - 3

```
MODULE Stelligkeit EXPORTS Main;
(* Dieses Programm berechnet die Stelligkeit von Zahlen *)
IMPORT IO;

PROCEDURE IstEinstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=0) AND (i<10) );
END IstEinstellig;

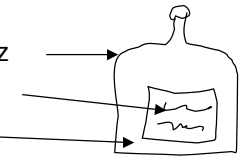
PROCEDURE IstZweistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=10) AND (i<100) );
END IstZweistellig;

PROCEDURE IstDreistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=100) AND (i<1000) );
END IstDreistellig;

PROCEDURE IstMinVierstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=1000) );
END IstMinVierstellig;
```


Idee und Anwendung

- **Idee:**
  - allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens ***durch Rückführung auf sich selbst***
- **Rekursive Funktion**
  - darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen***
- **Beispiel: Matrioschka-Puppen**

```
PROCEDURE BetrachtePuppe (puppe);
(* Pseudocode zum Betrachten einer Matrioschka *)
BEGIN (* BetrachtePuppe *)
  IF NOT Massiv (puppe) THEN
    OeffnePuppe (puppe); (* gibt Inhalt der Puppe frei *)
    BetrachtePuppe (inhaltDerPuppe); (* Rekursion *)
    SchliessePuppe (puppe);
  END (* IF *);
END BetrachtePuppe;
```



Beispiel




```
PROCEDURE BetrachtePuppe (puppe);
BEGIN
  IF NOT Massiv (puppe) THEN
   BetrachtePuppe(OeffnePuppe (puppe));
   SchliessePuppe (puppe);
  END (* IF *);
END BetrachtePuppe;
```

Jeder rekursive Aufruf erzeugt eine neue **Inkarnation** der Funktion.

Es wird solange rekursiv aufgerufen, bis der **Rekursions-abbruch** erreicht wird.

Anschließend werden die erzeugten Inkarnationen in **umgekehrter** Reihenfolge ausgeführt

Anmerkungen

- **Ziel:**
  - es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen
  - dies führt immer zu einem ***Laufzeitfehler***
- **Konsequenz**
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung***
    - Rekursionsabbruch: in definierten Fällen wird der rekursive Aufruf nicht ausgeführt
  - Muster:
    - ```
      IF … THEN
        rekursiver Aufruf
      ELSE
        Rekusionsabbruch
      ```
- **Bemerkung**
  - Rekursion führt zu ***prägnaten, knappen*** und ***eleganten*** Algorithmen für bestimmte Problemstellungen oder -lösungen
      ```

Exkurs: Speicherverwaltung

- **In traditionellen Programmiersprachen (z.B. FORTRAN)**
  - Zuordnung Variable ⟺ Speicher (statisch, zur Übersetzung)
  - Speicherplatz
  - Bezeichner
  - Wert



  - Vorteile: einfach, statisch zu überprüfen
  - Nachteile: - Speicherverschwendung
    - eine Variable kann nicht mehrfach existieren (Rekursion?)
    - keine Rekursion
- **Ansatz in block-orientierten Sprachen**
  - keine feste Zuordnung Variable ⟺ Speicher
  - Variablen werden dann auf den Speicher abgebildet, wenn der entsprechende Block ***aktiv ist*** (block-orientiert)

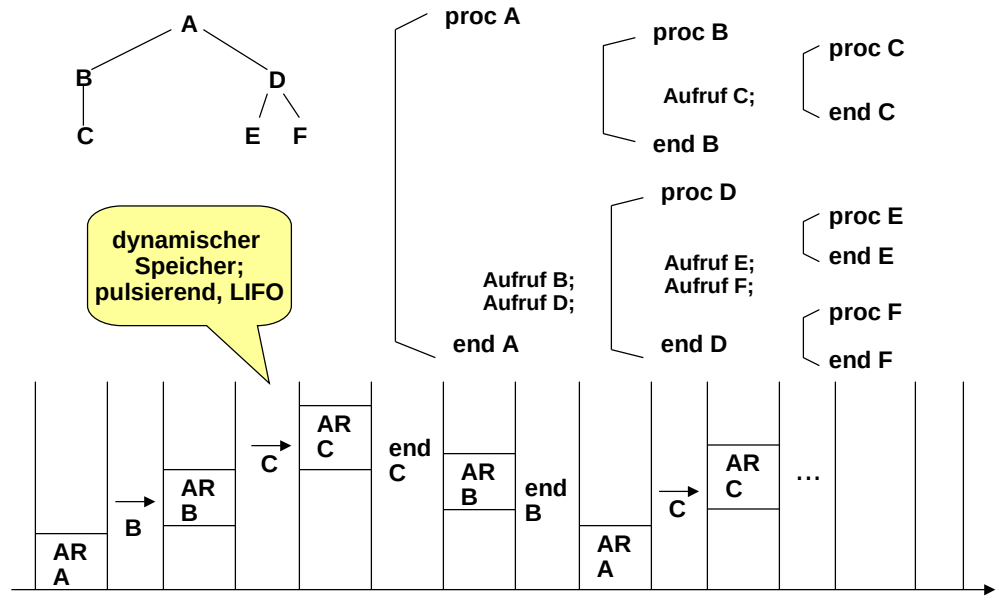
Programm-Kellerspeicher - 1

- **Blöcke sind baumartig angeordnet**
  - Vater-Block endet niemals vor dem Sohn-Block!
- **Wird ein Block aktiv, dann wird ein Aktivierungsblock (Inkarnation) erzeugt**

Parameter
Rückkehradresse
Lokale Variable
. .

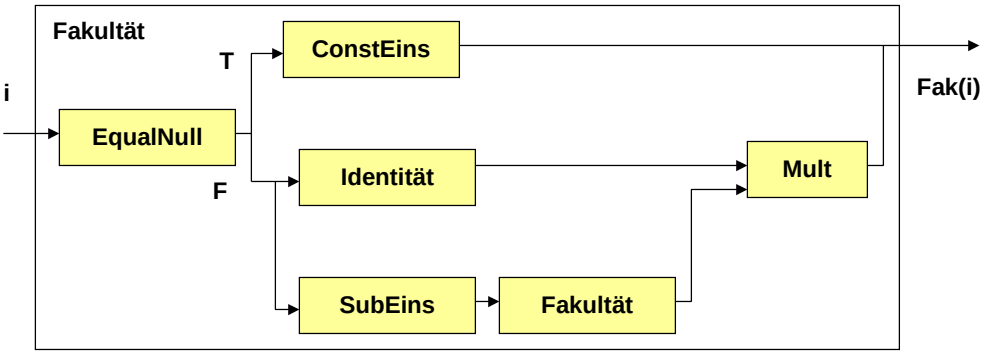
  - Inkarnationen werden nach dem LIFO-Prinzip (Keller-Prinzip) verwaltet
  - automatisch, unter Kontrolle des Laufzeitsystems (Laufzeitkeller)
  - nur die "jüngste" Inkarnation ist aktiv
  - Lebensdauer der Objekte ist direkt gekoppelt an die Inkarnation
  - es können mehrere Inkarnationen eines Blocks auf dem Stack liegen
  - es wird kein Speicher verschwendet

Programm-Kellerspeicher - 2

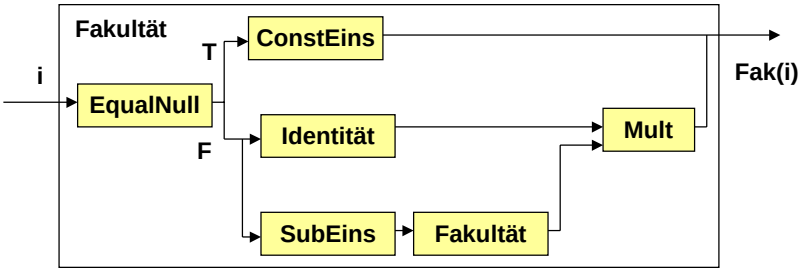


Beispiel Fakultät - 1

- **Fakultät is definiert:**
  - fak : n 1 , falls n = 0
    n * fak(n-1) n ∈ N
- **Funktionnetz für Fakultät**



Beispiel Fakultät - 2




```
PROCEDURE Fakultaet (i : INTEGER) : INTEGER =
BEGIN
  IF EqualNull(i)
  THEN RETURN(ConstEins(i))
  ELSE RETURN (Mult(Identitaet(i), Fakultaet(SubEins(i))));
  END;
END Fakultaet ;
...
  SIO.PutInt(Fakultaet(SIO.GetInt()));
```


Beispiel Fakultät - 3

```
PROCEDURE EqualNull ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( i = 0 );
END EqualNull;

PROCEDURE ConstEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( 1 );
END ConstEins;

PROCEDURE Mult ( i, j : INTEGER) : INTEGER =
BEGIN
  RETURN ( i * j );
END Mult;

PROCEDURE Identitaet ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i );
END Identitaet;

PROCEDURE SubEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i - 1 );
END SubEins ;
```

Diese Funktionen sind nur im Sinne der **reinen funktionalen** Programmierung notwendig.

Sie können in der Definition der Funktion "Fakultaet" durch entsprechende **Ausdrücke** ersetzt werden!

Beispiel Fakultät - 4

```
MODULE FakultaetM1 EXPORTS Main;
(* Dieses Programm berechnet die Fakultaetsfunktion *)

IMPORT SIO;


PROCEDURE Fakultaet ( i : INTEGER) : INTEGER =
BEGIN
  IF i = 0
  THEN RETURN 1
  ELSE RETURN (i * Fakultaet(i-1));
  END;
END Fakultaet ;


BEGIN
  SIO.PutInt(Fakultaet(SIO.GetInt()));
END FakultaetM1.
```


Fibonacci-Funktion (rekursiv)

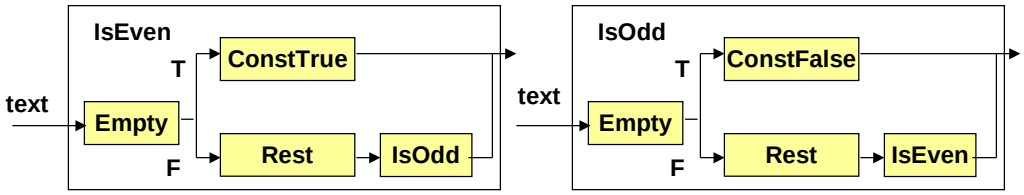
- **Wachstum einer Kaninchen-Population**
  - Wieviele Kaninchen-Pärchen gibt es nach n Jahren
    - Jahr 1 : 1 Pärchen
    - Jedes Pärchen hat ab dem zweiten Jahr je ein Pärchen Nachwuchs
  -

Jahr	1	2	3	4	5	6	7	8	9
Anzahl	1	1	2	3	5	8	13	21	34

```
PROCEDURE Fibonacci (arg: INTEGER): INTEGER =
BEGIN
  IF arg <= 2 THEN
    RETURN 1
  ELSE
    RETURN (Fibonacci (arg -1) + Fibonacci (arg - 2))
  END;
END Fibonacci ;
```

Indirekte Rekursion

- **Definition:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Beispiel:**
  - Die beiden Funktionen (IsEven, IsOdd) sind ***indirekt*** rekursiv
  - können festzustellen, ob eine Zeichenfolge eine gerade oder ungerade Anzahl von Zeichen enthält



Beispiel - Indirekte Rekursion

```
MODULE EvenOdd EXPORTS Main;
(* Beispiel fuer die indirekte Rekursion *)
IMPORT SIO, Text;

PROCEDURE IsEven (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN TRUE;
  ELSE RETURN (IsOdd (Text.Sub(str,1)));
  END;
END IsEven ;

PROCEDURE IsOdd (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN FALSE;
  ELSE RETURN (IsEven (Text.Sub(str, 1)));
  END;
END IsOdd ;

BEGIN
  SIO.PutBool ( IsEven (SIO.GetWord()));     SIO.Nl();
  SIO.PutBool ( IsOdd  (SIO.GetWord()));
END EvenOdd.
```

Funktional, Rekursion und Iteration

- **Funktionale Algorithmen**
  - kennen keine ***Variablen*** zur Zwischenspeicherung von Ergebnissen
- **Nichtfunktionale Algorithmen**
  - speichern Zwischenergebnisse in ***Variablen*** ab
- **Rekursive Algorithmen**
  - lösen ein Problem, in dem sie sich ***selbst wieder aufrufen*** (direkt oder indirekt)
- **Iterative Algorithmen**
  - besitzen Abschnitte, die bei der Ausführung ***mehrmals*** durchlaufen werden
  - Jeder rekursive Algorithmus kann in einen iterativen umgewandelt werden
    - allgemein (siehe Compiler)
    - problemspezifisch

Was haben wir gelernt

- **Funktionale Programmierung**
  - Funktion
  - Parameter
  - Vernetzung von Funktionen
- **Verwendung von Anweisungen und Ausdrücken**
- **Verwendung elementarer Datentypen**
- **Konzept der Rekursion**
  - direkt rekursive Funktionen
  - indirekt rekursive Funktionen
  - Realisierung durch Laufzeitkeller

Glossar

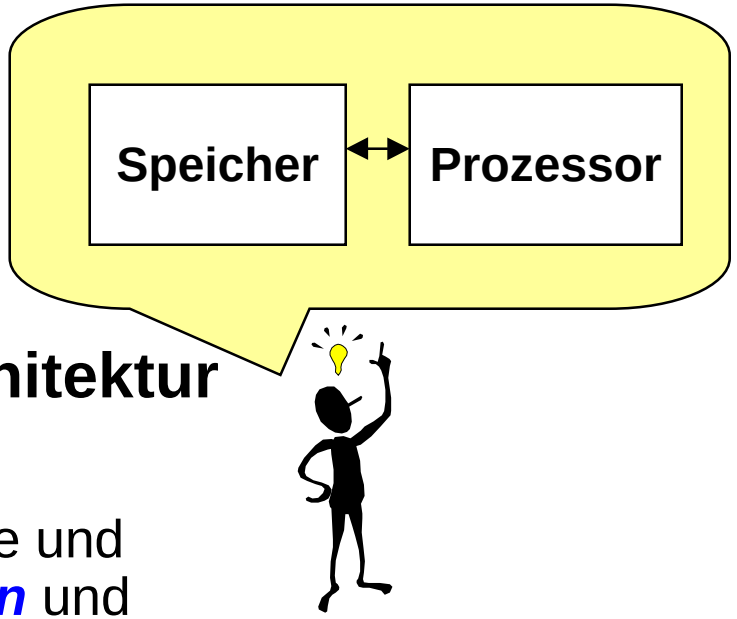
- **Funktionale Programmierung**
- **Funktion, Funktionskopf, -rumpf**
- **Funktionalform**
- **Parameter**
  - Formal- , Aktual-
- **Seiteneffektfreiheit**
- **Rekursion**
  - direkt, indirekt
- **Datentyp (Wiederholung)**
  - einfacher elementarer Typ
  - Ordinaltyp
- **Speicherverwaltung nach Kellerprinzip (Laufzeitkeller), statische Speicherverwaltung**
- **Aktivierungsblock**

Imperative Programmierung

- Konzepte der imperativen Programmierung
- Variable und Wertzuweisung
- Prozeduren
- rekursive Prozeduren
- Parameterübergabe
- Gültigkeitsbereich und Lebensdauer

Modell der imperativen Programmierung

- **Synomym:**
  - Befehls-orientierte Programmierung

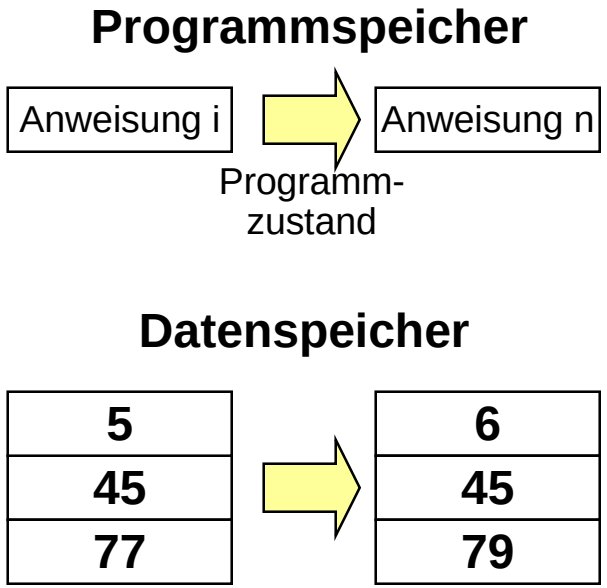


- **Geprägt durch die von-Neumann-Architektur**
  - Die CPU führt ***Maschinenbefehle*** aus
  - Deshalb müssen über den sog. Bus Befehle und Daten vom Speicher in die CPU ***übertragen*** und die Ergebnisse ***rückübertragen*** werden.

- **Mit imperativen Programmiersprachen setzen wir Entwürfe um:**
  - ***Aktionen*** fassen Folgen von Maschinenbefehlen zusammen,
  - ***Variable*** abstrahieren vom physischen Speicherplatz.

Semantik eines imperativen Progamms

- **Wesentliches Merkmal eines Programms**
  - ***Zustand*** der Daten im Speicher
  - Stand des Befehlszählers
- **Semantik eines Befehls**
  - Übergang von ZUSTAND1 -> ZUSTAND2
- **Programmierer beschreibt einen Prozeß von Zustandsübergängen**
  - durch elementare Anweisungen
  - durch den Kontrollfluß (Programmpfad)
- ***Variable*** **und** ***Wertzuweisung*** **modellieren Zustand und Zustands-übergang im Datenspeicher**



Imperatives Programmieren

Aufgaben des Programmierers

- Planung der ***Speicherbelegung***
  - Welche Daten braucht mein Programm?
- Planung der ***Unterprogramme***
  - Aus welchen Funktionen und Prozeduren soll das Programm bestehen?
  - Wie sollen diese die Werte der Daten verändern?
- Planung des ***Kontrollflusses***
  - In welcher Reihenfolge sollen die Operationen ausgeführt werden?
- Planung des ***Datenflusses***
  - Welche Daten müssen von welchen Operationen an andere übergeben werden?

Deklaration von Daten

Deklaration von Unterprogrammen

Anweisungen

Reihenfolge in Programmpfad bzw. Parameterübergabe

Objekte und Aktionen

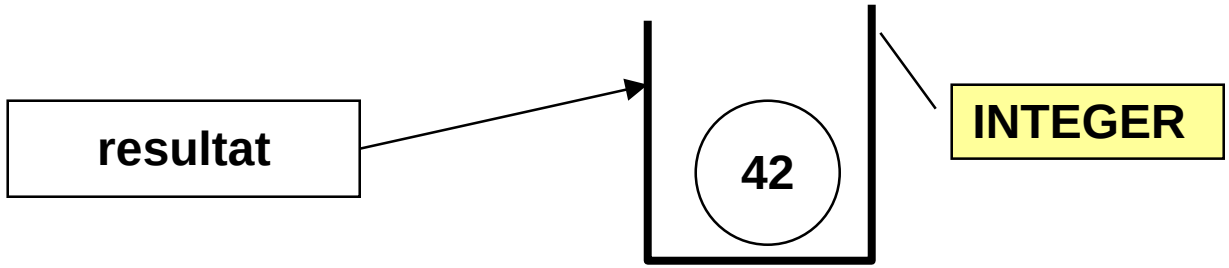
- **Wir unterscheiden bei Datenobjekten:**
  - ***Konstante***: Objekte, deren Wert während der Ausführung des Algorithmus unverändert bleibt.
  - ***Variable***: Objekte, deren Wert sich während der Ausführung des Algorithmus verändern kann.
  - Für beide gilt, daß ihre Werte einen ***Typ*** haben. Diese sind zunächst die elementaren Datentypen.

- **Wir unterscheiden bei Aktionen:**
  - Veränderung des Werts einer Variablen durch ***Zuweisung***.
  - Festlegung der nächsten Aktion durch den sog. ***Kontrollfluß*** (Ablaufsteuerung).

Charakterisierung

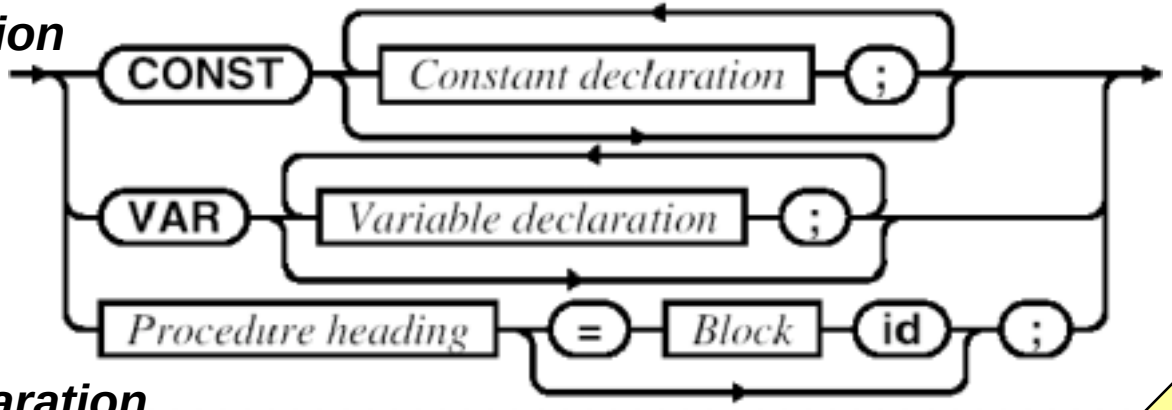
Variable

- Speicherplatz mit seinem Wert
- besitzt einen **Namen**, unter dem man ihn ansprechen kann
- bei Sprachen mit Typsystem muß jede Variable einem **Datentyp** zugeordnet sein
- Datentyp legt fest, welche **Struktur** und **Werte** eine Variable besitzt/ annehmen kann und wie Werte im Programm als Literale/Aggregate hingeschrieben werden
- Ferner legt ein Datentyp fest, welche **Operationen** ausgeführt werden dürfen
- Variablen werden im Deklarationsteil von Programmeinheiten (Blöcke, Module) vereinbart (deklariert)
- Variable kann als Behälter betrachtet werden: hat Wert und Typ

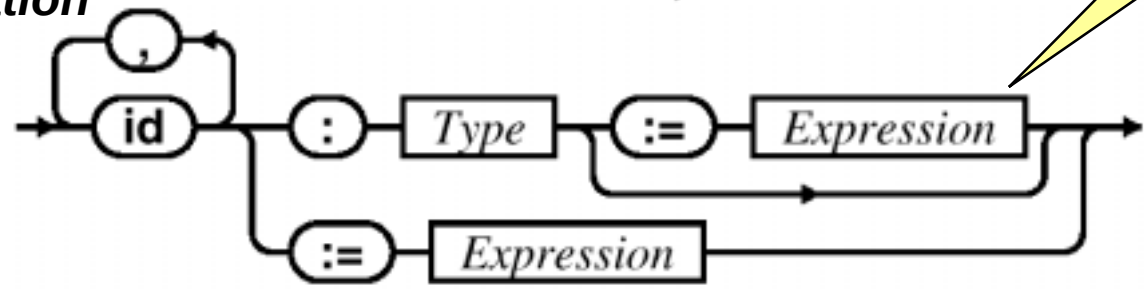


Deklarationen (erweitert)

Declaration



Variable declaration



Initialisierung der Variablen

Syntaktisch korrekt

Laufzeitfehler?

```
VAR r : INTEGER := 4;
    x : INTEGER := (4 * 8 -2);

    b := FALSE;
    c := TRUE;
    d := (b = c);

    y : REAL;
```

```
VAR x  : INTEGER := 4;
    xx : INTEGER := (2 * x);
    y  : INTEGER := (4 DIV y);
```

Wertzuweisung

Wertzuweisung

- dient dazu, den Wert einer Variablen zu verändern
- Syntax (vereinfacht): **Variable := Ausdruck**
- Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variablen zugewiesen
  - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
  - LH value, RH value
- Als Ausdruck auf der rechten Seite verwenden wir meist arithmetische und Boolesche Ausdrücke, Vergleiche und Zeichen bzw. Zeichenketten.
- ***Typkompatibilität***:

  Der Typ des Bezeichners muß zum Typ des Ausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Beispiele: Wertzuweisung

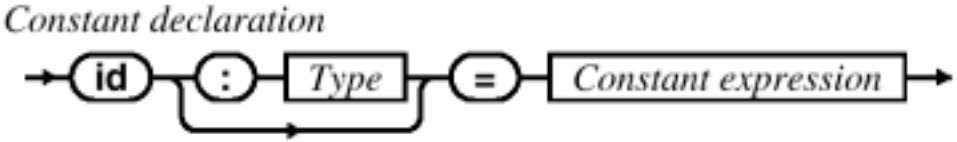
- **Deklaration**
  - `VAR x, y, z: CARDINAL;`
- **einige (korrekte und inkorrekte) Wertzuweisungen für x:**
  - ```
    x :=  0;
    x := MAX (CARDINAL);
    x :=  y;   (* sicher richtig *)
    ```
  - ```
    x := -1;
    x := MAX (CARDINAL)+1;
    x := -y-1; (* sicher falsch  *)
    ```
  - ```
    x := y+z; x := z-y;(* richtig oder falsch,  *)
                          (* je nach Wert von y, z *)
    ```
    ```

Beispiel: Wertzuweisung

```
MODULE Vertausche1 EXPORTS Main;
(* Vertauscht zwei eingebenene Werte *)

IMPORT SIO;

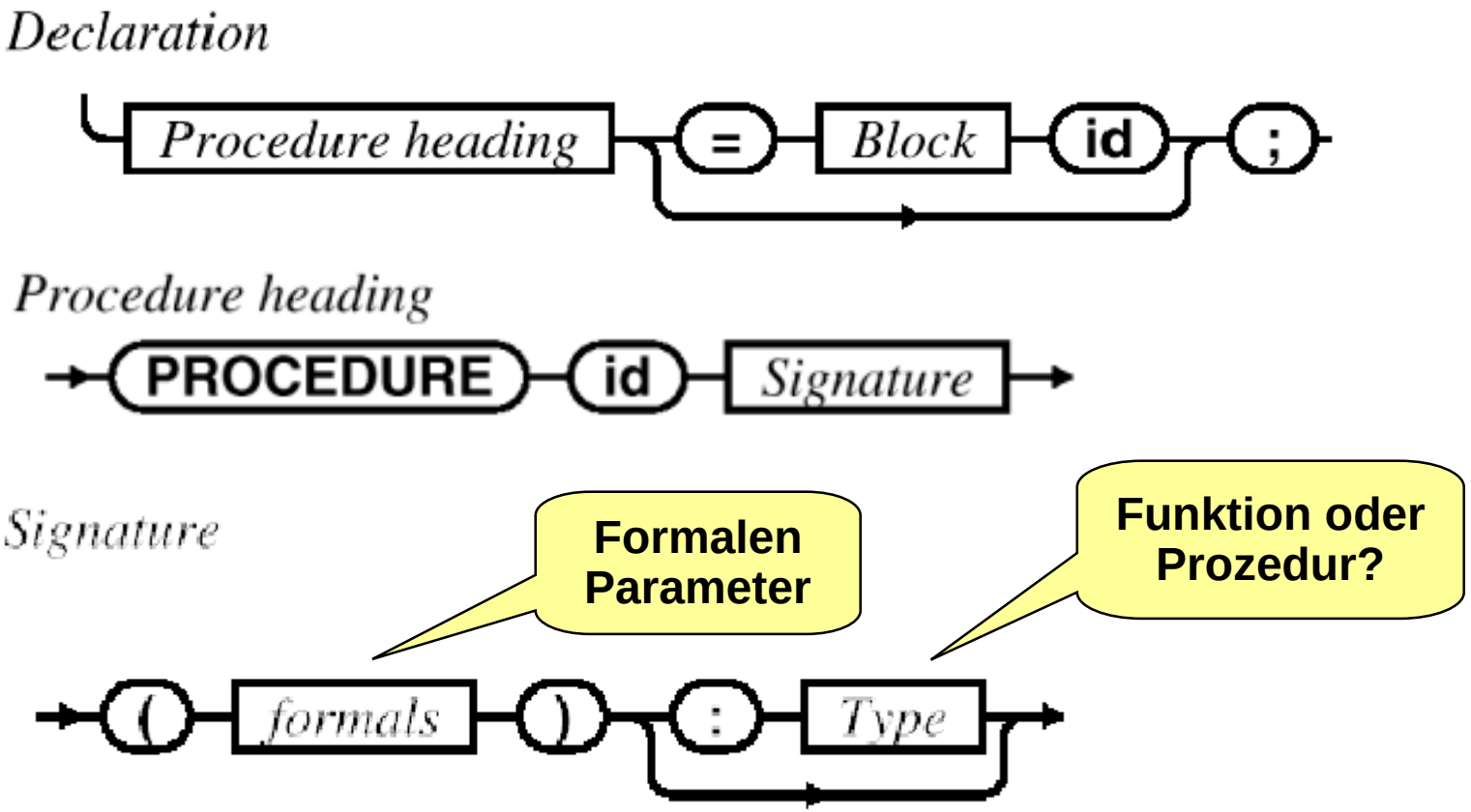
VAR x, y, h : INTEGER;

BEGIN
  x := SIO.GetInt();
  y := SIO.GetInt();

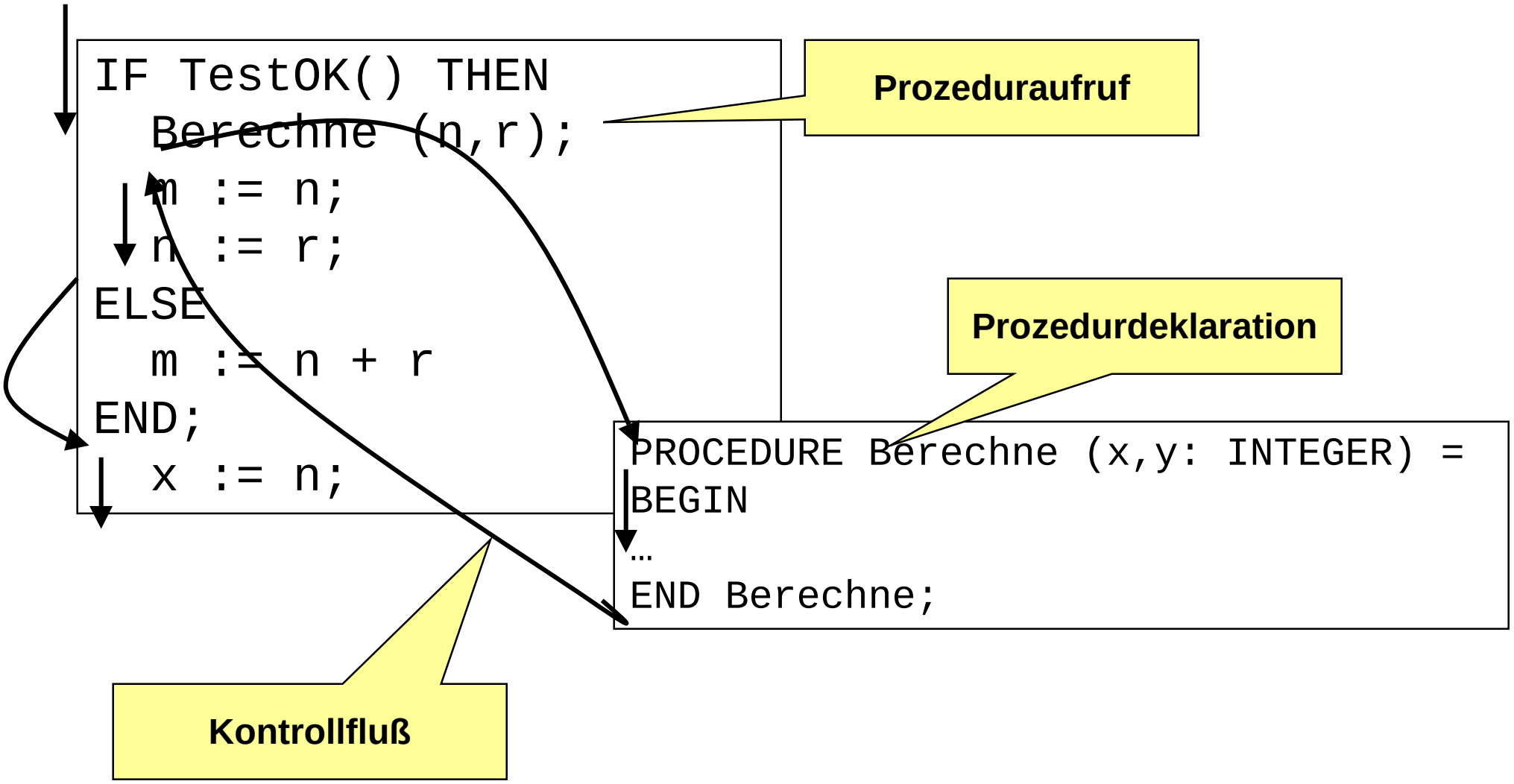
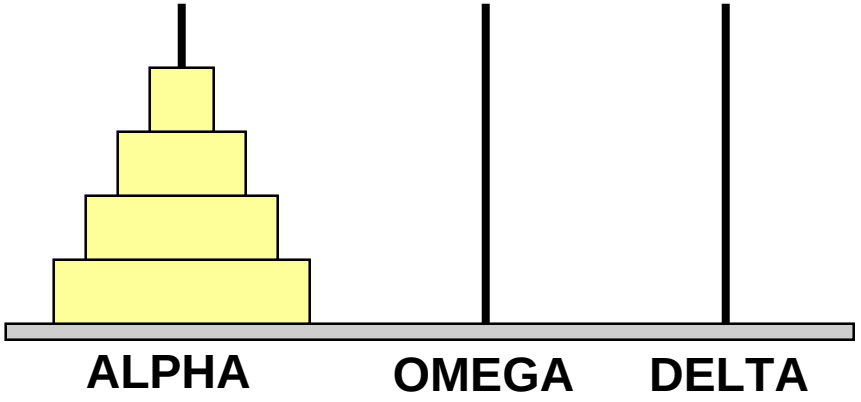
  h     := x;
  x     := y;
  y     := h;

  SIO.PutText("x = "); SIO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); SIO.PutInt(y);
END Vertausche1.
```

Deklaration der Variablen

Initialisierung der Variablen

Diskussion der RETURN-Anweisung

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
  VAR min: INTEGER;
BEGIN
  IF m <= n THEN
    min := m;
  ELSE
    min := n
  END;
  RETURN min;
END Minimum ;
```

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
BEGIN
  IF m <= n THEN
    RETURN m
  ELSE
    RETURN n
  END;
END Minimum ;
```

- **Anmerkung:**
  - mehrere RETURN-Anweisungen machen eine Funktion unübersichtlich
- **Empfehlung**
  - Code-Effizienz gegen Lesbarkeit abwägen!

Symbolische Konstanten

- **Verwendung von Zahlen-Konstanten ("Literalen") in Ausdrücken führt zu Problemen bei der Wartung!**

- **Konstante**
  - Bezeichner mit einem ***festen*** Wert
  - hat einen Datentyp
  - muß deklariert werden
  - überall, wo der Konstantenbezeichner auftritt, wird der Konstantenwert eingesetzt
  - Nach der Deklaration kann ihr ***kein Wert zugewiesen*** werden



- **Beispiel:**

```
CONST PI = 3.141;
VAR umfang, radius : REAL;

…

umfang := 2 * PI * radius
```

```
CONST
  Arbeitstage   = 5;
  Arbeitszeit   = 8;
  Wochenstunden = Arbeitstage *
                  Arbeitszeit;
```

Der Prozedurbegriff

- **Prozedur ist ein zentraler Begriff der prozeduralen Programmierung.**
- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.
  - Die Parameter erlauben die Prozedur mehrfach zu nutzen

Algorithmische Abstraktion

- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.
- **Abstraktion wird hier sowohl als *Vorgang* als auch als *Ergebnis des Vorgangs* verstanden:**
  - Im Vorgang der algorithmischen Abstraktion sehen wir von der konkreten Anweisungsfolge ab und bringen diese auf *"**einen Begriff**"*.
  - Das Ergebnis ist eine Entwurfs- und ***Programmeinheit*** – die Prozedur.

Beispiel: Abstraktionsprozeß

```
Programm Telefonieren
  Hörer_abheben;
  Telefonnummer_wählen;
  Gespräch_führen;
  Hörer_auflegen;
ENDE Telefonieren.
```

Algorithmische Abstraktion
durch **Prozeduren**:
Statt einer Anweisungsfolge
wird ein **Name** verwendet.

Ergebnis
Programmeinheit

```
Prozedur Telefonnummer_wählen
 IF Telefonnummer_gespeichert
THEN
   Kurzwahltaste_drücken
 ELSE
   Telefonnummer_eintippen
 END
ENDE Telefonnummer_wählen.
```

Kennzeichen von Prozeduren

- **Um den Prozedurbegriff verstehen zu können, benötigen wir drei Begriffe:**
- **Parametrisierung:**
  - Der Mechanismus zum ***Datenaustausch*** zwischen Prozedur und Umgebung.
- **Sichtbarkeit:**
  - Der Programmbereich, in dem ***Namen bekannt*** sind.
- **Lebensdauer:**
  - Der ***Zeitraum***, in dem die Werte von Programmobjekten zugegriffen werden können.

Prozeduren vs. Funktionen

- **Wurden bisher zur Ausgabe verwendet**
  - SIO.PutText ("Hallo")
- **Sind Funktionen "ähnlich"**
  - werden mit PROCEDURE eingeleitet, können auch rekursiv sein
- **Unterschied zu Funktionen**
  - Prozeduren liefern ***kein*** Ergebnis im Sinne eines Funktionsergebnisses!
  - Konsequenz:
    - Prozeduren besitzen keinen ***Ergebnistyp***
    - Aufruf einer Prozedur ist kein Ausdruck, sondern eine Anweisung
- **Zweck einer Prozedur**
  - ***Zusammenfassen*** einer "Funktionalität" (im Sinne der Lokalität)
  - ***Verändern*** der ihr übergebenen Parameter
  - Durchführung einer ***Nebenwirkung*** (z.B. Ausgabe einer Meldung)

Prozedurdeklaration

- **Syntax:**



- **Beispiele:**

```
PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL) = ...

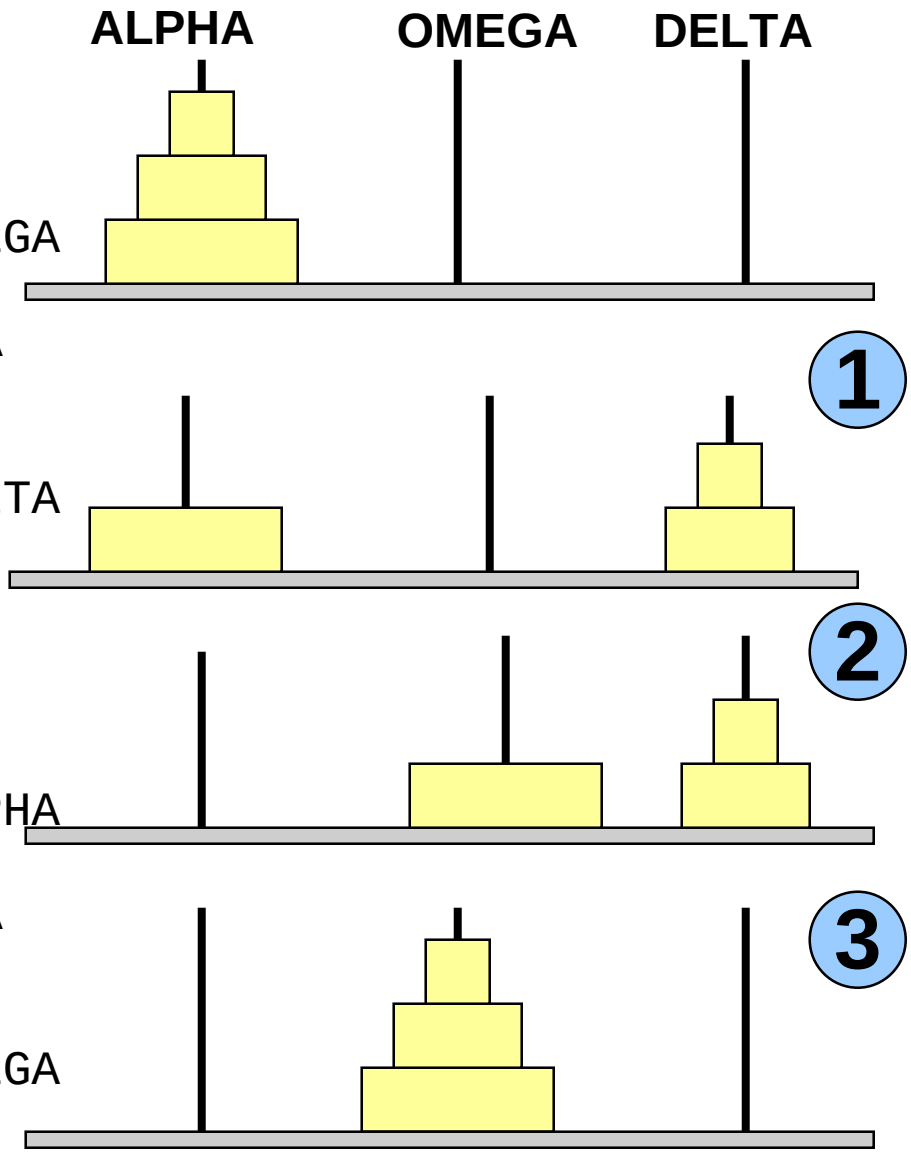
PROCEDURE Minimum(n,m : INTEGER): INTEGER = ...
```

Prozeduraufruf: Syntax

- **Beim Aufruf einer Prozedur werden die aktuellen Parameter an die formalen übergeben.**
- **Zur Übersetzungszeit wird überprüft:**
  - Der Name im Aufruf muß ***gleich*** dem Prozedurnamen sein.
  - Die ***Anzahl*** der aktuellen Parameter muß gleich der Anzahl der formalen sein.
  - Die Bindung der jeweiligen Parameter wird entsprechend ihrer ***Position*** im Aufruf und in der Prozedurdeklaration vorgenommen.
  - Die aktuellen Parameter müssen ***typkompatibel*** zu den formalen Parametern sein (d.h. meist typgleich).

```
PROCEDURE Minimum ( m, n : INTEGER) : INTEGER = …


res := Minimum (x, y); (* korrekter Aufruf)
res := Mini (x, y);
res := Minimum (x, y, z);
```

Prozeduraufruf: Semantik

- **Der Prozeduraufruf ist die explizite Anweisung,**
  - daß die Prozedur ***ausgeführt*** werden soll.
- **Eine Prozedur ist *aktiv*,**
  - nachdem sie gerufen wurde und in der Ausführung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- **Für den Prozeduraufruf in imperativen Sprachen ist charakteristisch:**
  - Beim Aufruf wechselt die ***Kontrolle*** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die aktuellen Parameter an die formalen ***gebunden***.
  - Prozeduren können wieder Prozeduren aufrufen. Dabei wird der Aufrufer unterbrochen (suspended), so daß die Kontrolle stets bei einer Prozedur ist.
  - Nach der Ausführung der Prozedur kehrt die Kontrolle zum Rufer zurück; die Ausführung wird mit der ***Anweisung nach dem Aufruf*** fortgesetzt.

Beispiel: Prozeduraufruf



Rekursion: Aufgabe

Aufgabe: Türme von Hanoi

- bewege die Scheiben des Turms von ALPHA nach OMEGA
- es darf immer ***nur eine Scheibe*** bewegt werden
- niemals darf eine Scheibe auf eine kleinere bewegt werden



Lösungsstrategie (Divide & Conquer)

- allgemeine Lösung für einen Turm der Höhe h von ALPHA nach OMEGA
  - h = 0 gar nichts machen
  - h > 0 1. Turm der Höhe h-1 von ALPHA nach DELTA über OMEGA
    2. Scheibe von ALPHA nach OMEGA legen
    3. Turm der Höhe h-1 von DELTA nach OMEGA über ALPHA

Rekursion: Programmtext

```
MODULE Hanoi EXPORTS Main;
(* Ausgabe der Zugfolge fuer Tuerme von Hanoi *)

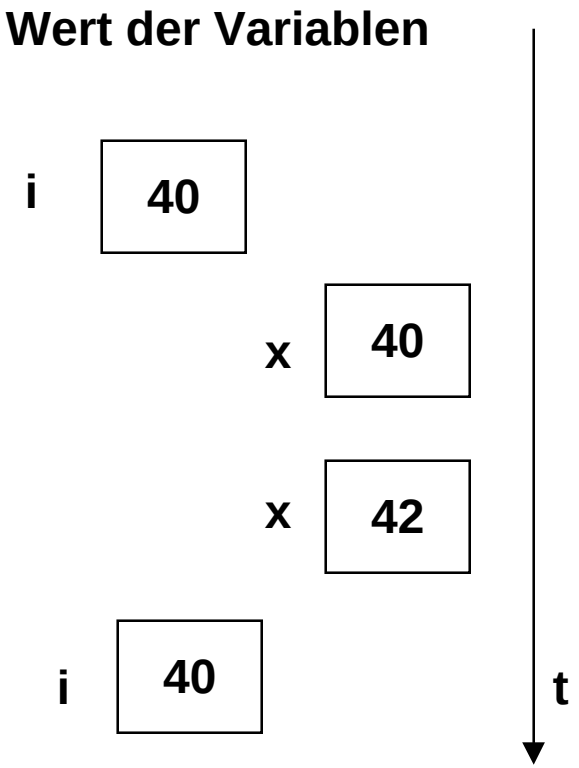
IMPORT SIO;

PROCEDURE DruckeZug (hoehe: CARDINAL; von, nach : TEXT)  =
BEGIN
  SIO.PutText ("Scheibe "); SIO.PutInt (hoehe);
  SIO.PutText (" von " & von & " nach " & nach); SIO.Nl();
END DruckeZug ;

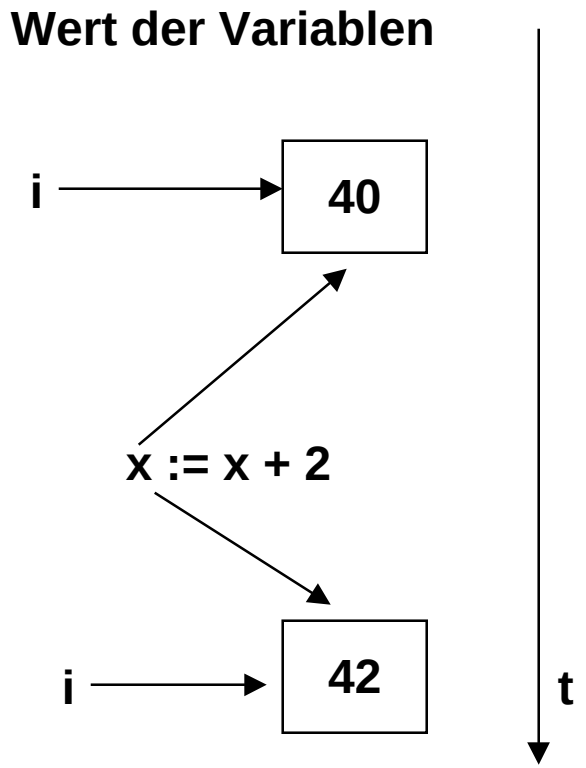
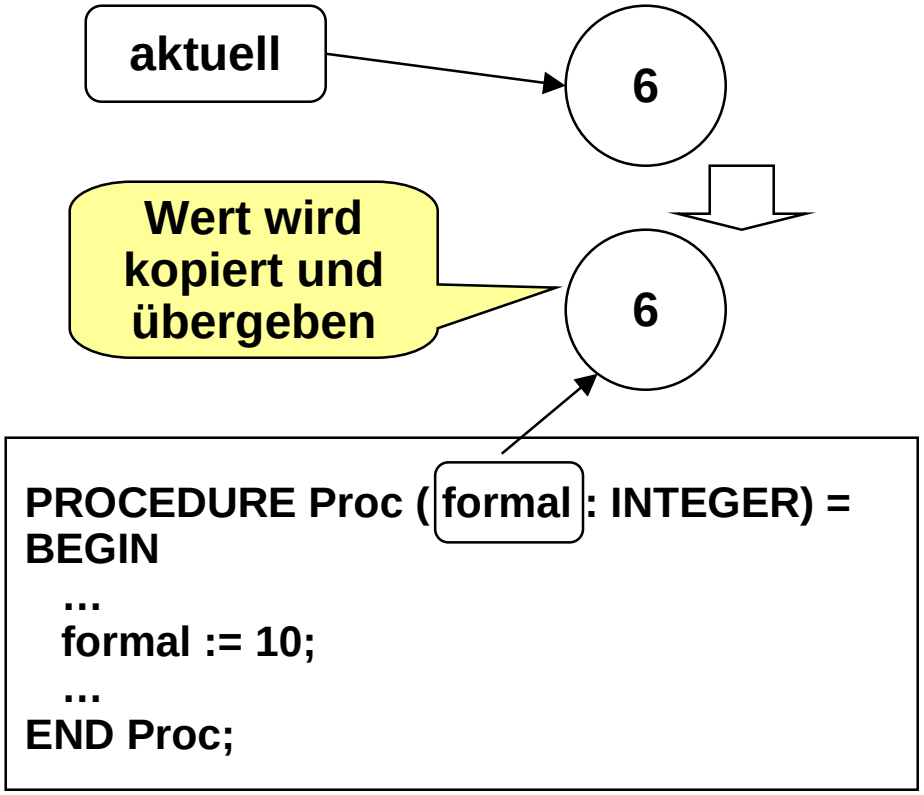
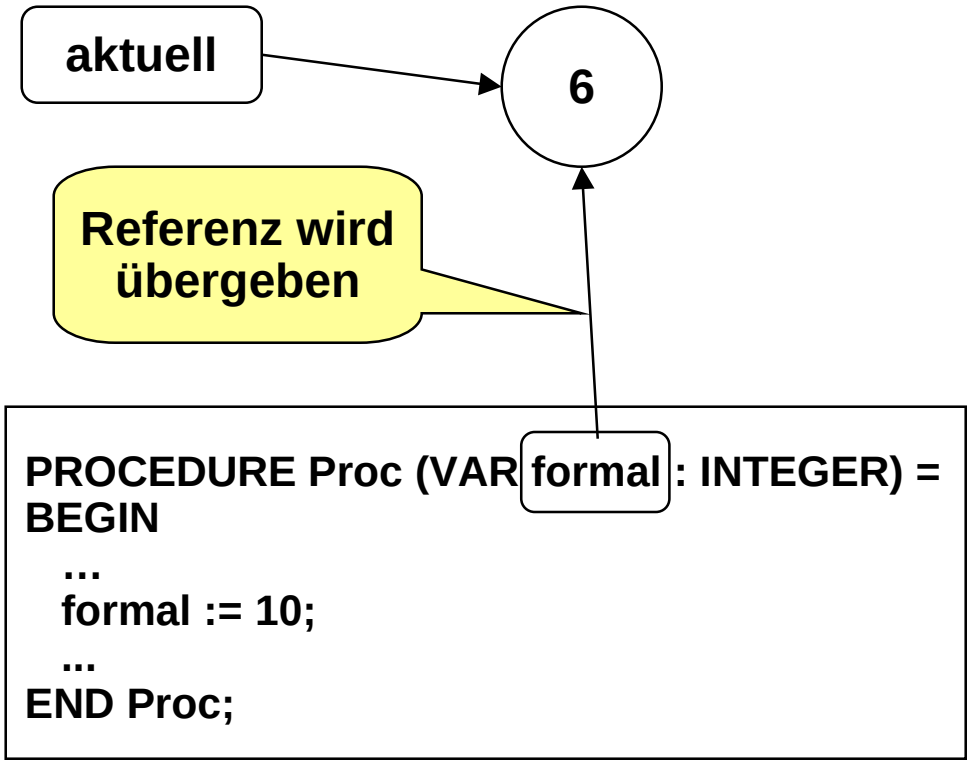
PROCEDURE BewegeTurm ( hoehe : CARDINAL; von, nach, ueber: TEXT)  =
BEGIN
  IF hoehe > 0 THEN
    BewegeTurm (hoehe-1, von, ueber, nach);
    DruckeZug  (hoehe, von, nach);
    BewegeTurm (hoehe-1, ueber, nach, von);
  END;
END BewegeTurm;

BEGIN
  BewegeTurm(SIO.GetInt(), "ALPHA", "OMEGA", "DELTA" );
END Hanoi.
```

Rekursion: Laufzeitgeschehen

```
3 ALPHA OMEGA DELTA
  ❶ 2 ALPHA DELTA OMEGA
    ❶ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
    ❷ Scheibe 2 von ALPHA nach DELTA
    ❸ 1 OMEGA DELTA ALPHA
      ❶ 0 OMEGA ALPHA DELTA
      ❷ Scheibe 1 von OMEGA nach DELTA
      ❸ 0 ALPHA DELTA OMEGA
  ❷ Scheibe 3 von ALPHA nach OMEGA
  ❸ 2 DELTA OMEGA ALPHA
    ❶ 1 DELTA ALPHA OMEGA
      ❶ 0 DELTA OMEGA ALPHA
      ❷ Scheibe 1 von DELTA nach ALPHA
      ❸ 0 OMEGA ALPHA DELTA
    ❷ Scheibe 2 von DELTA nach OMEGA
    ❸ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
```



Parameter

- **Bisher**
  - Funktionen besitzen ausnahmslos ***Eingabeparameter***
  - Wert dieser Parameter kann nicht ***geändert*** werden
- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
    - vor dem Aufruf wird der aktuelle Parameter ausgewertet und dem formalen Parameter zugewiesen (***call-by-value***)
  - ***Ausgabeparameter***
    - dienen dazu, Ergebnisse einer Prozedur an den Aufrufer zurückzugeben
    - Wert ist zum Zeitpunkt des Aufrufs undefiniert (***call-by-reference***)
  - ***Ein- / Ausgabeparameter (Transienten)***
    - vereinen Eigenschaften beider Arten
- **Modula-3 nutzt dazu zwei Parameterübergabemechanismen**

Übergabemechanismen: Call by Value

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
    - realisieren Eingangsparameter
    - Der aktuelle Parameter muß ein ***Ausdruck*** sein (Spezialfall: Variable, d.h. Bezeichner für ein Objekt).
    - Beim Aufruf der Prozedur wird ein dem Typ des formalen Parameters entsprechendes ***lokales Objekt*** angelegt. Ist der aktuelle Parameter eine Variable, so entsteht dabei eine ***Kopie*** des Parameter-Objekts.
    - In jedem Falle wird der Wert des aktuellen Parameters ***berechnet*** und dem formalen Parameter (-Objekt) zugewiesen.
    - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
    - Schlüsselwort ***VALUE*** zeigt einen Wertparameter an
    - steht kein Schlüsselwort, ist es ***per default*** ein Wertparameter

Beispiel: Call by Value

```
PROCEDURE Proc1 (VALUE x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Nichts passiert")
ELSE
  SIO.PutLine ("kein Call by Value")
END;
```



Übergabemechanismen: Call by Reference

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
    - realisieren Ausgangs und Ein- / Ausgangsparameter
    - Der aktuelle Parameter muß ein ***Bezeichner für ein Objekt*** sein.
    - Beim Aufruf wird der formale Parameter durch einen ***Verweis*** auf den aktuellen Parameter ersetzt.
      D.h. der formale Parameter wird als lokaler Bezeichner für das aktuelle Parameterobjekt ***substituiert***.
    - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.
    - Veränderungen des formalen Parameters in der Prozedur haben auf den aktuellen Parameter Auswirkung, d.h. Objekte im Namensraum des Rufers können ***verändert*** werden. Auf diese Weise können von einer Prozedur ***Ergebnisse*** zurückgegeben werden.
    - Schlüsselwort ***VAR*** zeigt einen Variablenparameter an

Beispiel: Call by Reference

```
PROCEDURE Proc1 (VAR x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Call by Value")
ELSE
  SIO.PutLine ("Call by Reference")
END;
```



Call-by-value <-> Call-by-reference

```
VAR aktuell : INTEGER
…
aktuell := 6;
Proc (aktuell);
```





Beispiel: Wert- Variablenparameter

```
MODULE Parameter EXPORTS Main;
IMPORT IO, SIO;

VAR aktuell: INTEGER;

PROCEDURE Proc1 (VALUE formal : INTEGER) =
BEGIN
  formal := 10;
END Proc1;

PROCEDURE Proc2 (VAR formal : INTEGER) =
BEGIN
  formal := 10;
END Proc2;

BEGIN
  aktuell := 6;          Proc1 (aktuell);
  SIO.PutText("aktuell (Proc1) = "); IO.PutInt(aktuell); SIO.Nl();
  aktuell := 6;          Proc2 (aktuell);
  SIO.PutText("aktuell (Proc2) = "); IO.PutInt(aktuell);
END Parameter.
```

Wertparameter

Variablen-parameter

```
aktuell (Proc1) = 6
aktuell (Proc2) = 10
```

Ausgabe des Programms

Beispiel: Variablenparameter

```
MODULE Vertausche2 EXPORTS Main;
IMPORT IO, SIO;

PROCEDURE Vertausche (VAR w1, w2 : INTEGER) =
VAR hilfe : INTEGER := w1;
BEGIN
  w1    := w2;
  w2    := hilfe;
END Vertausche;

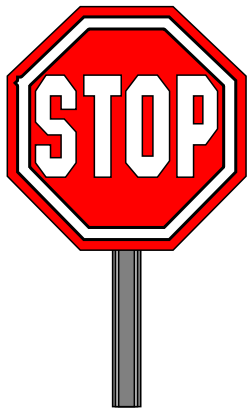
VAR x, y : INTEGER;

BEGIN
  x := IO.GetInt();
  y := IO.GetInt();
  Vertausche (x, y);
  SIO.PutText("x = "); IO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); IO.PutInt(y);
END Vertausche2.
```

Vertauscht die Werte der beiden Parameter

Beispiel: Ausgabeparameter

- **Ausgabeparameter**
  - dient dazu, einen Wert an den Aufrufer zurückzugeben

```
MODULE QuadratM1 EXPORTS Main;

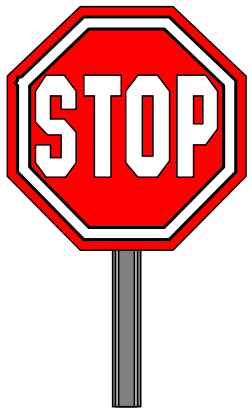
IMPORT SIO;
CONST eingabe = 2.5;
VAR  ergebnis : REAL;

PROCEDURE Quadrat ( VALUE x : REAL; VAR wert : REAL )=
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Ausgabeparameter oder Ein- Ausgabeparameter ?

Wert ist beim Aufruf undefiniert

Datenaustausch: Beispiel - 1

In einer Reihe von Meßwerten soll der **laufende Mittelwert** berechnet werden. Benötigte Objekte und Aktionen:

```
Wert, Mittelwert, Summe : REAL;
Anzahl : INTEGER;
(* BerechneMWert
      Addiere neuen Wert und Summe,
      Erhöhe Anzahl um 1,
      Mittelwert := Summe / Anzahl *)
```

Austausch über Parameter:

```
PROCEDURE BerechneMWert (     wert      : REAL;
                          VAR anzahl    : INTEGER;
                          VAR summe, mw : REAL) =

BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  mw := summe / anzahl;
END BerechneMWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;
BerechneMWert (wert,anzahl,summe,mw);
BerechneMWert (2*wert,anzahl,summe,mw);
```

Datenaustausch: Beispiel - 2

Datenaustausch über Funktionsergebnis:

```
PROCEDURE MWert (    wert  : REAL;
                 VAR anzahl: INTEGER;
                 VAR summe : REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl:= anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```



Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;

SIO.PutReal (MWert(wert,anzahl,summe));
...
SIO.PutReal (MWert(2*wert,anzahl,summe));
```

Datenaustausch: Beispiel - 3

Datenaustausch über Funktionsergebnis:

```
VAR summe: REAL; anzahl: INTEGER;

PROCEDURE MWert (wert:REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```



Verwendung:

```
summe := 0.0; anzahl := 0;

SIO.PutReal (MWert(4.0);
...
SIO.PutReal (MWert(10.0));
```

Diskussion der Beispiele - 1

- **Beispiel 1:**
  - Die Verwendung von VAR-Parametern in einer Prozedur zur Rückgabe von Ergebnissen ist in der imperativen Programmierung üblich.
  - Sie führt oft zu ***Verständnisproblemen***, da sowohl in der Deklaration als auch in der Verwendung klar sein muß, was das eigentliche Ergebnisobjekt (hier: der Mittelwert) ist.
- **Beispiel 2:**
  - Die Verwendung einer Funktion zur Berechnung genau eines Wertes ist dann sauber, wenn alle anderen Parameter als ***Wertparameter*** verwendet werden.
  - Die Modellierung einer Zustandsveränderung (hier: Summe und Anzahl) außerhalb der Funktion mit Hilfe von VAR-Parametern ist ein ***Seiteneffekt***, der die ***Lokalität*** der Funktion zerstört.

Diskussion der Beispiele - 2

- **Beispiel 3:**
  - Die Verwendung von ***globalen*** Variablen, die in einer Prozedur oder Funktion als Seiteneffekt verändert werden, ist die ***schlechteste*** Lösung.
  - Zustandsveränderungen über globale Variablen, die in der Signatur der Prozedur nicht aufgeführt sind, sind unverständlich und extrem ***fehleranfällig***.
- **Funktionsprozeduren**
  - Eine Funktion kann in imperativen Sprachen nur dann sauber modelliert werden, wenn
    - alle Parameter als ***Wertparameter*** übergeben werden,
    - in der Funktion ***keine globalen Variablen*** verwendet werden.
  - In Funktionen sollte auch keine globalen Variablen ***lesend verwendet*** werden, da dies Funktionen an ihren Verwendungskontext ankoppelt.
  - Merke: Funktionen sollten "***in sich***" verständlich sein.

Regeln für die Parameterverwendung

- **Wertparameter sind Variablenparameter vorzuziehen**
  - Änderung an Parametern in Prozeduren ist häufig eine ***Fehlerquelle***, die schwer zu finden ist.
- **Variablenparameter sollten nur verwendet werden, wenn**
  - Prozedur-Ergebnisse ***übergeben*** werden sollen (Ausgabeparameter)
  - Kopieren des Parameters ***nicht möglich*** ist (bei gewissen Datenstrukturen)
  - Kopieren zu ***ineffizient*** ist (bei sehr großen Datenstrukturen)
- **Funktionen haben nur Wertparameter**
  - liefern ihr Ergebnis durch ihren ***Namen*** zurück
  - geht in Modula-3 nur, wenn
    - genau ein Wert zurückgegeben werden soll
  - Rückgabe durch Namen und Variablenparameter muß ***vermieden*** werden

Zusammenfassung: Prozeduren und Parameter

- **In imperativen Sprachen wird oft die Prozedur in den Vordergrund gestellt.**
- **Die Funktion ist gelegentlich (z.B. Modula-3) nur eingeschränkt verwendbar, kann dafür aber *Seiteneffekte* erzeugen (nicht wünschenswert).**
- **Nur durch eine saubere Definition der *Signaturen* von Routinen kann ein verständlicher und weiterverwendbarer Entwurf erreicht werden, d.h. vor allem, jeder Datenaustausch mit der Umgebung sollte *explizit* sein.**
- **Die Modellierung von Zuständen und ihrer Veränderung ist nur in Verbindung mit einem entsprechenden *Modulkonzept* softwaretechnisch sauber zu lösen.**

  Das lernen wir später!

Def. Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - ***Sichtbarkeitsbereich*** ist Teil des Gültigkeitsbereichs
  - Gültigkeitsbereich/Sichtbarkeitsbereich wird durch den Compiler überwacht
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
  - macht nur Sinn für Objekte, die Speicher belegen
    - Typen belegen keinen Speicher
- **Es ist wichtig, beide Begriffe klar zu unterscheiden!**

Gültigkeitsbereich: lokale Deklarationen

- **Regeln für die Gültigkeit von Bezeichnern;**
  - Alle in einer Prozedur oder einem Modul deklarierten Bezeichner sind in der gesamten Prozedur / im gesamten Modul gültig.
  - Das gilt auch für den textuell vor der Deklaration eines Bezeichers liegenden Bereich
  - Davon ausgenommen sind Prozedur- und Modulkopf

```
PROCEDURE Proc ( in : REAL )=

  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Hier können die Bezeichner X und C verwendet werden.

```
PROCEDURE Proc ( in : REAL )=

  CONST CC : C * C
  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Korrekte Vorwärts-referenz

Gültigkeitsbereich: Erweiterung

- **Durch IMPORT kann der Gültigkeitsbereich von Bezeichnern auf das *importierende Modul* erweitert werden.**

```
INTERFACE SIO;
...
PROCEDURE GetReal …;

PROCEDURE PutReal ...;

PROCEDURE GetText ...;

PROCEDURE PutText ...;

…
END SIO;
```

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR   ergebnis : REAL;

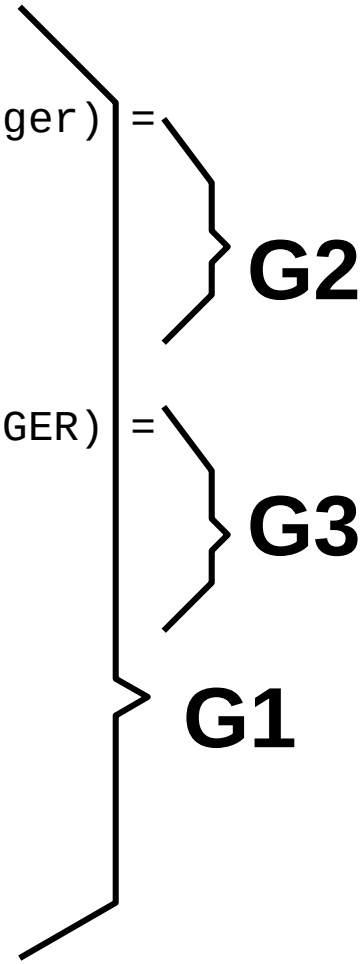
PROCEDURE Quadrat … =
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Qualifizierter Bezeichner muß verwendet werden.

Prozeduren als Namensraum - 1

Namensraum

- Eine Prozedur bildet gegenüber der (textuellen) Umgebung ihrer Deklaration einen eigenen ***Namensraum***, d.h. sie kann lokale Objekte benennen und verwalten.
- Die Namen der formalen Parameter sowie die im Deklarationsteil des Prozedurrumpfs deklarierten Bezeichner sind nur ***im Prozedurrumpf gültig***, d.h. bekannt.
- In einer Prozedur können Bezeichner der Umgebung ***neu lokal*** deklariert werden; diese Bezeichner ***verdecken*** die global deklarierten Bezeichner, die zugehörigen Objekte sind in der Prozedur ***nicht sichtbar***.

Lebensdauer

- Die Lebensdauer von prozedurlokalen Objekten entspricht dem Zeitraum, in dem der Aufruf der Prozedur abgearbeitet wird. Sie werden zu Beginn der Prozedurausführung angelegt.
- Werte gehen ***verloren***, wenn die Ausführung der Prozedur beendet ist.

Prozeduren als Namensraum - 2

```
PROCEDURE Proc ()=

  VAR x : INTEGER;

  PROCEDURE Proc1 (x :Integer) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Proc;
```



- **Module und Prozeduren definieren eigene Namensräume**
  - Prozeduren können verschachtelt werden
  - Hierarchie von Gültigkeits-bereichen (Namensräumen)
- **Überlappung von Gültigkeitsbereichen**
  - Liegt innerhalb des Gültigkeitsbereiches G1 von X mit der Bedeutung B1 ein weiterer Gültigkeitsbereich G2 von X mit der Bedeutung B2, so handelt es sich um zwei ***verschiedene*** Objekte
  - innerhalb von G2 gilt nur B2, alle anderen Bedeutungen sind ***unsichtbar***.

Lebensdauer - 1

- **Durch Ausführung einer Prozedur P entsteht eine *Inkarnation*.**
- **Zur Inkarnation gehören *zur Laufzeit*:**
  - ein ***Ausführungspunkt*** (also ein Zeiger auf den gerade auszuführenden oder ausgeführten Befehl)
  - ***Speicherplätze*** für alle Bezeichner von Variablen und Wertparameter
  - ***Bezüge*** auf die konkreten Variablenparameter.
- **Informationen existieren bis zum Ende der Ausführung von P.**
- **Beispiel**
  - ```
    PROCEDURE Test (ch: CHAR; VAR x:INTEGER)=
      VAR y: REAL
    ```
- **`Test ('a', z)` führt zu einer Inkarnation mit**
  - Speicherplatz für ch und y
  - unter dem lokalen Bezeichner `x` einen Bezug (einer Referenz) auf z
  - nach Abschluß werden diese Speicherplätze wieder freigegeben
    ```

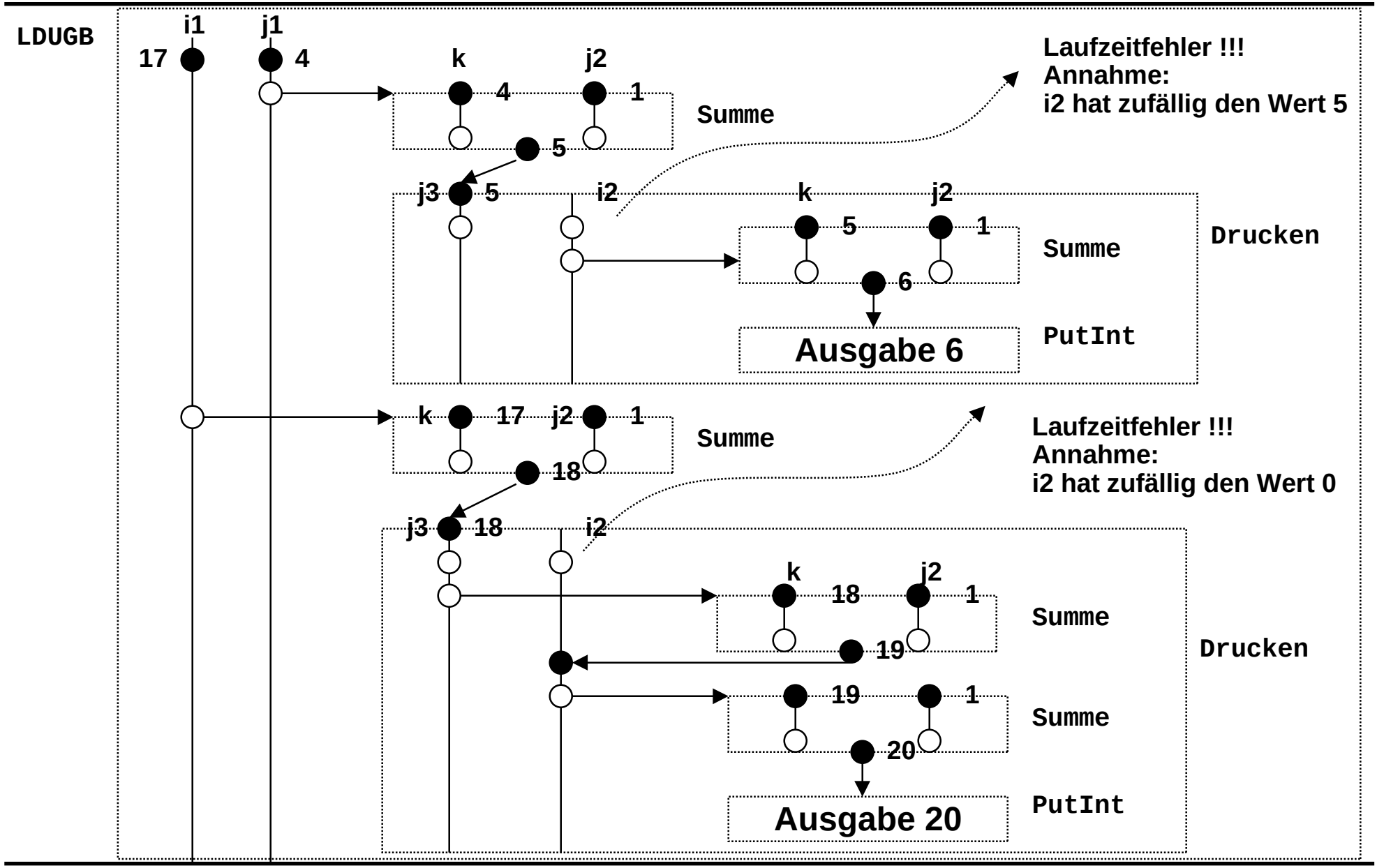
Lebensdauer - 2

- **Bemerkungen:**
  - **Sei M ein Modul,**
    - dann wird ***Speicherplatz*** für die Variablen permanent für die gesamte Laufzeit des Programms reserviert.
    - Eine eigentliche Inkarnation wird nur von dem Rumpf des Moduls gebildet; dieser hat weder Variablen noch Parameter.
  - Zu irgendeinem Zeitpunkt existieren i.a. neben der Inkarnation des ablaufenden Moduls ***Inkarnationen verschiedener*** Prozeduren, bei **rekursiven** Aufrufen auch mehrere der gleichen Prozedur.
  - Nur in der ***jüngsten*** aller existierenden Inkarnationen wandert der Ausführungspunkt weiter; diese wird als erste beendet.
  - Die Lebensdauer einer Variablen ist ***identisch*** mit der Existenz der zugehörigen Prozedur-Inkarnation.
  - Zu Beginn der Lebensdauer ist der Wert einer Variablen ***undefiniert***, darf also nicht verwendet werden.

Beispiel: Lebensdauer - 1

```
MODULE LDUGB EXPORTS Main;
IMPORT SIO;
VAR i  , j  : INTEGER;

PROCEDURE Summe (k  : INTEGER) : INTEGER =
VAR j  : INTEGER;
BEGIN
  IF i  <10 THEN j  := 10 ELSE j  :=1 END;
  RETURN j  + k  ;
END Summe;

PROCEDURE Drucken (j  : INTEGER) =
VAR i  : INTEGER;
BEGIN
  IF i   #   j   THEN i  := Summe(j  ) END;
  SIO.PutInt (Summe(i  )); SIO.Nl();
END Drucken;

BEGIN
  i  := 17; j  := 4;
  Drucken (Summe(j  ));
  Drucken (Summe(i  );
END LDUGB.
```

Beispiel: Lebensdauer - 2



Terminologie

- **"Objekt" bezeichnet alles,**
  - was durch einen Bezeichner eingeführt wird (Modul, Prozedur, Konstante, Typ, Variable, Parameter),
  - keine Objekte sind demnach Operatoren oder Wortsymbole (z.B. VAR, END)
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

Beispiel: lokal, global, global relativ

```
MODULE Gueltigkeit EXPORTS Main;
  IMPORT SIO;
  VAR x : INTEGER;

  PROCEDURE Proc1 (x: INTEGER) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Gueltigkeit.
```

x ist global sichtbar im Modul

x ist lokal in der Prozedur Proc1

x ist global relativ in der Prozedur Proc2

y ist lokal in der Prozedur Proc2

Lokalität

- **Ziel**
  - Programme sollten mit ***möglichst wenig*** Aufwand korrigier- und modifizierbar sein.
- **Strategie**
  - ***hohe Lokalität*** durch enge Gültigkeitsbereiche.
  - Größtmögliche Lokalität ist daher ***vorrangiges Ziel*** einer guten Programmierung!
- **Ein Programm sollte dafür folgende Merkmale aufweisen:**
  - Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind ***überschaubar***.
  - Die Objekte sind so ***lokal*** wie möglich definiert, jeder Bezeichner hat nur eine ***einzige***, bestimmte Bedeutung.
  - Die ***Kommunikation*** zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variable.

Vorteile lokaler Variabler

- **Lokale Variablen haben softwaretechnisch einige Vorteile.**
  - Deklaration und Verwendung stehen in einem ***textlichen*** Zusammenhang. Das erhöht die Lesbarkeit.
  - Die ***unfreiwillige*** Verwendung von globalen Variablen wird vermieden, d.h. globale Variablen müssen nicht vollständig bekannt sein.
  - Der Speicherverbrauch durch Prozeduren wird ***minimiert***, da Speicher für lokale Variablen nur während ihrer Aktivierung vorgehalten werden muß.
  - Lokalität:
    - Variablen sollen ***nur in dem Kontext*** deklariert werden, wo sie bekannt sein müssen. Eine Verteilung erschwert die Änderbarkeit.
  - Kapselung:
    - Außerhalb eines Kontextes (Modul, Prozedur) sollen nur relevante Objekte sichtbar sein. Implementationsdetails werden im Inneren ***verborgen*** und sind nicht zugreifbar.

Was haben wir gelernt!

- **Modell der imperativen Programmierung**
  - Zustand, Zustandsübergang
- **Konzept der Variablen**
  - Wertzuweisung
- **Parameterübergabemechanismen**
  - Wertparameter
  - Variablenparameter
  - Wie geht man damit um
- **Gültigkeit von Bezeichnern**
- **Lebensdauer von Objekten**
- **Konzept der Lokalität**

Glossar

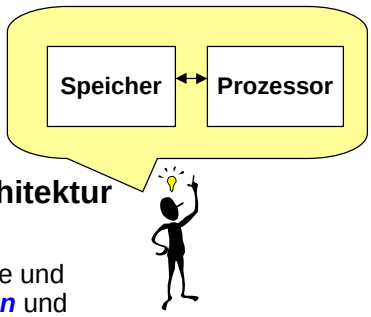
- Modelle der imperativen Programmierung (von-Neumann-Sprachen)
- Programm- und Datenspeicherzustände und -übergänge bei imperativen Programmen
- imperative Programmierung: Aufgaben und Hilfsmittel der Programmiersprache
- Programmiersprachliche Objekte
- Variablenbegriff: typisiert, Initialisierung, LH-Value, RH-Value, Wertzuweisung, Veränderung bei Parameterübergabe
- Deklarationen: verschiedene Arten, Zweck dieser Arten
- Konstantenbegriff von Modula-3
- Prozeduren: Aufgaben, Parameter, Gültigkeit, Lebensdauer, Unterscheidung zu Funktionen
- Prozedurdeklaration: Prozedurkopf/Signatur, Prozedurrumpf, Formalparameterliste, lokale Deklarationen, Verwendung der Formalparameter im Rumpf
- Prozeduraufruf: Aktualparameterliste, Konsistenz, Aufruf und Deklaration, Semantik durch Inkarnation
- Parameter: Arten, Übergabemechanismen, Methodikregeln, Call-by-Value (Aufruf über den Wert), Call-by-Reference (Aufruf über die Adresse)
- rekursive Prozeduren (direkt und indirekt): Verwendung bei Divide & Conquer-Entwicklungsstrategie, statisches Verständnis der Rekursion, Laufzeitgeschehen bei rekursiven Prozeduren
- Datenaustausch über Prozeduren
- Namensraum von Prozeduren, Gültigkeitsbereich, Erweiterung durch Importe, lokale Variable, Vorteile im Sinne der Programmiermethodik
- Lebensdauer von Prozedurinkarnationen, darin enthaltener Variablen

Imperative Programmierung

- Konzepte der imperativen Programmierung
- Variable und Wertzuweisung
- Prozeduren
- rekursive Prozeduren
- Parameterübergabe
- Gültigkeitsbereich und Lebensdauer

Modell der imperativen Programmierung

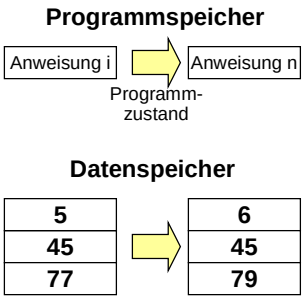
- **Synomym:**
  - Befehls-orientierte Programmierung



- **Geprägt durch die von-Neumann-Architektur**
  - Die CPU führt ***Maschinenbefehle*** aus
  - Deshalb müssen über den sog. Bus Befehle und Daten vom Speicher in die CPU ***übertragen*** und die Ergebnisse ***rückübertragen*** werden.
- **Mit imperativen Programmiersprachen setzen wir Entwürfe um:**
  - ***Aktionen*** fassen Folgen von Maschinenbefehlen zusammen,
  - ***Variable*** abstrahieren vom physischen Speicherplatz.

Semantik eines imperativen Progamms

- **Wesentliches Merkmal eines Programms**
  - ***Zustand*** der Daten im Speicher
  - Stand des Befehlszählers
- **Semantik eines Befehls**
  - Übergang von ZUSTAND1 -> ZUSTAND2
- **Programmierer beschreibt einen Prozeß von Zustandsübergängen**
  - durch elementare Anweisungen
  - durch den Kontrollfluß (Programmpfad)
- ***Variable*** **und** ***Wertzuweisung*** **modellieren Zustand und Zustands-übergang im Datenspeicher**



Imperatives Programmieren

- **Aufgaben des Programmierers**
  - Planung der ***Speicherbelegung***
    - Welche Daten braucht mein Programm?
  - Planung der ***Unterprogramme***
    - Aus welchen Funktionen und Prozeduren soll das Programm bestehen?
    - Wie sollen diese die Werte der Daten verändern?
  - Planung des ***Kontrollflusses***
    - In welcher Reihenfolge sollen die Operationen ausgeführt werden?
  - Planung des ***Datenflusses***
    - Welche Daten müssen von welchen Operationen an andere übergeben werden?

Deklaration von Daten

Deklaration von Unterprogrammen

Anweisungen

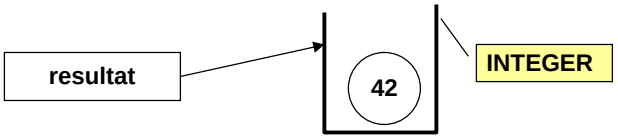
Reihenfolge in Programmpfad bzw. Parameterübergabe

Objekte und Aktionen

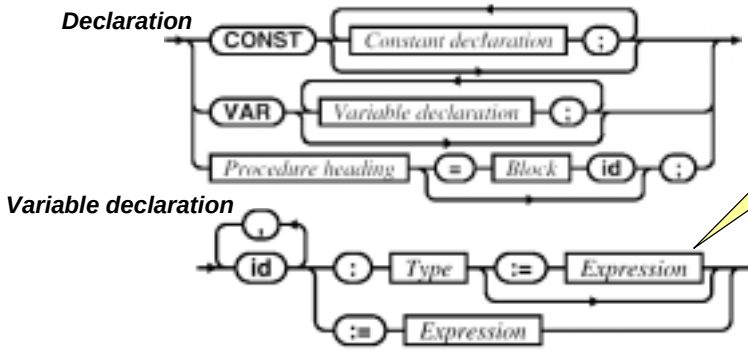
- **Wir unterscheiden bei Datenobjekten:**
  - ***Konstante***: Objekte, deren Wert während der Ausführung des Algorithmus unverändert bleibt.
  - ***Variable***: Objekte, deren Wert sich während der Ausführung des Algorithmus verändern kann.
  - Für beide gilt, daß ihre Werte einen ***Typ*** haben. Diese sind zunächst die elementaren Datentypen.
- **Wir unterscheiden bei Aktionen:**
  - Veränderung des Werts einer Variablen durch ***Zuweisung***.
  - Festlegung der nächsten Aktion durch den sog. ***Kontrollfluß*** (Ablaufsteuerung).

Charakterisierung

- **Variable**
  - Speicherplatz mit seinem Wert
  - besitzt einen **Namen**, unter dem man ihn ansprechen kann
  - bei Sprachen mit Typsystem muß jede Variable einem **Datentyp** zugeordnet sein
  - Datentyp legt fest, welche **Struktur** und **Werte** eine Variable besitzt/ annehmen kann und wie Werte im Programm als Literale/Aggregate hingeschrieben werden
  - Ferner legt ein Datentyp fest, welche **Operationen** ausgeführt werden dürfen
  - Variablen werden im Deklarationsteil von Programmeinheiten (Blöcke, Module) vereinbart (deklariert)
  - Variable kann als Behälter betrachtet werden: hat Wert und Typ



Deklarationen (erweitert)



Initialisierung der Variablen

Syntaktisch korrekt

Laufzeitfehler?

```
VAR r : INTEGER := 4;
    x : INTEGER := (4 * 8 -2);

    b := FALSE;
    c := TRUE;
    d := (b = c);

    y : REAL;
```

```
VAR x  : INTEGER := 4;
    xx : INTEGER := (2 * x);
    y  : INTEGER := (4 DIV y);
```


Wertzuweisung

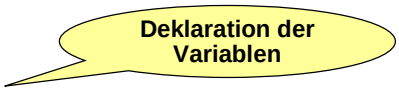
- **Wertzuweisung**
  - dient dazu, den Wert einer Variablen zu verändern
  - Syntax (vereinfacht): **Variable := Ausdruck**
  - Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variablen zugewiesen
    - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
    - LH value, RH value
  - Als Ausdruck auf der rechten Seite verwenden wir meist arithmetische und Boolesche Ausdrücke, Vergleiche und Zeichen bzw. Zeichenketten.
  - ***Typkompatibilität***:
    Der Typ des Bezeichners muß zum Typ des Ausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Beispiele: Wertzuweisung

- **Deklaration**
  - `VAR x, y, z: CARDINAL;`
- **einige (korrekte und inkorrekte) Wertzuweisungen für x:**
  -
```
x :=  0;
x := MAX (CARDINAL);
x :=  y;   (* sicher richtig *)
```
  -
```
x := -1;
x := MAX (CARDINAL)+1;
x := -y-1; (* sicher falsch  *)
```
  -
```
x := y+z; x := z-y;(* richtig oder falsch,  *)
                        (* je nach Wert von y, z *)
```


Beispiel: Wertzuweisung

```
MODULE Vertausche1 EXPORTS Main;
(* Vertauscht zwei eingebenene Werte *)

IMPORT SIO;

VAR x, y, h : INTEGER;

BEGIN
  x := SIO.GetInt();
  y := SIO.GetInt();

  h     := x;
  x     := y;
  y     := h;

  SIO.PutText("x = "); SIO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); SIO.PutInt(y);
END Vertausche1.
```



Diskussion der RETURN-Anweisung

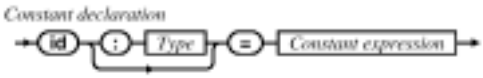
```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
  VAR min: INTEGER;
BEGIN
  IF m <= n THEN
    min := m;
  ELSE
    min := n
  END;
  RETURN min;
END Minimum ;
```

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
BEGIN
  IF m <= n THEN
    RETURN m
  ELSE
    RETURN n
  END;
END Minimum ;
```

- **Anmerkung:**
  - mehrere RETURN-Anweisungen machen eine Funktion unübersichtlich
- **Empfehlung**
  - Code-Effizienz gegen Lesbarkeit abwägen!

Symbolische Konstanten

- **Verwendung von Zahlen-Konstanten ("Literalen") in Ausdrücken führt zu Problemen bei der Wartung!**
- **Konstante**



  - Bezeichner mit einem ***festen*** Wert
  - hat einen Datentyp
  - muß deklariert werden
  - überall, wo der Konstantenbezeichner auftritt, wird der Konstantenwert eingesetzt
  - Nach der Deklaration kann ihr ***kein Wert zugewiesen*** werden
- **Beispiel:**

```
CONST PI = 3.141;
VAR umfang, radius : REAL;

…

umfang := 2 * PI * radius
```

```
CONST
  Arbeitstage   = 5;
  Arbeitszeit   = 8;
  Wochenstunden = Arbeitstage *
                  Arbeitszeit;
```

Der Prozedurbegriff

- **Prozedur ist ein zentraler Begriff der prozeduralen Programmierung.**
- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.
  - Die Parameter erlauben die Prozedur mehrfach zu nutzen

Algorithmische Abstraktion

- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.
- **Abstraktion wird hier sowohl als *Vorgang* als auch als *Ergebnis des Vorgangs* verstanden:**
  - Im Vorgang der algorithmischen Abstraktion sehen wir von der konkreten Anweisungsfolge ab und bringen diese auf *"**einen Begriff**"*.
  - Das Ergebnis ist eine Entwurfs- und ***Programmeinheit*** – die Prozedur.

Beispiel: Abstraktionsprozeß

```
Programm Telefonieren
  Hörer_abheben;
  Telefonnummer_wählen;
  Gespräch_führen;
  Hörer_auflegen;
ENDE Telefonieren.
```

Algorithmische Abstraktion
durch **Prozeduren**:
Statt einer Anweisungsfolge wird ein **Name** verwendet.

Ergebnis
Programmeinheit

```
Prozedur Telefonnummer_wählen
 IF Telefonnummer_gespeichert
THEN
   Kurzwahltaste_drücken
 ELSE
   Telefonnummer_eintippen
 END
ENDE Telefonnummer_wählen.
```


Kennzeichen von Prozeduren

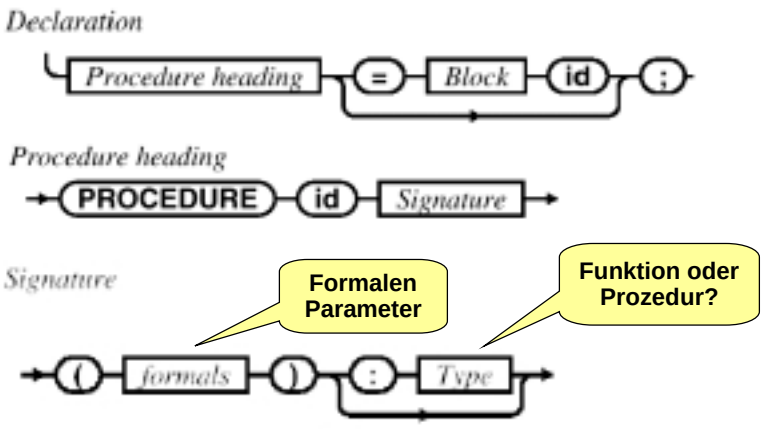
- **Um den Prozedurbegriff verstehen zu können, benötigen wir drei Begriffe:**
- **Parametrisierung:**
  - Der Mechanismus zum ***Datenaustausch*** zwischen Prozedur und Umgebung.
- **Sichtbarkeit:**
  - Der Programmbereich, in dem ***Namen bekannt*** sind.
- **Lebensdauer:**
  - Der ***Zeitraum***, in dem die Werte von Programmobjekten zugegriffen werden können.

Prozeduren vs. Funktionen

- **Wurden bisher zur Ausgabe verwendet**
  - SIO.PutText ("Hallo")
- **Sind Funktionen "ähnlich"**
  - werden mit PROCEDURE eingeleitet, können auch rekursiv sein
- **Unterschied zu Funktionen**
  - Prozeduren liefern ***kein*** Ergebnis im Sinne eines Funktionsergebnisses!
  - Konsequenz:
    - Prozeduren besitzen keinen ***Ergebnistyp***
    - Aufruf einer Prozedur ist kein Ausdruck, sondern eine Anweisung
- **Zweck einer Prozedur**
  - ***Zusammenfassen*** einer "Funktionalität" (im Sinne der Lokalität)
  - ***Verändern*** der ihr übergebenen Parameter
  - Durchführung einer ***Nebenwirkung*** (z.B. Ausgabe einer Meldung)

Prozedurdeklaration

- **Syntax:**



- **Beispiele:**

```
PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL) = ...

PROCEDURE Minimum(n,m : INTEGER): INTEGER = ...
```

Prozeduraufruf: Syntax

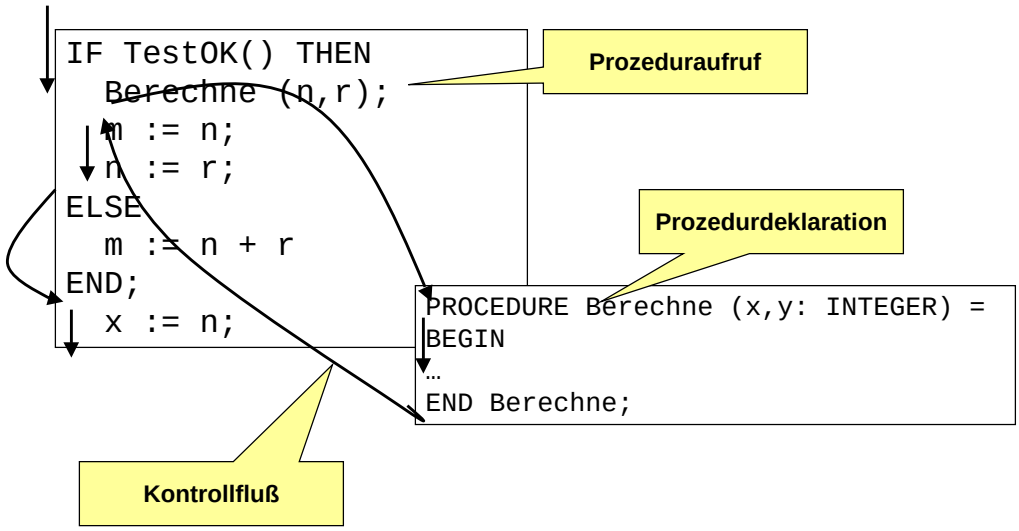
- **Beim Aufruf einer Prozedur werden die aktuellen Parameter an die formalen übergeben.**
- **Zur Übersetzungszeit wird überprüft:**
  - Der Name im Aufruf muß ***gleich*** dem Prozedurnamen sein.
  - Die ***Anzahl*** der aktuellen Parameter muß gleich der Anzahl der formalen sein.
  - Die Bindung der jeweiligen Parameter wird entsprechend ihrer ***Position*** im Aufruf und in der Prozedurdeklaration vorgenommen.
  - Die aktuellen Parameter müssen ***typkompatibel*** zu den formalen Parametern sein (d.h. meist typgleich).

```
PROCEDURE Minimum ( m, n : INTEGER) : INTEGER = …


res := Minimum (x, y); (* korrekter Aufruf)
res := Mini (x, y);
res := Minimum (x, y, z);
```


Prozeduraufruf: Semantik

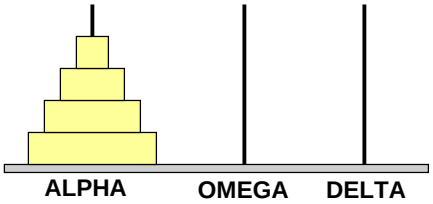
- **Der Prozeduraufruf ist die explizite Anweisung,**
  - daß die Prozedur ***ausgeführt*** werden soll.
- **Eine Prozedur ist *aktiv*,**
  - nachdem sie gerufen wurde und in der Ausführung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- **Für den Prozeduraufruf in imperativen Sprachen ist charakteristisch:**
  - Beim Aufruf wechselt die ***Kontrolle*** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die aktuellen Parameter an die formalen ***gebunden***.
  - Prozeduren können wieder Prozeduren aufrufen. Dabei wird der Aufrufer unterbrochen (suspended), so daß die Kontrolle stets bei einer Prozedur ist.
  - Nach der Ausführung der Prozedur kehrt die Kontrolle zum Rufer zurück; die Ausführung wird mit der ***Anweisung nach dem Aufruf*** fortgesetzt.

Beispiel: Prozeduraufruf



Rekursion: Aufgabe

- **Aufgabe: Türme von Hanoi**
  - bewege die Scheiben des Turms von ALPHA nach OMEGA
  - es darf immer ***nur eine Scheibe*** bewegt werden
  - niemals darf eine Scheibe auf eine kleinere bewegt werden



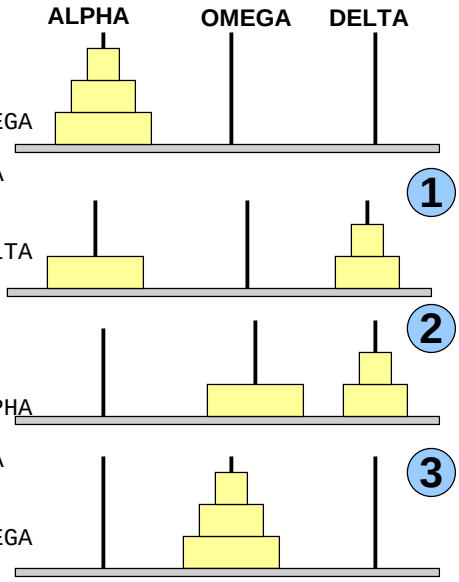
- **Lösungsstrategie (Divide & Conquer)**
  - allgemeine Lösung für einen Turm der Höhe h von ALPHA nach OMEGA
    - h = 0 gar nichts machen
    - h > 0 1. Turm der Höhe h-1 von ALPHA nach DELTA über OMEGA
      2. Scheibe von ALPHA nach OMEGA legen
      3. Turm der Höhe h-1 von DELTA nach OMEGA über ALPHA

Rekursion: Programmtext

```
MODULE Hanoi EXPORTS Main;
(* Ausgabe der Zugfolge fuer Tuerme von Hanoi *)

IMPORT SIO;

PROCEDURE DruckeZug (hoehe: CARDINAL; von, nach : TEXT)  =
BEGIN
  SIO.PutText ("Scheibe "); SIO.PutInt (hoehe);
  SIO.PutText (" von " & von & " nach " & nach); SIO.Nl();
END DruckeZug ;

PROCEDURE BewegeTurm ( hoehe : CARDINAL; von, nach, ueber: TEXT)  =
BEGIN
  IF hoehe > 0 THEN
    BewegeTurm (hoehe-1, von, ueber, nach);
    DruckeZug  (hoehe, von, nach);
    BewegeTurm (hoehe-1, ueber, nach, von);
  END;
END BewegeTurm;

BEGIN
  BewegeTurm(SIO.GetInt(), "ALPHA", "OMEGA", "DELTA" );
END Hanoi.
```


Rekursion: Laufzeitgeschehen

```
3 ALPHA OMEGA DELTA
  ❶ 2 ALPHA DELTA OMEGA
    ❶ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
    ❷ Scheibe 2 von ALPHA nach DELTA
    ❸ 1 OMEGA DELTA ALPHA
      ❶ 0 OMEGA ALPHA DELTA
      ❷ Scheibe 1 von OMEGA nach DELTA
      ❸ 0 ALPHA DELTA OMEGA
  ❷ Scheibe 3 von ALPHA nach OMEGA
  ❸ 2 DELTA OMEGA ALPHA
    ❶ 1 DELTA ALPHA OMEGA
      ❶ 0 DELTA OMEGA ALPHA
      ❷ Scheibe 1 von DELTA nach ALPHA
      ❸ 0 OMEGA ALPHA DELTA
    ❷ Scheibe 2 von DELTA nach OMEGA
    ❸ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
```



Parameter

- **Bisher**
  - Funktionen besitzen ausnahmslos ***Eingabeparameter***
  - Wert dieser Parameter kann nicht ***geändert*** werden
- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
    - vor dem Aufruf wird der aktuelle Parameter ausgewertet und dem formalen Parameter zugewiesen (***call-by-value***)
  - ***Ausgabeparameter***
    - dienen dazu, Ergebnisse einer Prozedur an den Aufrufer zurückzugeben
    - Wert ist zum Zeitpunkt des Aufrufs undefiniert (***call-by-reference***)
  - ***Ein- / Ausgabeparameter (Transienten)***
    - vereinen Eigenschaften beider Arten
- **Modula-3 nutzt dazu zwei Parameterübergabemechanismen**

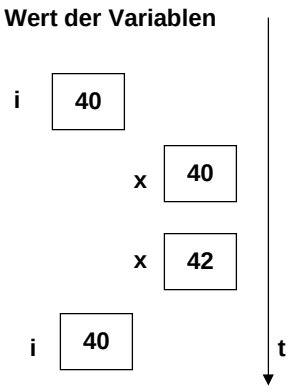
Übergabemechanismen: Call by Value

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
    - realisieren Eingangsparameter
    - Der aktuelle Parameter muß ein ***Ausdruck*** sein (Spezialfall: Variable, d.h. Bezeichner für ein Objekt).
    - Beim Aufruf der Prozedur wird ein dem Typ des formalen Parameters entsprechendes ***lokales Objekt*** angelegt. Ist der aktuelle Parameter eine Variable, so entsteht dabei eine ***Kopie*** des Parameter-Objekts.
    - In jedem Falle wird der Wert des aktuellen Parameters ***berechnet*** und dem formalen Parameter (-Objekt) zugewiesen.
    - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
    - Schlüsselwort ***VALUE*** zeigt einen Wertparameter an
    - steht kein Schlüsselwort, ist es ***per default*** ein Wertparameter

Beispiel: Call by Value

```
PROCEDURE Proc1 (VALUE x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Nichts passiert")
ELSE
  SIO.PutLine ("kein Call by Value")
END;
```



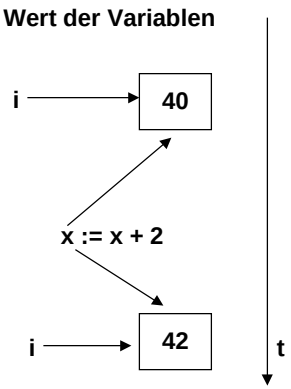
Übergabemechanismen: Call by Reference

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
    - realisieren Ausgangs und Ein- / Ausgangsparameter
    - Der aktuelle Parameter muß ein ***Bezeichner für ein Objekt*** sein.
    - Beim Aufruf wird der formale Parameter durch einen ***Verweis*** auf den aktuellen Parameter ersetzt.
      D.h. der formale Parameter wird als lokaler Bezeichner für das aktuelle Parameterobjekt ***substituiert***.
    - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.
    - Veränderungen des formalen Parameters in der Prozedur haben auf den aktuellen Parameter Auswirkung, d.h. Objekte im Namensraum des Rufers können ***verändert*** werden. Auf diese Weise können von einer Prozedur ***Ergebnisse*** zurückgegeben werden.
    - Schlüsselwort ***VAR*** zeigt einen Variablenparameter an

Beispiel: Call by Reference

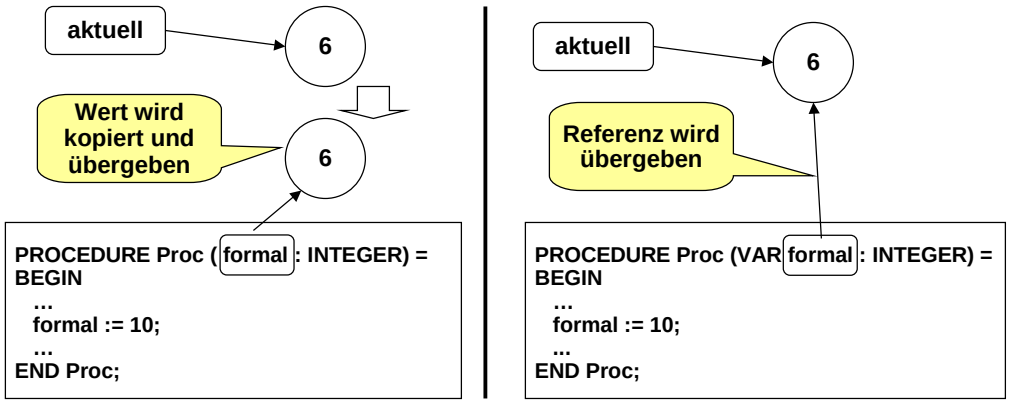
```
PROCEDURE Proc1 (VAR x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Call by Value")
ELSE
  SIO.PutLine ("Call by Reference")
END;
```



Call-by-value <-> Call-by-reference

```
VAR aktuell : INTEGER
…
aktuell := 6;
Proc (aktuell);
```



Beispiel: Wert- Variablenparameter

```
MODULE Parameter EXPORTS Main;
IMPORT IO, SIO;

VAR aktuell: INTEGER;

PROCEDURE Proc1 (VALUE formal : INTEGER) =
BEGIN
  formal := 10;
END Proc1;

PROCEDURE Proc2 (VAR formal : INTEGER) =
BEGIN
  formal := 10;
END Proc2;

BEGIN
  aktuell := 6;        Proc1 (aktuell);
  SIO.PutText("aktuell (Proc1) = "); IO.PutInt(aktuell); SIO.Nl();
  aktuell := 6;        Proc2 (aktuell);
  SIO.PutText("aktuell (Proc2) = "); IO.PutInt(aktuell);
END Parameter.
```

Wertparameter

Variablen-parameter

aktuell (Proc1) = 6
aktuell (Proc2) = 10

Ausgabe des Programms

Beispiel: Variablenparameter

```
MODULE Vertausche2 EXPORTS Main;
IMPORT IO, SIO;

PROCEDURE Vertausche (VAR w1, w2 : INTEGER) =
VAR hilfe : INTEGER := w1;
BEGIN
  w1    := w2;
  w2    := hilfe;
END Vertausche;

VAR x, y : INTEGER;

BEGIN
  x := IO.GetInt();
  y := IO.GetInt();
  Vertausche (x, y);
  SIO.PutText("x = "); IO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); IO.PutInt(y);
END Vertausche2.
```

Vertauscht die Werte der beiden Parameter

Beispiel: Ausgabeparameter

- **Ausgabeparameter**
  - dient dazu, einen Wert an den Aufrufer zurückzugeben

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR  ergebnis : REAL;

PROCEDURE Quadrat ( VALUE x : REAL; VAR wert : REAL )=
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Ausgabeparameter oder Ein- Ausgabeparameter ?

Wert ist beim Aufruf undefiniert

Datenaustausch: Beispiel - 1

In einer Reihe von Meßwerten soll der **laufende Mittelwert** berechnet werden. Benötigte Objekte und Aktionen:

```
Wert, Mittelwert, Summe : REAL;
Anzahl : INTEGER;
(* BerechneMWert
      Addiere neuen Wert und Summe,
      Erhöhe Anzahl um 1,
      Mittelwert := Summe / Anzahl *)
```

Austausch über Parameter:

```
PROCEDURE BerechneMWert (     wert     : REAL;
                          VAR anzahl    : INTEGER;
                          VAR summe, mw : REAL) =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  mw := summe / anzahl;
END BerechneMWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;
BerechneMWert (wert,anzahl,summe,mw);
BerechneMWert (2*wert,anzahl,summe,mw);
```

Datenaustausch: Beispiel - 2

Datenaustausch über Funktionsergebnis:

```
PROCEDURE MWert (    wert  : REAL;
                 VAR anzahl: INTEGER;
                 VAR summe : REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl:= anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;

SIO.PutReal (MWert(wert,anzahl,summe));
...
SIO.PutReal (MWert(2*wert,anzahl,summe));
```


Datenaustausch: Beispiel - 3

Datenaustausch über Funktionsergebnis:

```
VAR summe: REAL; anzahl: INTEGER;

PROCEDURE MWert (wert:REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0;

SIO.PutReal (MWert(4.0);
...
SIO.PutReal (MWert(10.0));
```

Diskussion der Beispiele - 1

- **Beispiel 1:**
  - Die Verwendung von VAR-Parametern in einer Prozedur zur Rückgabe von Ergebnissen ist in der imperativen Programmierung üblich.
  - Sie führt oft zu ***Verständnisproblemen***, da sowohl in der Deklaration als auch in der Verwendung klar sein muß, was das eigentliche Ergebnisobjekt (hier: der Mittelwert) ist.
- **Beispiel 2:**
  - Die Verwendung einer Funktion zur Berechnung genau eines Wertes ist dann sauber, wenn alle anderen Parameter als ***Wertparameter*** verwendet werden.
  - Die Modellierung einer Zustandsveränderung (hier: Summe und Anzahl) außerhalb der Funktion mit Hilfe von VAR-Parametern ist ein ***Seiteneffekt***, der die ***Lokalität*** der Funktion zerstört.

Diskussion der Beispiele - 2

- **Beispiel 3:**
  - Die Verwendung von ***globalen*** Variablen, die in einer Prozedur oder Funktion als Seiteneffekt verändert werden, ist die ***schlechteste*** Lösung.
  - Zustandsveränderungen über globale Variablen, die in der Signatur der Prozedur nicht aufgeführt sind, sind unverständlich und extrem ***fehleranfällig***.
- **Funktionsprozeduren**
  - Eine Funktion kann in imperativen Sprachen nur dann sauber modelliert werden, wenn
    - alle Parameter als ***Wertparameter*** übergeben werden,
    - in der Funktion ***keine globalen Variablen*** verwendet werden.
  - In Funktionen sollte auch keine globalen Variablen ***lesend verwendet*** werden, da dies Funktionen an ihren Verwendungskontext ankoppelt.
  - Merke: Funktionen sollten "***in sich***" verständlich sein.

Regeln für die Parameterverwendung

- **Wertparameter sind Variablenparameter vorzuziehen**
  - Änderung an Parametern in Prozeduren ist häufig eine ***Fehlerquelle***, die schwer zu finden ist.
- **Variablenparameter sollten nur verwendet werden, wenn**
  - Prozedur-Ergebnisse ***übergeben*** werden sollen (Ausgabeparameter)
  - Kopieren des Parameters ***nicht möglich*** ist (bei gewissen Datenstrukturen)
  - Kopieren zu ***ineffizient*** ist (bei sehr großen Datenstrukturen)
- **Funktionen haben nur Wertparameter**
  - liefern ihr Ergebnis durch ihren ***Namen*** zurück
  - geht in Modula-3 nur, wenn
    - genau ein Wert zurückgegeben werden soll
  - Rückgabe durch Namen und Variablenparameter muß ***vermieden*** werden

Zusammenfassung: Prozeduren und Parameter

- **In imperativen Sprachen wird oft die Prozedur in den Vordergrund gestellt.**
- **Die Funktion ist gelegentlich (z.B. Modula-3) nur eingeschränkt verwendbar, kann dafür aber *Seiteneffekte* erzeugen (nicht wünschenswert).**
- **Nur durch eine saubere Definition der *Signaturen* von Routinen kann ein verständlicher und weiterverwendbarer Entwurf erreicht werden, d.h. vor allem, jeder Datenaustausch mit der Umgebung sollte *explizit* sein.**
- **Die Modellierung von Zuständen und ihrer Veränderung ist nur in Verbindung mit einem entsprechenden *Modulkonzept* softwaretechnisch sauber zu lösen.**

  Das lernen wir später!

Def. Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - ***Sichtbarkeitsbereich*** ist Teil des Gültigkeitsbereichs
  - Gültigkeitsbereich/Sichtbarkeitsbereich wird durch den Compiler überwacht
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
  - macht nur Sinn für Objekte, die Speicher belegen
    - Typen belegen keinen Speicher
- **Es ist wichtig, beide Begriffe klar zu unterscheiden!**

Gültigkeitsbereich: lokale Deklarationen

- **Regeln für die Gültigkeit von Bezeichnern;**
  - Alle in einer Prozedur oder einem Modul deklarierten Bezeichner sind in der gesamten Prozedur / im gesamten Modul gültig.
  - Das gilt auch für den textuell vor der Deklaration eines Bezeichers liegenden Bereich
  - Davon ausgenommen sind Prozedur- und Modulkopf

```
PROCEDURE Proc ( in : REAL )=

  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Hier können die Bezeichner X und C verwendet werden.

```
PROCEDURE Proc ( in : REAL )=

  CONST CC : C * C
  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Korrekte Vorwärts-referenz

Gültigkeitsbereich: Erweiterung

- **Durch IMPORT kann der Gültigkeitsbereich von Bezeichnern auf das *importierende Modul* erweitert werden.**

```
INTERFACE SIO;
...
PROCEDURE GetReal …;

PROCEDURE PutReal ...;

PROCEDURE GetText ...;

PROCEDURE PutText ...;

…
END SIO;
```

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR   ergebnis : REAL;

PROCEDURE Quadrat … =
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Qualifizierter Bezeichner muß verwendet werden.

Prozeduren als Namensraum - 1

- **Namensraum**
  - Eine Prozedur bildet gegenüber der (textuellen) Umgebung ihrer Deklaration einen eigenen ***Namensraum***, d.h. sie kann lokale Objekte benennen und verwalten.
  - Die Namen der formalen Parameter sowie die im Deklarationsteil des Prozedurrumpfs deklarierten Bezeichner sind nur ***im Prozedurrumpf gültig***, d.h. bekannt.
  - In einer Prozedur können Bezeichner der Umgebung ***neu lokal*** deklariert werden; diese Bezeichner ***verdecken*** die global deklarierten Bezeichner, die zugehörigen Objekte sind in der Prozedur ***nicht sichtbar***.
- **Lebensdauer**
  - Die Lebensdauer von prozedurlokalen Objekten entspricht dem Zeitraum, in dem der Aufruf der Prozedur abgearbeitet wird. Sie werden zu Beginn der Prozedurausführung angelegt.
  - Werte gehen ***verloren***, wenn die Ausführung der Prozedur beendet ist.

Prozeduren als Namensraum - 2

```
PROCEDURE Proc ()=

  VAR x : INTEGER;

  PROCEDURE Proc1 (x :Integer) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Proc;
```

G2

G3

G1

- **Module und Prozeduren definieren eigene Namensräume**
  - Prozeduren können verschachtelt werden
  - Hierarchie von Gültigkeits-bereichen (Namensräumen)
- **Überlappung von Gültigkeitsbereichen**
  - Liegt innerhalb des Gültigkeitsbereiches G1 von X mit der Bedeutung B1 ein weiterer Gültigkeitsbereich G2 von X mit der Bedeutung B2, so handelt es sich um zwei ***verschiedene*** Objekte
  - innerhalb von G2 gilt nur B2, alle anderen Bedeutungen sind ***unsichtbar***.

Lebensdauer - 1

- **Durch Ausführung einer Prozedur P entsteht eine *Inkarnation*.**
- **Zur Inkarnation gehören *zur Laufzeit*:**
  - ein ***Ausführungspunkt*** (also ein Zeiger auf den gerade auszuführenden oder ausgeführten Befehl)
  - ***Speicherplätze*** für alle Bezeichner von Variablen und Wertparameter
  - ***Bezüge*** auf die konkreten Variablenparameter.
- **Informationen existieren bis zum Ende der Ausführung von P.**
- **Beispiel**
  - `PROCEDURE Test (ch: CHAR; VAR x:INTEGER)=`
    `VAR y: REAL`
- **`Test ('a', z)` führt zu einer Inkarnation mit**
  - Speicherplatz für ch und y
  - unter dem lokalen Bezeichner `x` einen Bezug (einer Referenz) auf z
  - nach Abschluß werden diese Speicherplätze wieder freigegeben

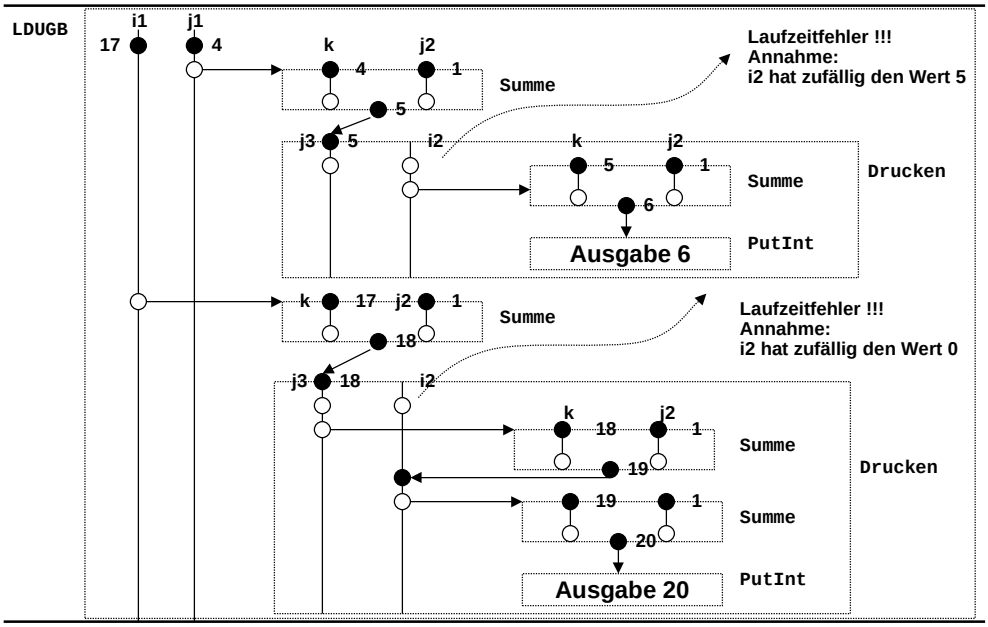
Lebensdauer - 2

- **Bemerkungen:**
  - **Sei M ein Modul,**
    - dann wird ***Speicherplatz*** für die Variablen permanent für die gesamte Laufzeit des Programms reserviert.
    - Eine eigentliche Inkarnation wird nur von dem Rumpf des Moduls gebildet; dieser hat weder Variablen noch Parameter.
  - Zu irgendeinem Zeitpunkt existieren i.a. neben der Inkarnation des ablaufenden Moduls ***Inkarnationen verschiedener*** Prozeduren, bei **rekursiven** Aufrufen auch mehrere der gleichen Prozedur.
  - Nur in der ***jüngsten*** aller existierenden Inkarnationen wandert der Ausführungspunkt weiter; diese wird als erste beendet.
  - Die Lebensdauer einer Variablen ist ***identisch*** mit der Existenz der zugehörigen Prozedur-Inkarnation.
  - Zu Beginn der Lebensdauer ist der Wert einer Variablen ***undefiniert***, darf also nicht verwendet werden.

Beispiel: Lebensdauer - 1

```
MODULE LDUGB EXPORTS Main;
IMPORT SIO;
VAR i  , j  : INTEGER;

PROCEDURE Summe (k  : INTEGER) : INTEGER =
VAR j  : INTEGER;
BEGIN
  IF i  <10 THEN j  := 10 ELSE j  :=1 END;
  RETURN j  + k  ;
END Summe;

PROCEDURE Drucken (j  : INTEGER) =
VAR i  : INTEGER;
BEGIN
  IF i   #   j   THEN i  := Summe(j  ) END;
  SIO.PutInt (Summe(i  )); SIO.Nl();
END Drucken;

BEGIN
  i  := 17; j  := 4;
  Drucken (Summe(j  ));
  Drucken (Summe(i  );
END LDUGB.
```

Beispiel: Lebensdauer - 2



Terminologie

- **"Objekt" bezeichnet alles,**
  - was durch einen Bezeichner eingeführt wird (Modul, Prozedur, Konstante, Typ, Variable, Parameter),
  - keine Objekte sind demnach Operatoren oder Wortsymbole (z.B. VAR, END)
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

Beispiel: lokal, global, global relativ

```
MODULE Gueltigkeit EXPORTS Main;
  IMPORT SIO;
  VAR x : INTEGER;

  PROCEDURE Proc1 (x: INTEGER) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Gueltigkeit.
```

x ist global sichtbar im Modul

x ist lokal in der Prozedur Proc1

x ist global relativ in der Prozedur Proc2

y ist lokal in der Prozedur Proc2

Lokalität

- **Ziel**
  - Programme sollten mit ***möglichst wenig*** Aufwand korrigier- und modifizierbar sein.
- **Strategie**
  - ***hohe Lokalität*** durch enge Gültigkeitsbereiche.
  - Größtmögliche Lokalität ist daher ***vorrangiges Ziel*** einer guten Programmierung!
- **Ein Programm sollte dafür folgende Merkmale aufweisen:**
  - Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind ***überschaubar***.
  - Die Objekte sind so ***lokal*** wie möglich definiert, jeder Bezeichner hat nur eine ***einzige***, bestimmte Bedeutung.
  - Die ***Kommunikation*** zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variable.

Vorteile lokaler Variabler

- **Lokale Variablen haben softwaretechnisch einige Vorteile.**
  - Deklaration und Verwendung stehen in einem ***textlichen*** Zusammenhang. Das erhöht die Lesbarkeit.
  - Die ***unfreiwillige*** Verwendung von globalen Variablen wird vermieden, d.h. globale Variablen müssen nicht vollständig bekannt sein.
  - Der Speicherverbrauch durch Prozeduren wird ***minimiert***, da Speicher für lokale Variablen nur während ihrer Aktivierung vorgehalten werden muß.
  - Lokalität:
    - Variablen sollen ***nur in dem Kontext*** deklariert werden, wo sie bekannt sein müssen. Eine Verteilung erschwert die Änderbarkeit.
  - Kapselung:
    - Außerhalb eines Kontextes (Modul, Prozedur) sollen nur relevante Objekte sichtbar sein. Implementationsdetails werden im Inneren ***verborgen*** und sind nicht zugreifbar.

Was haben wir gelernt!

- **Modell der imperativen Programmierung**
  - Zustand, Zustandsübergang
- **Konzept der Variablen**
  - Wertzuweisung
- **Parameterübergabemechanismen**
  - Wertparameter
  - Variablenparameter
  - Wie geht man damit um
- **Gültigkeit von Bezeichnern**
- **Lebensdauer von Objekten**
- **Konzept der Lokalität**

Glossar

- Modelle der imperativen Programmierung (von-Neumann-Sprachen)
- Programm- und Datenspeicherzustände und -übergänge bei imperativen Programmen
- imperative Programmierung: Aufgaben und Hilfsmittel der Programmiersprache
- Programmiersprachliche Objekte
- Variablenbegriff: typisiert, Initialisierung, LH-Value, RH-Value, Wertzuweisung, Veränderung bei Parameterübergabe
- Deklarationen: verschiedene Arten, Zweck dieser Arten
- Konstantenbegriff von Modula-3
- Prozeduren: Aufgaben, Parameter, Gültigkeit, Lebensdauer, Unterscheidung zu Funktionen
- Prozedurdeklaration: Prozedurkopf/Signatur, Prozedurrumpf, Formalparameterliste, lokale Deklarationen, Verwendung der Formalparameter im Rumpf
- Prozeduraufruf: Aktualparameterliste, Konsistenz, Aufruf und Deklaration, Semantik durch Inkarnation
- Parameter: Arten, Übergabemechanismen, Methodikregeln, Call-by-Value (Aufruf über den Wert), Call-by-Reference (Aufruf über die Adresse)
- rekursive Prozeduren (direkt und indirekt): Verwendung bei Divide & Conquer-Entwicklungsstrategie, statisches Verständnis der Rekursion, Laufzeitgeschehen bei rekursiven Prozeduren
- Datenaustausch über Prozeduren
- Namensraum von Prozeduren, Gültigkeitsbereich, Erweiterung durch Importe, lokale Variable, Vorteile im Sinne der Programmiermethodik
- Lebensdauer von Prozedurinkarnationen, darin enthaltener Variablen

Kontroll-strukturen

- **Ablaufkontrolle**
- **Fallunterscheidungen**
  - IF
  - CASE
- **Wiederholungsanweisungen (Schleifen)**
  - WHILE, FOR
  - REPEAT-UNTIL, LOOP-EXIT

Kontrollfluß

- **Ablaufsteuerung in der von Neumann-Maschine:**
  - Befehle stehen ***hintereinander*** im Speicher, werden vom Steuerwerk in den zentralen Prozessor geholt, dort decodiert und verarbeitet.
  - Durch ***Sprungbefehle*** kann von der Reihenfolge der gespeicherten Befehle abgewichen werden.
- **Abstraktion:**
  - Die Ausführungsreihenfolge der Aktionen eines Algorithmus entspricht zunächst der textuellen Anordnung (***Sequenz***). Davon kann aber abgewichen werden. Dazu gibt es eigene Anweisungen.
  - Ablaufsteuerung:
    - Fallunterscheidung,
    - Wiederholungen.
- **Ablaufsteuerung in imperativen Programmiersprachen:**
    - Anweisungsfolgen,
    - Fallunterscheidungen **`IF`**- und **`CASE`**-Anweisungen,
    - Schleifenformen **`WHILE`**-, **`UNTIL`**-, **`LOOP`**-Anweisungen.

Kontrollstrukturen

- **Berechnungen in imperativen Programmen erfolgen durch die *Auswertung von Ausdrücken* und die *Zuweisung* von Werten zu Variablen.**
- **Zur Erhöhung der Flexibilität von Programmen sind zwei weitere Sprachmechanismen notwendig:**
  - die Steuerung des Kontrollflusses zur ***Auswahl*** zwischen unterschiedlichen Anweisungen, (Fallunterscheidungen)
  - die ***wiederholte*** Ausführung einer Folge von Anweisungen.
- **Sprachmechanismen, die dies leisten,**
  - heißen ***Kontrollstrukturen***.
- **Kontrollstrukturen**
  - zusammen mit den Anweisungsfolgen, die von ihnen kontrolliert werden, stellen den Anweisungsteil eines Programms dar.

Beispiel GOTO (Sprunganweisung)

FORTRAN-Programm zur Bewertung von Meßdaten

```
Beispiel GOTO in einem FORTRAN-Programm
100   format()
200   format(56h anzahl messwerte gut brauchbar schlecht unbrauchbar)
      ischl = 0
      ibr = 0
      igut = 0
      do 1 i = 1,n
      read (5,100) x
      abso = abs(x - s);
      if (abso .le. 0.01) goto 10
      if (abso .le. 0.05) goto 11
      if (abso .le. 0.2)  goto 12
      iunb = iunb + 1
      goto 1
10    igut = igut + 1
      goto 1
11    ibr = ibr + 1
      goto 1
12    ischl = ischl + 1
1     continue
      ...
```

Diskussion: Kontrollstrukturen

- **Von Mitte der 60er bis Mitte der 70er wurde in der Informatik viel über *Kontrollstrukturen* diskutiert.**
- **Ein Ergebnis war:**
  - Obwohl eine einzige ***Anweisungsform*** (bedingter oder unbedingter Sprung) ausreicht, sollte genau diese Anweisung durch eine ***kleine Zahl*** von Kontrollanweisungen ***ersetzt*** werden.
- **Boehm und Jacopini haben 1966 nachgewiesen, daß alle Flußdiagramm-Programme durch zwei Kontrollstrukturen implementierbar sind:**
  - ***Auswahl*** zwischen alternativen Kontrollflüssen,
  - logisch kontrollierte ***Wiederholung***.
- **Kontrollstrukturen sollen *einen Einstieg* und *einen Ausstieg* (single entry, single exit) besitzen.**

Diskussion Goto

- **Obwohl die *unbedingte Verzweigung* (Goto) ausreicht,**
  - alle anderen Kontrollstrukturen nachzubilden, führt ihre uneingeschränkte Verwendung zu unlesbaren und unzuverlässigen Programmen.
- **Hauptgrund:**
  - Durch Goto kann im Ablauf ***jede beliebige Reihenfolge*** von Anweisungen unabhängig von ihrer textlichen Anordnung erreicht werden.
  - In seinem Artikel ("Goto statement considered harmful", CACM, 1968, Vol.11, No.3, pp.147-149) schreibt E.W. Dijkstra: "The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."
- **Dies hat die Goto-Debatte entzündet,**
  - die zwar zur softwaretechnischen Ablehnung des ***uneingeschränkten Goto*** geführt hat,
  - aber nur ***wenige*** Programmiersprachen haben völlig auf dieses Konstrukt verzichtet (Modula-2, Modula-3, Java).

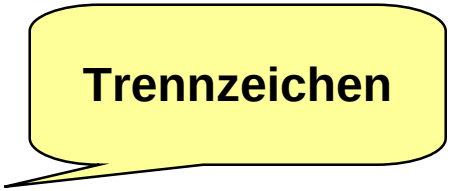
Konzept: Sequenz

Sequenz von Aktionen:

- Eine Aktion wird ***nach der anderen*** abgearbeitet.
- Dazu muß nur klar sein, wie zwei Aktionsbeschreibungen voneinander ***getrennt*** sind.
- Ein Aktion ist auch "tue nichts".

Beispiel:

- Algorithmus Telefonieren:
  - `hebe den Hörer ab;`
  - `wähle die Telefonnummer;`
  - `führe das Gespräch;`
  - `lege den Hörer auf`



Konzept: Bedingte Anweisung (if)

- **Fallunterscheidungen kommen vor als:**
  - Einwegauswahl (one-way selection)
  - Zweiwegauswahl (two-way-selection)
  - allgemeine bedingte Anweisung

```
IF b THEN
  stmts;
ELSE
  stmts;
END;
```

Zweiwegauswahl, zweiseitig bedingte Anweisung

```
IF b THEN
  stmts;
END;
```

Einwegauswahl, einseitig bedingte Anweisung

...

allgemeine bedingte Anweisung

- **Anwendbar**
  - Typ des Ausdrucks, der die Auswahl bestimmt, ist BOOLEAN

Auswahlanweisung (case)

- **Die *Auswahlanweisung* ermöglicht die Auswahl aus einer *beliebigen* Anzahl explizit angegebener Alternativen.**
- **Designentscheidungen sind:**
  - Welche Form und Typ von Ausdruck ***kontrolliert*** die Mehrfachselektion?
  - Welche Form von Anweisung kann ***ausgewählt*** werden?
  - Wie ist der ***Kontrollfluß*** innerhalb der Mehrfachselektion?
  - Wie werden Selektionswerte behandelt, für die es ***keine*** passende Anweisungsalternative gibt?
- **Programmiersprachen realisieren die Auswahl durch die**
  - CASE-Anweisung

CASE-Anweisung - 1

- **Hängen die Fälle einer Fallunterscheidung nur von unterschiedlichen Werten *eines Ausdrucks* ab,**
  - können wir die in modernen imperativen Sprachen vorhandene *CASE-Anweisung* verwenden.

- **Der ELSE-Teil ist im Beispiel leer, um einen *Fehlerfall* zu vermeiden.**
- **Dies ist eine häufige, aber *schlechte* Programmiertechnik.**
- **Besser:**
  - möglichen Fehlerfall explizit behandeln.

```
CASE zweierpotenz OF

  0 => ergebnis := 1;
| 1 => ergebnis := 2;
| 2 => ergebnis := 4;
| 3 => ergebnis := 8;
| 4 => ergebnis := 16;
ELSE
END;
```

CASE-Anweisung - 2

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  ergebnis := 0;

  CASE zweierpotenz OF
     0 => ergebnis := 1;
   | 1 => ergebnis := 2;
   | 2 => ergebnis := 4;
   | 3 => ergebnis := 8;
   | 4 => ergebnis := 16;
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;

  SIO.Nl(); SIO.PutInt(ergebnis);
END CaseDemo.
```

Bemerkung:

Werden nicht alle Werte als Alternativen aufgeführt und fehlt des ELSE-Zweig, dann kann das zu *Laufzeitfehlern* führen!!

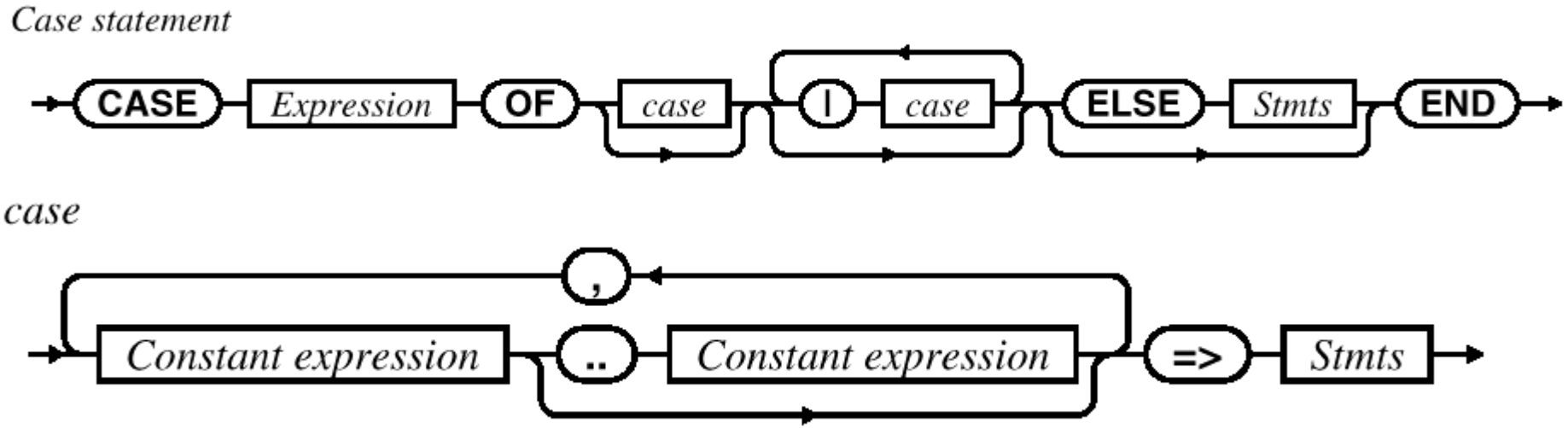
Fehlerbehandlung im ELSE-Zweig ???

CASE-Anweisung - 3

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  IF (zweierpotenz >= 0 AND zweierpotenz <= 4) THEN
    CASE zweierpotenz OF
      0 => ergebnis := 1;
    | 1 => ergebnis := 2;
    | 2 => ergebnis := 4;
    | 3 => ergebnis := 8;
    | 4 => ergebnis := 16;
    END;
    SIO.PutInt(ergebnis);
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;
END CaseDemo.
```

Fehlersituation wird explizit vor Ausführung der CASE-Anweisung behandelt!

Syntax CASE-Anweisung



Auswahllisten: Bequemlichkeit!

- **Zusatzbedingungen für die Case-Anweisung:**
  - Ergebnis von *Expression* und Ergebnis der *Constant expressions* müssen ***denselben*** Typ haben.
  - Zugelassen sind nur ***Ordinaltypen*** (z.B. BOOLEAN oder CHAR)
  - Die Werte dürfen jeweils ***nur einmal*** auftreten.

Konzept: Schleifen

- **Schleifen (Wiederholungsanweisungen, Iterationen) werden benötigt,**
  - um ***iterative Algorithmen*** zu formulieren.
- **Die historisch ersten Sprachkonstrukte zur Wiederholung waren**
  - für die ***Array-Bearbeitung*** gedacht,
  - da anfangs vorrangig numerische Probleme gelöst werden mußten.
- **Designentscheidungen sind:**
  - Wie wird die Wiederholung kontrolliert?
    - durch einen ***logischen Ausdruck***,
    - durch ***Abzählen***
  - Wo steht der Kontrollmechanismus im Programmtext?
    - am ***Anfang***/***Ende***, ***automatisch***/***benutzerdefiniert***

Zählschleife (FOR-Anweisung)

- **Die FOR-Anweisung ist eine spezielle Schleifenform**
  - Anzahl der Wiederholungen ist ***im voraus*** bekannt
  - Wiederholungen werden durch Abzählen kontrolliert
- **Syntax:**

- **Anmerkungen:**
  - Die Steuerung und der Abbruch geschieht mit Hilfe der sog. ***Laufvariablen***, deren Schrittweite und Laufrichtung festgelegt werden kann.
  - In Modula-3: Die Laufvariable muß ***nicht deklariert*** werden.
  - Bei jedem Durchlauf wird die Laufvariable "***automatisch***" erhöht oder erniedrigt.
  - FOR-Schleifen werden oft bei der Bearbeitung von ***Arrays*** verwendet.

Beispiel: FOR-Anweisung

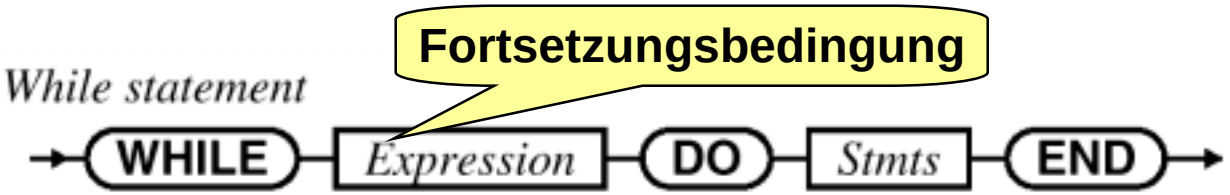
- **Ein Beispiel für die FOR-Schleife:**
  - ```
    (* Berechne Summe I = 1 bis 100 ueber I *)
    r := 0;
    FOR i := 1 TO 100 DO
      r := r + i
    END;
    ```
- **FOR-Schleife mit Schrittweite:**
  - ```
    (* Berechne Summe I = 1 bis 100 ueber I *)
    r := 0;
    FOR i := 100 TO 1 BY -1 DO
      r := r + i
    END;
    ```

Innerhalb der FOR-Schleife muß die Laufvariable wie eine ***Konstante*** behandelt werden, darf also nicht auf der linken Seite einer Wertzuweisung oder als Referenzparameter stehen und damit ***manipulierbar*** sein.

Bedingte Schleife mit vorheriger Prüfung

- **WHILE-Schleife**
  - "***ablehnende Schleife***", da Prüfung vor Schleifendurchlauf gemacht wird
- **Syntax:**

```
WHILE E DO
    S;
END;
```



- **Bemerkung**
  - Ergebnis der "Expression" muß vom ***Typ BOOLEAN*** sein
  - Der Programmierer muß dafür sorgen, daß das Ergebnis von "Expression" ***irgendwann FALSE*** ist; ansonsten Endlosschleife; Programm terminiert nicht
- **Semantik:**

Beispiel: WHILE-Anweisung

- **Aufgabe:**
  - Man berechne den Quotienten und den Rest der ganz-zahligen Division zweier positiver ganzer Zahlen a und b.

```
MODULE GanzzahlDivision EXPORTS Main;
IMPORT SIO;
VAR a, b, c: INTEGER;

BEGIN
  a := SIO.GetInt();
  b := SIO.GetInt();

  c := 0;  (*enthalte den Quotienten *)
  WHILE a >= b DO
    a := a - b;
    c := c + 1;
  END;

  SIO.PutText(" Quo :");  SIO.PutInt(c);
  SIO.PutText(" Rest :"); SIO.PutInt(a);

END GanzzahlDivision.
```

Bedingte Schleife mit nachfolgender Prüfung

REPEAT-UNTIL Anweisung (UNTIL-Schleifen)

- "***annehmende***" Schleife, da eine erste Ausführung stattfindet, ohne daß die Bedingung (in diesem Fall eine ***Abbruch-Bedingung***) geprüft wird

Syntax:

Bemerkung

- REPEAT-UNTIL erfordert besondere Vorsicht
- REPEAT-UNTIL ist immer dann sinnvoll, wenn der Bedingungswert erst durch die ***Anweisungen der Schleife*** entsteht
- z.B. Eingabe mit Fehlerbehandlung

Beispiel: UNTIL

```
MODULE Einleseschleife EXPORTS Main;
IMPORT SIO;
VAR a: INTEGER;

BEGIN

  REPEAT
    SIO.PutText("Geben Sie eine Zahl zwischen 1 und 5 ein! ");
    SIO.Nl();
    a := SIO.GetInt();
  UNTIL (a >=1 AND a <=5);

  SIO.PutText("Ihre Eingabe war korrekt! ");

END Einleseschleife.
```

Vergleich WHILE - UNTIL

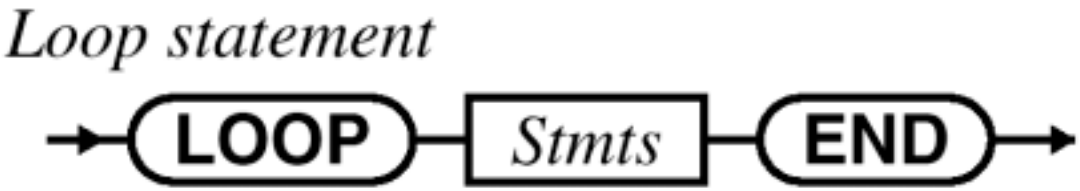
```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t : TEXT := "";
BEGIN
  c := SIO.GetChar();
  WHILE c # Punkt DO
    t := t & Text.FromChar(c);
    c := SIO.GetChar();
  END;
  RETURN t;
END Einlesen;
```

```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t : TEXT := "";
BEGIN
  c := SIO.GetChar();
  IF c # Punkt THEN
    REPEAT
      t := t & Text.FromChar(c);
      c := SIO.GetChar();
    UNTIL c = Punkt;
  END;
  RETURN t;
END Einlesen;
```

```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t, cText : TEXT := "";
BEGIN
  REPEAT
    t := t & cText;
    c := SIO.GetChar();
    cText := Text.FromChar(c);
  UNTIL c = Punkt
  RETURN t;
END Einlesen;
```

Endlosschleife

- **LOOP-Anweisung**
  - die Schleife wird solange durchlaufen, bis eine darin enthaltene ***EXIT-Anweisung*** ausgeführt wird
- **Syntax:**



- **Bemerkungen:**
  - in einer LOOP-Anweisung können ***mehrere*** EXIT-Anweisungen stehen
  - single-entry - single-exit wird dadurch verletzt
  - ist keine EXIT-Anweisung enthalten, so realisiert die LOOP-Anweisung eine ***nichtterminierende Schleife***

Beispiel: LOOP-Anweisung - 1

Algorithmus GGT

❶ Falls n < m ist, so vertausche man n und m

❷ Falls m = 0 ist, dann ist n der ggt(n,m) und man beende den Algorithmus

❸ Falls m≠ 0 ist, so bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

```
MODULE GGT EXPORTS Main;
VAR n, m, hilf, r : INTEGER;
BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  LOOP
    IF n < m THEN  (* 1 *)
      hilf := m;
      m := n;
      n := m;
    END;
    IF m = 0 THEN  (* 2 *)
      EXIT;
    ELSE           (* 3 *)
      r := n MOD m;
      n := m;
      m := r;
    END;
  END (* LOOP *);
  SIO.PutText("GGT = ");SIO.PutInt(n);
END GGT.
```

Beispiel: LOOP-Anweisung - 2

Bemerkungen

- LOOP-Anweisungen terminieren nicht!
- Die Ausführung eines Exits terminiert die Loop-Anweisung. Die Kontrolle geht zur Anweisung ***nach dem END***.
- Bei geschachtelten Loop-Strukturen bewirkt die Exit-Anweisung nur das Verlassen der ***zugehörigen*** Loop-Struktur.

Empfehlung

- Aus Gründen der besseren Lesbarkeit sollte man überall dort WHILE- und REPEAT-Schleifen zu verwenden, wo nur eine Abfrage am ***Anfang*** oder ***Ende*** der Schleife erforderlich ist!

```
MODULE GGT1 EXPORTS Main;
IMPORT SIO;
VAR n, m, r : INTEGER;

BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  r := n MOD m;
  LOOP
    IF r = 0 THEN
      EXIT;
    ELSE
      n := m;
      m := r;
      r := n MOD m;
    END;
  END;
  SIO.PutText ("GGT = ");
  SIO.PutInt(m);
END GGT1.
```

LOOP versus WHILE

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
LOOP
  IF r = 0 THEN
    EXIT;
  ELSE
    n := m;
    m := r;
    r := n MOD m;
  END;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
WHILE r # 0 DO
  n := m;
  m := r;
  r := n MOD m;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

- **Hier bietet sich die WHILE-Anweisung an,**
  - da am Beginn der Schleife die Bedingung geprüft werden soll.

Wahl des Schleifenkonstrukts

REPEAT...UNTIL

- ist mit Vorsicht zu verwenden, denn die Anweisung in der Schleife wird ***mindestens einmal*** durchlaufen.
- Sie ist nur sinnvoll, wenn der Bedingungswert ***erst in der Schleife entsteht***
- kann durch Schleife ohne Prüfung realisiert werden (sollte man aber nicht!)

```
REPEAT
  SIO.PutText("Zahl zwischen 1 und 5! ");
  SIO.Nl();
  a := SIO.GetInt();
UNTIL (a >=1 AND a <=5);

LOOP
 SIO.PutText("Zahl zwischen 1 und 5! ");
 SIO.Nl();
 a := SIO.GetInt();
 IF (a >=1 AND a <=5) THEN EXIT END;
END;
```

Was haben wir gelernt!

Fallunterscheidungen

- einseitig bedingte — IF-THEN
- zweiseitig bedingte — IF-THEN-ELSE
- allgemeine bedingte — mit ELSIF
- Auswahlanweisung — CASE

Wiederholungsanweisungen (Schleifen)

- WHILE
- FOR
- REPEAT-UNTIL
- LOOP-EXIT

Einsatz der Wiederholungsanweisungen

- Die Wahl der geeigneten Ausdrucksmittel hängt vom jeweiligen Konstruktionsproblem ab.
- Implementationsgesichtspunkte sind ggf. zu berücksichtigen.
- Softwaretechnische Überlegungen wie Verständlichkeit und Sicherheit spielen bei der Verwendung eine zentrale Rolle.

Glossar

- **Ablaufkontrolle, Kontrollfluß, zugehörige Kontrollstrukturen für zusammengesetzte Anweisungen, Kontrollstrukturen als Konstruktoren für die Ablaufkontrolle**
- **Fallunterscheidung und Zusammenführung im Kontrollfluß**
- **Kontrollstrukturen für Fallunterscheidungen: bedingte Anweisung (einseitig, zweiseitig, mehrseitig), Auswahlanweisung (Mehrfachselektion), Steuerung des Kontrollflusses durch Boolesche Ausdrücke bzw. Aufzählen der Fälle durch Auswahllisten**
- **GOTO-Kontroverse, wohlstrukturierte Programme (Sequenz, Fallunterscheidung, Iteration), wohlstrukturierte Programme mit Escape (Exit), Umwandlung beliebiger Programme in wohlstrukturierte**
- **Schleifenformen: Zählschleife, WHILE-Schleife, UNTIL-Schleife, Endlosschleife**
- **Anwendungsfälle für verschiedene Schleifenformen**
- **Termination bei verschiedenen Schleifenformen**

Kontroll-strukturen

- **Ablaufkontrolle**
- **Fallunterscheidungen**
  - IF
  - CASE
- **Wiederholungsanweisungen (Schleifen)**
  - WHILE, FOR
  - REPEAT-UNTIL, LOOP-EXIT

Kontrollfluß

- **Ablaufsteuerung in der von Neumann-Maschine:**
  - Befehle stehen ***hintereinander*** im Speicher, werden vom Steuerwerk in den zentralen Prozessor geholt, dort decodiert und verarbeitet.
  - Durch ***Sprungbefehle*** kann von der Reihenfolge der gespeicherten Befehle abgewichen werden.
- **Abstraktion:**
  - Die Ausführungsreihenfolge der Aktionen eines Algorithmus entspricht zunächst der textuellen Anordnung (***Sequenz***). Davon kann aber abgewichen werden. Dazu gibt es eigene Anweisungen.
  - Ablaufsteuerung:
    - Fallunterscheidung,
    - Wiederholungen.
- **Ablaufsteuerung in imperativen Programmiersprachen:**
    - Anweisungsfolgen,
    - Fallunterscheidungen **`IF`**- und **`CASE`**-Anweisungen,
    - Schleifenformen **`WHILE`**-, **`UNTIL`**-, **`LOOP`**-Anweisungen.

Kontrollstrukturen

- **Berechnungen in imperativen Programmen erfolgen durch die *Auswertung von Ausdrücken* und die *Zuweisung* von Werten zu Variablen.**
- **Zur Erhöhung der Flexibilität von Programmen sind zwei weitere Sprachmechanismen notwendig:**
  - die Steuerung des Kontrollflusses zur ***Auswahl*** zwischen unterschiedlichen Anweisungen, (Fallunterscheidungen)
  - die ***wiederholte*** Ausführung einer Folge von Anweisungen.
- **Sprachmechanismen, die dies leisten,**
  - heißen ***Kontrollstrukturen***.
- **Kontrollstrukturen**
  - zusammen mit den Anweisungsfolgen, die von ihnen kontrolliert werden, stellen den Anweisungsteil eines Programms dar.

Beispiel GOTO (Sprunganweisung)

- **FORTRAN-Programm zur Bewertung von Meßdaten**

```
Beispiel GOTO in einem FORTRAN-Programm
100   format()
200   format(56h anzahl messwerte gut brauchbar schlecht unbrauchbar)
      ischl = 0
      ibr = 0
      igut = 0
      do 1 i = 1,n
      read (5,100) x
      abso = abs(x - s);
      if (abso .le. 0.01) goto 10
      if (abso .le. 0.05) goto 11
      if (abso .le. 0.2)  goto 12
      iunb = iunb + 1
      goto 1
10    igut = igut + 1
      goto 1
11    ibr = ibr + 1
      goto 1
12    ischl = ischl + 1
1     continue
      ...
```

Diskussion: Kontrollstrukturen

- **Von Mitte der 60er bis Mitte der 70er wurde in der Informatik viel über *Kontrollstrukturen* diskutiert.**
- **Ein Ergebnis war:**
  - Obwohl eine einzige ***Anweisungsform*** (bedingter oder unbedingter Sprung) ausreicht, sollte genau diese Anweisung durch eine ***kleine Zahl*** von Kontrollanweisungen ***ersetzt*** werden.
- **Boehm und Jacopini haben 1966 nachgewiesen, daß alle Flußdiagramm-Programme durch zwei Kontrollstrukturen implementierbar sind:**
  - ***Auswahl*** zwischen alternativen Kontrollflüssen,
  - logisch kontrollierte ***Wiederholung***.
- **Kontrollstrukturen sollen *einen Einstieg* und *einen Ausstieg* (single entry, single exit) besitzen.**

Diskussion Goto

- **Obwohl die *unbedingte Verzweigung* (Goto) ausreicht,**
  - alle anderen Kontrollstrukturen nachzubilden, führt ihre uneingeschränkte Verwendung zu unlesbaren und unzuverlässigen Programmen.
- **Hauptgrund:**
  - Durch Goto kann im Ablauf ***jede beliebige Reihenfolge*** von Anweisungen unabhängig von ihrer textlichen Anordnung erreicht werden.
  - In seinem Artikel ("Goto statement considered harmful", CACM, 1968, Vol.11, No.3, pp.147-149) schreibt E.W. Dijkstra: "The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."
- **Dies hat die Goto-Debatte entzündet,**
  - die zwar zur softwaretechnischen Ablehnung des ***uneingeschränkten Goto*** geführt hat,
  - aber nur ***wenige*** Programmiersprachen haben völlig auf dieses Konstrukt verzichtet (Modula-2, Modula-3, Java).

Konzept: Sequenz

- **Sequenz von Aktionen:**
  - Eine Aktion wird ***nach der anderen*** abgearbeitet.
  - Dazu muß nur klar sein, wie zwei Aktionsbeschreibungen voneinander ***getrennt*** sind.
  - Ein Aktion ist auch "tue nichts".

- **Beispiel:**
  - Algorithmus Telefonieren:
    - `hebe den Hörer ab;`
    - `wähle die Telefonnummer;`
    - `führe das Gespräch;`
    - `lege den Hörer auf`

Trennzeichen

Konzept: Bedingte Anweisung (if)

- **Fallunterscheidungen kommen vor als:**
  - Einwegauswahl (one-way selection)
  - Zweiwegauswahl (two-way-selection)
  - allgemeine bedingte Anweisung

```
IF b THEN
  stmts;
ELSE
  stmts;
END;
```

Zweiwegauswahl, zweiseitig bedingte Anweisung

```
IF b THEN
  stmts;
END;
```

Einwegauswahl, einseitig bedingte Anweisung

...

allgemeine bedingte Anweisung

- **Anwendbar**
  - Typ des Ausdrucks, der die Auswahl bestimmt, ist BOOLEAN

Auswahlanweisung (case)

- **Die *Auswahlanweisung* ermöglicht die Auswahl aus einer *beliebigen* Anzahl explizit angegebener Alternativen.**
- **Designentscheidungen sind:**
  - Welche Form und Typ von Ausdruck ***kontrolliert*** die Mehrfachselektion?
  - Welche Form von Anweisung kann ***ausgewählt*** werden?
  - Wie ist der ***Kontrollfluß*** innerhalb der Mehrfachselektion?
  - Wie werden Selektionswerte behandelt, für die es ***keine*** passende Anweisungsalternative gibt?
- **Programmiersprachen realisieren die Auswahl durch die**
  - CASE-Anweisung

CASE-Anweisung - 1

- **Hängen die Fälle einer Fallunterscheidung nur von unterschiedlichen Werten *eines Ausdrucks* ab,**
  - können wir die in modernen imperativen Sprachen vorhandene *CASE-Anweisung* verwenden.
- **Der ELSE-Teil ist im Beispiel leer, um einen *Fehlerfall* zu vermeiden.**
- **Dies ist eine häufige, aber *schlechte* Programmiertechnik.**
- **Besser:**
  - möglichen Fehlerfall explizit behandeln.

```
CASE zweierpotenz OF

  0 => ergebnis := 1;
| 1 => ergebnis := 2;
| 2 => ergebnis := 4;
| 3 => ergebnis := 8;
| 4 => ergebnis := 16;
ELSE
END;
```

CASE-Anweisung - 2

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  ergebnis := 0;

  CASE zweierpotenz OF
     0 => ergebnis := 1;
   | 1 => ergebnis := 2;
   | 2 => ergebnis := 4;
   | 3 => ergebnis := 8;
   | 4 => ergebnis := 16;
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;

  SIO.Nl(); SIO.PutInt(ergebnis);
END CaseDemo.
```

Bemerkung:

Werden nicht alle Werte als Alternativen aufgeführt und fehlt des ELSE-Zweig, dann kann das zu *Laufzeitfehlern* führen!!

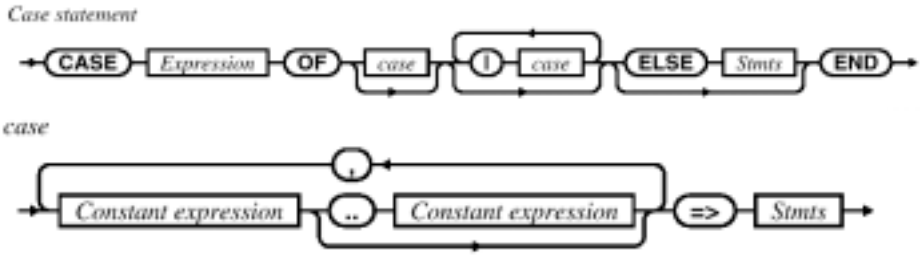
Fehlerbehandlung im ELSE-Zweig ???

CASE-Anweisung - 3

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  IF (zweierpotenz >= 0 AND zweierpotenz <= 4) THEN
    CASE zweierpotenz OF
      0 => ergebnis := 1;
    | 1 => ergebnis := 2;
    | 2 => ergebnis := 4;
    | 3 => ergebnis := 8;
    | 4 => ergebnis := 16;
    END;
    SIO.PutInt(ergebnis);
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;
END CaseDemo.
```

Fehlersituation wird explizit vor Ausführung der CASE-Anweisung behandelt!

Syntax CASE-Anweisung



Auswahlisten: Bequemlichkeit!

- **Zusatzbedingungen für die Case-Anweisung:**
  - Ergebnis von *Expression* und Ergebnis der *Constant expressions* müssen ***denselben*** Typ haben.
  - Zugelassen sind nur ***Ordinaltypen*** (z.B. BOOLEAN oder CHAR)
  - Die Werte dürfen jeweils ***nur einmal*** auftreten.

Konzept: Schleifen

- **Schleifen (Wiederholungsanweisungen, Iterationen) werden benötigt,**
  - um ***iterative Algorithmen*** zu formulieren.
- **Die historisch ersten Sprachkonstrukte zur Wiederholung waren**
  - für die ***Array-Bearbeitung*** gedacht,
  - da anfangs vorrangig numerische Probleme gelöst werden mußten.
- **Designentscheidungen sind:**
  - Wie wird die Wiederholung kontrolliert?
    - durch einen ***logischen Ausdruck***,
    - durch ***Abzählen***
  - Wo steht der Kontrollmechanismus im Programmtext?
    - am ***Anfang***/***Ende***, ***automatisch***/***benutzerdefiniert***

Zählschleife (FOR-Anweisung)

- **Die FOR-Anweisung ist eine spezielle Schleifenform**
  - Anzahl der Wiederholungen ist ***im voraus*** bekannt
  - Wiederholungen werden durch Abzählen kontrolliert
- **Syntax:**

- **Anmerkungen:**
  - Die Steuerung und der Abbruch geschieht mit Hilfe der sog. ***Laufvariablen***, deren Schrittweite und Laufrichtung festgelegt werden kann.
  - In Modula-3: Die Laufvariable muß ***nicht deklariert*** werden.
  - Bei jedem Durchlauf wird die Laufvariable "***automatisch***" erhöht oder erniedrigt.
  - FOR-Schleifen werden oft bei der Bearbeitung von ***Arrays*** verwendet.

Beispiel: FOR-Anweisung

- **Ein Beispiel für die FOR-Schleife:**

```
(* Berechne Summe I = 1 bis 100 ueber I *)
r := 0;
FOR i := 1 TO 100 DO
  r := r + i
END;
```

- **FOR-Schleife mit Schrittweite:**

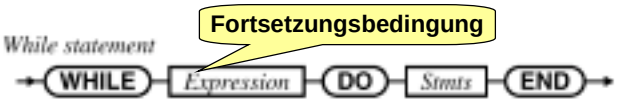
```
(* Berechne Summe I = 1 bis 100 ueber I *)
r := 0;
FOR i := 100 TO 1 BY -1 DO
  r := r + i
END;
```

Innerhalb der FOR-Schleife muß die Laufvariable wie eine ***Konstante*** behandelt werden, darf also nicht auf der linken Seite einer Wertzuweisung oder als Referenzparameter stehen und damit ***manipulierbar*** sein.

Bedingte Schleife mit vorheriger Prüfung

- **WHILE-Schleife**
  - "***ablehnende Schleife***", da Prüfung vor Schleifendurchlauf gemacht wird
- **Syntax:**

```
WHILE E DO
    S;
END;
```



- **Bemerkung**
  - Ergebnis der "Expression" muß vom ***Typ BOOLEAN*** sein
  - Der Programmierer muß dafür sorgen, daß das Ergebnis von "Expression" ***irgendwann FALSE*** ist; ansonsten Endlosschleife; Programm terminiert nicht
- **Semantik:**

Beispiel: WHILE-Anweisung

- **Aufgabe:**
  - Man berechne den Quotienten und den Rest der ganzzahligen Division zweier positiver ganzer Zahlen a und b.

```
MODULE GanzzahlDivision EXPORTS Main;
IMPORT SIO;
VAR a, b, c: INTEGER;

BEGIN
  a := SIO.GetInt();
  b := SIO.GetInt();

  c := 0;  (*enthalte den Quotienten *)
  WHILE a >= b DO
    a := a - b;
    c := c + 1;
  END;

  SIO.PutText(" Quo :");  SIO.PutInt(c);
  SIO.PutText(" Rest :"); SIO.PutInt(a);

END GanzzahlDivision.
```

Wiederhol-bedingung

Bedingte Schleife mit nachfolgender Prüfung

- **REPEAT-UNTIL Anweisung (UNTIL-Schleifen)**
  - "***annehmende***" Schleife, da eine erste Ausführung stattfindet, ohne daß die Bedingung (in diesem Fall eine ***Abbruch-Bedingung***) geprüft wird
- **Syntax:**

- **Bemerkung**
  - REPEAT-UNTIL erfordert besondere Vorsicht
  - REPEAT-UNTIL ist immer dann sinnvoll, wenn der Bedingungswert erst durch die ***Anweisungen der Schleife*** entsteht
  - z.B. Eingabe mit Fehlerbehandlung

Beispiel: UNTIL

```
MODULE Einleseschleife EXPORTS Main;
IMPORT SIO;
VAR a: INTEGER;

BEGIN

  REPEAT
    SIO.PutText("Geben Sie eine Zahl zwischen 1 und 5 ein! ");
    SIO.Nl();
    a := SIO.GetInt();
  UNTIL (a >=1 AND a <=5);

  SIO.PutText("Ihre Eingabe war korrekt! ");

END Einleseschleife.
```

Abbruch-bedingung

Vergleich WHILE - UNTIL

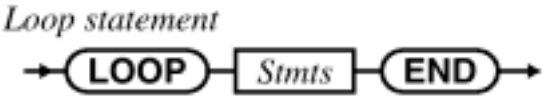
```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t : TEXT := "";
BEGIN
  c := SIO.GetChar();
  WHILE c # Punkt DO
    t := t & Text.FromChar(c);
    c := SIO.GetChar();
  END;
  RETURN t;
END Einlesen;
```

```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t : TEXT := "";
BEGIN
  c := SIO.GetChar();
  IF c # Punkt THEN
    REPEAT
      t := t & Text.FromChar(c);
      c := SIO.GetChar();
    UNTIL c = Punkt;
  END;
  RETURN t;
END Einlesen;
```

```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t, cText : TEXT := "";
BEGIN
  REPEAT
    t := t & cText;
    c := SIO.GetChar();
    cText := Text.FromChar(c);
  UNTIL c = Punkt
  RETURN t;
END Einlesen;
```


Endlosschleife

- **LOOP-Anweisung**
  - die Schleife wird solange durchlaufen, bis eine darin enthaltene ***EXIT-Anweisung*** ausgeführt wird
- **Syntax:**



- **Bemerkungen:**
  - in einer LOOP-Anweisung können ***mehrere*** EXIT-Anweisungen stehen
  - single-entry - single-exit wird dadurch verletzt
  - ist keine EXIT-Anweisung enthalten, so realisiert die LOOP-Anweisung eine ***nichtterminierende Schleife***

Beispiel: LOOP-Anweisung - 1

Algorithmus GGT

❶ Falls n < m ist, so vertausche man n und m

❷ Falls m = 0 ist, dann ist n der ggt(n,m) und man beende den Algorithmus

❸ Falls m≠ 0 ist, so bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

```
MODULE GGT EXPORTS Main;
VAR n, m, hilf, r : INTEGER;
BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  LOOP
    IF n < m THEN  (* 1 *)
      hilf := m;
      m := n;
      n := m;
    END;
    IF m = 0 THEN  (* 2 *)
      EXIT;
    ELSE           (* 3 *)
      r := n MOD m;
      n := m;
      m := r;
    END;
  END (* LOOP *);
  SIO.PutText("GGT = ");SIO.PutInt(n);
END GGT.
```


Beispiel: LOOP-Anweisung - 2

Bemerkungen

- LOOP-Anweisungen terminieren nicht!
- Die Ausführung eines Exits terminiert die Loop-Anweisung. Die Kontrolle geht zur Anweisung ***nach dem END***.
- Bei geschachtelten Loop-Strukturen bewirkt die Exit-Anweisung nur das Verlassen der ***zugehörigen*** Loop-Struktur.

Empfehlung

- Aus Gründen der besseren Lesbarkeit sollte man überall dort WHILE- und REPEAT-Schleifen zu verwenden, wo nur eine Abfrage am ***Anfang*** oder ***Ende*** der Schleife erforderlich ist!

```
MODULE GGT1 EXPORTS Main;
IMPORT SIO;
VAR n, m, r : INTEGER;

BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  r := n MOD m;
  LOOP
    IF r = 0 THEN
      EXIT;
    ELSE
      n := m;
      m := r;
      r := n MOD m;
    END;
  END;
  SIO.PutText ("GGT = ");
  SIO.PutInt(m);
END GGT1.
```

LOOP versus WHILE

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
LOOP
  IF r = 0 THEN
    EXIT;
  ELSE
    n := m;
    m := r;
    r := n MOD m;
  END;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
WHILE r # 0 DO
  n := m;
  m := r;
  r := n MOD m;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

- **Hier bietet sich die WHILE-Anweisung an,**
  - da am Beginn der Schleife die Bedingung geprüft werden soll.

Wahl des Schleifenkonstrukts

- **REPEAT...UNTIL**
  - ist mit Vorsicht zu verwenden, denn die Anweisung in der Schleife wird ***mindestens einmal*** durchlaufen.
  - Sie ist nur sinnvoll, wenn der Bedingungswert ***erst in der Schleife entsteht***
  - kann durch Schleife ohne Prüfung realisiert werden (sollte man aber nicht!)

```
REPEAT
  SIO.PutText("Zahl zwischen 1 und 5! ");
  SIO.Nl();
  a := SIO.GetInt();
UNTIL (a >=1 AND a <=5);

LOOP
 SIO.PutText("Zahl zwischen 1 und 5! ");
 SIO.Nl();
 a := SIO.GetInt();
 IF (a >=1 AND a <=5) THEN EXIT END;
END;
```

Was haben wir gelernt!

- **Fallunterscheidungen**
  - einseitig bedingte — IF-THEN
  - zweiseitig bedingte — IF-THEN-ELSE
  - allgemeine bedingte — mit ELSIF
  - Auswahlanweisung — CASE
- **Wiederholungsanweisungen (Schleifen)**
  - WHILE
  - FOR
  - REPEAT-UNTIL
  - LOOP-EXIT
- **Einsatz der Wiederholungsanweisungen**
  - Die Wahl der geeigneten Ausdrucksmittel hängt vom jeweiligen Konstruktionsproblem ab.
  - Implementationsgesichtspunkte sind ggf. zu berücksichtigen.
  - Softwaretechnische Überlegungen wie Verständlichkeit und Sicherheit spielen bei der Verwendung eine zentrale Rolle.

Glossar

- **Ablaufkontrolle, Kontrollfluß, zugehörige Kontrollstrukturen für zusammengesetzte Anweisungen, Kontrollstrukturen als Konstruktoren für die Ablaufkontrolle**
- **Fallunterscheidung und Zusammenführung im Kontrollfluß**
- **Kontrollstrukturen für Fallunterscheidungen: bedingte Anweisung (einseitig, zweiseitig, mehrseitig), Auswahlanweisung (Mehrfachselektion), Steuerung des Kontrollflusses durch Boolesche Ausdrücke bzw. Aufzählen der Fälle durch Auswahllisten**
- **GOTO-Kontroverse, wohlstrukturierte Programme (Sequenz, Fallunterscheidung, Iteration), wohlstrukturierte Programme mit Escape (Exit), Umwandlung beliebiger Programme in wohlstrukturierte**
- **Schleifenformen: Zählschleife, WHILE-Schleife, UNTIL-Schleife, Endlosschleife**
- **Anwendungsfälle für verschiedene Schleifenformen**
- **Termination bei verschiedenen Schleifenformen**

Zusammenfassung

Programmiersprachen-Konstrukte

- **Ablaufkontrolle**
- **Datenstrukturierung**
- **Entsprechung Kontroll- und Datenstrukturen**
- **Strenge Typisierung**
- **Typäquivalenz**

Kontrollstrukturen zur Bildung zusammengesetzer Anweisungen

- **Formulierung der Berechnung, des Berechnungsablauf**
- **Bildung zusammengesetzter Anweisungen mit Konstruktionselementen (Kontrollstrukturen)**
  - Aneinanderreihung von Anweisungen
  - Fallunterscheidung: bedingte Anweisung **IF**, Auswahlanweisung **CASE**
  - Iteration: Zählschleife **FOR**, bedingte Schleifen **WHILE, UNTIL**
  - Kontrollierter Sprung **EXIT**
  - (unkontrollierter Sprung **GOTO**)

- rekursiver Aufbau:
  - einfache Anweisung —**Kontroll-strukturen**→ zusammengesetzte Anweisungen

  Kontroll-strukturen

- dynamischer Ablauf ist Programmpfad des statischen Berechnungsablaufs

- heute kaum noch Diskussion in Programmiersprachen

Datentypkonstruktoren zur Bildung zusammenges. Typen

- **Formulierung des Datenaufbaus durch Datentypdeklarationen**
- **Zusammengesetzte / skalare Datentypen; vordefinierte / selbstdefinierte; statische / dynamische**
- **Datentypen**

zusammengesetzte:
- Felder
- Verbunde
- Mengen (eingeschränkt)

} statisch

einfache:
- vordefinierte ganzzahlige, reelle
- selbstdef. Aufzählungsdatentypen
- Unterbereichstypen

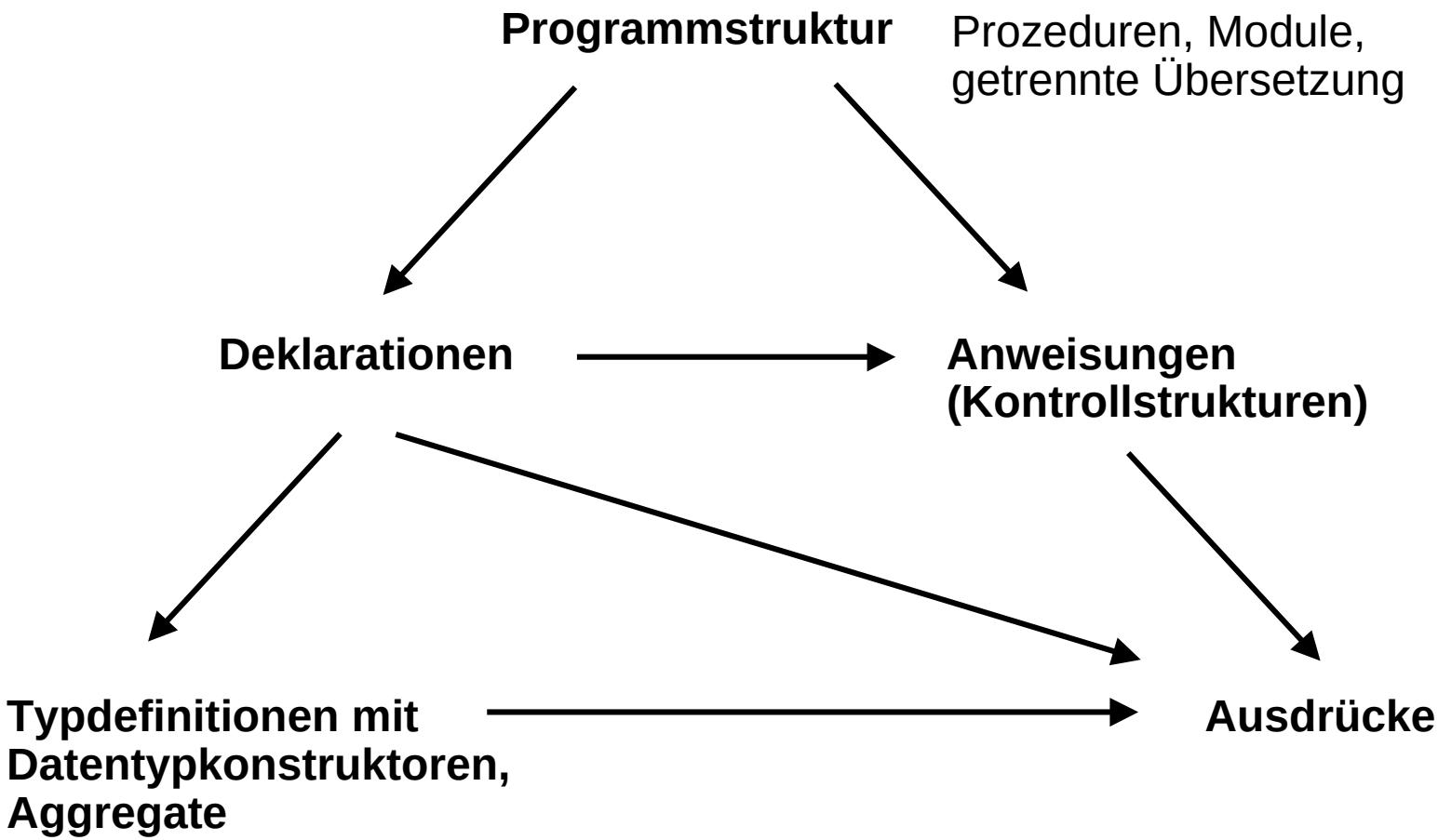
- rekursiver Aufbau:
  - skalare Datentypen —**Datentyp-konstruktor**→ zusammengesetzte Datentypen

- Aufbau eines Datentyps:
  - bei statischen fest, sinnvolle Werte hängen vom Berechnungsablauf ab
  - bei dynamischen Datentypen ergibt sich ein sinnvoller Aufbau durch Berechnung
- abgeklärte Konzepte seit Pascal

Entsprechung der Konstruktionselemente

Konstruktionsmuster	Anweisungsform	Datentyp
Atomares Element	Zuweisung	skalarer Datentyp
Zusammenfassung unterschiedlicher Elemente	Anweisungsfolge	Verbundtyp
Wiederholung/ Zusammenfassung gleicher Elemente, Anzahl bekannt	Zählschleife	Feldtyp
Auswahl	Bedingte Anweisung IF Auswahlanweisung CASE	(varianter Verbund, Vereinigungstyp)
Wiederholung, Anzahl unbekannt	Bedingte Schleifen WHILE, UNTIL	Sequenz (Datei)
Rekursion	rekursive Prozeduren	rekursive Datentypen
Allgemeiner Graph	Sprunganweisung	verkettete Strukturen auf Halde mit Zeigern

Gruppierung der Sprachkonstrukte (s. kontextfreie Syntax)



lexikalische Syntax

Charakterisierung und Vorteile

- **Typ: Struktur, Wertebereich, Operationen, Literale/Aggregate:**

  Abstraktion, Vokabular eines Systems/Anwendungsbereichs (Person, Liste): schwer änderbar

- **strenge Typisierung: Regeln für Sicherheit, Effizienz**
- **statische Typisierung: kontextsensitive Regeln zur Compilezeit**
- **Deklarationsgebot (bis auf vordefinierte Datentypen): Typen und Variablen**

Strukturäquivalenz

- **Zwei Typen sind gleich, wenn ihre Typen nach Expansion der Typdefinitionen gleich sind**
- **für skalare Datentypen, wenn Datentypdefinition gleich ist**
  - Name eines vordefinierten Typs oder gleiche Aufzählungs- bzw. Unterbereichsdefinition
- **für zusammengesetzte Datentypen**
  - Felder: gleicher Komponententyp, gleiche Indextypen in gleicher Reihenfolge, gleiche Anzahl von Elementen für jeden Indextyp
  - Verbunde: gleiche Komponententypen, in gleicher Reihenfolge, gleiche Selektornamen
  - Mengen: gleicher Typ für Universum
- **für Prozedurtypen**
  - gleiche Parameterzahl und Parametertypen, in gleicher Reihenfolge, gleicher Bindungsmodus, gleicher Ergebnistyp

Was haben wir gelernt

- **Konstruktionselemente Ablaufkontrolle, Datentypkonstruktion**
- **Entsprechung Kontrollstrukturen und Datentypen, Konstruktionsmuster für Ablaufkontrolle und Datenstrukturierung**
- **Sprachkonstrukte: Aufbau und gegenseitiger Bezug**
- **Typäquivalenz in Modula-3: Strukturäquivalenz**

Glossar

- **Ablaufkontrolle, Strukturierung des Anweisungsteils, Kontrollstrukturen, zusammengesetzte Anweisungen, Programmpfad**
- **Datenspeicherstrukturierung, Typdefinitionen für zusammengesetzte Datentypen, Datentypkonstruktoren**
- **Gruppen von Konstrukten in imperativen Sprachen**
- **Typisierung: strenge Typisierung, statische Typisierung, Typgleichheit (Typäquivalenz), Namens- oder Srukturäquivalenz**

Zusammenfassung

Programmiersprachen-Konstrukte

- **Ablaufkontrolle**
- **Datenstrukturierung**
- **Entsprechung Kontroll- und Datenstrukturen**
- **Strenge Typisierung**
- **Typäquivalenz**

Ablauf-kontrolle

Kontrollstrukturen zur Bildung zusammengesetzer Anweisungen

- **Formulierung der Berechnung, des Berechnungsablauf**
- **Bildung zusammengesetzter Anweisungen mit Konstruktionselementen (Kontrollstrukturen)**
  - Aneinanderreihung von Anweisungen
  - Fallunterscheidung: bedingte Anweisung **IF**, Auswahlanweisung **CASE**
  - Iteration: Zählschleife **FOR**, bedingte Schleifen **WHILE, UNTIL**
  - Kontrollierter Sprung **EXIT**
  - (unkontrollierter Sprung **GOTO**)

- rekursiver Aufbau:
  - einfache Anweisung **Kontroll-strukturen** zusammengesetzte Anweisungen **Kontroll-strukturen**

- dynamischer Ablauf ist Programmpfad des statischen Berechnungsablaufs

- heute kaum noch Diskussion in Programmiersprachen

Datentypkonstruktoren zur Bildung zusammenges. Typen

- **Formulierung des Datenaufbaus durch Datentypdeklarationen**
- **Zusammengesetzte / skalare Datentypen; vordefinierte / selbstdefinierte; statische / dynamische**
- **Datentypen**

zusammengesetzte:
- Felder
- Verbunde
- Mengen (eingeschränkt)

} statisch

einfache:
- vordefinierte ganzzahlige, reelle
- selbstdef. Aufzählungsdatentypen
- Unterbereichstypen

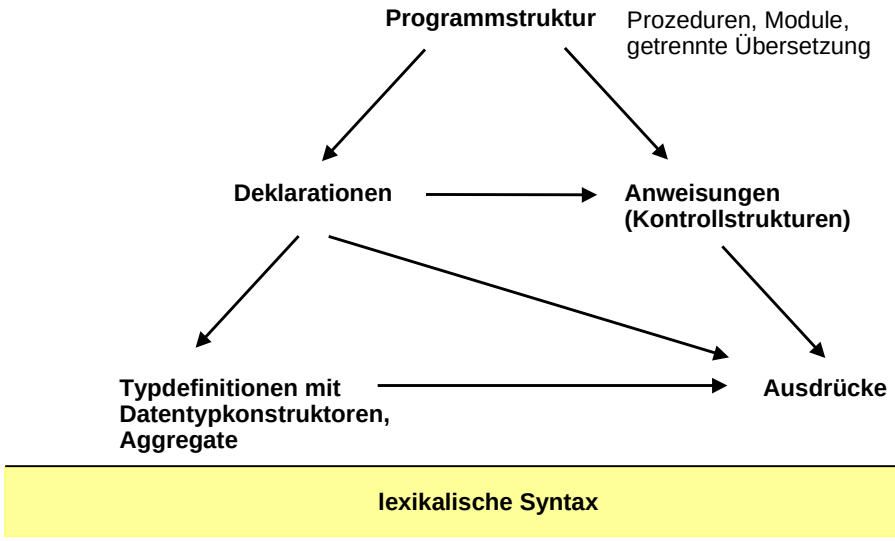
- rekursiver Aufbau:
  - skalare Datentypen ⇒ (Datentyp-konstruktor) zusammengesetzte Datentypen
  - zusammengesetzte Datentypen ⇒ (Datentyp-konstruktor) zusammengesetzte Datentypen
- Aufbau eines Datentyps:
  - bei statischen fest, sinnvolle Werte hängen vom Berechnungsablauf ab
  - bei dynamischen Datentypen ergibt sich ein sinnvoller Aufbau durch Berechnung
- abgeklärte Konzepte seit Pascal

Entsprechung der Konstruktionselemente

Konstruktionsmuster	Anweisungsform	Datentyp
Atomares Element	Zuweisung	skalarer Datentyp
Zusammenfassung unterschiedlicher Elemente	Anweisungsfolge	Verbundtyp
Wiederholung/ Zusammenfassung gleicher Elemente, Anzahl bekannt	Zählschleife	Feldtyp
Auswahl	Bedingte Anweisung IF Auswahlanweisung CASE	(varianter Verbund, Vereinigungstyp)
Wiederholung, Anzahl unbekannt	Bedingte Schleifen WHILE, UNTIL	Sequenz (Datei)
Rekursion	rekursive Prozeduren	rekursive Datentypen
Allgemeiner Graph	Sprunganweisung	verkettete Strukturen auf Halde mit Zeigern

Gruppierung der Konstruktions-elemente

Gruppierung der Sprachkonstrukte (s. kontextfreie Syntax)



Typisierung

Charakterisierung und Vorteile

- **Typ: Struktur, Wertebereich, Operationen, Literale/Aggregate:**

  Abstraktion, Vokabular eines Systems/Anwendungsbereichs (Person, Liste): schwer änderbar
- **strenge Typisierung: Regeln für Sicherheit, Effizienz**
- **statische Typisierung: kontextsensitive Regeln zur Compilezeit**
- **Deklarationsgebot (bis auf vordefinierte Datentypen): Typen und Variablen**

Typäquivalenz

Strukturäquivalenz

- **Zwei Typen sind gleich, wenn ihre Typen nach Expansion der Typdefinitionen gleich sind**
- **für skalare Datentypen, wenn Datentypdefinition gleich ist**
  - Name eines vordefinierten Typs oder gleiche Aufzählungs- bzw. Unterbereichsdefinition
- **für zusammengesetzte Datentypen**
  - Felder: gleicher Komponententyp, gleiche Indextypen in gleicher Reihenfolge, gleiche Anzahl von Elementen für jeden Indextyp
  - Verbunde: gleiche Komponententypen, in gleicher Reihenfolge, gleiche Selektornamen
  - Mengen: gleicher Typ für Universum
- **für Prozedurtypen**
  - gleiche Parameterzahl und Parametertypen, in gleicher Reihenfolge, gleicher Bindungsmodus, gleicher Ergebnistyp

Was haben wir gelernt

- **Konstruktionselemente Ablaufkontrolle, Datentypkonstruktion**
- **Entsprechung Kontrollstrukturen und Datentypen, Konstruktionsmuster für Ablaufkontrolle und Datenstrukturierung**
- **Sprachkonstrukte: Aufbau und gegenseitiger Bezug**
- **Typäquivalenz in Modula-3: Strukturäquivalenz**

Glossar

- **Ablaufkontrolle, Strukturierung des Anweisungsteils, Kontrollstrukturen, zusammengesetzte Anweisungen, Programmpfad**
- **Datenspeicherstrukturierung, Typdefinitionen für zusammengesetzte Datentypen, Datentypkonstruktoren**
- **Gruppen von Konstrukten in imperativen Sprachen**
- **Typisierung: strenge Typisierung, statische Typisierung, Typgleichheit (Typäquivalenz), Namens- oder Srukturäquivalenz**

Teil III

Methodik des Programmierens

- Allgemeines
- Beispiel 1: Entwickeln, Verbessern, Effizienzbetrachtung, Dokumentieren, Test
- Beispiel 2: Partielle Korrektheit, Termination
- Beispiel 3: Handhabung vieler Fälle und Entscheidungstabellen

Allgemeines

Lösungsentwicklung

- **Bisher**
  - Problemformulierung (als Übungsaufgabe)
  - direkt Programm erstellt
    - **Hilfsmittel**: schrittweise Verfeinerung
      Wahl geeigneter Bezeichner
      Verwendung von Kommentaren
    - dabei möglichst gute Gestaltung der Bedieneroberfäche
- **Später (Softwaretechnik)**
  - Problemformulierung, Festlegung der Bedieneroberfläche etc. ist **wichtiger Teil** der Softwareerstellung
  - Programmerstellung, Programmveränderung geht **in „Phasen“**
  - **viele andere Dokumente** ausser Quelltext nötig
  - Programm aus **Bausteinen** zusammengesetzt
  - bei Programmerstellung wirken **viele Personen** mit

Welche Lösung?

- **Problem → zugehöriges Programm**

  es gibt (unendlich) viele Lösungen

- **Lösung** — **schwach**, **stark (funktional)** } **äquivalent**

- **Welche Lösung?**
- **Welche Eigenschaft soll die Lösung haben?**
  - Suchen insbesondere ***effiziente(ste) Lösung***

- **Programme sind im allgemeinen falsch**
  - (große Programmsysteme sind immer falsch)
  - Welche Maßnahmen sind notwendig, um Korrektheit zu „erzielen“?

Übersicht Teil III

- **Im folgenden einige Beispiele zu systematischem PIK**
- **Systematisch in Bezug auf**
  - Programmentwicklung
  - Korrektheitsüberlegungen
  - Testdatenableitung
  - Effizienzanalyse
  - Verwendung von graphischen Notationen
  - Verwendung von Notationen zur Problem- oder Algorithmusfestlegung ***vor*** der Programmierung

Beispiel 1

- Entwickeln
- Verbessern
- Effizienzbetrachtung
- Dokumentieren
- Test

Präzisierung der Aufgaben

- **Aufgabenformulierung:**

  In einer Datei befindet sich eine unbekannte Anzahl von Zahlen. Diese soll gelesen und nach ihrer Größe ausgedruckt werden.

- **Unklarheiten**
  - Format: Wie ausdrucken?
  - Mehrfachvorkommnisse: erlaubt, mehrfach aufgeführt?
  - Sortierung: aufsteigend, absteigend?
  - Sortierung in einem Feld: max. Anzahl unbekannt?
    (internes Sortieren/externers Sortieren)
  - Welcher Typ der einzelnen Elemente: INTEGER, REAL?
  - Feste Randbedingungen: Zahlen < größte in der Basismaschine darstellbare Zahl

- **Neue Formulierung:**

  In einer Datei befinden sich max. 100 nicht notwendigerweise verschiedene INTEGER-Zahlen. Diese Zahlen sollen gelesen und der Größe nach aufsteigend sortiert werden und entsprechend ihrer Vorkommenshäufigkeit einzelnen ausgegeben werden und zwar je 10 pro Zeile.

Programmerstellung durch schrittweise Verfeinerung

- **Deklarationen**
- **Pseudocode**

```
MODULE Main;
    FROM ...
    CONST Max = 100;
TYPE Index = [1..MAX];
VAR Anzahl: CARDINAL;
VAR Behaelter; ARRAY Index OF INTEGER;

BEGIN
    ZahlenEinlesen;
    ZahlenSortieren;
    ZahlenDrucken
END.
```

Verfeinerung

```
(* Zahlen einlesen *)
      Anzahl := 0;
      x := GetInt();
      WHILE NOT End() DO
            Anzahl := Anzahl + 1;
            Behaelter[Anzahl] := x;
            x := GetInt();
      END;
```

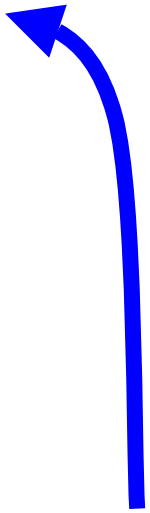
Ergänzung der Deklarationen:

```
VAR   x: INTEGER;
      nr: Index;
```

```
(* Zahlen drucken *)
      PutText("Die Zahlenfolge in aufsteigender Reihenfolge: "); Nl(),
      FOR nr := 1 TO Anzahl DO
            PutInt (Behaelter[nr], 10);

            IF nr MOD 10 = 0 THEN
                  Nl();
            END;
      END;
```

Sortierverfahren

- **Hauptteil Sortieren**
  - einfache Verfahren: n2
  - gute Verfahren: n log n
  - Sortierstrategien
    - Einfügen
    - Auswählen
    - Vertauschen ← Bubblesort
    - Verschmelzen



→ Vorlesung Datenstrukturen und Algorithmen

Bubblesort

```
(* Sortieren *)
      FOR k := 1 TO Anzahl DO
            FOR i := 1 TO Anzahl-1 DO
                  IF Behaelter[ i ] > Behaelter [ i+1 ] THEN

                        (* Vertausche *)
                        hilf := Behaelter [ i ];
                        Behaelter [ i ] := Behaelter [ i+1 ];
                        Behaelter [ i+1 ] := hilf;

                  END;
            END;
      END;
```

Ergänzung der Deklarationen:

```
VAR   i: Index;
      hilf: INTEGER;
```

1. Schritt

i	a					
Undef.	16	33	94	82	6	58
1	16	33	94	82	6	58
2	16	33	94	82	6	58
3	16	33	82	94	6	58
4	16	33	82	6	94	58
5	16	33	82	6	58	94

1. *Durchlauf größte Zahl am richtigen Platz*
2. *Durchlauf zweitgrößte Zahl am richtigen Platz*

...

- **Brauchen nur maximal Anzahl-1 Durchläufe, letzte Zahl ist dann automatisch am richtigen Platz**

  => letzter Durchlauf ist überflüssig

- **Brauchen beim k-ten Durchlauf nur bis Anzahl-k zu laufen**

- **Abbruchkriterium einführen:**

  Nicht mehr durchlaufen, wenn im letzten Durchgang nicht vertauscht worden ist

Verbessern Programm nach 1. Schritt: Bubblesort 2

```
(* Sortieren *)
      FOR k  := 1 TO Anzahl-1 DO
            getauscht := FALSE;
            FOR i := 1 TO Anzahl-k DO
                  IF Behaelter [ i ] > Behaelter [ i+1 ] THEN
                        (* Vertausche *)
                        hilf := Behaelter [ i ];
                        Behaelter [ i ] := Behaelter [ i+1 ];
                        Behaelter [ i+1] := hilf;
                        getauscht := TRUE;
                  END;
            END;
            If NOT getauscht THEN Exit;
      END
```

Ergänzung Deklaration:
`VAR getauscht: BOOLEAN;`

2. Schritt

k	i	a					
		16	33	94	82	6	58
1	5	16	33	82	6	58	94
2	1	16	33	82	6	58	94
2	2	16	33	82	6	58	94
2	3	16	33	6	82	58	94
2	4	16	33	6	58	82	94
3	1	16	33	6	58	82	94
	2	16	6	33	58	82	94
	3	16	6	33	58	82	94

Überflüssig bis Anzahl-k zu laufen, wenn im letzten Durchlauf nicht bis dorthin vertauscht wurde.

Verbessern Programm nach 2. Schritt: Bubblesort 3

Rechtes Ende nicht nur um 1 verkleinern pro Durchlauf

```
...
VAR rechtesEnde, letztesI: Index;
...
letztesI := Anzahl;
REPEAT
      rechtesEnde := letztesI; letztesI := 0;
      For i := 1 TO rechtesEnde -1 DO
            IF Behaelter[i] > Behaelter[i+1] THEN
              (* Vertausche *)
                  hilf := Behaelter[i];
                  Behaelter[i] := Behaelter[i+1];
                  Behaelter[i+1] := hilf;
                  letztesI := i;
            END
      END
UNTIL letztesI := 0
```

Effizienzbetrachtung für Bubblesort 3

best-case-Analyse: geordnet
1 Schleifendurchlauf
n-1 Vergleiche, 0 Vertauschungen
hier Bubblesort gut !

worst-case-Analyse: Umgekehrt geordnet
n-1 Durchläufe
k-ter Durchlauf: innere Schleife: (n-k)-mal vertauscht

$$\sum_{k=1}^{n-1} (n-k) = \frac{(n-1)(n-2)}{2} = \frac{n^2}{2} - \frac{3n}{2} - 1$$

Anzahl der Vergleiche/
Vertauschungen

O-Notation

=> Bubblesort $O(n^2)$

d.h. $\exists$ Konstante a, b, c mit

f^{wc}_{Zeit} (Bubblesort, FeldDerLaenge) $\leq an^2 + bn + c$

Konstanten nicht ausgerchnet : Vorsicht !

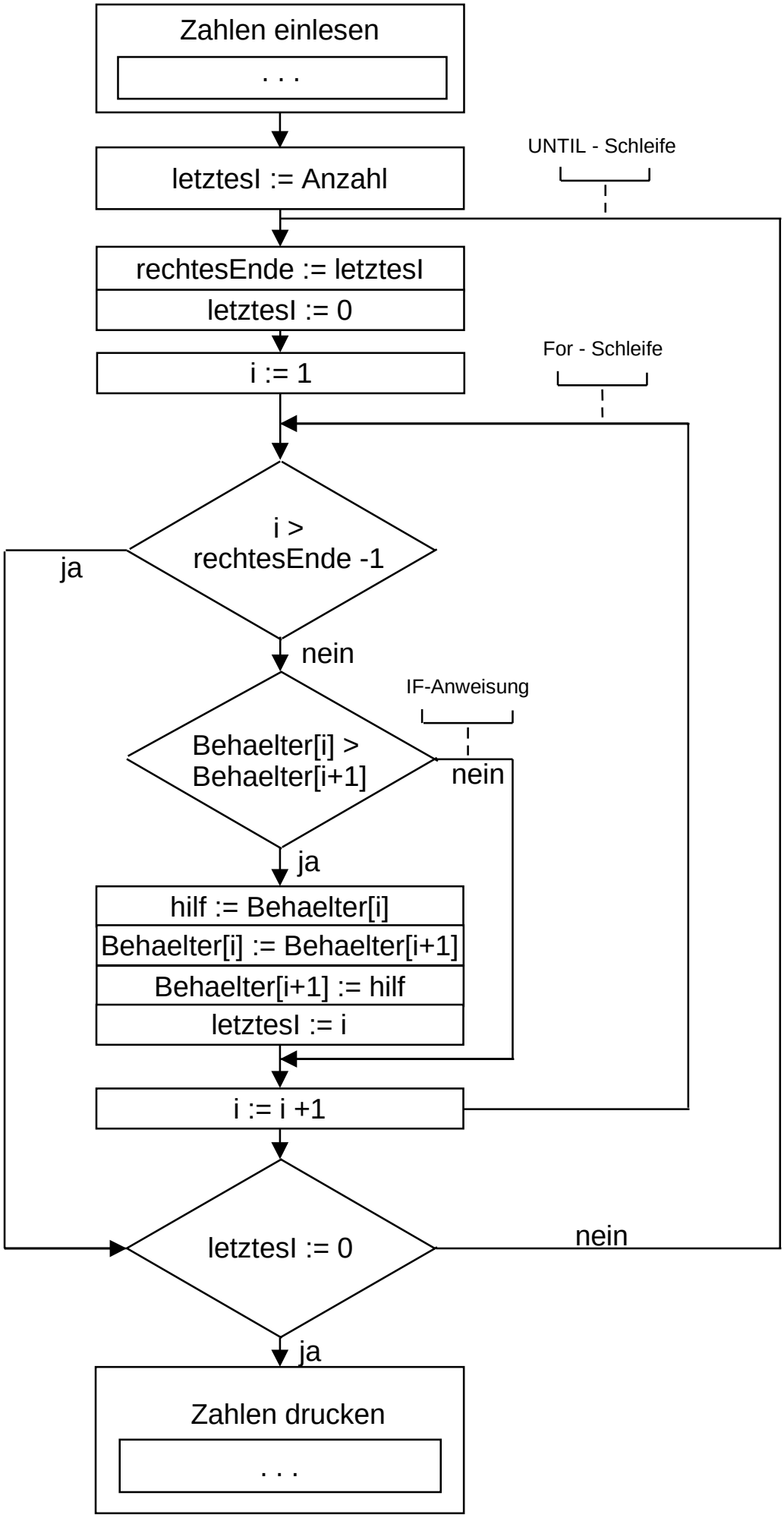
Gibt bessere Verfahren O(n log n)
=> Datenstrukturvorlesung, auch für O-Notation

Dokumentation: Allgemein

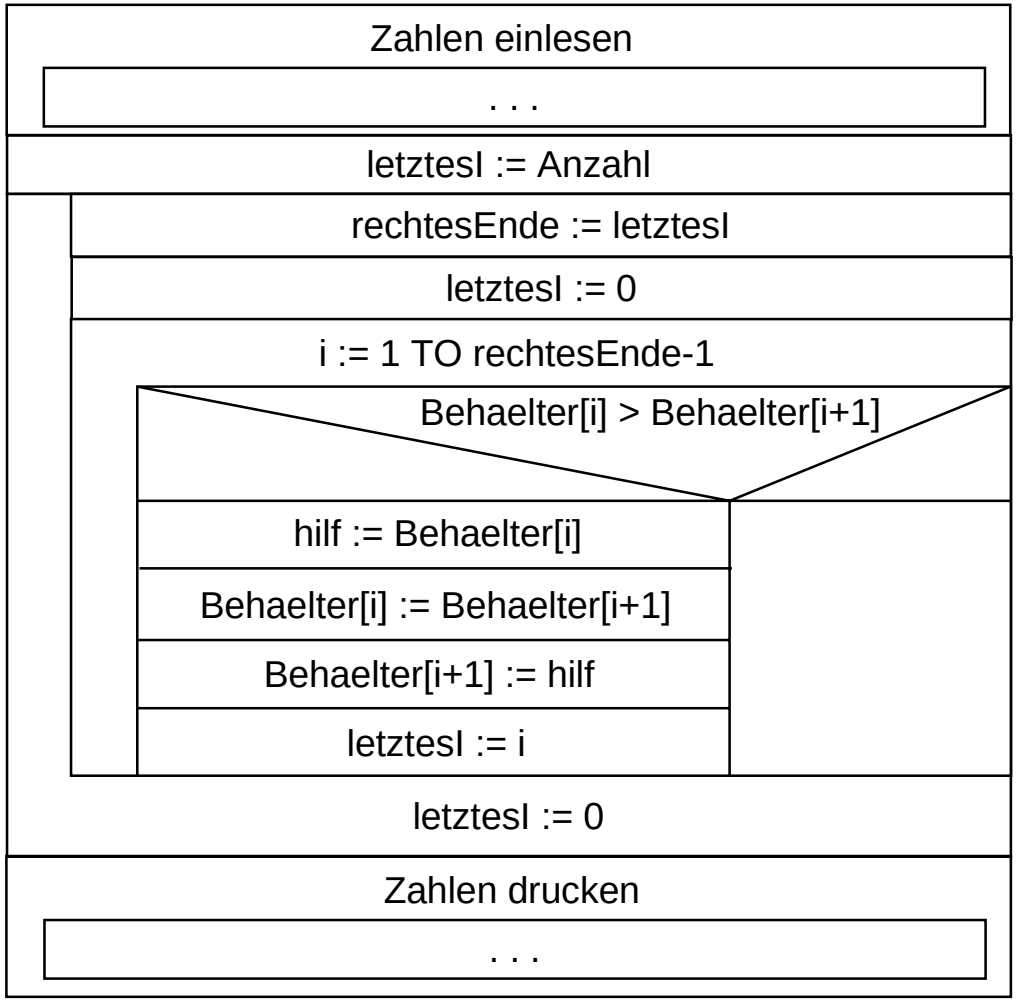
- **Bisher im Programm durch Kommentare,**
  Programm selbst durch Lesbarkeit
- **„parallele“ Notation :** **Flußdiagramme**
  Struktogramme s.u.
- **große Projekte :** **Anforderungsspezifikation**
  Entwurfsspezifikation
  Rationales
  etc.

Flußdiagramm zu Bubblesort 3

Dokumentieren



Struktogramm zu Bubblesort 3



Black-Box-Test

z.B. **Normalfall - Extremfall - Betrachtung**

Normalfall

16 31 94 82 6 58
...

Extremfall

1	2	3	4	5	6
6	5	4	3	2	1
5	5	5	5	5	5
6					

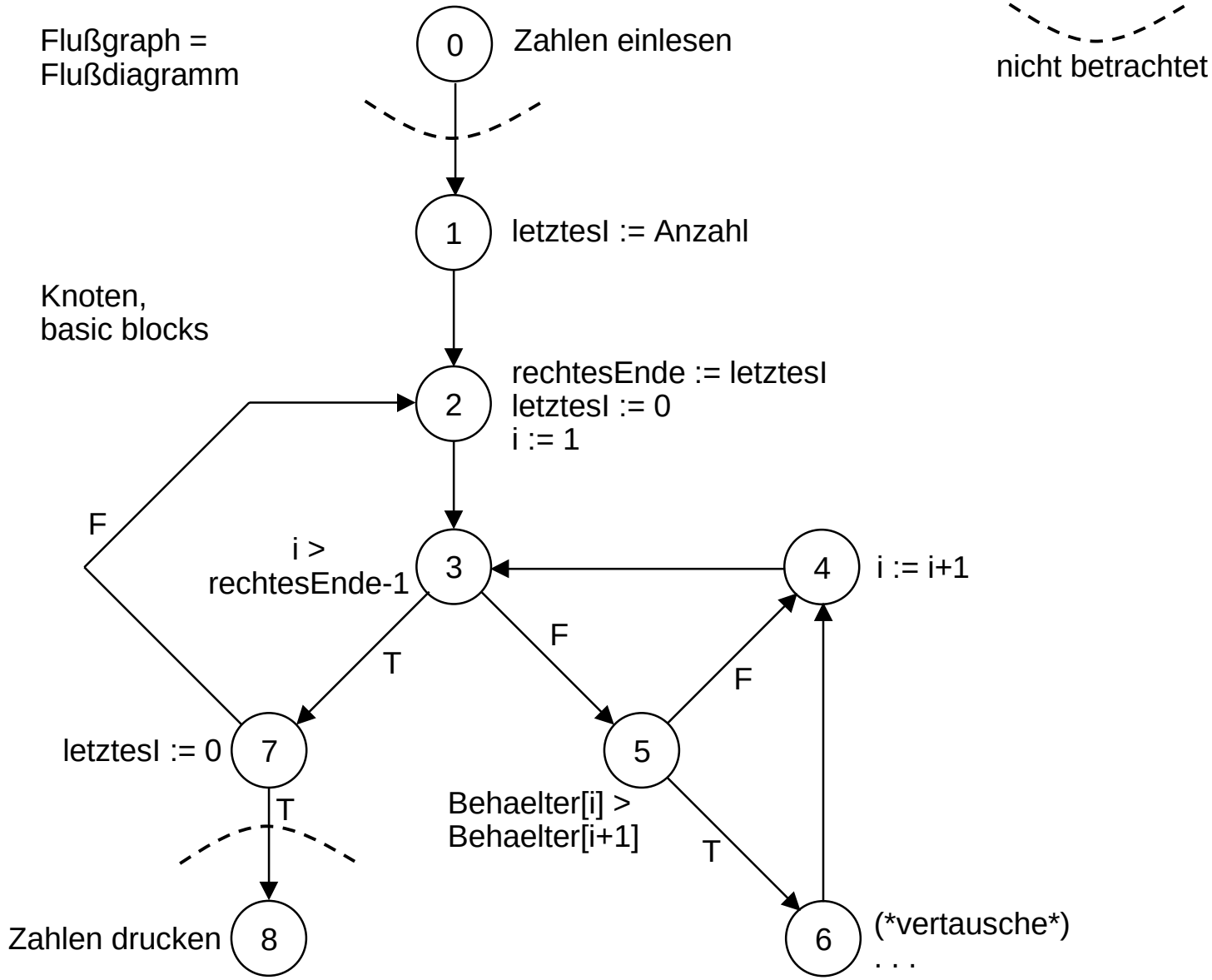
leere Eingabe

- **Verbesserte Methoden**
  - Äquivalenzklassen, Ursache-Wirkungs-Graph etc. -> spätere Vorlesungen
- **Test ist der Qualitätssicherung -> spätere Vorlesungen**

Testen und Testdatenermittlung: White-Box-Test

Basic Block =
seq. Stück

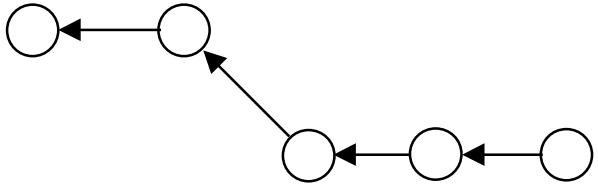
Flußgraph =
Flußdiagramm/
seq. Stücke



White-Box-Test: mögliche Graphmuster für Durchlauf

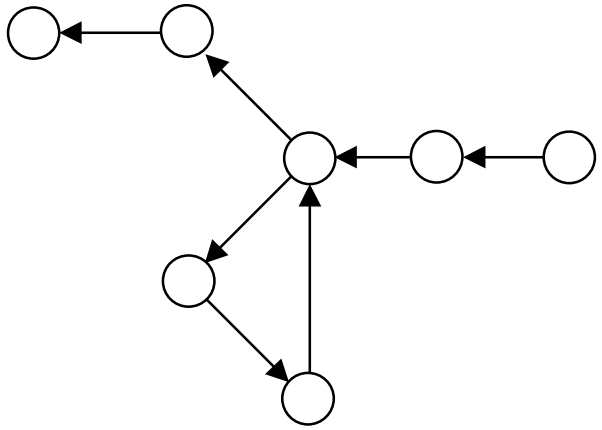
(1)

eine Zahl,
leere Liste



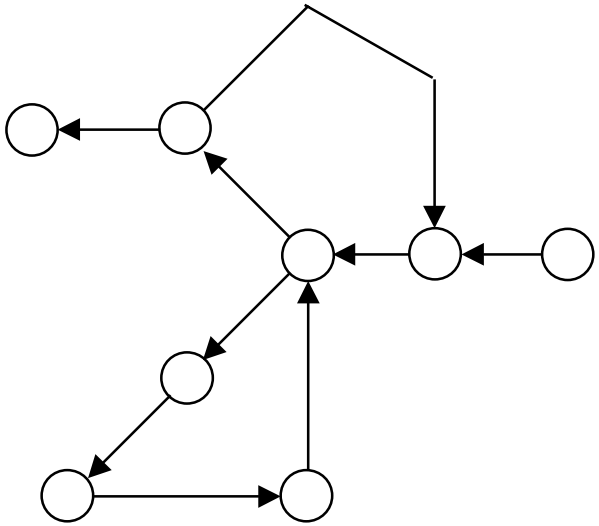
(2)

vollständig
sortiert



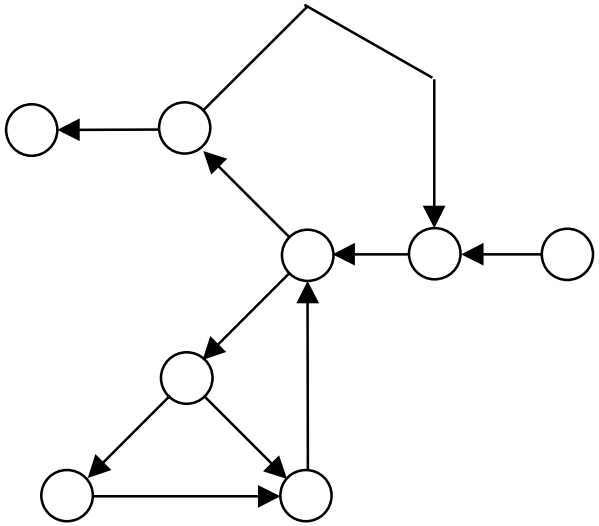
(3)

umgekehrt
sortiert

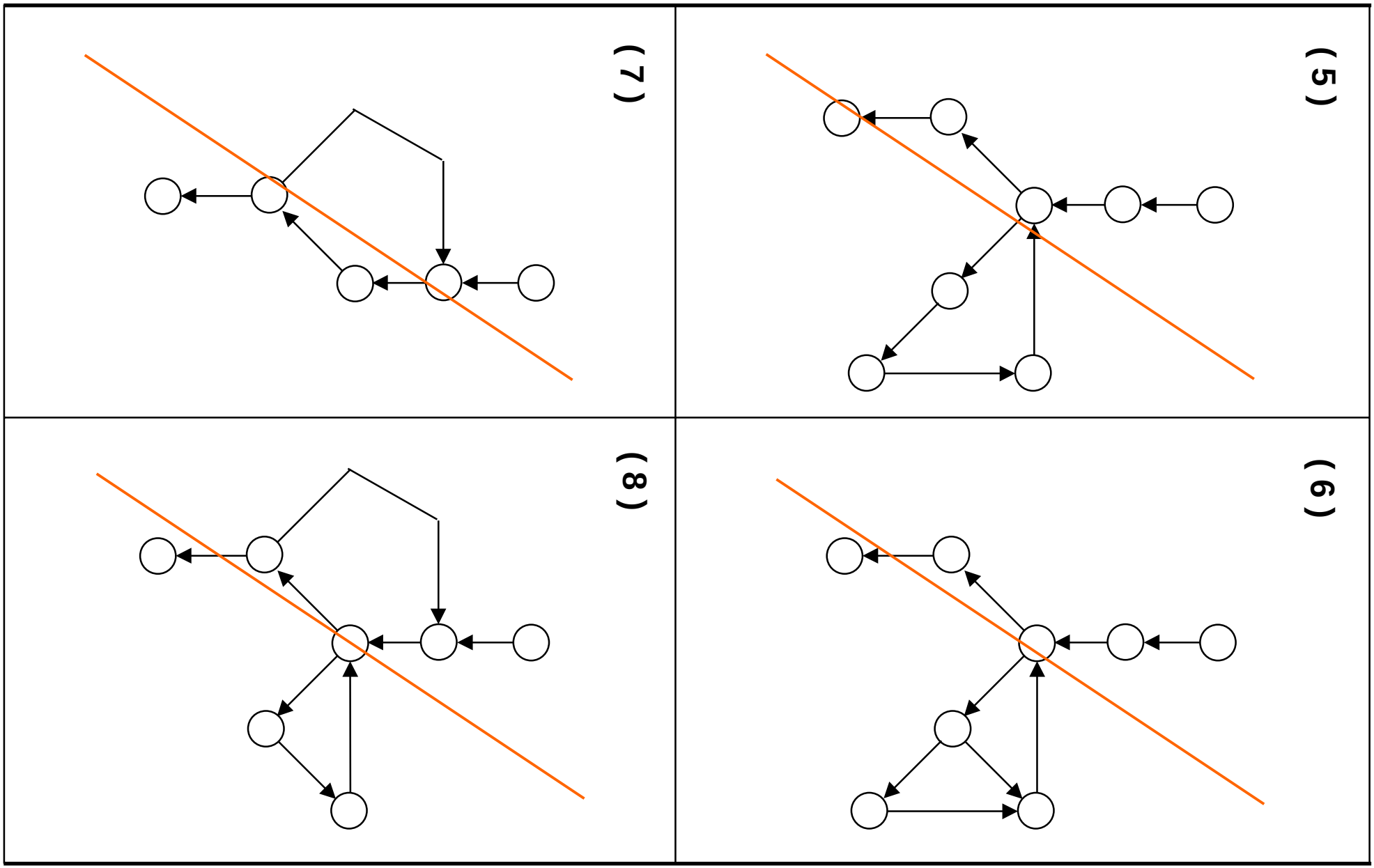


(4)

Normalfall



White-Box-Test: Nicht mögliche Graphmuster



White-Box-Test: Allgemeines

- **Testen und Durchführbarkeit:**
  - Ermittlung aller dyn. Programmpfade nicht möglich!
  - Ermittlung aller Programmpfade (bis auf vielfache von Schleifendurchläufen, wie oben) nur bei kleinen Programmen möglich.
  - Kantenüberdeckung und Bedingungsüberdeckung üblich!
    ⇒ Softwaretechnik, Qualitätssicherungs-Vorlesung

- **Testen und Korrektheit:**

  „Durch Testen kann nur die Anwesenheit aber nicht die Abwesenheit von Fehlern gezeigt werden“ (Dijkstra)

Was haben wir gelernt

- Problemstellung erarbeiten, präzisieren
- Ein Sortierverfahren nach Vertauschstrategie,
- Vermeiden unnötiger Durchläufe, Teildurchläufe,
- best- und worst-case-Analyse für ein Beispiel
- „Dokumentation“ durch Flußdiagramme und Struktogramme
- Grenzen des Testens
- Black-Box-Test und White-Box-Test

Glossar

- Problemformulierung durch Anforderungsspezifikation (Pflichtenheft)
- starke/schwache Äquivalenz von Programmen
- Wiederholung: schrittweise Verfeinerung, Wahl suggestiver Bezeichner, Kommentierung, Bedienungsoberflächengestaltung
- Sortierprinzipien: Einfügen, Auswählen, Vertauschen, Verschmelzung, Bubblesort: Sortierverfahren durch Vertauschen
- Programmverbesserung: Schritte bei Bubblesort
- Effizienzanalyse: best case, worst case
- Dokumentation durch Flußdiagramme, Nassi-Shneiderman-Diagramme
- Test als experimentelle, stichprobenartige Qualitätssicherung: Black-Box-Test (BBT), White-Box-Test (WBT), Normalfall-/Extermfallbetrachtung für BBT, Flußgraphenmethode für WBT
- Testen und Programmkorrektheit

Teil III

Methodik des Programmierens

- Allgemeines
- Beispiel 1: Entwickeln, Verbessern, Effizienzbetrachtung, Dokumentieren, Test
- Beispiel 2: Partielle Korrektheit, Termination
- Beispiel 3: Handhabung vieler Fälle und Entscheidungstabellen

Allgemeines

Lösungsentwicklung

- **Bisher**
  - Problemformulierung (als Übungsaufgabe)
  - direkt Programm erstellt
    - **Hilfsmittel**: schrittweise Verfeinerung
      Wahl geeigneter Bezeichner
      Verwendung von Kommentaren
    - dabei möglichst gute Gestaltung der Bedieneroberfäche
- **Später (Softwaretechnik)**
  - Problemformulierung, Festlegung der Bedieneroberfläche etc. ist **wichtiger Teil** der Softwareerstellung
  - Programmerstellung, Programmveränderung geht **in „Phasen“**
  - **viele andere Dokumente** ausser Quelltext nötig
  - Programm aus **Bausteinen** zusammengesetzt
  - bei Programmerstellung wirken **viele Personen** mit

Welche Lösung?

- **Problem → zugehöriges Programm**
  es gibt (unendlich) viele Lösungen
- **Lösung** **schwach**
  stark (funktional) } **äquivalent**
- **Welche Lösung?**
- **Welche Eigenschaft soll die Lösung haben?**
  - Suchen insbesondere ***effiziente(ste) Lösung***
- **Programme sind im allgemeinen falsch**
  - (große Programmsysteme sind immer falsch)
  - Welche Maßnahmen sind notwendig, um Korrektheit zu „erzielen“?

Übersicht Teil III

- **Im folgenden einige Beispiele zu systematischem PIK**
- **Systematisch in Bezug auf**
  - Programmentwicklung
  - Korrektheitsüberlegungen
  - Testdatenableitung
  - Effizienzanalyse
  - Verwendung von graphischen Notationen
  - Verwendung von Notationen zur Problem- oder Algorithmusfestlegung ***vor*** der Programmierung

Beispiel 1

- **Entwickeln**
- **Verbessern**
- **Effizienzbetrachtung**
- **Dokumentieren**
- **Test**

Entwickeln

Präzisierung der Aufgaben

- **Aufgabenformulierung:**

  In einer Datei befindet sich eine unbekannte Anzahl von Zahlen. Diese soll gelesen und nach ihrer Größe ausgedruckt werden.

- **Unklarheiten**
  - Format: Wie ausdrucken?
  - Mehrfachvorkommnisse: erlaubt, mehrfach aufgeführt?
  - Sortierung: aufsteigend, absteigend?
  - Sortierung in einem Feld: max. Anzahl unbekannt?
    (internes Sortieren/externers Sortieren)
  - Welcher Typ der einzelnen Elemente: INTEGER, REAL?
  - Feste Randbedingungen: Zahlen < größte in der Basismaschine darstellbare Zahl

- **Neue Formulierung:**

  In einer Datei befinden sich max. 100 nicht notwendigerweise verschiedene INTEGER-Zahlen. Diese Zahlen sollen gelesen und der Größe nach aufsteigend sortiert werden und entsprechend ihrer Vorkommenshäufigkeit einzelnen ausgegeben werden und zwar je 10 pro Zeile.

Entwickeln

Programmerstellung durch schrittweise Verfeinerung

- **Deklarationen**
- **Pseudocode**

```
MODULE Main;
    FROM ...
    CONST Max = 100;
TYPE Index = [1..MAX];
VAR Anzahl: CARDINAL;
VAR Behaelter; ARRAY Index OF INTEGER;

BEGIN
    ZahlenEinlesen;
    ZahlenSortieren;
    ZahlenDrucken
END.
```

Entwickeln

Verfeinerung

```
(* Zahlen einlesen *)
      Anzahl := 0;
      x := GetInt();
      WHILE NOT End() DO
            Anzahl := Anzahl + 1;
            Behaelter[Anzahl] := x;
            x := GetInt();
      END;
```

Ergänzung der Deklarationen:

```
VAR   x: INTEGER;
      nr: Index;
```

```
(* Zahlen drucken *)
      PutText("Die Zahlenfolge in aufsteigender Reihenfolge: "); Nl(),
      FOR nr := 1 TO Anzahl DO
            PutInt (Behaelter[nr], 10);

            IF nr MOD 10 = 0 THEN
                  Nl();
            END;
      END;
```


Entwickeln

Sortierverfahren

- **Hauptteil Sortieren**
  - einfache Verfahren: n2
  - gute Verfahren: n log n
  - Sortierstrategien
    - Einfügen
    - Auswählen
    - Vertauschen ← Bubblesort
    - Verschmelzen

→ Vorlesung Datenstrukturen und Algorithmen

Entwickeln

Bubblesort

```
(* Sortieren *)
     FOR k := 1 TO Anzahl DO
          FOR i := 1 TO Anzahl-1 DO
               IF Behaelter[ i ] > Behaelter [ i+1 ] THEN

                    (* Vertausche *)
                    hilf := Behaelter [ i ];
                    Behaelter [ i ] := Behaelter [ i+1 ];
                    Behaelter [ i+1 ] := hilf;

               END;
          END;
     END;
```

Ergänzung der Deklarationen:

```
VAR   i: Index;
      hilf: INTEGER;
```


Verbessern

1. Schritt

i	a					
Undef.	16	33	94	82	6	58
1	16	33	94	82	6	58
2	16	33	94	82	6	58
3	16	33	82	94	6	58
4	16	33	82	6	94	58
5	16	33	82	6	58	94

1. *Durchlauf größte Zahl am richtigen Platz*
2. *Durchlauf zweitgrößte Zahl am richtigen Platz*

...

- **Brauchen nur maximal Anzahl-1 Durchläufe, letzte Zahl ist dann automatisch am richtigen Platz**

  => letzter Durchlauf ist überflüssig

- **Brauchen beim k-ten Durchlauf nur bis Anzahl-k zu laufen**

- **Abbruchkriterium einführen:**

  Nicht mehr durchlaufen, wenn im letzten Durchgang nicht vertauscht worden ist

Verbessern **Programm nach 1. Schritt: Bubblesort 2**

```
(* Sortieren *)
     FOR k  := 1 TO Anzahl-1 DO
          getauscht := FALSE;
          FOR i := 1 TO Anzahl-k DO
               IF Behaelter [ i ] > Behaelter [ i+1 ] THEN
                    (* Vertausche *)
                    hilf := Behaelter [ i ];
                    Behaelter [ i ] := Behaelter [ i+1 ];
                    Behaelter [ i+1] := hilf;
                    getauscht := TRUE;
               END;
          END;
          If NOT getauscht THEN Exit;
     END
```

Ergänzung Deklaration:
`VAR getauscht: BOOLEAN;`

Verbessern

2. Schritt

k	i	a					
		16	33	94	82	6	58
1	5	16	33	82	6	58	94
2	1	16	33	82	6	58	94
2	2	16	33	82	6	58	94
2	3	16	33	6	82	58	94
2	4	16	33	6	58	82	94
3	1	16	33	6	58	82	94
	2	16	6	33	58	82	94
	3	16	6	33	58	82	94

Überflüssig bis Anzahl-k zu laufen, wenn im letzten Durchlauf nicht bis dorthin vertauscht wurde.

Verbessern **Programm nach 2. Schritt: Bubblesort 3**

Rechtes Ende nicht nur um 1 verkleinern pro Durchlauf

```
...
VAR rechtesEnde, letztesI: Index;
...
letztesI := Anzahl;
REPEAT
      rechtesEnde := letztesI; letztesI := 0;
      For i := 1 TO rechtesEnde -1 DO
            IF Behaelter[i] > Behaelter[i+1] THEN
              (* Vertausche *)
                  hilf := Behaelter[i];
                  Behaelter[i] := Behaelter[i+1];
                  Behaelter[i+1] := hilf;
                  letztesI := i;
            END
      END
UNTIL letztesI := 0
```


Effizienz-betrachtung **Effizienzbetrachtung für Bubblesort 3**

best-case-Analyse: geordnet
1 Schleifendurchlauf
n-1 Vergleiche, 0 Vertauschungen
hier Bubblesort gut !

worst-case-Analyse: Umgekehrt geordnet
n-1 Durchläufe
k-ter Durchlauf: innere Schleife: (n-k)-mal vertauscht

$$\sum_{k=1}^{n-1} (n-k) = \frac{(n-1)(n-2)}{2} = \frac{n^2}{2} - \frac{3n}{2} - 1$$

Anzahl der Vergleiche/ Vertauschungen

O-Notation

=> Bubblesort $O(n^2)$

d.h. $\exists$ Konstante a, b, c mit

f^{wc}_{Zeit} (Bubblesort, FeldDerLaenge) $\leq an^2 + bn + c$

Konstanten nicht ausgerchnet : Vorsicht !

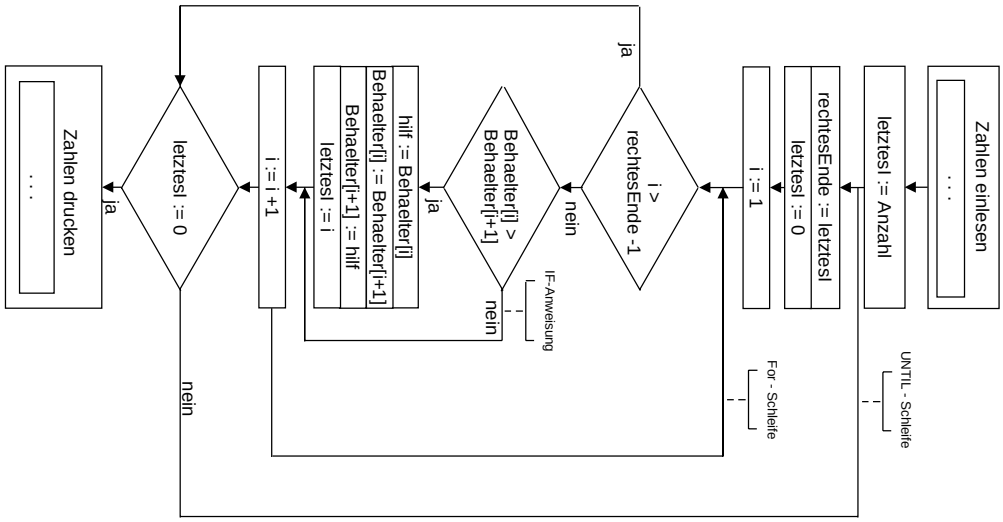
Gibt bessere Verfahren O(n log n)
=> Datenstrukturvorlesung, auch für O-Notation

Dokumentation: Allgemein

- **Bisher im Programm durch Kommentare, Programm selbst durch Lesbarkeit**
- **„parallele“ Notation : Flußdiagramme Struktogramme s.u.**
- **große Projekte : Anforderungsspezifikation Entwurfsspezifikation Rationales etc.**

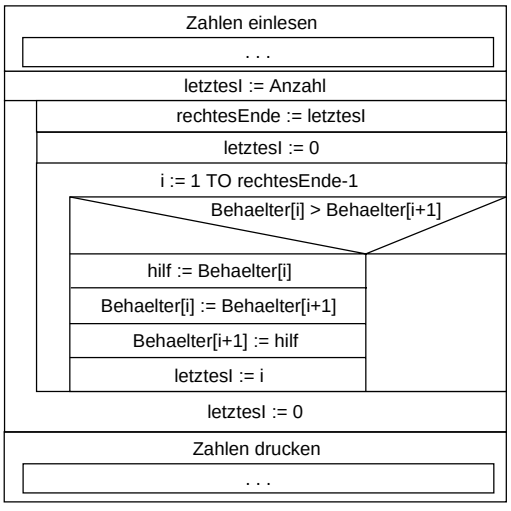
Dokumentieren

Flußdiagramm zu Bubblesort 3



Dokumentieren

Struktogramm zu Bubblesort 3



Black-Box-Test

z.B. **Normalfall - Extremfall - Betrachtung**

Normalfall

16 31 94 82 6 58
...

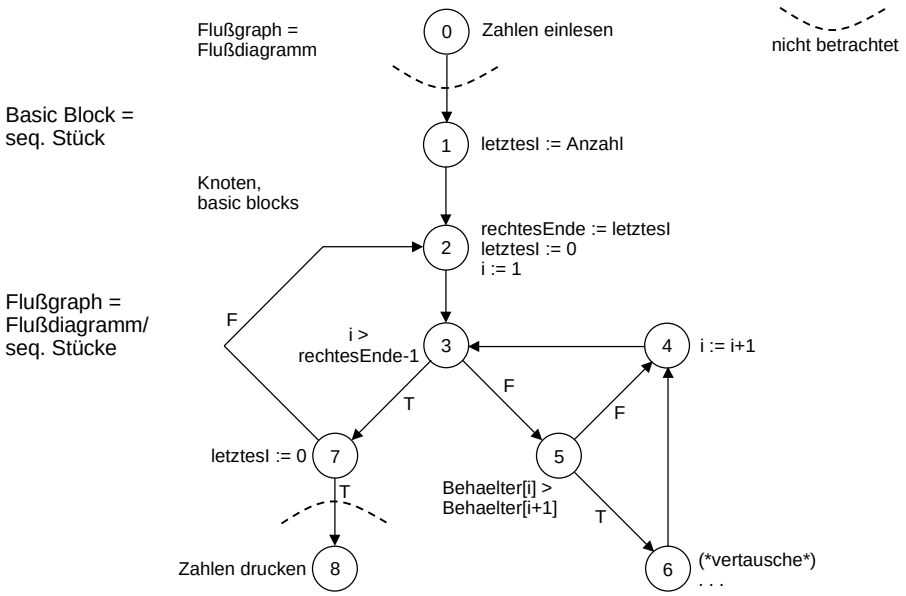
Extremfall

1	2	3	4	5	6
6	5	4	3	2	1
5	5	5	5	5	5
6					

leere Eingabe

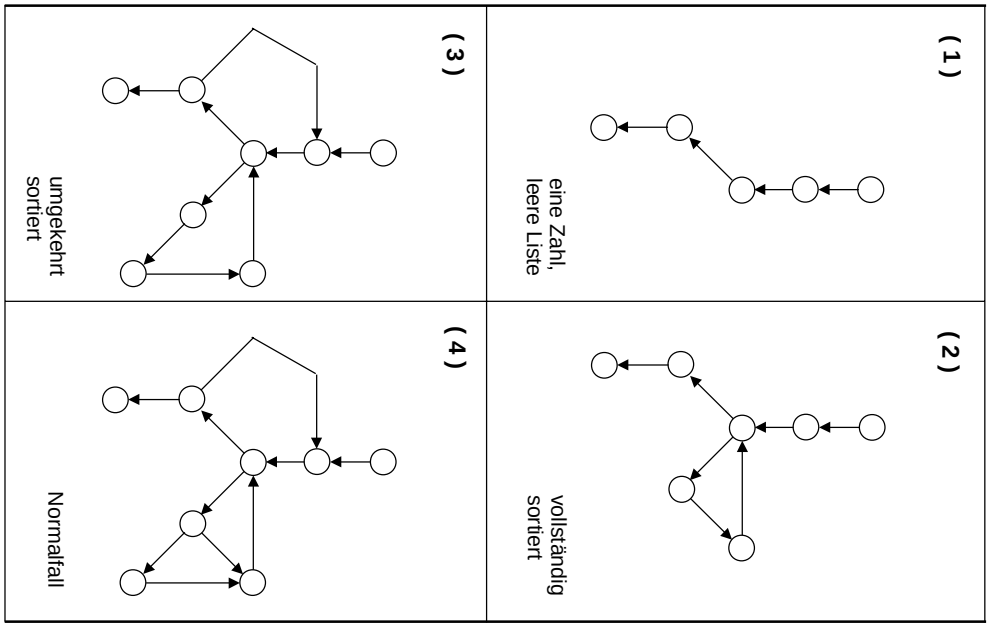
- **Verbesserte Methoden**
  - Äquivalenzklassen, Ursache-Wirkungs-Graph etc. -> spätere Vorlesungen
- **Test ist der Qualitätssicherung -> spätere Vorlesungen**

Testen und Testdatenermittlung: White-Box-Test



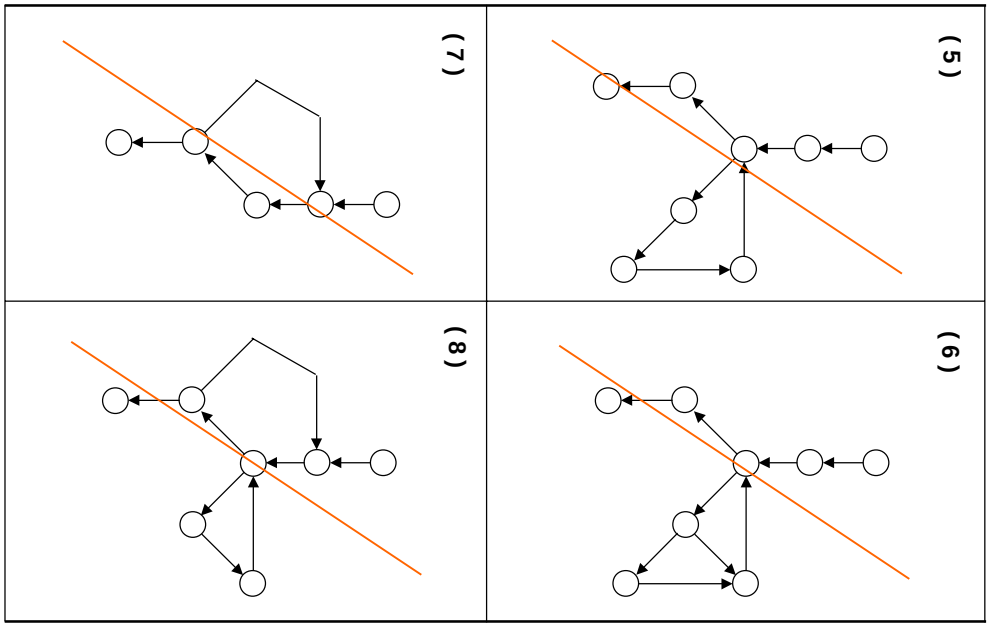
Test

White-Box-Test: mögliche Graphmuster für Durchlauf



Test

White-Box-Test: Nicht mögliche Graphmuster



White-Box-Test: Allgemeines

- **Testen und Durchführbarkeit:**
  - Ermittlung aller dyn. Programmpfade nicht möglich!
  - Ermittlung aller Programmpfade (bis auf vielfache von Schleifendurchläufen, wie oben) nur bei kleinen Programmen möglich.
  - Kantenüberdeckung und Bedingungsüberdeckung üblich!
    ⇒ Softwaretechnik, Qualitätssicherungs-Vorlesung

- **Testen und Korrektheit:**

  „Durch Testen kann nur die Anwesenheit aber nicht die Abwesenheit von Fehlern gezeigt werden“ (Dijkstra)

Was haben wir gelernt

- **Problemstellung erarbeiten, präzisieren**
- **Ein Sortierverfahren nach Vertauschstrategie,**
- **Vermeiden unnötiger Durchläufe, Teildurchläufe,**
- **best- und worst-case-Analyse für ein Beispiel**
- **„Dokumentation“ durch Flußdiagramme und Struktogramme**
- **Grenzen des Testens**
- **Black-Box-Test und White-Box-Test**

Glossar

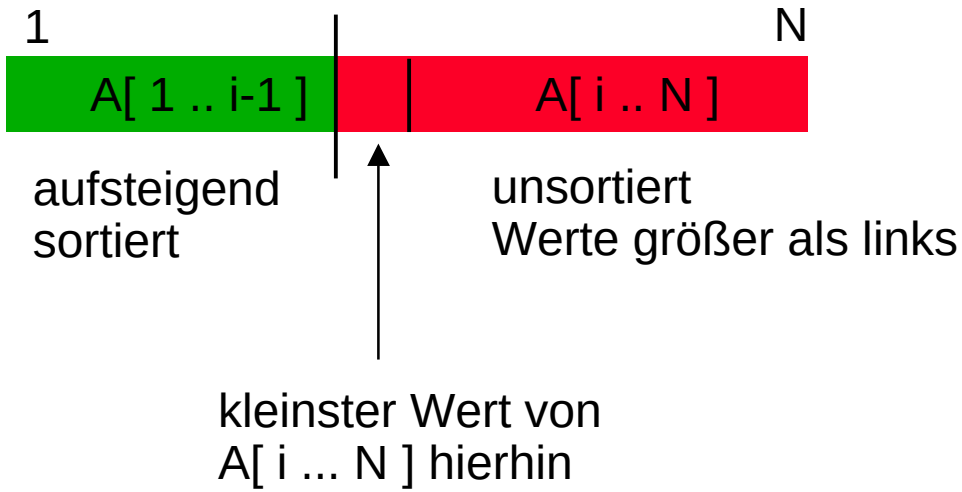
- **Problemformulierung durch Anforderungsspezifikation (Pflichtenheft)**
- **starke/schwache Äquivalenz von Programmen**
- **Wiederholung: schrittweise Verfeinerung, Wahl suggestiver Bezeichner, Kommentierung, Bedienungsoberflächengestaltung**
- **Sortierprinzipien: Einfügen, Auswählen, Vertauschen, Verschmelzung, Bubblesort: Sortierverfahren durch Vertauschen**
- **Programmverbesserung: Schritte bei Bubblesort**
- **Effizienzanalyse: best case, worst case**
- **Dokumentation durch Flußdiagramme, Nassi-Shneiderman-Diagramme**
- **Test als experimentelle, stichprobenartige Qualitätssicherung: Black-Box-Test (BBT), White-Box-Test (WBT), Normalfall-/Extermfallbetrachtung für BBT, Flußgraphenmethode für WBT**
- **Testen und Programmkorrektheit**

Beispiel 2

- **Partielle Korrektheit**
- **Termination**

Problem (Sortieren durch Auswählen)

- **Situation während des Programmlaufs**



- **Programmentwicklung und Verifikation *stets zusammen***
  - z.B. bei schrittweiser Verfeinerung
  - also Verifikation nie im Nachhinein

Logische Bedingungen (graphisch) 1

i = 1
A[1 .. 0] ist leer

anfangs 

jeder Wert in A[i .. N] ist ≥
jedem Wert in A[1 .. i-1], trivialerweiser richtig für i = 1

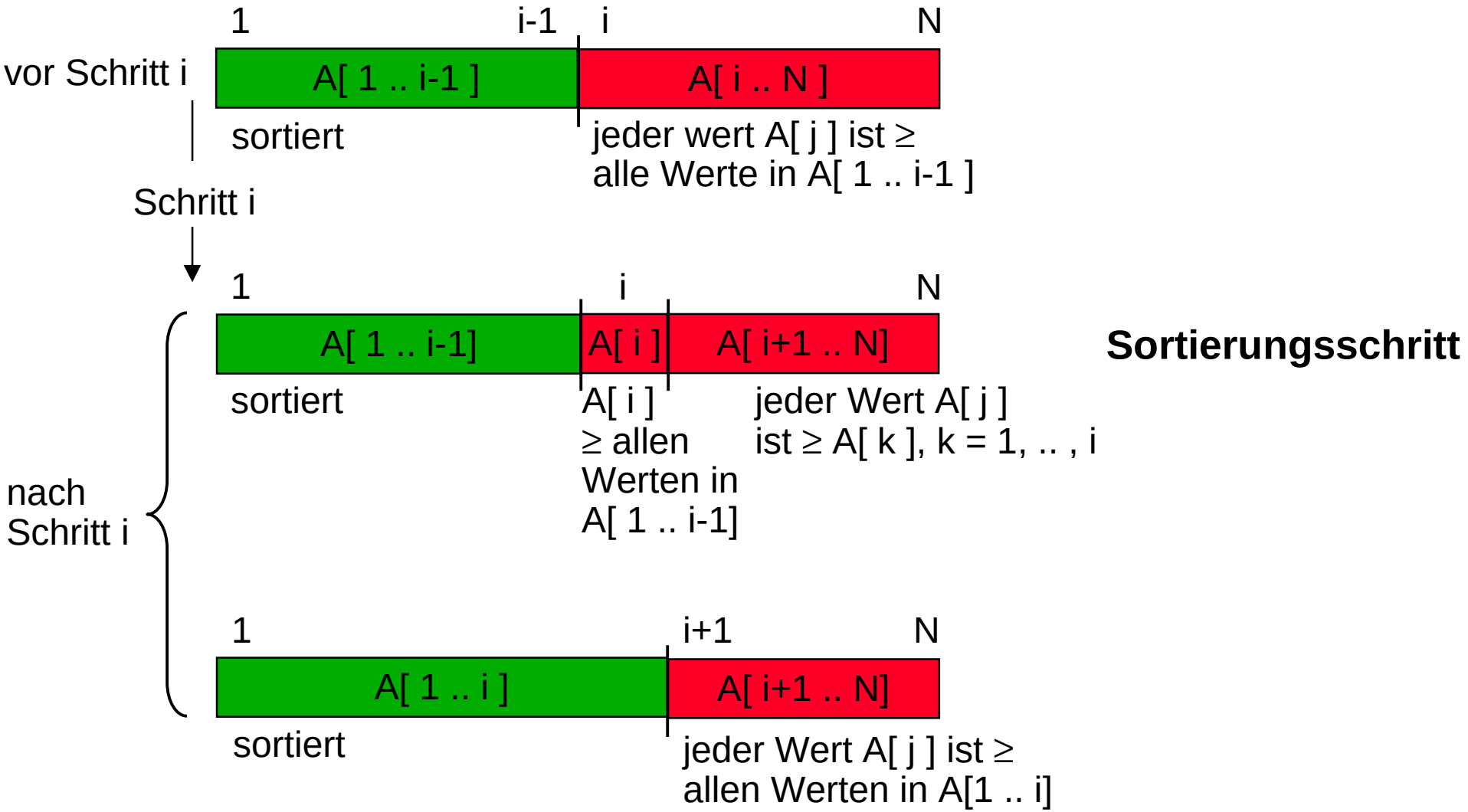
Sortierungsschritt

A[1..N-1] Sortiert

A[N] ≥ jedem Wert in
A[1 .. N-1]

⇒ A[1 .. N] ist sortiert

Logische Bedingungen (graphisch) 2



Verifikationsansatz für Sortieren durch Auswählen

- **Programmtext mit Vor- und Nachbedinung**

```
(* Zahlen einlesen und Zahlen ausdrucken, vgl.
   Bubblesort *)
(* Sortieren *)
      {Feld möglicherweise unsortiert}
      FOR i := 1 TO N-1 DO
            permutiere Werte von A[i .. N],
            um den kleinsten nach A[i] zu bringen
      END
      {Feldsortiert[1 .. N] ≡
      ∀ p,q((1≤p≤q≤N) ⇒ (A[p]≤A[q]))}
```

- **Anzuwendende Verifikationsregel**

$$\frac{\{(1 \le i \le N\text{-}1) \wedge R([\ 1\ ..\ i\ ])\}\ \text{permutiere ...}\ \{\ R([\ 1\ ..\ i\ ])\ \}}{\{R([\ ])\}\ \text{FOR i := 1 TO N-1 DO permutiere ...}\ \{\ R([\ 1\ ..\ N\text{-}1\ ])\ \}}$$

Wie finden wir R? - Ansatz

A[1 .. i-1] sortiert:

$Sortiert([\ 1\ ..\ i)) \equiv \forall\ p,q((1 \le p < q < i) \Rightarrow (A[\ p\ ] \le A[\ q\ ]))$

jeder Wert A[j] ist größer als alle Werte in A[1 .. i-1]:

$größereWertehinten([\ 1\ ..\ i\ )) \equiv$
$\forall\ l,m\ ((1 \le l < i \le m \le N) \Rightarrow (A[\ l\ ] \le A[\ m\ ]))$

$R([\ 1\ ..\ i)) \equiv Sortiert([1\ ..\ i)) \wedge größereWertehinten([\ 1\ ..\ i))$

Anschlüsse

- Am Anfang

  R[ ] ≡ R([1 .. 1)) ist Formalisierung von Feldmöglicherweiseunsortiert und trifft zu

  ≡ Sortiert([1 .. 1)) ∧ größereWertehinten([1 .. 1))

  ≡ ∀ p,q ((1≤p<q<1) ⇒ (A[p] ≤ A[q])) ∧ ∀ l,m ((1≤l<1≤ m≤N) ⇒ (A[l] ≤ A[m])) } W

  F W

  F W

- Am Ende

  nach letztem Schleifendurchlauf

  R([1 .. N-1]) ≡

  R([1 .. N)) → Feldsortiert([1 .. N])

  ≡ Sortiert([1 .. N)) ∧ ∀ l,m ((1≤l<N≤m ≤N) ⇒ ((A[l] ≤ A[m]))

  insb. richtig für m = N

Induktionsschritt

- **Haben jetzt zu zeigen**

{R([1 .. i))} (*) `permutiere Werte von A[i .. N ] um den kleinsten nach A[i] zu holen` {R([1 .. i])} ≡ {R([1 .. i+1))}

Nach (*) sollte gelten:

listkleinstes([i .. N]) ≡ ∀ q ((i≤q≤N) ⇒ (A[i]≤A[q]))

Nach (*) sei R([1 .. i)) nach wie vor gültig

Sortiert([1 .. i)) ∧ größereWertehinten([1 .. i)) ⟶ Sortiert([1 .. i+1))

d.h. mit Hinzunahme eines größeren Elements erhalten wir sortiertes Feld

größereWertehinten([1 .. i)) ∧ listkleinstes([i .. N)) ⟶
größereWertehinten([1 .. i+1))

d.h. A([i+1 .. N]) enthält nur Werte größer A[1 .. i]

somit R([1 .. i+1)) ≡ Sortiert([1 .. i+1)) ∧ größereWertehinten([1 .. i+1)) wahr

Innere Schleife: Vor- und Nachbedingungen

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
```

{R([1 .. i))}

```
    m:=i;

    FOR j:=i+1 TO N DO

       IF A[j] < A[m] THEN

          m := j;

       END

    END;
```

vertausche A[i] mit A[m]

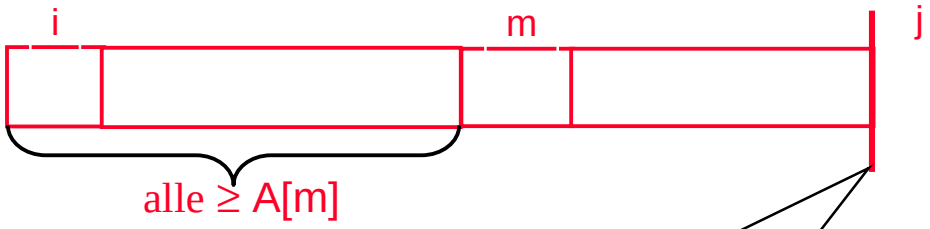
{listkleinestes([i .. N])}
{R([1..i))} klar

Innere Schleife: Schleifeninvariante

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
{R([1 .. i))}
    m:=i;

    FOR j:=i+1 TO N DO

       IF A[j] < A[m] THEN

          m := j;

       END

    END;


    vertausche A[i] mit A[m]

{listkleinestes([i .. N])}
{R([1..i))}
```

Schleifeninvariante:

Sind_größer_gl_A_m([i+1 .. j)) ≡
∀q ((i≤q<j) ⇒ (A[m] ≤ A[q]))



bis hierhin ist die FOR-Schleife gelaufen

Innere Schleife: Anschlussstellen

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
{R([1 .. i))}
    m:=i;
{Sind_größer_gl_A_m([])}
    FOR j:=i+1 TO N DO
        IF A[j] < A[m] THEN
            m := j;
        END
    END;
{Sind_größer_gl_A_m([i+1 .. N])}
    vertausche A[i] mit A[m]
{listkleinestes([i .. N])}
{R([1..i))}
```

≡ Sind_größer_gl_A_m([i+1 .. i+1)) ≡
∀q ((i≤q<i+1) ⇒ (A[m] ≤ A[q]))
d.h. A[m] ≤ A[i]

≡ ∀q ((i≤q≤N) ⇒ (A[m] ≤ A[q]))

daraus folgt Nachbedingung nach Ausführung von vertauschen

Innere Schleife: Induktionsschritt

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
```

{R([1 .. i))}

```
    m:=i;

    FOR j:=i+1 TO N DO
```

{Sind_größer_gl_A_m([i+1 .. j))}

```
        IF A[j] < A[m] THEN

            m := j;

        END
```

{Sind_größer_gl_A_m([i+1 .. j])}

```
    END;


    vertausche A[i] mit A[m]
```

{listkleinestes([i .. N])}

zu zeigen

wegen

{Sind_größer_gl_A_m([i+1 .. j)) ∧ (A[j] < A[m])}
`m:= j` {Sind_größer_gl_A_m([i+1 .. j])}

und

Sind_größer_gl_A_m([i+1 .. j) ∧ ¬(A[j] < A[m])
⇒ Sind_größer_gl_A_m([i+1 .. j])

Wann ist die Termination zu beweisen?

- **bei Sortieren durch Auswählen in Beispiel 2**
  - nichts zu zeigen:
    - nur FOR-Schleifen in `Zahlen einlesen`,
    - `Sortieren` und `Zahlen ausgeben`
- **zurück zu Bubblesort:**
  - dort ebenfalls nur Sortierteil

Termination im Hauptteil von Bubblesort (3. Version)

```
LetztesI:=Anzahl;
REPEAT
     rechtesEnde:=letztesI;
     letztesI:=0;
     0 ≤ letztesI < rechtesEnde in folgender Schleife
          FOR i:=1 TO rechtesEnde-1 DO
               IF Behaelter[i]>Behaelter[i+1] THEN
                    (* Vertausche *)                    stets i <
                    hilf:=Behaelter[i];                 rechtesEnde
                    Behaelter[i]:=Behaelter[i+1];
                    Behaelter[i+1]:=hilf;
                    letztesI:=i;
               END
          END
UNTIL letzesI=0;
```

Hält UNTIL-Schleife stets an?

Termination - Beweisskizze

Haben

`rechtesEnde, letztesI` ganzzahlig
`rechtesEnde` anfangs gleich `Anzahl`
`rechtesEnde` ≥ 0

Zeigen

`rechtesEnde` ist streng monoton fallend
über Schleifendurchläufe
hier max. `Anzahl` Durchläufe durch REPEAT-Schleife
mit `rechtesEnde` ≥ 0 (ggfs. weniger)

Wegen $0 \leq$ `letztesI` $<$ `rechtesEnde` muß
Terminationsbedingung nach max. `Anzahl`
Durchläufen zutreffen

Terminationsbeweis: Regeln u. a.

- **Allgemein für WHILE- und UNTIL-Schleifen**
  - suche ganzzahlige Größe, die
    - aufsteigend monoton und nach oben beschränkt ist, oder
    - absteigend monoton und nach unten beschränkt ist

- **Dokumentations-, Effizienz- und Testdatenüberlegungen analog zu Bubblesort**

Was haben wir gelernt?

- Sortierverfahren nach Strategie ’Sortieren durch Auswählen’
- grafische Notation von Prädikaten
- Schrittweise Verfeinerung mit Verifikationsüberlegungen
- Anwendungen der Floyd/Hoare-Regeln
- Terminationsbeweis

Glossar

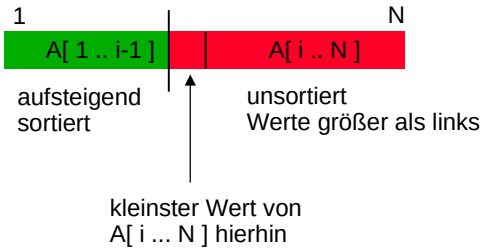
- **Partielle Korrektheit, Temination, Korrektheit**
- **Floyd-Hoare-Regeln für Kontrollstrukturen (s. Anhang)**
- **Vor-, Nachbedingungen und Schleifeninvarianten als prädikatenlogische Ausdrücke**
- **Terminationsbeweis als Monotonieüberlegung**

Beispiel 2

- **Partielle Korrektheit**
- **Termination**

Problem (Sortieren durch Auswählen)

- **Situation während des Programmlaufs**



- **Programmentwicklung und Verifikation *stets zusammen***
  - z.B. bei schrittweiser Verfeinerung
  - also Verifikation nie im Nachhinein

Logische Bedingungen (graphisch) 1

i = 1
A[1 .. 0] ist leer

anfangs 

jeder Wert in A[i .. N] ist ≥
jedem Wert in A[1 .. i-1], trivialerweiser richtig für i = 1

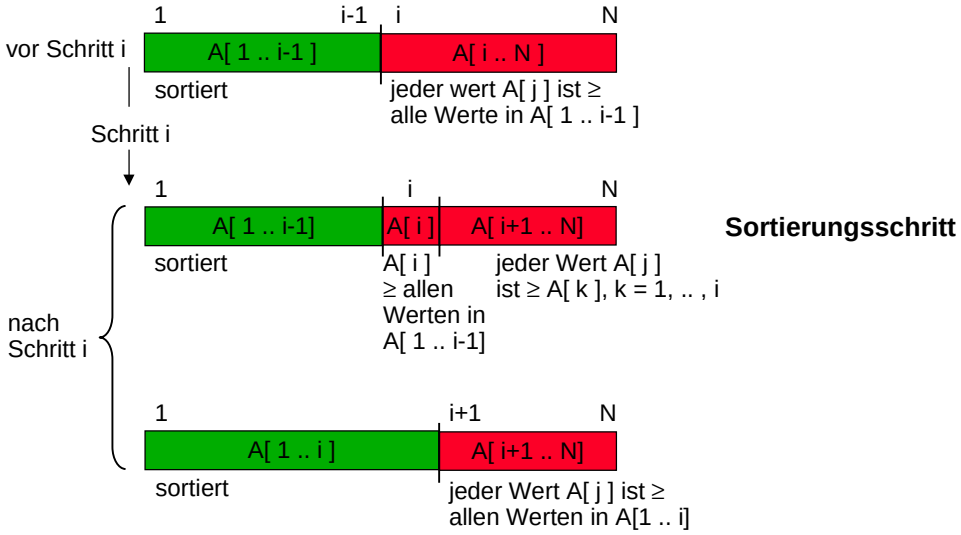
Sortierungsschritt

A[1..N-1] Sortiert

A[N] ≥ jedem Wert in
A[1 .. N-1]

⇒ A[1 .. N] ist sortiert

Logische Bedingungen (graphisch) 2



Verifikationsansatz für Sortieren durch Auswählen

- **Programmtext mit Vor- und Nachbedinung**

```
(* Zahlen einlesen und Zahlen ausdrucken, vgl.
   Bubblesort *)
(* Sortieren *)
      {Feld möglicherweise unsortiert}
      FOR i := 1 TO N-1 DO
            permutiere Werte von A[i .. N],
            um den kleinsten nach A[i] zu bringen
      END
      {Feldsortiert[1 .. N] ≡
      ∀ p,q((1≤p≤q≤N) ⇒ (A[p]≤A[q]))}
```

- **Anzuwendende Verifikationsregel**

$$\frac{\{(1 \le i \le N\text{-}1) \wedge R([\,1 \,..\, i\,))\}\ \text{permutiere ...}\ \{\, R([\,1 \,..\, i\,])\,\}}{\{R([\,])\}\ \text{FOR i := 1 TO N-1 DO permutiere ...}\ \{\, R([\,1 \,..\, N\text{-}1\,])\,\}}$$

Wie finden wir R? - Ansatz

A[1 .. i-1] sortiert:

Sortiert([1 .. i)) ≡ ∀ p,q((1≤p<q<i) ⇒ (A[p] ≤A[q]))

jeder Wert A[j] ist größer als alle Werte in A[1 .. i-1]:

größereWertehinten([1 .. i)) ≡
∀ l,m ((1 ≤l<i ≤m ≤N) ⇒ (A[l] ≤A[m]))

R([1 .. i)) ≡ Sortiert([1 .. i)) ∧ größereWertehinten([1 .. i))

Anschlüsse

- Am Anfang
  R[ ] ≡ R([1 .. 1)) ist Formalisierung von Feldmöglicherweiseunsortiert und trifft zu

  ≡ Sortiert([1 .. 1)) ∧ größereWertehinten([1 .. 1))
  ≡ ∀ p,q ((1≤p<q<1) ⇒ (A[p] ≤ A[q])) ∧
  F W W
  ∀ l,m ((1≤l<1≤ m≤N) ⇒ (A[l] ≤ A[m]))
  F W

- Am Ende
  R([1 .. N-1]) ≡
  R([1 .. N)) → Feldsortiert([1 .. N])
  ≡ Sortiert([1 .. N)) ∧ ∀ l,m ((1≤l<N≤m ≤N) ⇒ ((A[l] ≤ A[m]))
  insb. richtig für m = N

  nach letztem Schleifendurchlauf

Induktionsschritt

- **Haben jetzt zu zeigen**

{R([1 .. i))} (*)
```
permutiere Werte von A[i .. N ]
um den kleinsten nach A[i] zu holen
```
{R([1 .. i])} ≡ {R([1 .. i+1))}

Nach (*) sollte gelten:
listkleinstes([i .. N]) ≡ ∀ q ((i≤q≤N) ⇒ (A[i]≤A[q]))

Nach (*) sei R([1 .. i)) nach wie vor gültig
Sortiert([1 .. i)) ∧ größereWertehinten([1 .. i)) ⟶ Sortiert([1 .. i+1))
d.h. mit Hinzunahme eines größeren Elements erhalten wir sortiertes Feld

größereWertehinten([1 .. i)) ∧ listkleinstes([i .. N)) ⟶
größereWertehinten([1 .. i+1))
d.h. A([i+1 .. N]) enthält nur Werte größer A[1 .. i]

somit R([1 .. i+1)) ≡ Sortiert([1 .. i+1)) ∧ größereWertehinten([1 .. i+1)) wahr

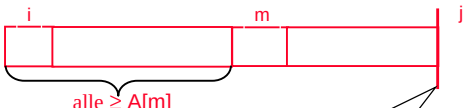
Innere Schleife: Vor- und Nachbedingungen

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
```
{R([1 .. i))}
```
    m:=i;

    FOR j:=i+1 TO N DO

       IF A[j] < A[m] THEN

          m := j;

       END

    END;


    vertausche A[i] mit A[m]
```
{listkleinestes([i .. N])}
{R([1..i))} klar

Innere Schleife: Schleifeninvariante

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
```
{R([1 .. i))}
```
    m:=i;

    FOR j:=i+1 TO N DO

       IF A[j] < A[m] THEN

          m := j;

       END

    END;


    vertausche A[i] mit A[m]
```
{listkleinestes([i .. N])}
{R([1..i))}

Schleifeninvariante:
Sind_größer_gl_A_m([i+1 .. j)) ≡
∀q ((i≤q<j) ⇒ (A[m] ≤ A[q]))



bis hierhin ist die FOR-Schleife gelaufen

Innere Schleife: Anschlussstellen

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
{R([1 .. i))}
    m:=i;
{Sind_größer_gl_A_m([])}
    FOR j:=i+1 TO N DO
       IF A[j] < A[m] THEN
          m := j;
       END
    END;
{Sind_größer_gl_A_m([i+1 .. N])}
    vertausche A[i] mit A[m]
{listkleinestes([i .. N])}
{R([1..i))}
```

≡ Sind_größer_gl_A_m([i+1 .. i+1)) ≡
∀q ((i≤q<i+1) ⇒ (A[m] ≤ A[q]))
d.h. A[m] ≤ A[i]

≡ ∀q ((i≤q≤N) ⇒ (A[m] ≤ A[q]))
daraus folgt Nachbedingung nach Ausführung von vertauschen

Innere Schleife: Induktionsschritt

```
(* permutiere Werte von A[i .. N], um den
   Kleinsten nach A[i] zu bringen *)
{R([1 .. i))}
    m:=i;
    FOR j:=i+1 TO N DO
{Sind_größer_gl_A_m([i+1 .. j))}
       IF A[j] < A[m] THEN
          m := j;
       END
{Sind_größer_gl_A_m([i+1 .. j])}
    END;
    vertausche A[i] mit A[m]
{listkleinestes([i .. N])}
```

zu zeigen

wegen

{Sind_größer_gl_A_m([i+1 .. j)) ∧ (A[j] < A[m])}
`m:= j` {Sind_größer_gl_A_m([i+1 .. j])}

und

Sind_größer_gl_A_m([i+1 .. j) ∧ ¬(A[j] < A[m])
⇒ Sind_größer_gl_A_m([i+1 .. j])

Wann ist die Termination zu beweisen?

- **bei Sortieren durch Auswählen in Beispiel 2**
  - nichts zu zeigen:
    - nur FOR-Schleifen in `Zahlen einlesen`,
    - `Sortieren` und `Zahlen ausgeben`
- **zurück zu Bubblesort:**
  - dort ebenfalls nur Sortierteil

Termination im Hauptteil von Bubblesort (3. Version)

```
LetztesI:=Anzahl;
REPEAT
     rechtesEnde:=letztesI;
     letztesI:=0;
     0 ≤ letztesI < rechtesEnde in folgender Schleife
          FOR i:=1 TO rechtesEnde-1 DO
               IF Behaelter[i]>Behaelter[i+1] THEN
                    (* Vertausche *)                    stets i <
                    hilf:=Behaelter[i];                 rechtesEnde
                    Behaelter[i]:=Behaelter[i+1];
                    Behaelter[i+1]:=hilf;
                    letztesI:=i;
               END
          END
UNTIL letzesI=0;
```

Hält UNTIL-Schleife stets an?

Termination - Beweisskizze

Haben

`rechtesEnde, letztesI` ganzzahlig
`rechtesEnde` anfangs gleich `Anzahl`
`rechtesEnde` ≥ 0

Zeigen

`rechtesEnde` ist streng monoton fallend
über Schleifendurchläufe
hier max. `Anzahl` Durchläufe durch REPEAT-Schleife
mit `rechtesEnde` ≥ 0 (ggfs. weniger)

Wegen $0 \leq$ `letztesI` $<$ `rechtesEnde` muß
Terminationsbedingung nach max. `Anzahl`
Durchläufen zutreffen

Terminationsbeweis: Regeln u. a.

- **Allgemein für WHILE- und UNTIL-Schleifen**
  - suche ganzzahlige Größe, die
    - aufsteigend monoton und nach oben beschränkt ist, oder
    - absteigend monoton und nach unten beschränkt ist
- **Dokumentations-, Effizienz- und Testdatenüberlegungen analog zu Bubblesort**

Was haben wir gelernt?

- Sortierverfahren nach Strategie ’Sortieren durch Auswählen’
- grafische Notation von Prädikaten
- Schrittweise Verfeinerung mit Verifikationsüberlegungen
- Anwendungen der Floyd/Hoare-Regeln
- Terminationsbeweis

Glossar

- Partielle Korrektheit, Temination, Korrektheit
- Floyd-Hoare-Regeln für Kontrollstrukturen (s. Anhang)
- Vor-, Nachbedingungen und Schleifeninvarianten als prädikatenlogische Ausdrücke
- Terminationsbeweis als Monotonieüberlegung

Beispiel 3

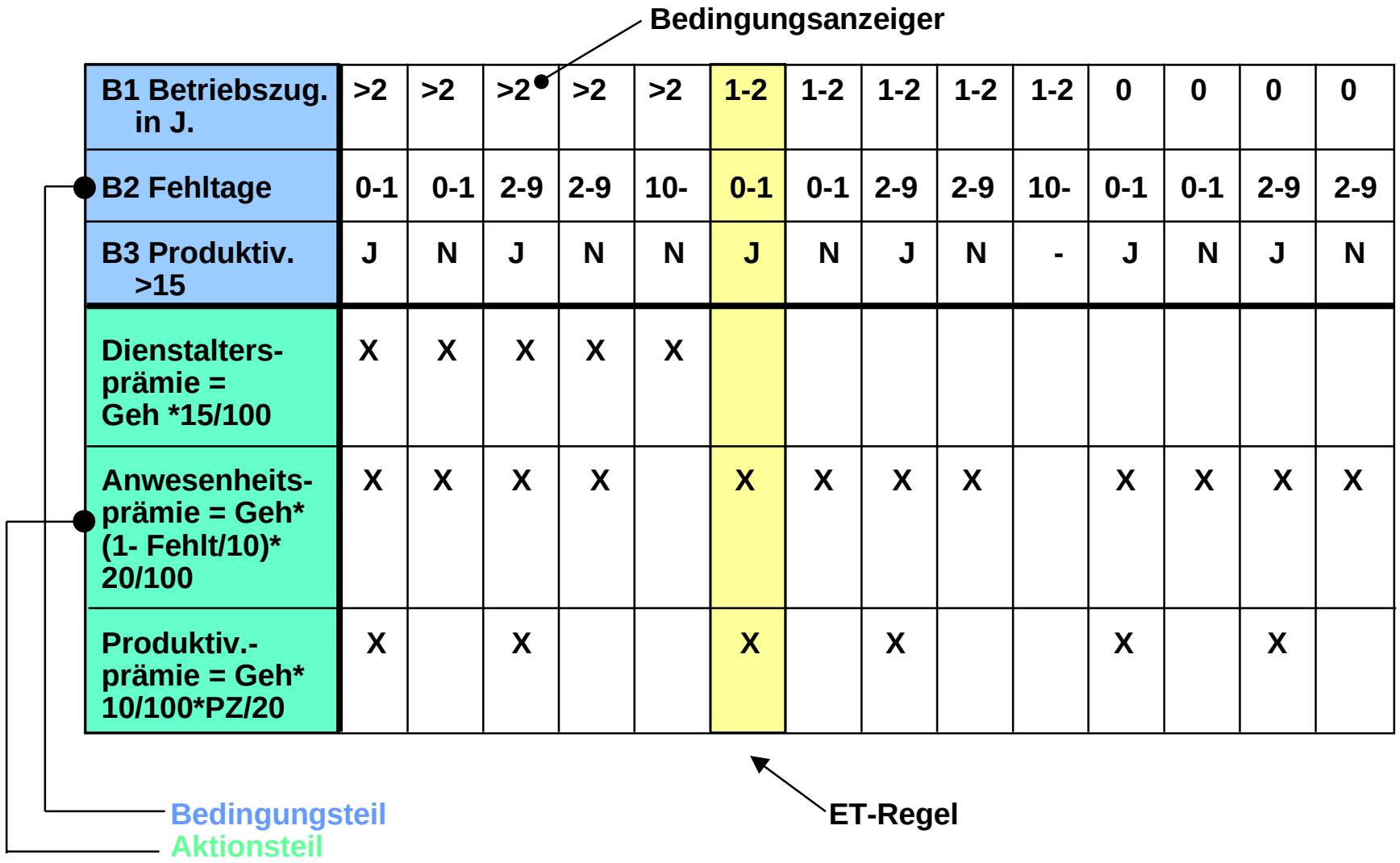
- **Entscheidungstabellen**
- **Programmcode für ETn**

Handhabung verschiedener Fälle

- **Berechnung Jahresprämie unter Berücksichtigung von**
  - Betriebszugehörigkeit
  - Anzahl der Fehltage
  - Produktivität

- **Präzisierung der Aufgabenstellung durch Entscheidungstabelle**

Formulierung durch Entscheidungstabelle



B1 Betriebszug. in J.	>2	>2	>2	>2	>2	1-2	1-2	1-2	1-2	1-2	0	0	0	0
B2 Fehltage	0-1	0-1	2-9	2-9	10-	0-1	0-1	2-9	2-9	10-	0-1	0-1	2-9	2-9
B3 Produktiv. >15	J	N	J	N	N	J	N	J	N	-	J	N	J	N
Dienstalters-prämie = Geh *15/100	X	X	X	X	X									
Anwesenheits-prämie = Geh* (1- Fehlt/10)* 20/100	X	X	X	X		X	X	X	X		X	X	X	X
Produktiv.-prämie = Geh* 10/100*PZ/20	X		X			X		X			X		X	

d.h. ET ist Hilfsmittel bei der Problemausarbeitung

ETn: Kombinatorische Vielfalt und Prüfungen

- **ETn für Übersicht u. Handhabung vieler Fälle und dabei auszuführende Aktionen**
  - Angabe der richtigen Bedingung
  - dann auszuführende Aktionen

- **bei festgelegtem Bedingungs- und Aktionsteil**

Prüfung auf Redundanzfreiheit
Vollständigkeit
Widerspruchsfreiheit
} **mechanisch überprüfbar**

in obigem Beispiel wurden z.B. vergessen

>2
10-
J
X

0
10-
-

Entscheidungstabellen: Begriffe

- **einfache ET: ohne Irrelevanzanzeiger**
- **komplexe ET: mit Irrelevanzanzeriger**
- **erweiterte ET: anstelle J/N im Bedingungsteil weitere Bedingungen**
- **geschachtelte ET**
- **ET mit sonst-Regel**

Modula-3 - Programm

```
MODULE Praemie;

FROM  ...

CONST MinProd = 15;

VAR   Betriebszugeh, Fehltage, Produktivität,
      Gehalt, DAP, ANWP, PRODP, PR : CARDINAL;

PROCEDURE Dienstalterspr (FGeh, FBetriebszugeh : CARDINAL) : CARDINAL;

BEGIN
RETURN (FGeh * 15 DIV 100)
END;

...
BEGIN

...
      s. nächste Folie
...

END Praemie.
```

analoge Funktion für

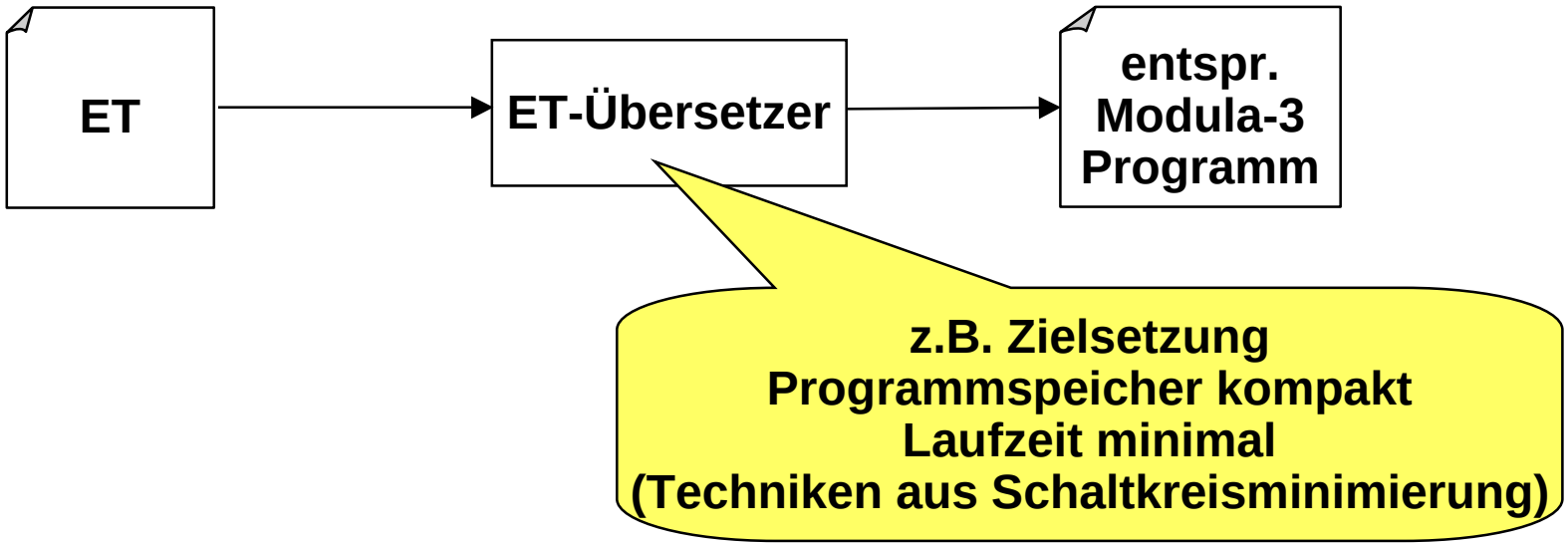
Produktivitätsprämie, Anwesenheitsprämie

Kernteil: geschaltete Fallunterscheidungen

```
IF Betriebszugeh > 2 THEN
   CASE Fehltage OF
      0,1 =>  IF Produktivitaet > MinProd THEN
                  PRODP:= Produktivitaetpr (Gehalt, Produktivitaet);
              ELSE
                  PRODP:= 0;
              END;
              ANWP:= Anwesenheitspr (Gehalt, Fehltage);
      2..9 => IF Produktivitaet > MinProd THEN
                  PRODP:= Produktivitaetpr (Gehalt, Produktivitaet);
              ELSE
                  PRODP:= 0;
              END;
              ANWP:= Anwesenheitspr (Gehalt, Fehltage);
   ELSE (*Fehltage : mehr als 9*)
      PRODP :=0; ANWP :=0;
   END;
   DAP := Dienstalterspr (Gehalt, Betriebszugeh);
ELSIF Betriebszugeh = 1 or Betriebszugeh = 2 THEN  (*analog*)
ELSE (* Betriebszugeh : unter einem Jahr*)    (*analog*)
...
END;
PR := DAP + ANWP + PRODP;
```

Autom. Übersetzung/Ausführung anstelle von Handcodierung

- **Mechanische manuelle Übertragung der Tabelle: viele Fälle, Fälle werden vergessen:**
  - **mechanische automatische Übertragung durch ET-Übersetzer**



- **direkte Ausführung der Tabellen: ET-Interpretierer**
  - **z.B. in Modula-3 schreibbar**

Rolle der Entscheidungstabellen u. a.

- **ET hier für**
  - Problemfestlegung
  - Ausgangspunkt der Programmentwicklung
  - Hilfsmittel der Überprüfung
  - Dokumentation
  - Hilfsmittel der Testdatenbestimmung
- **Korrektheit, Termination, Effizienz, Dokumentation, Testdatengenerierung für Beispiel 3**

Was haben wir gelernt?

- Beherrschung der kombinatorischen Explosion von Fällen durch ET
- ET als Hilfsmittel der Programmfestlegung
- ET als Hilfsmittel für Programmentwicklung, Überprüfung, Dokumentation, Testdatenerstellung
- ET-Begriffe
- ET Analyse auf Vollständigkeit, Widerspruchsfreiheit und Redundanzfreiheit
- ET-Übersetzung
- Manuelle Codierung einer ET

Glossar

- Bedingungs- und Aktionsteil von ET, ET-Regel
- einfache, komplexe, erweiterte und geschachtelte ETn
- äquivalenter Code zu ET
- ET-Übersetzung, ET-Interpretation
- Rolle von ETn: Problemdefinition, ...

Beispiel 3

- **Entscheidungstabellen**
- **Programmcode für ETn**

Handhabung verschiedener Fälle

- **Berechnung Jahresprämie unter Berücksichtigung von**
  - Betriebszugehörigkeit
  - Anzahl der Fehltage
  - Produktivität
- **Präzisierung der Aufgabenstellung durch Entscheidungstabelle**

Formulierung durch Entscheidungstabelle

Bedingungsanzeiger

B1 Betriebszug. in J.	>2	>2	>2	>2	>2	1-2	1-2	1-2	1-2	1-2	0	0	0	0
B2 Fehltage	0-1	0-1	2-9	2-9	10-	0-1	0-1	2-9	2-9	10-	0-1	0-1	2-9	2-9
B3 Produktiv. >15	J	N	J	N	N	J	N	J	N	-	J	N	J	N
Dienstalters-prämie = Geh *15/100	X	X	X	X	X									
Anwesenheits-prämie = Geh* (1- Fehlt/10)* 20/100	X	X	X	X		X	X	X	X		X	X	X	X
Produktiv.-prämie = Geh* 10/100*PZ/20	X		X			X		X			X		X	

Bedingungsteil
Aktionsteil
ET-Regel

d.h. ET ist Hilfsmittel bei der Problemausarbeitung

ETn: Kombinatorische Vielfalt und Prüfungen

- **ETn für Übersicht u. Handhabung vieler Fälle und dabei auszuführende Aktionen**
  - Angabe der richtigen Bedingung
  - dann auszuführende Aktionen

- **bei festgelegtem Bedingungs- und Aktionsteil**

Prüfung auf Redundanzfreiheit
Vollständigkeit
Widerspruchsfreiheit
} **mechanisch überprüfbar**

in obigem Beispiel wurden z.B. vergessen

>2
10-
J
X

0
10-
-

Entscheidungstabellen: Begriffe

- einfache ET: ohne Irrelevanzanzeiger
- komplexe ET: mit Irrelevanzanzeriger
- erweiterte ET: anstelle J/N im Bedingungsteil weitere Bedingungen
- geschachtelte ET
- ET mit sonst-Regel

Modula-3 - Programm

```
MODULE Praemie;

FROM  ...

CONST MinProd = 15;

VAR   Betriebszugeh, Fehltage, Produktivität,
      Gehalt, DAP, ANWP, PRODP, PR : CARDINAL;

PROCEDURE Dienstalterspr (FGeh, FBetriebszugeh : CARDINAL) : CARDINAL;

BEGIN
RETURN (FGeh * 15 DIV 100)
END;

...
BEGIN

...
      s. nächste Folie
...

END Praemie.
```

analoge Funktion für

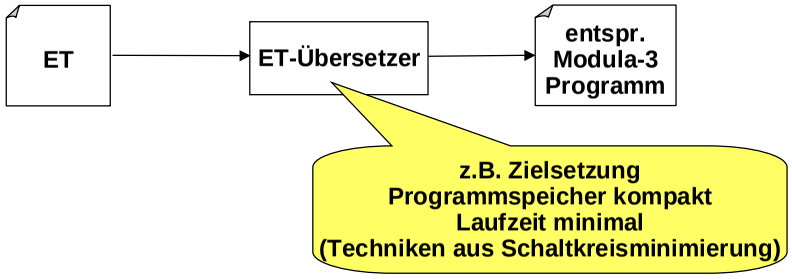
Produktivitätsprämie, Anwesenheitsprämie

Kernteil: geschaltete Fallunterscheidungen

```
IF Betriebszugeh > 2 THEN
  CASE Fehltage OF
    0,1 =>  IF Produktivitaet > MinProd THEN
              PRODP:= Produktivitaetpr (Gehalt, Produktivitaet);
            ELSE
              PRODP:= 0;
            END;
            ANWP:= Anwesenheitspr (Gehalt, Fehltage);
    2..9 => IF Produktivitaet > MinProd THEN
              PRODP:= Produktivitaetpr (Gehalt, Produktivitaet);
            ELSE
              PRODP:= 0;
            END;
            ANWP:= Anwesenheitspr (Gehalt, Fehltage);
  ELSE (*Fehltage : mehr als 9*)
    PRODP :=0; ANWP :=0;
  END;
  DAP := Dienstalterspr (Gehalt, Betriebszugeh);
ELSIF Betriebszugeh = 1 or Betriebszugeh = 2 THEN  (*analog*)
ELSE (* Betriebszugeh : unter einem Jahr*)   (*analog*)
  ...
END;
PR := DAP + ANWP + PRODP;
```


Autom. Übersetzung/Ausführung anstelle von Handcodierung

- **Mechanische manuelle Übertragung der Tabelle: viele Fälle, Fälle werden vergessen:**
  - **mechanische automatische Übertragung durch ET-Übersetzer**



- **direkte Ausführung der Tabellen: ET-Interpretierer**
  - **z.B. in Modula-3 schreibbar**

Programmcode für ETn

Rolle der Entscheidungstabellen u. a.

- **ET hier für**
  - Problemfestlegung
  - Ausgangspunkt der Programmentwicklung
  - Hilfsmittel der Überprüfung
  - Dokumentation
  - Hilfsmittel der Testdatenbestimmung
- **Korrektheit, Termination, Effizienz, Dokumentation, Testdatengenerierung für Beispiel 3**

Was haben wir gelernt?

- **Beherrschung der kombinatorischen Explosion von Fällen durch ET**
- **ET als Hilfsmittel der Programmfestlegung**
- **ET als Hilfsmittel für Programmentwicklung, Überprüfung, Dokumentation, Testdatenerstellung**
- **ET-Begriffe**
- **ET Analyse auf Vollständigkeit, Widerspruchsfreiheit und Redundanzfreiheit**
- **ET-Übersetzung**
- **Manuelle Codierung einer ET**

Glossar

- Bedingungs- und Aktionsteil von ET, ET-Regel
- einfache, komplexe, erweiterte und geschachtelte ETn
- äquivalenter Code zu ET
- ET-Übersetzung, ET-Interpretation
- Rolle von ETn: Problemdefinition, ...

Beispiel einer Programmentwicklung

- Aufgabenstellung
- Prozedurale Lösung
- Objekt- und klassenbasierte Lösung
- Lösung mit Objektorientierung

Informell festgehaltene Anforderungen

- **Herr X**, ein engagierter Jungunternehmer, hat viele geschäftliche Verbindungen.

  Da er sich die Adressen seiner Geschäftspartner nicht merken kann und die Sammlung der Visitenkarten im Geldbeutel bereits sehr unübersichtlich geworden ist, wünscht sich Herr X eine Möglichkeit, um die **Adressen** seiner Geschäftspartner zu verwalten.

- Er möchte folgende Angaben für jede Adresse vorfinden:

  Name, **Vorname**, **PLZ**, **Ort** und **Telefon-Nummer**.

Verschiedene Lösungen

- **Texteditor: keine Lösung**
  - Suchen und Sortieren ist eher mühsam.
  - Daten können leicht unbeabsichtigt zerstört oder verfälscht werden (cut & paste)

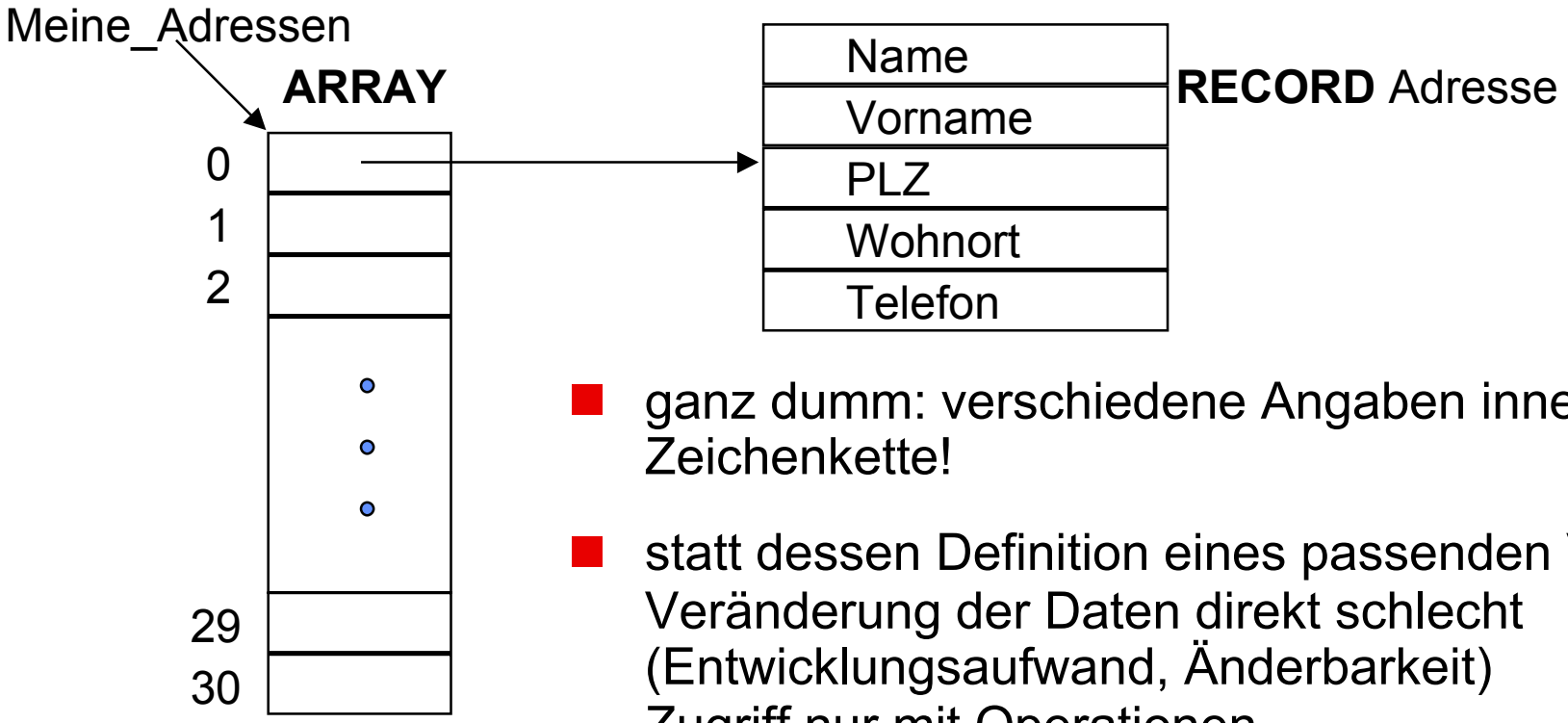
- **Adreßverwaltungsprogramm:**
  - Eingabefehler abprüfen
  - Sicherheit der Dateien gewährleistet

Diskutieren der Lösungsansätze von schlecht bis gut

Offene Datenstrukturen

Eine Adresse

Kugler	**Hans**	**8021**	**Zürich**	**01-4330570**	. . .



- ganz dumm: verschiedene Angaben innerhalb einer Zeichenkette!
- statt dessen Definition eines passenden Verbundtyps
  Veränderung der Daten direkt schlecht
  (Entwicklungsaufwand, Änderbarkeit)
  Zugriff nur mit Operationen
- Verbundtypdefinition und Operationen in ein Modul zusammengefaßt

Modul Adress_Operationen-1

Prozedurale Lösung

```
INTERFACE Adress_Operationen;

IMPORT Rd,Wr;

TYPE Adresse = RECORD
       Name : TEXT;
       Vorname : TEXT;
       PLZ : TEXT;
       Ort : TEXT;
       Tel : TEXT;
     END;

CONST Max_Adr = 30;

TYPE Listen_Laenge = [1 .. Max_Adr];
     Adressen_Liste = ARRAY Listen_Laenge OF Adresse;

VAR  Meine_Adressen   : Adressen_Liste;  (* Datensaetze *)

(*-------------------------------------------------------------*)

PROCEDURE Setze_Name(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Vorname(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_PLZ(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Ort(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Tel(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Hat_Namen(I: Listen_Laenge; Wert : TEXT): BOOLEAN;

PROCEDURE Zeige_Adresse(I: Listen_Laenge);

PROCEDURE Lese_Adresse(I: Listen_Laenge; File : Rd.T);

PROCEDURE Schreibe_Adresse(I: Listen_Laenge; File : Wr.T);

END Adress_Operationen.
```

Adressen-Typ

Zentrale Variable

Deklaration der Operationen

```
MODULE Adressen_Verwaltung EXPORTS Main;
IMPORT Adress_Operationen AS AO;

BEGIN  (* -- des Hauptprogramms*)
  Adress_Datei_lesen := IO.OpenRead(Gesch_Adr);
  I := 0;
  WHILE NOT SIO.EOF(Adress_Datei_lesen) DO
    I := I + 1;
    AO.Lese_Adresse (I, Adress_Datei_lesen);
  END;
  Anz := I;
  Rd.Close(Adress_Datei_lesen);
(*-------------------------------------------------------------*)
  SIO.PutText ("Den zu suchenden Namen bitte > ");
  Name:=SIO.GetLine ();
  I := 1;
  WHILE (NOT Gefunden) AND (I <= Anz) DO   (* -- Daten suchen*)
    Gefunden := AO.Hat_Namen(I, Name);
    I := I + 1;
  END;
(*-------------------------------------------------------------*)
  IF Gefunden THEN
    AO.Zeige_Adresse(I-1);
    SIO.PutLine("Die neuen Angaben bitte!");
    SIO.PutText("PLZ > ");          (* PLZ *)
    PLZ := SIO.GetLine ();
    AO.Setze_PLZ(I-1, PLZ);

    SIO.PutText("Ort > ");          (* Wohnort*)
    Ort := SIO.GetLine ();
    AO.Setze_Ort(I-1, Ort);

    SIO.PutText("Telefon-Nr. > "); (* Telefon*)
    Tel := SIO.GetLine ();
    AO.Setze_Tel(I-1, Tel);

    AO.Zeige_Adresse(I-1);          (* Daten anzeigen*)
  ELSE
    SIO.PutLine("Spezifizierte Adresse nicht vorhanden!");
  END;
(*-------------------------------------------------------------*)
  Adress_Datei_schreiben := IO.OpenWrite (Gesch_Adr);
  FOR J := 1 TO Anz DO
    AO.Schreibe_Adresse(J, Adress_Datei_schreiben);
  END;
  Wr.Close (Adress_Datei_schreiben);

END Adressen_Verwaltung.
```

Import der Operationen

Aufruf der Operationen

Diskussion 1

- **Es ist möglich, die definierten Operationen zu umgehen**
  - Die Operationen müssen im Hauptprogramm nicht verwendet werden.
- **Die direkte Manipulation der Variable `Meine_Adressen` ist im Hauptprogramm immer noch möglich!**

```
  SIO.PutLine("Die neuen Angaben
bitte!");
  SIO.PutText("PLZ > ");          (*
PLZ *)
  PLZ := SIO.GetLine ();
  AO.Setze_PLZ(I-1, PLZ);

  Meine_Adressen[I-1].PLZ :=
'abcd';
```

Aufbau der Datenstruktur ist bekannt. Sie kann direkt manipuliert werden!

Diskussion 2

- **Die einzelnen Verwaltungsfunktionen Suchen, Einfügen, Löschen sind im Änderungsteil des Hauptprogramms fest „eingebacken“**

  Besser Prozeduren hierfür deklarieren, die dann noch externe Prüfungen übernehmen und die die Operationen der Datenspeicherung aufrufen

- **Die E/A Handhabung sollte aus den Verwaltungsoperationen herausgezogen werden: kann sich leicht ändern**

Neue Anforderungen

- **Herr X hat einen Software-Engineering-Kurs gehört.**
  - Dort hat er gelernt, daß das Geheimnisprinzip angewendet werden muß, wenn Informationen geschützt werden sollen
  - Er entscheidet sich, die Variable `Meine_Adressen` durch ein Objektmodul (Datenkapsel) zu schützen.
- **Gleichzeitig soll sein Programm um folgende Funktionalität erweitert werden: Es soll**
  - ein Datensatz eingegeben werden können
  - nach einem Datensatz gesucht werden können (Name)
  - ein Datensatz modifiziert werden können
  - ein Datensatz angezeigt werden können
  - auf den Anfang der Sammlung positioniert werden können
  - geblättert werden können

Objektmodul für Adressen

```
INTERFACE Adress_OM;
(* Dieses Modul realisiert ein Objektmodul, mit dem maximal 30 Adressen verwaltet
werden koennen. Bevor eine Operation des Objektmoduls ausgefuehrt werden kann, muss die
Prozedur Lese_Adressen ausgefuehrt worden sein.*)

PROCEDURE Neue_Adresse();
(* Fordert den Benutzer auf, eine neue Adresse einzugeben*)

PROCEDURE Modifiziere_Adresse();
(*Fordert den Benutzer auf, Angaben fuer die aktuelle
  Adresse einzugeben.*)

PROCEDURE Suche_Adresse();
(*Fordert den Benutzer auf, einen Namen einzugeben. Ist eine Adresse mit diesem Namen
vorhanden, wird diese angezeigt. Diese Adresse wird die aktuell selektierte Adresse.*)

PROCEDURE Lese_Adressen();
(*Initialisiert das Modul und liest alle Adressen von Datei ein. Die erste Adresse ist
die aktuell selektierte Adr.*)

PROCEDURE Schreibe_Adressen();
(*Schreibt die Adressen zurueck auf Datei.*)

PROCEDURE Zeige_Adresse();
(*Zeigt die aktuelle Adresse auf dem Bildschirm.*)

PROCEDURE Zuruecksetzen();
(*Es wird auf den Anfang der Adressensammlung positioniert. Die erste Adresse wird die
aktuelle Adresse.*)

PROCEDURE Blaettern();
(*Zeigt ausgehend von der aktuellen Adresse die naechste Adresse am Bildschirm.*)

END Adress_OM.
```

Hauptprogramm 2

```
MODULE Adress_Verwaltung EXPORTS Main;
IMPORT Adress_OM AS Adr;
(*-------------------------------------------------*)
PROCEDURE Zeige_Adress_Menue(): CHAR =
BEGIN
  SIO.PutLine("--------------------");
  SIO.PutLine("Zurueecksetzen (r)");
  SIO.PutLine("Blaettern      (b)");
  SIO.PutLine("Suchen         (s)");
  SIO.PutLine("Aendern        (a)");
  SIO.PutLine("Eintragen      (e)");
  SIO.PutLine("Zeigen         (z)");
  SIO.PutLine("BEENDEN        (sonst)");
  SIO.PutLine("--------------------");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Adress_Menue;

PROCEDURE Menu() =
VAR Wahl : CHAR := 'r';
    Kommandos := SET OF CHAR {'r','b','s','a','z','e'};
BEGIN
  REPEAT
    Wahl := Zeige_Adress_Menue();
    IF Wahl IN Kommandos THEN
      CASE Wahl OF
         'r'    => Adr.Zurueecksetzen();
         'b'    => Adr.Blaettern();
         's'    => Adr.Suche_Adresse();
         'a'    => Adr.Modifiziere_Adresse();
         'z'    => Adr.Zeige_Adresse();
         'e'    => Adr.Neue_Adresse();
      END;
    ELSE
      SIO.PutLine ("ENDE DES PROGRAMMS");
    END;
  UNTIL NOT Wahl IN Kommandos;
END Menu;
(*-------------------------------------------------*)
BEGIN (* des Hauptprogramms*)
  Adr.Lese_Adressen();
  Menu();
  Adr.Schreibe_Adressen();
END Adress_Verwaltung.
```

Impl. des Objektmoduls - 1

```
MODULE Adress_OM;

IMPORT IO, Rd, Wr,Text;

TYPE Adresse = RECORD
       Name     : TEXT;
       Vorname  : TEXT;
       PLZ      : TEXT;
       Ort      : TEXT;
       Tel      : TEXT;
     END;

CONST Adr_Datei_Name = "gesch.adr";
      Max = 30;

TYPE
  Listen_Laenge = [1..Max];
  Adressen_Liste = ARRAY Listen_Laenge OF Adresse;

(*-----------------------------------------------*)
(* gekapselte Daten*)

VAR Meine_Adressen : Adressen_Liste;
    Letzte_Adresse : Listen_Laenge := 1;
    Akt_Adresse    : Listen_Laenge := 1;
```

Impl. des Objektmoduls - 2

```
PROCEDURE Lese_Adressen() =
VAR I : Listen_Laenge;
    Zeile : TEXT;
    Adr_Datei_lesen : Rd.T;
BEGIN
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  I := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Name := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Vorname :=  Zeile;
    ...
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Ort := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Tel := Zeile;

    Zeile:=IO.GetLine(Adr_Datei_lesen); (* Leerzeile*)
    INC(I);
  END;
  Letzte_Adresse := I - 1; Akt_Adresse := 1;
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;

PROCEDURE Zuruecksetzen() =
BEGIN
  Akt_Adresse := 1; Zeige_Adresse();
END Zuruecksetzen;

PROCEDURE Blaettern() =
BEGIN
  IF Akt_Adresse < Letzte_Adresse THEN
    INC(Akt_Adresse);
    Zeige_Adresse();
  ELSE
    IO.Put("Kein Datensatz mehr vorhanden!\n");
  END;
END Blaettern;
```

Impl. des Objektmoduls - 3

```
PROCEDURE Gebe_Adresse_Aus (Adr: Adresse;
                            Seperator :TEXT;
                            Wo : Wr.T := NIL) =
BEGIN
IO.Put(Adr.Name     & Seperator &
       Adr.Vorname  & Seperator &
       Adr.PLZ      & Seperator &
       Adr.Ort      & Seperator &
       Adr.Tel      & Seperator , Wo );
END Gebe_Adresse_Aus;

PROCEDURE Zeige_Adresse() =
BEGIN
  IO.Put("\n");
  Gebe_Adresse_Aus(Meine_Adressen[Akt_Adresse], " ");
  IO.Put("\n");
END Zeige_Adresse;

PROCEDURE Schreibe_Adressen() =
VAR Adr_Datei_schreiben: Wr.T;
BEGIN
  Adr_Datei_schreiben := IO.OpenWrite
(Adr_Datei_Name);
  FOR I:= 1 TO Letzte_Adresse DO
    Gebe_Adresse_Aus (Meine_Adressen[I],
                      "\n",
                      Adr_Datei_schreiben);
    IO.Put("\n",Adr_Datei_schreiben);
  END;
  Wr.Close (Adr_Datei_schreiben);
END Schreibe_Adressen;
```

Entwicklungsschritt

- Da Herr X mit der Verwaltung der Adressen seiner **Geschäftspartner** sehr zufrieden ist, möchte er die Adressen seiner **privaten Bekannten** ebenfalls mit seinem Programm verwalten.

Wie kann er dies einfach erreichen?

Lösungsalternativen

- Er kann den Programmcode ***duplizieren*** und eine neue Datei für die Adressen seiner privaten Bekannten verwenden.
- Er kann sein Programm mit der Adressen-Datei ***parametrisieren***.
- Er kann seine Datenkapsel zu einem ***Abstrakten Datentyp*** erweitern.

Vom Objektmodul zum ADT

- **Gekapselte Daten werden zu einem Typ zusammengefaßt**
  - typischerweise ein Record
- **Die Operationen des Objektmoduls werden um einen zusätzlichen Parameter erweitert.**
  - Dieser Parameter ist eine ADT-Exemplar und steht per Konvention an erster Stelle.
- **In den Operations-Rümpfen muß der neue Parameter entsprechend verwendet werden.**
- **Exemplare des ADTs werden im Hauptprogramm erzeugt und die Operationsaufrufe entsprechend angepaßt.**

Der ADT Adressen_Liste-1

Name des ADT

```
INTERFACE Adressen_Liste_ADT;
(* Realisiert einen abstrakten Datentyp Adressen_Liste*)

TYPE Adressen_Liste <: REFANY;

PROCEDURE Neue_Adresse         (VAR Liste : Adressen_Liste);
PROCEDURE Modifiziere_Adresse  (VAR Liste : Adressen_Liste);
PROCEDURE Suche_Adresse        (    Liste : Adressen_Liste);
PROCEDURE Zeige_Adresse        (    Liste : Adressen_Liste);
PROCEDURE Zuruecksetzen        (VAR Liste : Adressen_Liste);
PROCEDURE Blaettern            (VAR Liste : Adressen_Liste);

PROCEDURE Lese_Adressen        (VAR Liste : Adressen_Liste;
                                    Adr_Datei_Name : TEXT);
PROCEDURE Schreibe_Adressen    (    Liste : Adressen_Liste;
                                    Adr_Datei_Name : TEXT);

END Adressen_Liste_ADT.
```

Deklaration der Operationen

3. Hauptprogramm - 1

```
MODULE Adress_Verwaltung EXPORTS Main;
(*Testprogramm fuer den Abstrakten Datentypen.*)

IMPORT SIO, Text;
IMPORT Adressen_Liste_ADT AS AL;

CONST Private_Adr    = "priv.adr";
      Geschaefts_Adr = "gesch.adr";

VAR P_Adressen : AL.Adressen_Liste;
    G_Adressen : AL.Adressen_Liste;

…
PROCEDURE Interaktion()=
…

(*-------------------------------------------------------*)
BEGIN (* des Hauptprogramms*)

  AL.Lese_Adressen(P_Adressen, Private_Adr);
  AL.Lese_Adressen(G_Adressen, Geschaefts_Adr);

  Interaktion();

  AL.Schreibe_Adressen(P_Adressen, Private_Adr);
  AL.Schreibe_Adressen(G_Adressen, Geschaefts_Adr);

END Adress_Verwaltung.
```

Import des ADTs

Deklaration von Variablen des ADTs

Verwenden der Operationen des ADTs

3. Hauptprogramm - 2

```
PROCEDURE Zeige_Auswahl_Menue (): CHAR =
BEGIN
  SIO.PutLine("Private Adressen    (a)");
  SIO.PutLine("Geschaefts Adressen (b)");
  SIO.PutLine("BEENDEN             (sonst)");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Auswahl_Menue;

(*-------------------------  ---------------------------*)

PROCEDURE Interaktion()=
  VAR Auswahl : CHAR := 'r';
BEGIN
  REPEAT
    Auswahl:= Zeige_Auswahl_Menue();
    CASE Auswahl OF
      'a' => Menue(P_Adressen, "PRIVATE-ADRESSEN");     |
      'b' => Menue(G_Adressen, "GESCHAEFTS-ADRESSEN");
    ELSE
      SIO.PutLine ("ENDE DES PROGRAMMS!");
    END;
  UNTIL (Auswahl # 'a') AND (Auswahl # 'b');
END Interaktion;
```

3. Hauptprogramm - 3

Objekt- und Klassenbasierte Lösung

```
PROCEDURE Zeige_Adress_Menue(): CHAR =
BEGIN
  SIO.PutLine("--------------------");
  SIO.PutLine("Zuruecksetzen (r)");
  SIO.PutLine("Blaettern     (b)");
  SIO.PutLine("Suchen        (s)");
  SIO.PutLine("Aendern       (a)");
  SIO.PutLine("Eintragen     (e)");
  SIO.PutLine("Zeigen        (z)");
  SIO.PutLine("BEENDEN       (sonst)");
  SIO.PutLine("--------------------");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Adress_Menue;

(*-------------------------------------------------*)
PROCEDURE Menue(VAR Adress_Liste : AL.Adressen_Liste;
                    Ueberschrift : TEXT) =
VAR Wahl : CHAR;
    Kommandos := SET OF CHAR {'r','b','s','a','z','e'};
BEGIN
  REPEAT
    SIO.PutLine("--------------------");
    SIO.PutLine(Ueberschrift);

    Wahl := Zeige_Adress_Menue();

    IF Wahl IN Kommandos THEN
      CASE Wahl OF
        'r'  => AL.Zuruecksetzen       (Adress_Liste);
        'b'  => AL.Blaettern           (Adress_Liste);
        's'  => AL.Suche_Adresse       (Adress_Liste);
        'a'  => AL.Modifiziere_Adresse(Adress_Liste);
        'z'  => AL.Zeige_Adresse       (Adress_Liste);
        'e'  => AL.Neue_Adresse        (Adress_Liste);
      END;
    ELSE
      SIO.PutLine ("<---");
    END;
  UNTIL NOT Wahl IN Kommandos;
END Menue;
```

Verwenden der Operationen des ADTs

Blick ins Innere des ADT - 1

```
MODULE Adressen_Liste_ADT;

IMPORT IO, Rd, Wr, Text;

TYPE Adresse = RECORD
       Name : TEXT;
       Vorname : TEXT;
       PLZ : TEXT;
       Ort : TEXT;
       Tel : TEXT;
     END;

CONST Max = 30;

TYPE
  Listen_Laenge = [1..Max];

  AdressenSpeicher = RECORD
    Letzte_Adresse : Listen_Laenge := 1;
    Akt_Adresse    : Listen_Laenge := 1;
    Adressen       : ARRAY Listen_Laenge OF Adresse;
  END;

  REVEAL
    Adressen_Liste = BRANDED REF AdressenSpeicher;
```

Definition der Struktur des ADTs

Blick ins Innere des ADT - 2

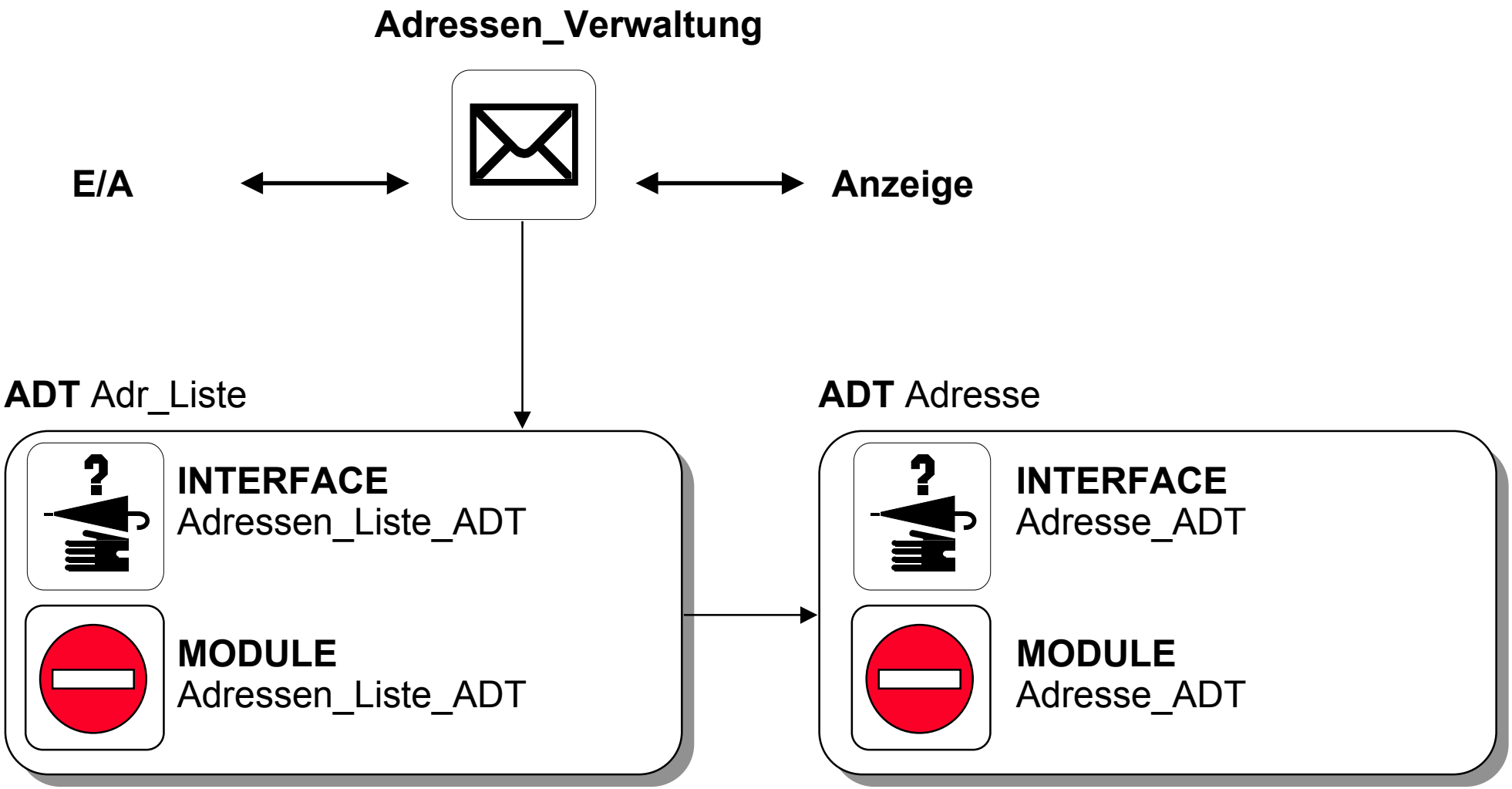
```
PROCEDURE Lese_Adressen(VAR Liste: Adressen_Liste;
                        Adr_Datei_Name : TEXT) =
VAR Zeile : TEXT;
    Adr_Datei_lesen : Rd.T;
BEGIN
  Liste:=NEW(Adressen_Liste);
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  Liste.Letzte_Adresse := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Name := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Vorname := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].PLZ := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Ort := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Tel := Zeile;

    Zeile:=IO.GetLine(Adr_Datei_lesen);    (* Leerzeile *)
    INC(Liste.Letzte_Adresse);
  END;
  Liste.Akt_Adresse := 1;
  DEC(Liste.Letzte_Adresse);
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;
```

Entwicklungsschritt

Fasziniert vom Konzept der ADTen beschließt Herr X
den Typ `Adresse` aus dem Paket `Adressen_ADT` zu **extrahieren**
und einen **eigenen ADT `Adresse`** zu bilden

- Damit wird das Programm besser erweiterbar und wartbar!
- `Adresse` Eintragstyp
- `Adressen_ADT` Kollektion

Architektur mit zwei ADTen



Schnittstelle ADT Adresse

```
INTERFACE Adresse_ADT;
(* realisiert den ADT Adresse mit folgenden Operationen*)

IMPORT Rd, Wr;

TYPE Adresse <: REFANY;

PROCEDURE Lese_Eine_Adresse(VAR Datei: Rd.T) : Adresse;
(* liest eine Adresse von der angegebenen Datei und
   liefert diese zurueck *)

PROCEDURE Schreibe_Eine_Adresse(Adr: Adresse; VAR Datei: Wr.T);
(* schreibt die Adresse Adr auf Datei *)

PROCEDURE Zeige_Eine_Adresse(Adr: Adresse);
(*  zeigt die Adresse Adr am Bildschirm an *)

PROCEDURE Lese_Eine_Neue_Adresse():Adresse;
(* Fordert den Benutzer auf, Angaben fuer ein neue Adresse
   einzugeben und liefert die eingegebene Adresse zurueck. *)

PROCEDURE Modifiziere_Eine_Adresse(VAR Adr: Adresse);
(* Fordert den Benutzer auf, neue Angaben fuer die Adresse Adr
   einzugeben.*)

PROCEDURE Liefere_Suchattribut():TEXT;
(* Liefert das Suchattribute fuer Adressen.*)

PROCEDURE Hat_Namen (Adr: Adresse;  Name: TEXT):BOOLEAN;
(* Testet, ob die Adresse Adr den angegebene Namen hat.*)

END Adresse_ADT.
```

Name des ADT

Deklaration der Operationen

Schnittstelle Adressen_Liste

Name des ADT

```
INTERFACE Adressen_Liste_ADT;
(* Realisiert einen abstrakten Datentyp Adressen_Liste*)

TYPE Adressen_Liste <: REFANY;

PROCEDURE Neue_Adresse         (VAR Liste : Adressen_Liste);
PROCEDURE Modifiziere_Adresse  (VAR Liste : Adressen_Liste);
PROCEDURE Suche_Adresse        (    Liste : Adressen_Liste);
PROCEDURE Zeige_Adresse        (    Liste : Adressen_Liste);
PROCEDURE Zuruecksetzen        (VAR Liste : Adressen_Liste);
PROCEDURE Blaettern            (VAR Liste : Adressen_Liste);

PROCEDURE Lese_Adressen        (VAR Liste : Adressen_Liste;
                                    Adr_Datei_Name : TEXT);
PROCEDURE Schreibe_Adressen    (    Liste : Adressen_Liste;
                                    Adr_Datei_Name : TEXT);

END Adressen_Liste_ADT.
```

Deklaration der Operationen

Rumpf ADT Adressen_Liste - 1

```
MODULE Adressen_Liste_ADT;

IMPORT IO, Rd, Wr;
IMPORT Adresse_ADT AS Adr;

CONST Max = 30;

TYPE Listen_Laenge = [1..Max];
     AdressenSpeicher = RECORD
       Letzte_Adresse : Listen_Laenge := 1;
       Akt_Adresse    : Listen_Laenge := 1;
       Adressen       : ARRAY Listen_Laenge OF Adr.Adresse;
     END;

     REVEAL Adressen_Liste = BRANDED REF AdressenSpeicher;

(*-------------------------------------------------------*)

PROCEDURE Zeige_Adresse(Liste: Adressen_Liste) =
BEGIN
  IO.Put("\n");
  Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
  IO.Put("\n");
END Zeige_Adresse;

PROCEDURE Zuruecksetzen(VAR Liste: Adressen_Liste) =
BEGIN
  Liste.Akt_Adresse := 1;
  Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
END Zuruecksetzen;

PROCEDURE Blaettern(VAR Liste: Adressen_Liste) =
BEGIN
  IF Liste.Akt_Adresse < Liste.Letzte_Adresse THEN
    INC(Liste.Akt_Adresse);
    Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
  ELSE
    IO.Put("Kein Datensatz mehr vorhanden!\n");
  END;
END Blaettern;
```

Import ADT Adresse

Rumpf ADT Adressen_Liste - 2

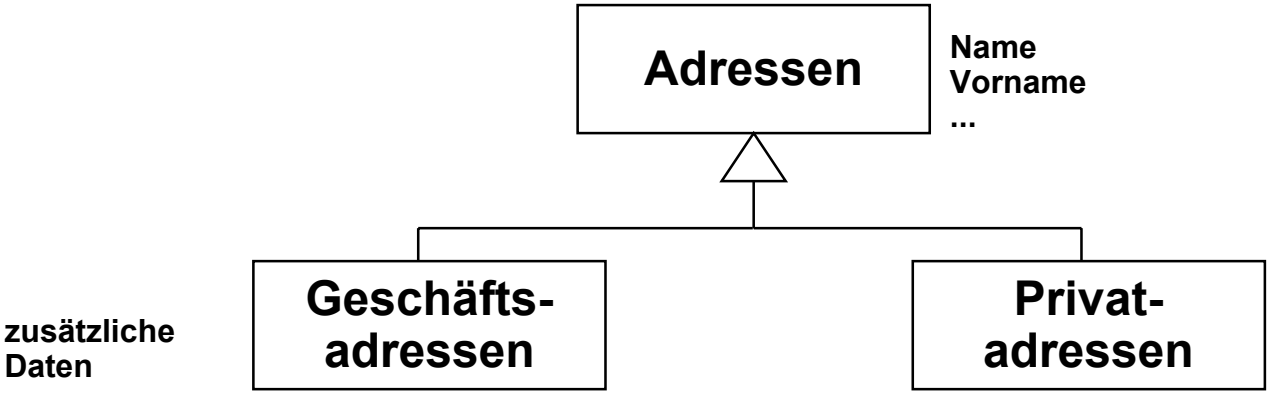
```
PROCEDURE Suche_Adresse(Liste: Adressen_Liste) =
VAR Suchbegriff : TEXT;
       Gefunden : BOOLEAN := FALSE;
BEGIN
  IO.Put("Es kann nur nach dem Attribut: " &
          Adr.Liefere_Suchattribut()      &
          " gesucht werden!\n");
  IO.Put ("Attributwert > ");
  Suchbegriff := IO.GetLine ();
  Liste.Akt_Adresse := 1;
  WHILE (NOT Gefunden) AND
        (Liste.Akt_Adresse <= Liste.Letzte_Adresse) DO
    IF Adr.Hat_Namen(Liste.Adressen[Liste.Akt_Adresse],
                     Suchbegriff) THEN
      Gefunden := TRUE;
      Zeige_Adresse(Liste);
    ELSE
      INC(Liste.Akt_Adresse);
    END;
  END;
  IF NOT Gefunden THEN
    IO.Put("Der spezifizierte Datensatz
            ist nicht vorhanden!\n");
  END;
END Suche_Adresse;

(*-------------------------------------------------------*)

PROCEDURE Lese_Adressen(VAR Liste: Adressen_Liste;
                            Adr_Datei_Name : TEXT) =
VAR Adr_Datei_lesen : Rd.T;
BEGIN
  Liste:= NEW(Adressen_Liste);
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  Liste.Letzte_Adresse := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Liste.Adressen[Liste.Letzte_Adresse]:=
                     Adr.Lese_Eine_Adresse(Adr_Datei_lesen);
    INC(Liste.Letzte_Adresse);
  END;
  DEC(Liste.Letzte_Adresse);
  Liste.Akt_Adresse := 1;
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;
```

```
PROCEDURE Schreibe_Adressen( Liste: Adressen_Liste;
                             Adr_Datei_Name: TEXT) =
VAR Adr_Datei_schreiben:Wr.T;
BEGIN
  Adr_Datei_schreiben := IO.OpenWrite (Adr_Datei_Name);
  FOR I:= 1 TO Liste.Letzte_Adresse DO
    Adr.Schreibe_Eine_Adresse(Liste.Adressen[I,
                              Adr_Datei_schreiben);
  END;
  Wr.Close (Adr_Datei_schreiben);
END Schreibe_Adressen;
(*-------------------------------------------------------*)

PROCEDURE Neue_Adresse(VAR Liste: Adressen_Liste) =
BEGIN
  IF Liste.Letzte_Adresse = Max THEN
    IO.Put("Es ist leider keine Platz mehr!\n");
  ELSE
    INC(Liste.Letzte_Adresse);
    Liste.Akt_Adresse := Liste.Letzte_Adresse;
    Liste.Adressen[Liste.Akt_Adresse] :=
                                  Adr.Lese_Eine_Neue_Adresse();
    IO.Put("Eingegebene Adresse :\n");
    Zeige_Adresse(Liste);
  END;
END Neue_Adresse;

(*-------------------------------------------------------*)

PROCEDURE Modifiziere_Adresse(VAR Liste:Adressen_Liste) =
BEGIN
  IO.Put("Adresse :");
  Zeige_Adresse(Liste);
  Adr.Modifiziere_Eine_Adresse
                           (Liste.Adressen[Liste.Akt_Adresse]);
  IO.Put("\nGeaenderte Adresse :");
  Zeige_Adresse(Liste);
END Modifiziere_Adresse;

BEGIN
END Adressen_Liste_ADT.
```

(Teilweise) Objektorientierte Lösung

- **Über Personen werden nicht nur Name, Vorname etc. gehalten, sondern auch weitere Daten**
- **Diese weiteren Daten können von Personengruppe zu Personengruppe variieren**
  - Geschäftsfreund: Handy-Nr.
  - private Freunde: Geburtsdatum, Vorlieben
- **Es macht keinen Sinn die gemeinsamen Teile mehrfach und gleich festzulegen**

Vererbungshierachie für Adressen



Was haben wir gelernt?

- **kleines Anwendungssystem in verschiedenen Ausbaustufen, Änderungsaufwand enorm, gründliche Überlegung vor der Realisierung**
- **schlechteste Lösung: offene Datentypen, Veränderung direkt an den Stellen der Anwendung**
- **prozedurale Lösung: alle wesentliche Funktionalität in Prozeduren, schlechter Entwurf: Prozeduren dienen unterschiedlichen Zwecken: Datenzugriff, E/A-Handhabung, Verwaltung des Ablaufs; keine Garantie, daß auf Daten nicht direkt zugegriffen werden wird**
- **objektbasierte Lösung: ein Datenbestand; Zugriffsoperationen sauber (weitere Prozeduren sollten geordnet nach Zweck in andere Module eingelagert werden: E/A, Verwaltung, Initialisierung/Abschluß**
- **ADT-Lösung: mehrere Datenbestände (ohne Quelltextvervielfältigung), Rest analog, man hat zwei ADT-Entscheidungen Adresse (Eintrag), Adreßliste (Kollektion): ADO-Baustein bei einer Kollektion, ADT-Baustein bei mehreren**
- **objektorientierte Lösung:**
  - Wirft man alles weg und macht alles zu Klassen (typische OO-Denkweise)?
  - Konzentriert man OO-?? auf die Stellen, wo Ähnlichkeiten zwischen ADTs gehandhabt werden sollen? Haben letzteres gemacht.
  - Modellierung der Ähnlichkeiten, Verschiedenheiten der Einträge durch OO (Übungsaufgaben)

Beispiel einer Programmentwicklung

- **Aufgabenstellung**
- **Prozedurale Lösung**
- **Objekt- und klassenbasierte Lösung**
- **Lösung mit Objektorientierung**

Aufgaben-stellung

Informell festgehaltene Anforderungen

- **Herr X**, ein engagierter Jungunternehmer, hat viele geschäftliche Verbindungen.

  Da er sich die Adressen seiner Geschäftspartner nicht merken kann und die Sammlung der Visitenkarten im Geldbeutel bereits sehr unübersichtlich geworden ist, wünscht sich Herr X eine Möglichkeit, um die **Adressen** seiner Geschäftspartner zu verwalten.

- Er möchte folgende Angaben für jede Adresse vorfinden:

  Name, **Vorname**, **PLZ**, **Ort** und **Telefon-Nummer**.

Aufgaben-stellung

Verschiedene Lösungen

- **Texteditor: keine Lösung**
  - Suchen und Sortieren ist eher mühsam.
  - Daten können leicht unbeabsichtigt zerstört oder verfälscht werden (cut & paste)

- **Adreßverwaltungsprogramm:**
  - Eingabefehler abprüfen
  - Sicherheit der Dateien gewährleistet

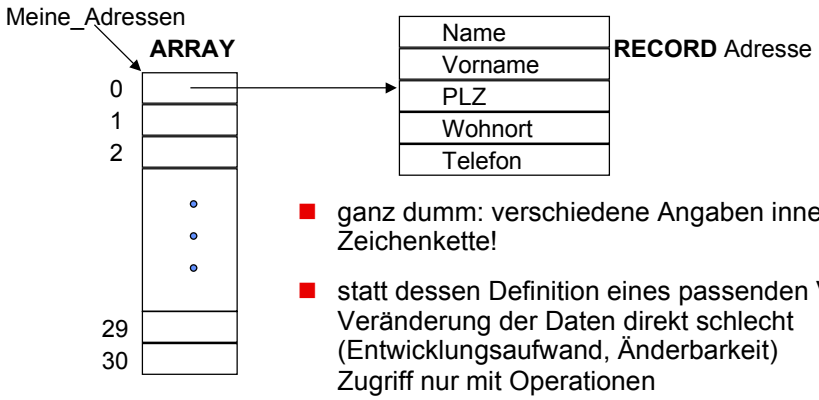
Diskutieren der Lösungsansätze von schlecht bis gut

Prozedurale Lösung

Offene Datenstrukturen

Eine Adresse

Kugler	**Hans**	**8021**	**Zürich**	**01-4330570**	. . .



- ganz dumm: verschiedene Angaben innerhalb einer Zeichenkette!
- statt dessen Definition eines passenden Verbundtyps
  Veränderung der Daten direkt schlecht (Entwicklungsaufwand, Änderbarkeit)
  Zugriff nur mit Operationen
- Verbundtypdefinition und Operationen in ein Modul zusammengefaßt

Prozedurale Lösung

Modul Adress_Operationen-1

```
INTERFACE Adress_Operationen;

IMPORT Rd,Wr;

TYPE Adresse = RECORD
       Name : TEXT;
       Vorname : TEXT;
       PLZ : TEXT;
       Ort : TEXT;
       Tel : TEXT;
     END;

CONST Max_Adr = 30;

TYPE Listen_Laenge = [1 .. Max_Adr];
     Adressen_Liste = ARRAY Listen_Laenge OF Adresse;

VAR  Meine_Adressen : Adressen_Liste;  (* Datensaetze *)

(*-------------------------------------------*)

PROCEDURE Setze_Name(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Vorname(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_PLZ(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Ort(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Setze_Tel(I: Listen_Laenge; Wert : TEXT);

PROCEDURE Hat_Namen(I: Listen_Laenge; Wert : TEXT): BOOLEAN;

PROCEDURE Zeige_Adresse(I: Listen_Laenge);

PROCEDURE Lese_Adresse(I: Listen_Laenge; File : Rd.T);

PROCEDURE Schreibe_Adresse(I: Listen_Laenge; File : Wr.T);

END Adress_Operationen.
```

Adressen-Typ

Zentrale Variable

Deklaration der Operationen

Prozedurale Lösung

Hauptprogramm

```
MODULE Adressen_Verwaltung EXPORTS Main;
IMPORT Adress_Operationen AS AO;

BEGIN  (* -- des Hauptprogramms*)
  Adress_Datei_lesen := IO.OpenRead(Gesch_Adr);
  I := 0;
  WHILE NOT SIO.EOF(Adress_Datei_lesen) DO
    I := I + 1;
    AO.Lese_Adresse (I, Adress_Datei_lesen);
  END;
  Anz := I;
  Rd.Close(Adress_Datei_lesen);
(*-------------------------------------------*)
  SIO.PutText ("Den zu suchenden Namen bitte > ");
  Name:=SIO.GetLine ();
  I := 1;
  WHILE (NOT Gefunden) AND (I <= Anz) DO   (* -- Daten suchen*)
    Gefunden := AO.Hat_Namen(I, Name);
    I := I + 1;
  END;
(*-------------------------------------------*)
  IF Gefunden THEN
    AO.Zeige_Adresse(I-1);
    SIO.PutLine("Die neuen Angaben bytte!");
    SIO.PutText("PLZ > ");                  (* PLZ *)
    PLZ := SIO.GetLine ();
    AO.Setze_PLZ(I-1, PLZ);

    SIO.PutText("Ort > ");
    Ort := SIO.GetLine ();                  (* Wohnort*)
    AO.Setze_Ort(I-1, Ort);

    SIO.PutText("Telefon-Nr. > "); (* Telefon*)
    Tel := SIO.GetLine ();
    AO.Setze_Tel(I-1, Tel);

    AO.Zeige_Adresse(I-1);                  (* Daten anzeigen*)
  ELSE
    SIO.PutLine("Spezifizierte Adresse nicht vorhanden!");
  END;
(*-------------------------------------------*)
  Adress_Datei_schreiben := IO.OpenWrite (Gesch_Adr);
  FOR J := 1 TO Anz DO
    AO.Schreibe_Adresse(J, Adress_Datei_schreiben);
  END;
  Wr.Close (Adress_Datei_schreiben);

END Adressen_Verwaltung.
```

Import der Operationen

Aufruf der Operationen

Diskussion 1

- **Es ist möglich, die definierten Operationen zu umgehen**
  - Die Operationen müssen im Hauptprogramm nicht verwendet werden.
- **Die direkte Manipulation der Variable `Meine_Adressen` ist im Hauptprogramm immer noch möglich!**

```
  SIO.PutLine("Die neuen Angaben
bitte!");
  SIO.PutText("PLZ > ");          (*
PLZ *)
  PLZ := SIO.GetLine ();
  AO.Setze_PLZ(I-1, PLZ);

  Meine_Adressen[I-1].PLZ :=
'abcd';
```

Aufbau der Datenstruktur ist bekannt. Sie kann direkt manipuliert werden!

Diskussion 2

- **Die einzelnen Verwaltungsfunktionen Suchen, Einfügen, Löschen sind im Änderungsteil des Hauptprogramms fest „eingebacken“**

  Besser Prozeduren hierfür deklarieren, die dann noch externe Prüfungen übernehmen und die die Operationen der Datenspeicherung aufrufen
- **Die E/A Handhabung sollte aus den Verwaltungsoperationen herausgezogen werden: kann sich leicht ändern**

Neue Anforderungen

- **Herr X hat einen Software-Engineering-Kurs gehört.**
  - Dort hat er gelernt, daß das Geheimnisprinzip angewendet werden muß, wenn Informationen geschützt werden sollen
  - Er entscheidet sich, die Variable `Meine_Adressen` durch ein Objektmodul (Datenkapsel) zu schützen.
- **Gleichzeitig soll sein Programm um folgende Funktionalität erweitert werden: Es soll**
  - ein Datensatz eingegeben werden können
  - nach einem Datensatz gesucht werden können (Name)
  - ein Datensatz modifiziert werden können
  - ein Datensatz angezeigt werden können
  - auf den Anfang der Sammlung positioniert werden können
  - geblättert werden können

Objektmodul für Adressen

```
INTERFACE Adress_OM;
(* Dieses Modul realisiert ein Objektmodul, mit dem maximal 30 Adressen verwaltet
werden koennen. Bevor eine Operation des Objektmoduls ausgefuehrt werden kann, muss die
Prozedur Lese_Adressen ausgefuehrt worden sein.*)

PROCEDURE Neue_Adresse();
(* Fordert den Benutzer auf, eine neue Adresse einzugeben*)

PROCEDURE Modifiziere_Adresse();
(*Fordert den Benutzer auf, Angaben fuer die aktuelle
  Adresse einzugeben.*)

PROCEDURE Suche_Adresse();
(*Fordert den Benutzer auf, einen Namen einzugeben. Ist eine Adresse mit diesem Namen
vorhanden, wird diese angezeigt. Diese Adresse wird die aktuell selektierte Adresse.*)

PROCEDURE Lese_Adressen();
(*Initialisiert das Modul und liest alle Adressen von Datei ein. Die erste Adresse ist
die aktuell selektierte Adr.*)

PROCEDURE Schreibe_Adressen();
(*Schreibt die Adressen zurueck auf Datei.*)

PROCEDURE Zeige_Adresse();
(*Zeigt die aktuelle Adresse auf dem Bildschirm.*)

PROCEDURE Zuruecksetzen();
(*Es wird auf den Anfang der Adressensammlung positioniert. Die erste Adresse wird die
aktuelle Adresse.*)

PROCEDURE Blaettern();
(*Zeigt ausgehend von der aktuellen Adresse die naechste Adresse am Bildschirm.*)

END Adress_OM.
```

Hauptprogramm 2

```
MODULE Adress_Verwaltung EXPORTS Main;
IMPORT Adress_OM AS Adr;
(*-------------------------------------------*)
PROCEDURE Zeige_Adress_Menue(): CHAR =
BEGIN
  SIO.PutLine("--------------------");
  SIO.PutLine("Zuruecksetzen (r)");
  SIO.PutLine("Blaettern     (b)");
  SIO.PutLine("Suchen        (s)");
  SIO.PutLine("Aendern       (a)");
  SIO.PutLine("Eintragen     (e)");
  SIO.PutLine("Zeigen        (z)");
  SIO.PutLine("BEENDEN       (sonst)");
  SIO.PutLine("--------------------");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Adress_Menue;

PROCEDURE Menu() =
VAR Wahl : CHAR := 'r';
    Kommandos := SET OF CHAR {'r','b','s','a','z','e'};
BEGIN
  REPEAT
    Wahl := Zeige_Adress_Menue();
    IF Wahl IN Kommandos THEN
      CASE Wahl OF
        'r'  => Adr.Zuruecksetzen();
      | 'b'  => Adr.Blaettern();
      | 's'  => Adr.Suche_Adresse();
      | 'a'  => Adr.Modifiziere_Adresse();
      | 'z'  => Adr.Zeige_Adresse();
      | 'e'  => Adr.Neue_Adresse();
      END;
    ELSE
      SIO.PutLine ("ENDE DES PROGRAMMS");
    END;
  UNTIL NOT Wahl IN Kommandos;
END Menu;
(*-------------------------------------------*)
BEGIN (* des Hauptprogramms*)
  Adr.Lese_Adressen();
  Menu();
  Adr.Schreibe_Adressen();
END Adress_Verwaltung.
```


Impl. des Objektmoduls - 1

```
MODULE Adress_OM;

IMPORT IO, Rd, Wr,Text;

TYPE Adresse = RECORD
                 Name     : TEXT;
                 Vorname  : TEXT;
                 PLZ      : TEXT;
                 Ort      : TEXT;
                 Tel      : TEXT;
               END;

CONST Adr_Datei_Name = "gesch.adr";
      Max = 30;

TYPE
  Listen_Laenge = [1..Max];
  Adressen_Liste = ARRAY Listen_Laenge OF Adresse;

(*-------------------------------------------*)
(* gekapselte Daten*)

VAR Meine_Adressen : Adressen_Liste;
    Letzte_Adresse : Listen_Laenge := 1;
    Akt_Adresse    : Listen_Laenge := 1;
```

Impl. des Objektmoduls - 2

```
PROCEDURE Lese_Adressen() =
VAR I : Listen_Laenge;
    Zeile : TEXT;
    Adr_Datei_lesen : Rd.T;
BEGIN
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  I := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Name := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Vorname := Zeile;
    ...
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Ort := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Meine_Adressen[I].Tel := Zeile;

    Zeile:=IO.GetLine(Adr_Datei_lesen); (* Leerzeile*)
    INC(I);
  END;
  Letzte_Adresse := I - 1; Akt_Adresse := 1;
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;

PROCEDURE Zuruecksetzen() =
BEGIN
  Akt_Adresse := 1; Zeige_Adresse();
END Zuruecksetzen;

PROCEDURE Blaettern() =
BEGIN
  IF Akt_Adresse < Letzte_Adresse THEN
    INC(Akt_Adresse);
    Zeige_Adresse();
  ELSE
    IO.Put("Kein Datensatz mehr vorhanden!\n");
  END;
END Blaettern;
```


Impl. des Objektmoduls - 3

```
PROCEDURE Gebe_Adresse_Aus (Adr: Adresse;
                            Seperator :TEXT;
                            Wo : Wr.T := NIL) =
BEGIN
IO.Put (Adr.Name     & Seperator &
        Adr.Vorname  & Seperator &
        Adr.PLZ      & Seperator &
        Adr.Ort      & Seperator &
        Adr.Tel      & Seperator , Wo );
END Gebe_Adresse_Aus;

PROCEDURE Zeige_Adresse() =
BEGIN
  IO.Put("\n");
  Gebe_Adresse_Aus(Meine_Adressen[Akt_Adresse], " ");
  IO.Put("\n");
END Zeige_Adresse;

PROCEDURE Schreibe_Adressen() =
VAR Adr_Datei_schreiben: Wr.T;
BEGIN
  Adr_Datei_schreiben := IO.OpenWrite
(Adr_Datei_Name);
  FOR I:= 1 TO Letzte_Adresse DO
    Gebe_Adresse_Aus (Meine_Adressen[I],
                      "\n",
                      Adr_Datei_schreiben);
    IO.Put("\n",Adr_Datei_schreiben);
  END;
  Wr.Close (Adr_Datei_schreiben);
END Schreibe_Adressen;
```

Entwicklungsschritt

- Da Herr X mit der Verwaltung der Adressen seiner **Geschäftspartner** sehr zufrieden ist, möchte er die Adressen seiner **privaten Bekannten** ebenfalls mit seinem Programm verwalten.

Wie kann er dies einfach erreichen?

- **Lösungsalternativen**
  - Er kann den Programmcode ***duplizieren*** und eine neue Datei für die Adressen seiner privaten Bekannten verwenden.
  - Er kann sein Programm mit der Adressen-Datei ***parametrisieren***.
  - Er kann seine Datenkapsel zu einem ***Abstrakten Datentyp*** erweitern.

Vom Objektmodul zum ADT

- **Gekapselte Daten werden zu einem Typ zusammengefaßt**
  - typischerweise ein Record
- **Die Operationen des Objektmoduls werden um einen zusätzlichen Parameter erweitert.**
  - Dieser Parameter ist eine ADT-Exemplar und steht per Konvention an erster Stelle.
- **In den Operations-Rümpfen muß der neue Parameter entsprechend verwendet werden.**
- **Exemplare des ADTs werden im Hauptprogramm erzeugt und die Operationsaufrufe entsprechend angepaßt.**

Der ADT Adressen_Liste-1

Name des ADT

```
INTERFACE Adressen_Liste_ADT;
(* Realisiert einen abstrakten Datentyp Adressen_Liste*)

TYPE Adressen_Liste <: REFANY;

PROCEDURE Neue_Adresse        (VAR Liste : Adressen_Liste);
PROCEDURE Modifiziere_Adresse (VAR Liste : Adressen_Liste);
PROCEDURE Suche_Adresse       (    Liste : Adressen_Liste);
PROCEDURE Zeige_Adresse       (    Liste : Adressen_Liste);
PROCEDURE Zuruecksetzen       (VAR Liste : Adressen_Liste);
PROCEDURE Blaettern           (VAR Liste : Adressen_Liste);

PROCEDURE Lese_Adressen       (VAR Liste : Adressen_Liste;
                                   Adr_Datei_Name : TEXT);
PROCEDURE Schreibe_Adressen   (    Liste : Adressen_Liste;
                                   Adr_Datei_Name : TEXT);

END Adressen_Liste_ADT.
```

Deklaration der Operationen

3. Hauptprogramm - 1

```
MODULE Adress_Verwaltung EXPORTS Main;
(*Testprogramm fuer den Abstrakten Datentypen.*)

IMPORT SIO, Text;
IMPORT Adressen_Liste_ADT AS AL;

CONST Private_Adr    = "priv.adr";
      Geschaefts_Adr = "gesch.adr";

VAR P_Adressen : AL.Adressen_Liste;
    G_Adressen : AL.Adressen_Liste;

…
PROCEDURE Interaktion()=
…

(*-------------------------------------------------------*)
BEGIN (* des Hauptprogramms*)

  AL.Lese_Adressen(P_Adressen, Private_Adr);
  AL.Lese_Adressen(G_Adressen, Geschaefts_Adr);

  Interaktion();

  AL.Schreibe_Adressen(P_Adressen, Private_Adr);
  AL.Schreibe_Adressen(G_Adressen, Geschaefts_Adr);

END Adress_Verwaltung.
```

Import des ADTs

Deklaration von Variablen des ADTs

Verwenden der Operationen des ADTs

3. Hauptprogramm - 2

```
PROCEDURE Zeige_Auswahl_Menue (): CHAR =
BEGIN
  SIO.PutLine("Private Adressen    (a)");
  SIO.PutLine("Geschaefts Adressen (b)");
  SIO.PutLine("BEENDEN             (sonst)");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Auswahl_Menue;

(*-------------------------  --------------------------*)

PROCEDURE Interaktion()=
  VAR Auswahl : CHAR := 'r';
BEGIN
  REPEAT
    Auswahl:= Zeige_Auswahl_Menue();
    CASE Auswahl OF
      'a' => Menue(P_Adressen, "PRIVATE-ADRESSEN");
      'b' => Menue(G_Adressen, "GESCHAEFTS-ADRESSEN");
    ELSE
      SIO.PutLine ("ENDE DES PROGRAMMS!");
    END;
  UNTIL (Auswahl # 'a') AND (Auswahl # 'b');
END Interaktion;
```


3. Hauptprogramm - 3

```
PROCEDURE Zeige_Adress_Menue(): CHAR =
BEGIN
  SIO.PutLine("-------------------------");
  SIO.PutLine("Zuruecksetzen (r)");
  SIO.PutLine("Blaettern     (b)");
  SIO.PutLine("Suchen        (s)");
  SIO.PutLine("Aendern       (a)");
  SIO.PutLine("Eintragen     (e)");
  SIO.PutLine("Zeigen        (z)");
  SIO.PutLine("BEENDEN       (sonst)");
  SIO.PutLine("-------------------------");
  SIO.PutText("AUSWAHL > ");
  RETURN Text.GetChar(SIO.GetLine(), 0);
END Zeige_Adress_Menue;

(*-----------------------------------------*)
PROCEDURE Menue(VAR Adress_Liste : AL.Adressen_Liste;
                    Ueberschrift : TEXT) =
  VAR Wahl : CHAR;
    Kommandos := SET OF CHAR {'r','b','s','a','z','e'};
BEGIN
  REPEAT
    SIO.PutLine("-------------------------");
    SIO.PutLine(Ueberschrift);

    Wahl := Zeige_Adress_Menue();

    IF Wahl IN Kommandos THEN
      CASE Wahl OF
        'r' => AL.Zuruecksetzen          (Adress_Liste);
      | 'b' => AL.Blaettern              (Adress_Liste);
      | 's' => AL.Suche_Adresse          (Adress_Liste);
      | 'a' => AL.Modifiziere_Adresse(Adress_Liste);
      | 'z' => AL.Zeige_Adresse          (Adress_Liste);
      | 'e' => AL.Neue_Adresse           (Adress_Liste);
      END;
    ELSE
      SIO.PutLine ("<---");
    END;
  UNTIL NOT Wahl IN Kommandos;
END Menue;
```

Verwenden der Operationen des ADTs

Blick ins Innere des ADT - 1

```
MODULE Adressen_Liste_ADT;

IMPORT IO, Rd, Wr, Text;

TYPE Adresse = RECORD
       Name : TEXT;
       Vorname : TEXT;
       PLZ : TEXT;
       Ort : TEXT;
       Tel : TEXT;
     END;

CONST Max = 30;

TYPE
  Listen_Laenge = [1..Max];

  AdressenSpeicher = RECORD
    Letzte_Adresse : Listen_Laenge := 1;
    Akt_Adresse    : Listen_Laenge := 1;
    Adressen       : ARRAY Listen_Laenge OF Adresse;
  END;

  REVEAL
    Adressen_Liste = BRANDED REF AdressenSpeicher;
```

Definition der Struktur des ADTs

Blick ins Innere des ADT - 2

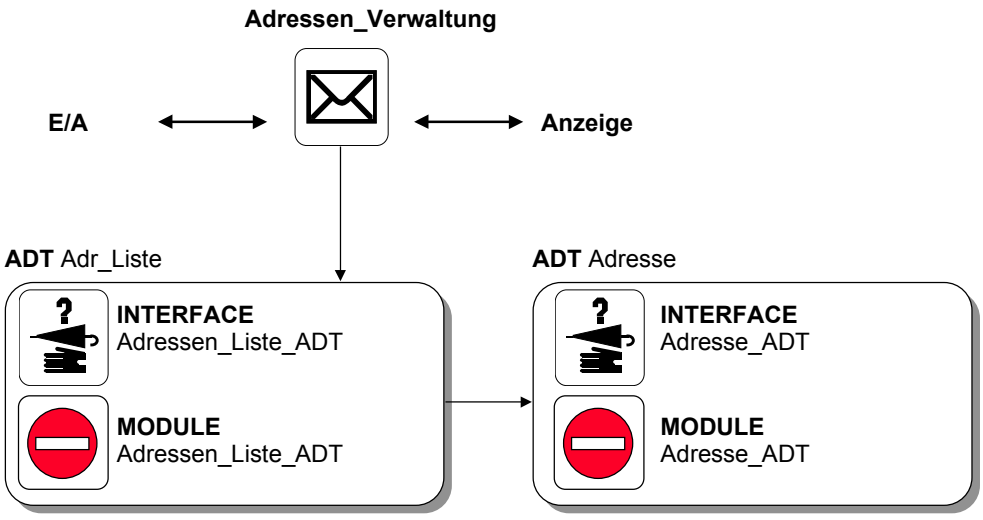
```
PROCEDURE Lese_Adressen(VAR Liste: Adressen_Liste;
                        Adr_Datei_Name : TEXT) =
VAR Zeile : TEXT;
    Adr_Datei_lesen : Rd.T;
BEGIN
  Liste:=NEW(Adressen_Liste);
  Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
  Liste.Letzte_Adresse := 1;
  WHILE NOT IO.EOF (Adr_Datei_lesen) DO
    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Name := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Vorname := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].PLZ := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Ort := Zeile;

    Zeile := IO.GetLine(Adr_Datei_lesen);
    Liste.Adressen[Liste.Letzte_Adresse].Tel := Zeile;

    Zeile:=IO.GetLine(Adr_Datei_lesen);    (* Leerzeile *)
    INC(Liste.Letzte_Adresse);
  END;
  Liste.Akt_Adresse := 1;
  DEC(Liste.Letzte_Adresse);
  Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;
```

Entwicklungsschritt

Fasziniert vom Konzept der ADTen beschließt Herr X
den Typ `Adresse` aus dem Paket `Adressen_ADT` zu **extrahieren**
und einen **eigenen ADT `Adresse`** zu bilden

- Damit wird das Programm besser erweiterbar und wartbar!
- `Adresse` Eintragstyp
- `Adressen_ADT` Kollektion

Architektur mit zwei ADTen



Schnittstelle ADT Adresse

```
INTERFACE Adresse_ADT;
(* realisiert den ADT Adresse mit folgenden Operationen*)

IMPORT Rd, Wr;

TYPE Adresse <: REFANY;

PROCEDURE Lese_Eine_Adresse(VAR Datei: Rd.T) : Adresse;
(* liest eine Adresse von der angegebenen Datei und
   liefert diese zurueck *)

PROCEDURE Schreibe_Eine_Adresse(Adr: Adresse; VAR Datei: Wr.T);
(* schreibt die Adresse Adr auf Datei *)

PROCEDURE Zeige_Eine_Adresse(Adr: Adresse);
(*  zeigt die Adresse Adr am Bildschirm an *)

PROCEDURE Lese_Eine_Neue_Adresse():Adresse;
(* Fordert den Benutzer auf, Angaben fuer ein neue Adresse
   einzugeben und liefert die eingegebene Adresse zurueck. *)

PROCEDURE Modifiziere_Eine_Adresse(VAR Adr: Adresse);
(* Fordert den Benutzer auf, neue Angaben fuer die Adresse Adr
   einzugeben.*)

PROCEDURE Liefere_Suchattribut():TEXT;
(* Liefert das Suchattribute fuer Adressen.*)

PROCEDURE Hat_Namen (Adr: Adresse;  Name: TEXT):BOOLEAN;
(* Testet, ob die Adresse Adr den angegebene Namen hat.*)

END Adresse_ADT.
```

Name des ADT

Deklaration der Operationen

Schnittstelle Adressen_Liste

Name des ADT

```
INTERFACE Adressen_Liste_ADT;
(* Realisiert einen abstrakten Datentyp Adressen_Liste*)

TYPE Adressen_Liste <: REFANY;

PROCEDURE Neue_Adresse        (VAR Liste : Adressen_Liste);
PROCEDURE Modifiziere_Adresse (VAR Liste : Adressen_Liste);
PROCEDURE Suche_Adresse       (    Liste : Adressen_Liste);
PROCEDURE Zeige_Adresse       (    Liste : Adressen_Liste);
PROCEDURE Zuruecksetzen       (VAR Liste : Adressen_Liste);
PROCEDURE Blaettern           (VAR Liste : Adressen_Liste);

PROCEDURE Lese_Adressen       (VAR Liste : Adressen_Liste;
                                   Adr_Datei_Name : TEXT);
PROCEDURE Schreibe_Adressen   (    Liste : Adressen_Liste;
                                   Adr_Datei_Name : TEXT);

END Adressen_Liste_ADT.
```

Deklaration der Operationen

Objekt- und Klassenbasierte Lösung

Rumpf ADT Adressen_Liste - 1

```
MODULE Adressen_Liste_ADT;

IMPORT IO, Rd, Wr;
IMPORT Adresse_ADT AS Adr;

CONST Max = 30;

TYPE Listen_Laenge = [1..Max];
     AdressenSpeicher = RECORD
        Letzte_Adresse : Listen_Laenge := 1;
        Akt_Adresse    : Listen_Laenge := 1;
        Adressen       : ARRAY Listen_Laenge OF Adr.Adresse;
     END;

     REVEAL Adressen_Liste = BRANDED REF AdressenSpeicher;

(*-------------------------------------------------------*)

PROCEDURE Zeige_Adresse(Liste: Adressen_Liste) =
BEGIN
   IO.Put("\n");
   Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
   IO.Put("\n");
END Zeige_Adresse;

PROCEDURE Zuruecksetzen(VAR Liste: Adressen_Liste) =
BEGIN
   Liste.Akt_Adresse := 1;
   Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
END Zuruecksetzen;

PROCEDURE Blaettern(VAR Liste: Adressen_Liste) =
BEGIN
   IF Liste.Akt_Adresse < Liste.Letzte_Adresse THEN
      INC(Liste.Akt_Adresse);
      Adr.Zeige_Eine_Adresse(Liste.Adressen[Liste.Akt_Adresse]);
   ELSE
      IO.Put("Kein Datensatz mehr vorhanden!\n");
   END;
END Blaettern;
```

Import ADT Adresse

Objekt- und Klassenbasierte Lösung

Rumpf ADT Adressen_Liste - 2

```
PROCEDURE Suche_Adresse(Liste: Adressen_Liste) =
VAR Suchbegriff : TEXT;
    Gefunden : BOOLEAN := FALSE;
BEGIN
   IO.Put("Es kann nur nach dem Attribut: " &
          Adr.Liefere_Suchattribut() &
          " gesucht werden!\n");
   IO.Put ("Attributwert > ");
   Suchbegriff := IO.GetLine ();
   Liste.Akt_Adresse := 1;
   WHILE (NOT Gefunden) AND
         (Liste.Akt_Adresse <= Liste.Letzte_Adresse) DO
      IF Adr.Hat_Namen(Liste.Adressen[Liste.Akt_Adresse],
                       Suchbegriff) THEN
         Gefunden := TRUE;
         Zeige_Adresse(Liste);
      ELSE
         INC(Liste.Akt_Adresse);
      END;
   END;
   IF NOT Gefunden THEN
      IO.Put("Der spezifizierte Datensatz
              ist nicht vorhanden!\n");
   END;
END Suche_Adresse;

(*-------------------------------------------------------*)

PROCEDURE Lese_Adressen(VAR Liste: Adressen_Liste;
                        Adr_Datei_Name : TEXT) =
VAR Adr_Datei_lesen : Rd.T;
BEGIN
   Liste:= NEW(Adressen_Liste);
   Adr_Datei_lesen := IO.OpenRead (Adr_Datei_Name);
   Liste.Letzte_Adresse := 1;
   WHILE NOT IO.EOF (Adr_Datei_lesen) DO
      Liste.Adressen[Liste.Letzte_Adresse]:=
                  Adr.Lese_Eine_Adresse(Adr_Datei_lesen);
      INC(Liste.Letzte_Adresse);
   END;
   DEC(Liste.Letzte_Adresse);
   Liste.Akt_Adresse := 1;
   Rd.Close (Adr_Datei_lesen);
END Lese_Adressen;
```

Rumpf ADT Adressen_Liste - 3

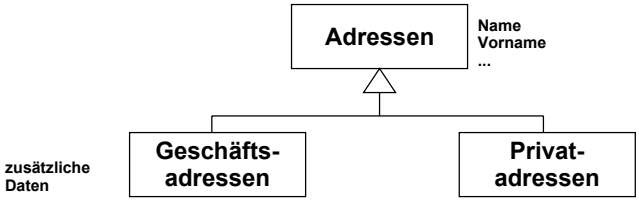
```
PROCEDURE Schreibe_Adressen( Liste: Adressen_Liste;
                             Adr_Datei_Name: TEXT) =
VAR Adr_Datei_schreiben:Wr.T;
BEGIN
  Adr_Datei_schreiben := IO.OpenWrite (Adr_Datei_Name);
  FOR I:= 1 TO Liste.Letzte_Adresse DO
    Adr.Schreibe_Eine_Adresse(Liste.Adressen[I,
                              Adr_Datei_schreiben);
  END;
  Wr.Close (Adr_Datei_schreiben);
END Schreibe_Adressen;
(*-------------------------------------*)

PROCEDURE Neue_Adresse(VAR Liste: Adressen_Liste) =
BEGIN
  IF Liste.Letzte_Adresse = Max THEN
    IO.Put("Es ist leider keine Platz mehr!\n");
  ELSE
    INC(Liste.Letzte_Adresse);
    Liste.Akt_Adresse := Liste.Letzte_Adresse;
    Liste.Adressen[Liste.Akt_Adresse] :=
                                Adr.Lese_Eine_Neue_Adresse();
    IO.Put("Eingegebene Adresse :\n");
    Zeige_Adresse(Liste);
  END;
END Neue_Adresse;

(*-------------------------------------*)

PROCEDURE Modifiziere_Adresse(VAR Liste:Adressen_Liste) =
BEGIN
  IO.Put("Adresse :");
  Zeige_Adresse(Liste);
  Adr.Modifiziere_Eine_Adresse
                    (Liste.Adressen[Liste.Akt_Adresse]);
  IO.Put("\nGeaenderte Adresse :");
  Zeige_Adresse(Liste);
END Modifiziere_Adresse;

BEGIN
END Adressen_Liste_ADT.
```


(Teilweise) Objektorientierte Lösung

- **Über Personen werden nicht nur Name, Vorname etc. gehalten, sondern auch weitere Daten**
- **Diese weiteren Daten können von Personengruppe zu Personengruppe variieren**
  - Geschäftsfreund: Handy-Nr.
  - private Freunde: Geburtsdatum, Vorlieben
- **Es macht keinen Sinn die gemeinsamen Teile mehrfach und gleich festzulegen**

Lösung mit OO

Vererbungshierachie für Adressen



Was haben wir gelernt?

- **kleines Anwendungssystem in verschiedenen Ausbaustufen, Änderungsaufwand enorm, gründliche Überlegung vor der Realisierung**
- **schlechteste Lösung: offene Datentypen, Veränderung direkt an den Stellen der Anwendung**
- **prozedurale Lösung: alle wesentliche Funktionalität in Prozeduren, schlechter Entwurf: Prozeduren dienen unterschiedlichen Zwecken: Datenzugriff, E/A-Handhabung, Verwaltung des Ablaufs; keine Garantie, daß auf Daten nicht direkt zugegriffen werden wird**
- **objektbasierte Lösung: ein Datenbestand; Zugriffsoperationen sauber (weitere Prozeduren sollten geordnet nach Zweck in andere Module eingelagert werden: E/A, Verwaltung, Initialisierung/Abschluß**
- **ADT-Lösung: mehrere Datenbestände (ohne Quelltextvervielfältigung), Rest analog, man hat zwei ADT-Entscheidungen Adresse (Eintrag), Adreßliste (Kollektion): ADO-Baustein bei einer Kollektion, ADT-Baustein bei mehreren**
- **objektorientierte Lösung:**
  - Wirft man alles weg und macht alles zu Klassen (typische OO-Denkweise)?
  - Konzentriert man OO-?? auf die Stellen, wo Ähnlichkeiten zwischen ADTs gehandhabt werden sollen? Haben letzteres gemacht.
  - Modellierung der Ähnlichkeiten, Verschiedenheiten der Einträge durch OO (Übungsaufgaben)

Datenabstraktion

- **Prozeß- und Datenabstraktion**
- **Objektmodule (Datenkapselung)**
- **Abstrakte Datentypen**

Programmieren im Großen

- **Programmieren im Kleinen**
  - befaßt sich mit der Konstruktion eines ***Programms***. Wir haben bisher die dazu notwendigen Konzepte kennengelernt:
    - Daten- und Kontrollstrukturen,
    - Typen,
    - Prozeduren und Funktionen.
- **Programmieren im Großen**
  - für die Entwicklung großer Softwaresysteme müssen weitere Konzepte hinzukommen. Das vorrangige Problem ist, die ***Komplexität*** großer Softwaresysteme zu beherrschen.
  - Die gewählte Lösung heißt ***Abstraktion***.
- **Für den modularen Softwareentwurf sind zwei Abstraktionskonzepte entscheidend:**
  - ***Prozeßabstraktion***,
  - ***Datenabstraktion***.

Abstraktion: Prozeß, Daten

- **Prozeßabstraktion (auch algorith. Abstraktion):**
  - Funktionen und Prozeduren werden zu ***abstrakten*** Konzepten.
  - Module können gleichartige zusammenfassen: funktionaler Modul.
  - Bekannt sind nur die ***Eingabe***- und ***Ausgabegrößen***, aber nicht die Implementation.
  - z.B.: Für eine mathematische Funktion ist nur wesentlich, daß sie ein korrektes Ergebnis liefert, aber nicht wie dies geschieht
- **Datenabstraktion:**
  - Die ***Details*** der verwendeten Daten werden verborgen.
  - Schließt konzeptionell die Prozeßabstraktion für die Zugriffsoperationen ein.
  - Beispiel:
    - Für eine Liste ist wichtig, daß sie ein Behälter für Elemente ist und daß bestimmte Zugriffsoperationen erlaubt sind,
    - aber nicht, wie die Elemente der Liste gespeichert und bearbeitet werden.
  - Eine weitere von Datenabstraktion abstrahiert auch von der ***Art der Elemente***, die in der Liste gespeichert werden.

Information Hiding

- **Prinzip:**
  - Es werden nur die Informationen zur Verfügung gestellt, die ***absolut notwendig*** sind!
  - Alle anderen, insbesondere die ***wichtigen Informationen*** werden ***versteckt***!
  - Der Zugriff auf diese Informationen geschieht über "***Vermittler***".
- **Beispiel: "Offene Bank"**
  - Funktioniert nicht, weil
    - die Buchhaltung nicht klappt (Änderungen werden nicht jedem bekannt sein)
    - doch gestohlen wird (was mißbraucht werden kann, wird mißbraucht)
- **Deshalb:**
  - Das wertvolle wird versteckt (Geld -Tresor)
  - Es werden Vermittler eingesetzt (Mitarbeiter, Automaten)

D. Parnas, 1972

Datenkapsel

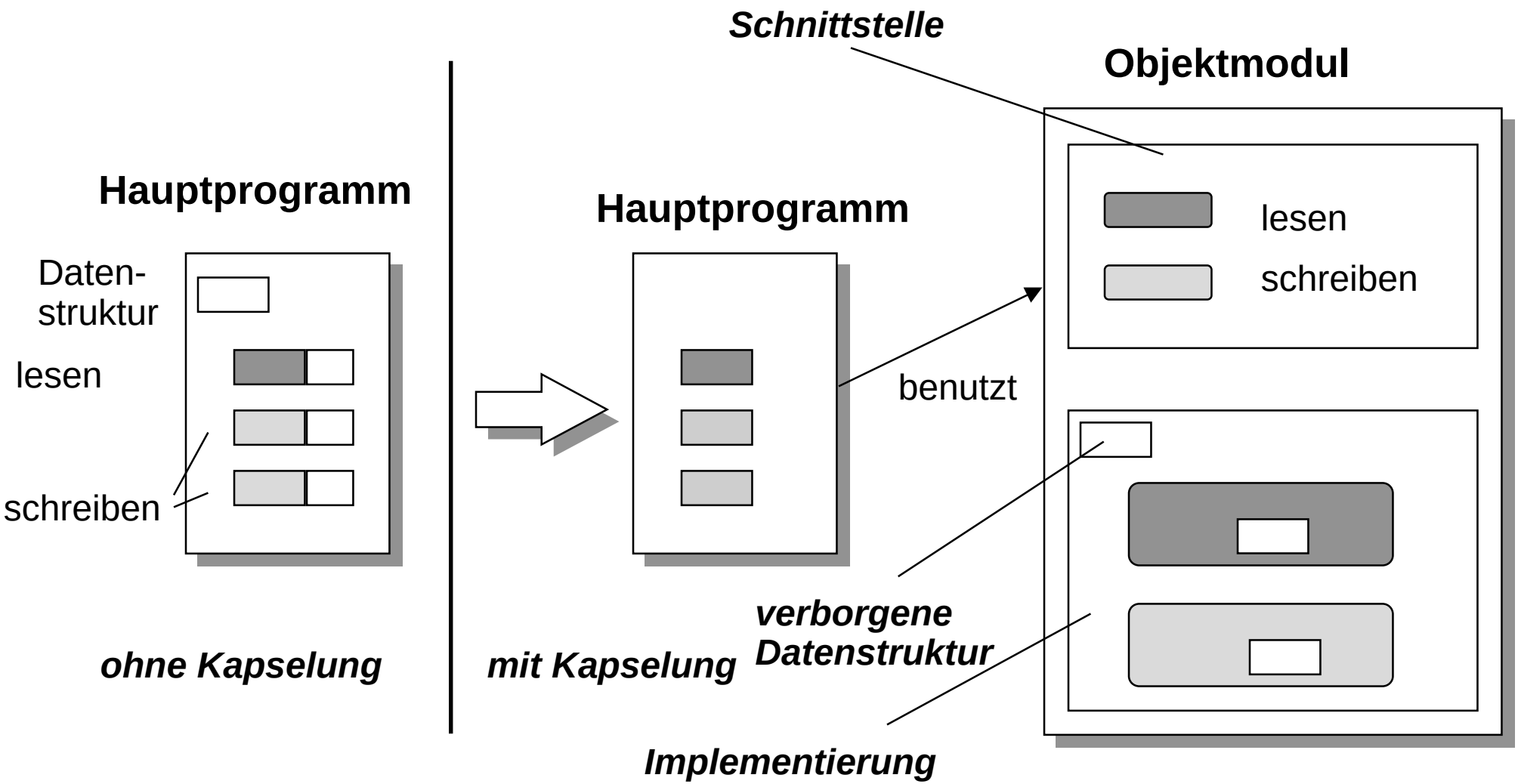
- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Datenstruktur von ihren sichtbaren Eigenschaften.

- **Merkmale:**
  - Eine Datenstruktur wird in einem Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen sichtbar***, die den allgemeinen Umgang mit der Datenstruktur beschreiben.
  - Die Datenstruktur selbst ist ***verborgen***.

- **Jedes Objektmodul**
  - beschreibt und realisiert nur eine ***einzige sog. abstrakte Datenstruktur***.

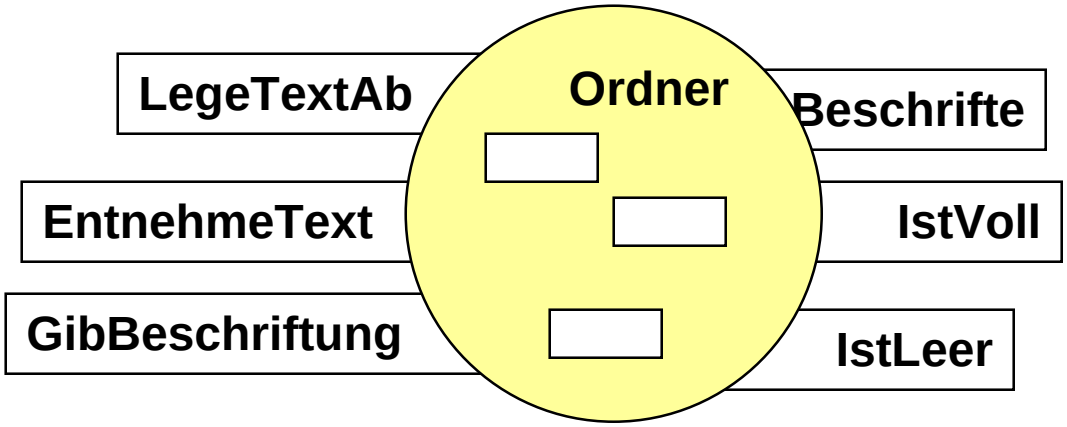
- **Kapselung von Daten**
  - ist ein zentraler Denkansatz und ein wesentliches ***Entwurfsprinzip***

Schema: Objektmodul



Beispiel: Das Objektmodul Ordner

- **"Ordner " als Konzept**
  - ***enthält*** Texte
  - Texte können ***abgelegt*** und ***entnommen*** werden
  - ein Ordner kann ***beschriftet*** werden
  - ein Ordner kann ***leer*** oder ***voll*** sein
- **Ein Objektmodul**
  - realisiert ein ***fachliches Konzept*** (z.B. das Konzept "Ordner")
  - als ***genau eine abstrakte Datenstruktur***
  - hat ***Zustand*** (Gedächtnis)



Realisierung Objektmodul

```
INTERFACE Ordner;

PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();

END Ordner.
```

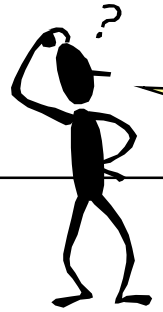
Objektmodul Ordner ist ein Behälter für Daten des Typs TEXT

Abstrakte Beschreibung des fachlichen Konzepts Ordner

Das eine Objekt Ordner soll mehrfach pro Programmablauf verwendbar sein

Verwendung eines Objektmoduls 1

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Ordner,  SIO;

BEGIN
  Ordner. Initialisier();
  Ordner.Beschrifte ("Kleine Gedichte");
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
  Ordner.LegeTextAb ("Herr von Ribeck auf Ribeck ...");
  Ordner.LegeTextAb ("Von drauss vom Walde komm ich her ...");
  SIO.PutLine (Ordner.GibBeschriftung());
  SIO.PutLine ("---------------------------");
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  Ordner.Initialisiere();
  ...
END Ordner_Test.
```



Diskussion:
Wie darf ein Objekt-modul verwendet werden?

Verwendung eines Objektmoduls 2

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in bestimmten Situationen (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, werden an der Schnittstelle entsprechende ***Testfunktionen*** wie `IstLeer` oder `IstVoll` zur Verfügung.

Einsatz der Testfunktionen

Prüfen der Vorbedingung

Das Objektmodul **Ordner** wird vor der ersten Verwendung initialisiert, d.h. der Inhalt wird gelöscht

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Vor Entnahme wird der Zustand des Objektmoduls Ordner geprüft und entsprechend gehandelt.

Entwurfskonzept

- **Um ein fachliches Konzept als Objektmodul zu realisieren, stellen wir *drei Arten von Operationen* zur Verfügung.**
- **Prozeduren:**
  - ***verändern*** den ***Zustand*** des Objekts. Meist sind sie von Vorbedingungen abhängig, d.h. sie können nicht in jedem Zustand ausgeführt werden.
- **Funktionen:**
  - liefern Informationen, ***ohne*** den Objektzustand nach außen sichtbar ***zu verändern***.
  - Fachliche Funktionen:
    - liefern ***fachliche Informationen*** und sind vom Objektzustand abhängig.
  - Testfunktionen:
    - ***prüfen*** den Objektzustand und werden zum Prüfen der Vorbedingungen verwendet.

Entwurfskonzept-Beispiel: Ordner

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Prozeduren:**
  - `Initialisiere`, `LegeTextAb`, `EntnehmeText`, `Beschrifte` sind verändernde Prozeduren.
- **Fachliche Funktionen:**
  - `GibBeschriftung` ist eine fachliche Funktion; sie verändert im Gegensatz zu `EntnehmeText` nicht den Ordnerzustand.
- **Textfunktionen:**
  - `IstLeer` und `IstVoll`

Ein Blick ins Innere - 1

```
MODULE Ordner EXPORTS Ordner;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
TYPE Texte = ARRAY Fassungsvermoegen OF TEXT;
VAR ordnerInhalt : Texte;
    anzahlTexte : CARDINAL;
    beschriftung : TEXT := "";

PROCEDURE LegeTextAb (t : TEXT)=
BEGIN
  anzahlTexte := anzahlTexte + 1;
  ordnerInhalt[anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText () : TEXT =
VAR t : TEXT;
BEGIN
  t := ordnerInhalt[anzahlTexte];
  ordnerInhalt[anzahlTexte] := "";
  anzahlTexte := anzahlTexte - 1;
  RETURN t;
END EntnehmeText;
```

Es wird ein Array verwendet

Es wird der zuletzt abgelegte Text entnommen

Ein Blick ins Innere - 2

```
PROCEDURE IstVoll () : BOOLEAN =
BEGIN
  RETURN (anzahlTexte = MaxTexte);
END IstVoll
...

PROCEDURE Beschrifte (t : TEXT)=
BEGIN
  beschriftung := t;
END Beschrifte;
...

PROCEDURE Initialisiere () =
BEGIN
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    ordnerInhalt[i] := "";
  END;
  anzahlTexte := 0;
END Initialisiere;

BEGIN
  Initialisiere ();
END Ordner.
```

`IstVoll` verwendet die Variable `anzahlTexte`

Zusammenfassung Objektmodul

Eigenschaften eines Objektmoduls:

- Das Modul verwaltet seine Daten ***selbst***.
- Die interne Repräsentation der Daten ist vollständig ***verborgen***.
- Die interne Repräsentation ist ***austauschbar***.
- Zur Laufzeit existiert immer ***nur ein Exemplar*** eines Objektmoduls, d.h. es können z.B. nicht mehrere Ordner erzeugt werden.
- Das Objektmodul kann von ***mehreren*** anderen "Kunden" verwendet werden.
- Dadurch kann z.B. ein ***gemeinsamer*** Ordner verwaltet werden.

Module als Sprachelement

- **Ein Modul kann zwar**
  - ***definiert*** und zur Laufzeit von anderen Modulen ***importiert*** werden,
  - aber es ist kein ***primäres*** Sprachelement (first class), d.h.
  - ein Modul kann ***nicht*** wie ein Typ ***zur Deklaration*** von Bezeichnern verwendet werden.

  - `o1, o2 : Ordner (* geht nicht, da Ordner Modul ist *)`

- **Folge:**
  - es gibt ***nur ein Exemplar*** eines Objektmoduls.

- **Problem:**
  - Es werden oft mehrere Exemplare eines durch ein Objektmodul realisierten Objekts gebraucht.

- **Lösungsansatz:**
  - Wir formulieren ***einen Typ*** für die im Modul beschriebenen Objekte.

Konzept Abstrakter Datentyp

- **Kann als *Verallgemeinerung* des Objektmoduls (Datenkapsel) betrachtet werden.**
- **Anstatt eines Objektes wird ein Typ (für diese Objekte) definiert.**
- **Betrachten wir den ADT als (formale) Spezifikation eines Typs,**
  - dann entwerfen wir ihn durch Angabe von
  - ***Typnamen***
  - ***Signaturen*** (Operationen)
    - zum Erzeugen von Objekten, zum Verändern etc.
  - ***Axiome***
    - formulieren den semantischen Zusammenhang der Operationen.
  - ***Vorbedingungen***
    - geben an, in welchem Zustand welche Operationen gültig sind.

Beispiel: ADT

TYPE

Stack [G]

FUNCTIONS

push: Stack[G] x G -> Stack[G]

pop: Stack[G] -> Stack[G]

item: Stack[G] -> G

empty: Stack[G] -> Bool

new: Stack[G]

programmiersprachenunabhängig, formale Formulierung

AXIOMS

For any x: G, s: Stack[G]

item (push(s,x)) = x

pop (push (s,x)) = s

empty(new)

not empty(push (s,x))

PRECONDITIONS

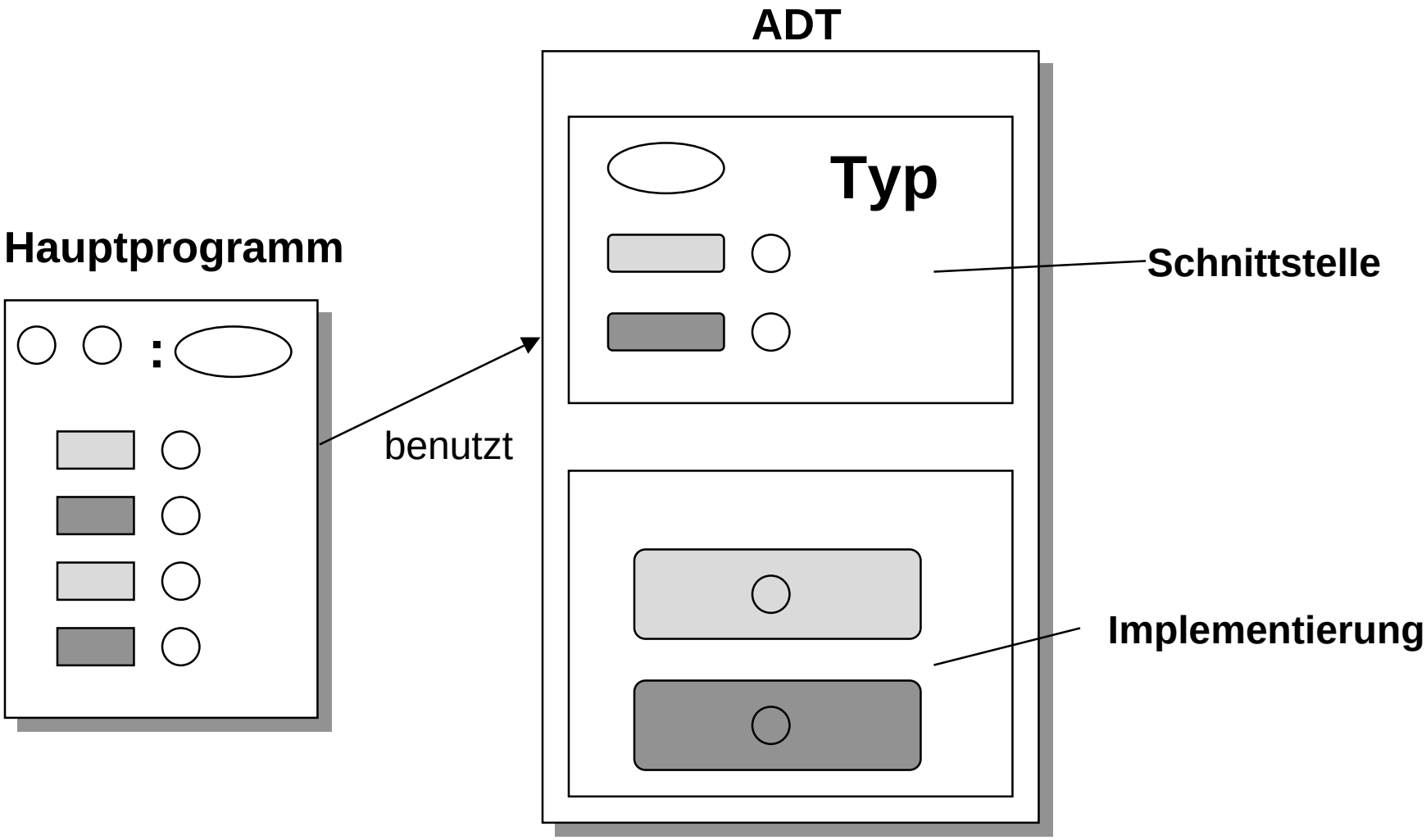
pop (s: Stack[G]) requires not empty (s)

item (s: Stack[G]) requires not empty (s)

Realisierung von ADTs durch Module

- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Menge gleichartiger Datenstrukturen von ihren allgemeinen Eigenschaften.
- **Merkmale:**
  - Die Beschreibung der gleichartigen Exemplare einer Datenstruktur wird in einem sog. ADT-Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen*** sichtbar, die den allgemeinen Umgang mit ***jedem Exemplar*** der Datenstruktur beschreiben.
  - Ein ***Bezeichner für den Typ*** der Datenstruktur wird exportiert.
  - Die Implementierung der Datenstruktur selbst ist ***verborgen***.
- **Jedes ADT-Modul beschreibt und realisiert eine Menge von Exemplaren der abstrakten Datenstruktur.**
  - Die Exemplare werden von ***ihren Kunden*** verwaltet. Ihre Bearbeitung geschieht nur mit Hilfe der exportierten Operationen des ADT-Moduls.

Realisierungsschema ADT



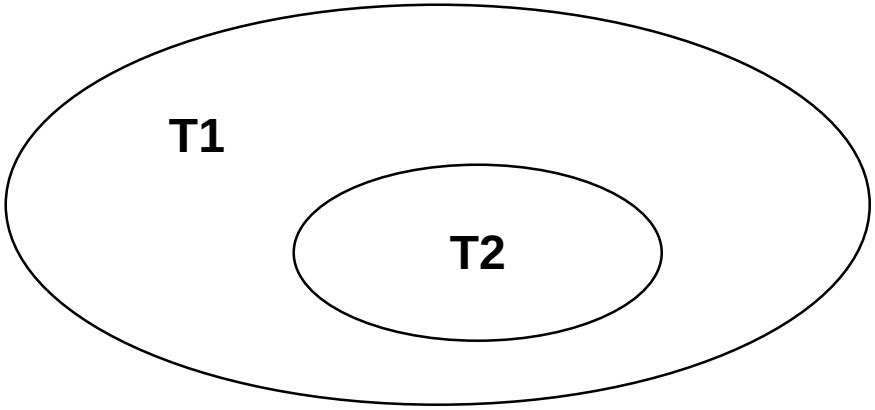
Exkurs: Subtypen in Modula-3

Subtypen in Modula-3

- Modula-3 kennt das Konzept der ***Subtyp-Bildung***
- Subtypbeziehung wird durch "***<:***" angezeigt

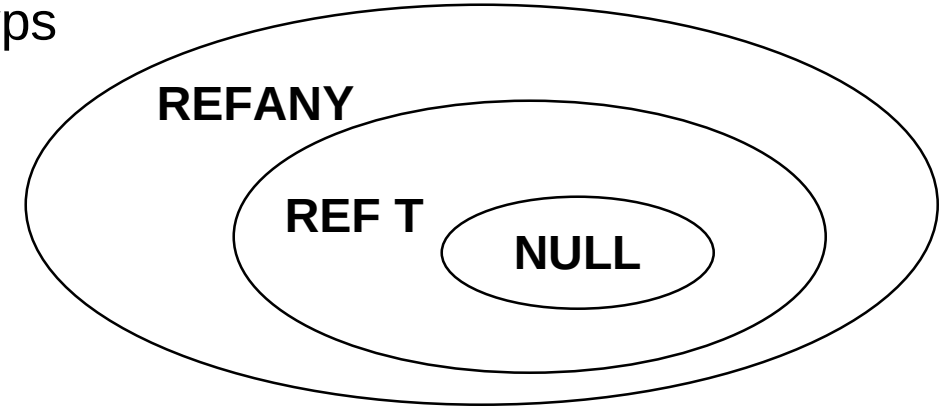
Definition

- Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
- T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
- Es gilt: ein Typ kann beliebig viele Subtypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Subtypen von Referenztypen

- **Modula-3**
  - definiert zwei vorgegebene Referenztypen
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Interpretation:**
  - jeder Referenztyp in ***Untertyp*** von REFANY
  - NULL ist ***Untertyp*** jedes Referenztyps
    - Der Referenztyp NULL kennt nur den Wert NIL
    - NIL ist Wert jedes Referenztyps



Realisierung eines ADT in Modula-3

- **Forderung:**
  - In der Schnittstelle darf die Typstruktur eines ADT ***nicht sichtbar*** sein.
  - Lediglich der ***Name*** soll bekannt gemacht werden.
- **Realisierung im Modula-3**
  - In der Schnittstelle wird ein ***opakter Typ*** (verdeckter Typ) deklariert
  - Dies geschieht, indem der Typ als ***Untertyp*** zum vordefinierten Typ REFANY deklariert wird.
  - ***Obertyp*** eines opakten Typs muß ein ***Referenztyp*** sein.
  - In der Implementierung des ADTs wird der opake Typ ***offengelegt*** ("enthüllt")
  - Bsp.:

```
TYPE Ordner <: REFANY;
```

Mithilfe des Subtyp-Konzepts und mit opaken Tyoen können die Forderungen umgesetzt werden

Schnittstelle des ADT-Moduls Ordner

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t :
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

Nur der Name des ADT ist sichtbar!

Alle Operationen erhalten als ersten Parameter das jeweilige Ordnerobjekt

Erzeugt ein neues Ordnerobjekt und gibt es zurück

Beispiel: ADT Ordner

Angabe des Typnamens

Angabe der Operationen

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

```
IMPORT OrdnerADT; (* Client *)
VAR
  ord1, ord2 : OrdnerADT.Ordner

BEGIN
  ord1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ord1, "Nicht immer sind bequeme ...");

  ord2 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord2, "Musikstuecke");
  OrdnerADT.LegeTextAb (ord2, "Tief im Westen ...");
```

Implementierung des ADT Ordner

```
MODULE OrdnerADT EXPORTS OrdnerADT;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
     Texte = ARRAY Fassungsvermoegen OF TEXT;

REVEAL Ordner = BRANDED REF RECORD
                  ordnerInhalt : Texte;
                  anzahlTexte  : Fassungsvermoegen;
                  beschriftung : TEXT := "";
                END;
```

REVEAL-Deklaration

- damit wird die ***interne Struktur*** des opakenTyps bekannt gegeben
- Steht immer in der ***Implementierung*** eines ADTs (information hiding).
- Der äußere Typ-Konstruktor ***muß*** ein mit einem ***Brandzeichen*** versehener Referenztyp sein
- Dieses unterscheidet den Typ von anderen ***strukturell gleichen*** Typen.

Ein Blick ins Innere des ADTs

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)=
BEGIN
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT =
VAR t : TEXT;
BEGIN
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  RETURN t;
END EntnehmeText;


PROCEDURE Anlegen () : Ordner =
VAR o : Ordner;
BEGIN
  o := NEW(Ordner);
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    o^.ordnerInhalt[i] := "";
  END;
  o^.anzahlTexte := 0;
  RETURN o;
END Anlegen;
```

Diskussion : ADT-Realisierung in Modula-3

- **Wir können feststellen**
  - Als Typ für einen ADT müssen wir einen opaken Referenztyp wählen.
  - Grund: So kann der Übersetzer für Objekte eines ADTs ausreichend Speicherplatz reservieren, ohne den inneren Aufbau des Typs zu kennen.
- **Konsequenz:**
  - Es können neben den Operationen der Schnittstelle des ADTs auch die Operationen
    - Zuweisung und
    - Vergleich auf Objekten des ADTs durchgeführt werden (Pointer-Zuweisung und Pointer-Vergleich).
  - Diese sollten jedoch nicht genutzt werden!
  - Das Brandzeichen verhindert, daß einem Objekt eines ADT zufälligerweise ein Wert eines strukturell gleichen Typs zugewiesen werden kann.

Zuweisung und Vergleich von ADT-Objekten

```
ordner1 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner1, "Nicht immer sind bequeme Stuehle ...");

ordner2 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner2, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner2, "Nicht immer sind bequeme Stuehle ...");

IF ordner1 = ordner2 THEN
  SIO.PutLine ("1 ordner sind gleich");
ELSE
  ordner2 := ordner1;
  OrdnerADT.Beschrifte (ordner1, "Romane");
END;
IF ordner1 = ordner2 THEN
  SIO.PutLine ("nach Zuweisung sind die Ordner gleich");
END;

SIO.PutLine (OrdnerADT.GibBeschriftung(ordner1));
SIO.PutLine ("----------------------------");
SIO.PutLine (OrdnerADT.GibBeschriftung(ordner2));
```

Vergleich der Referenzen (Adressen

ordner2 zeigt auf die selbe Adresse wie ordner1

Ändert ordner1 und ordner2

Modul als Entwicklungseinheit

- **Ein Modul ist charakterisiert durch eine *Entwurfs-entscheidung*.**
  - (z.B. wir wollen Ordner verarbeiten können)
- **Jedes Modul basiert auf genau einer Entwurfsentscheidung.**
- **Module *verbergen* Implementierungsentscheidungen.**
  - Jedes Modul hat sein Geheimnis.
- **Es werden immer die Entscheidungen**
  - in einem Modul gekapselt, die vielleicht ***revidiert*** werden müssen als andere.
  - z.B. Datenaufbau
- **Der Änderungsaufwand muß möglichst immer auf ein Modul begrenzt werden.**

Zusammenfassung ADT

- **In imperativen Sprachen**
  - mit einem ***Modulkonzept***
  - realisieren wir abstrakte Datentypen mit einem ADT-Modul.

- **Ein ADT-Modul zeigt folgende Eigenschaften:**
  - Das Modul liefert ***Exemplare einer Datenstruktur***, die beim Kunden aufbewahrt werden.
  - Dadurch können ***mehrere Exemplare*** eines ADT-Moduls bearbeitet werden.
  - Die Datenstruktur ist nur als ***Verweis*** auf die interne Repräsentation bekannt.
  - Die Repräsentation selbst ist ***verborgen*** und ***austauschbar***.
  - Die Datenstrukturen des ADT-Moduls können (fast) nur über die ***exportierten Operationen*** des Moduls bearbeitet werden.

Was haben wir gelernt?

- **Modularisierung**
  - Mittel, um ***handhabbare*** Programmeinheiten zu konstruieren
  - Module bestehen aus ***Schnittstelle*** und ***Implementierung***
- **Information Hiding**
  - Ziel:
    - Implementierung wesentlicher Details ist ***nicht*** nach ***außen sichtbar,*** insbesondere anwendbar auf Datenstrukturierung
  - Objektmodul:
    - Es wird in einem Modul ***ein Objekt*** gemäß Information Hiding realisiert, Datenabstraktion für ein Objekt
  - Abstrakter Datentyp:
    - Es wird ein ***geschützter Typ*** realisiert. Objekte des ADTs können nur mit den Operationen des ADTs manipuliert werden, Datenabstraktion für eine „Klasse“ von Objekten

Glossar

- **Information Hiding: Prozeß- oder funktionale Abstraktion, Datenabstraktion, funktionaler Module, Datenobjektmodul (Kapsel), abstrakter Datentyp**
- **opake Datenstruktur, opake (geschützte, abstrakte, geheime) Datentypen**
- **Schnittstellenoperationen eines Datenabstraktionsbausteins: Initialisierung (Löschung), Lesen, Verändern, Sicherheitsoperationen**
- **Realisierung opaker Datentypen durch Verweistypen**
- **Typname, Signatur, Axiome, Vorbedingungen eines ADT**
- **Subtypen in Modula-3 haben Referenzsemantik (keine Zuweisung und keine Gleichheitsabfrage nutzen!)**

Datenabstraktion

- **Prozeß- und Datenabstraktion**
- **Objektmodule (Datenkapselung)**
- **Abstrakte Datentypen**

Programmieren im Großen

- **Programmieren im Kleinen**
  - befaßt sich mit der Konstruktion eines ***Programms***. Wir haben bisher die dazu notwendigen Konzepte kennengelernt:
    - Daten- und Kontrollstrukturen,
    - Typen,
    - Prozeduren und Funktionen.
- **Programmieren im Großen**
  - für die Entwicklung großer Softwaresysteme müssen weitere Konzepte hinzukommen. Das vorrangige Problem ist, die ***Komplexität*** großer Softwaresysteme zu beherrschen.
  - Die gewählte Lösung heißt ***Abstraktion***.
- **Für den modularen Softwareentwurf sind zwei Abstraktionskonzepte entscheidend:**
  - ***Prozeßabstraktion***,
  - ***Datenabstraktion***.

Abstraktion: Prozeß, Daten

- **Prozeßabstraktion (auch algorith. Abstraktion):**
  - Funktionen und Prozeduren werden zu ***abstrakten*** Konzepten.
  - Module können gleichartige zusammenfassen: funktionaler Modul.
  - Bekannt sind nur die ***Eingabe***- und ***Ausgabegrößen***, aber nicht die Implementation.
  - z.B.: Für eine mathematische Funktion ist nur wesentlich, daß sie ein korrektes Ergebnis liefert, aber nicht wie dies geschieht
- **Datenabstraktion:**
  - Die ***Details*** der verwendeten Daten werden verborgen.
  - Schließt konzeptionell die Prozeßabstraktion für die Zugriffsoperationen ein.
  - Beispiel:
    - Für eine Liste ist wichtig, daß sie ein Behälter für Elemente ist und daß bestimmte Zugriffsoperationen erlaubt sind,
    - aber nicht, wie die Elemente der Liste gespeichert und bearbeitet werden.
  - Eine weitere von Datenabstraktion abstrahiert auch von der ***Art der Elemente***, die in der Liste gespeichert werden.

Information Hiding

- **Prinzip:**
  - Es werden nur die Informationen zur Verfügung gestellt, die ***absolut notwendig*** sind!
  - Alle anderen, insbesondere die ***wichtigen Informationen*** werden ***versteckt***!
  - Der Zugriff auf diese Informationen geschieht über "***Vermittler***".
- **Beispiel: "Offene Bank"**
  - Funktioniert nicht, weil
    - die Buchhaltung nicht klappt (Änderungen werden nicht jedem bekannt sein)
    - doch gestohlen wird (was mißbraucht werden kann, wird mißbraucht)
- **Deshalb:**
  - Das wertvolle wird versteckt (Geld -Tresor)
  - Es werden Vermittler eingesetzt (Mitarbeiter, Automaten)

D. Parnas, 1972

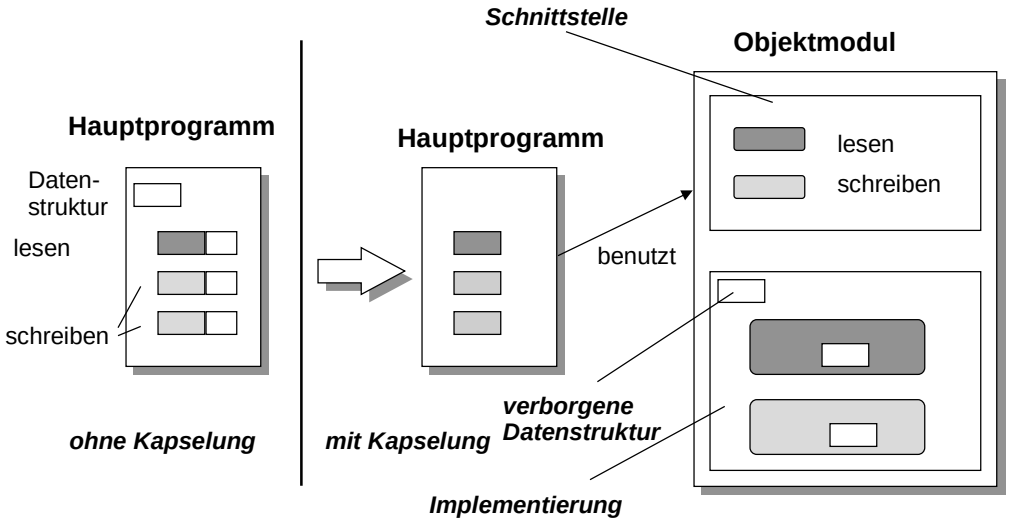
Objekt-modul

Datenkapsel

- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Datenstruktur von ihren sichtbaren Eigenschaften.
- **Merkmale:**
  - Eine Datenstruktur wird in einem Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen sichtbar***, die den allgemeinen Umgang mit der Datenstruktur beschreiben.
  - Die Datenstruktur selbst ist ***verborgen***.
- **Jedes Objektmodul**
  - beschreibt und realisiert nur eine ***einzige sog. abstrakte Datenstruktur***.
- **Kapselung von Daten**
  - ist ein zentraler Denkansatz und ein wesentliches ***Entwurfsprinzip***

Objekt-modul

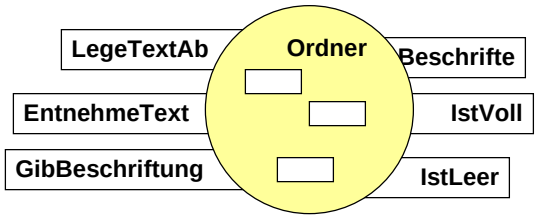
Schema: Objektmodul



Objekt-modul

Beispiel: Das Objektmodul Ordner

- **"Ordner " als Konzept**
  - ***enthält*** Texte
  - Texte können ***abgelegt*** und ***entnommen*** werden
  - ein Ordner kann ***beschriftet*** werden
  - ein Ordner kann ***leer*** oder ***voll*** sein
- **Ein Objektmodul**
  - realisiert ein ***fachliches Konzept*** (z.B. das Konzept "Ordner")
  - als ***genau eine abstrakte Datenstruktur***
  - hat ***Zustand*** (Gedächtnis)



Objekt-modul

Realisierung Objektmodul

```
INTERFACE Ordner;

PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();

END Ordner.
```

Objektmodul Ordner ist ein Behälter für Daten des Typs TEXT

Abstrakte Beschreibung des fachlichen Konzepts Ordner

Das eine Objekt Ordner soll mehrfach pro Programmablauf verwendbar sein

Objekt-modul

Verwendung eines Objektmoduls 1

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Ordner,  SIO;

BEGIN
  Ordner. Initialisier();
  Ordner.Beschrifte ("Kleine Gedichte");
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
  Ordner.LegeTextAb ("Herr von Ribeck auf Ribeck ...");
  Ordner.LegeTextAb ("Von drauss vom Walde komm ich her ...");
  SIO.PutLine (Ordner.GibBeschriftung());
  SIO.PutLine ("--------------------------");
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  Ordner.Initialisiere();
  ...
END Ordner_Test.
```

Diskussion: Wie darf ein Objekt-modul verwendet werden?

Objekt-modul

Verwendung eines Objektmoduls 2

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in bestimmten Situationen (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, werden an der Schnittstelle entsprechende ***Testfunktionen*** wie `IstLeer` oder `IstVoll` zur Verfügung.

Objekt-modul

Einsatz der Testfunktionen

Prüfen der Vorbedingung

Das Objektmodul **Ordner** wird vor der ersten Verwendung initialisiert, d.h. der Inhalt wird gelöscht

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Vor Entnahme wird der Zustand des Objektmoduls Ordner geprüft und entsprechend gehandelt.

Objekt-modul

Entwurfskonzept

- **Um ein fachliches Konzept als Objektmodul zu realisieren, stellen wir *drei Arten von Operationen* zur Verfügung.**
- **Prozeduren:**
  - ***verändern*** den ***Zustand*** des Objekts. Meist sind sie von Vorbedingungen abhängig, d.h. sie können nicht in jedem Zustand ausgeführt werden.
- **Funktionen:**
  - liefern Informationen, ***ohne*** den Objektzustand nach außen sichtbar ***zu verändern***.
  - Fachliche Funktionen:
    - liefern ***fachliche Informationen*** und sind vom Objektzustand abhängig.
  - Testfunktionen:
    - ***prüfen*** den Objektzustand und werden zum Prüfen der Vorbedingungen verwendet.

Objekt-modul

Entwurfskonzept-Beispiel: Ordner

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Prozeduren:**
  - `Initialisiere`, `LegeTextAb`, `EntnehmeText`, `Beschrifte` sind verändernde Prozeduren.
- **Fachliche Funktionen:**
  - `GibBeschriftung` ist eine fachliche Funktion; sie verändert im Gegensatz zu `EntnehmeText` nicht den Ordnerzustand.
- **Textfunktionen:**
  - `IstLeer` und `IstVoll`

Objekt-modul

Ein Blick ins Innere - 1

```
MODULE Ordner EXPORTS Ordner;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
TYPE Texte = ARRAY Fassungsvermoegen OF TEXT;
VAR ordnerInhalt : Texte;
    anzahlTexte : CARDINAL;
    beschriftung : TEXT := "";

PROCEDURE LegeTextAb (t : TEXT)=
BEGIN
  anzahlTexte := anzahlTexte + 1;
  ordnerInhalt[anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText () : TEXT =
VAR t : TEXT;
BEGIN
  t := ordnerInhalt[anzahlTexte];
  ordnerInhalt[anzahlTexte] := "";
  anzahlTexte := anzahlTexte - 1;
  RETURN t;
END EntnehmeText;
```

Es wird ein Array verwendet

Es wird der zuletzt abgelegte Text entnommen

Ein Blick ins Innere - 2

```
PROCEDURE IstVoll () : BOOLEAN =
BEGIN
  RETURN (anzahlTexte = MaxTexte);
END IstVoll
...

PROCEDURE Beschrifte (t : TEXT)=
BEGIN
  beschriftung := t;
END Beschrifte;
...

PROCEDURE Initialisiere () =
BEGIN
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    ordnerInhalt[i] := "";
  END;
  anzahlTexte := 0;
END Initialisiere;

BEGIN
  Initialisiere ();
END Ordner.
```

`IstVoll` verwendet die Variable `anzahlTexte`

Zusammenfassung Objektmodul

- **Eigenschaften eines Objektmoduls:**
  - Das Modul verwaltet seine Daten ***selbst***.
  - Die interne Repräsentation der Daten ist vollständig ***verborgen***.
  - Die interne Repräsentation ist ***austauschbar***.
  - Zur Laufzeit existiert immer ***nur ein Exemplar*** eines Objektmoduls, d.h. es können z.B. nicht mehrere Ordner erzeugt werden.
  - Das Objektmodul kann von ***mehreren*** anderen "Kunden" verwendet werden.
  - Dadurch kann z.B. ein ***gemeinsamer*** Ordner verwaltet werden.

Module als Sprachelement

- **Ein Modul kann zwar**
  - ***definiert*** und zur Laufzeit von anderen Modulen ***importiert*** werden,
  - aber es ist kein ***primäres*** Sprachelement (first class), d.h.
  - ein Modul kann ***nicht*** wie ein Typ ***zur Deklaration*** von Bezeichnern verwendet werden.

```
o1, o2 : Ordner (* geht nicht, da Ordner Modul ist *)
```

- **Folge:**
  - es gibt ***nur ein Exemplar*** eines Objektmoduls.
- **Problem:**
  - Es werden oft mehrere Exemplare eines durch ein Objektmodul realisierten Objekts gebraucht.
- **Lösungsansatz:**
  - Wir formulieren ***einen Typ*** für die im Modul beschriebenen Objekte.

Konzept Abstrakter Datentyp

- **Kann als *Verallgemeinerung* des Objektmoduls (Datenkapsel) betrachtet werden.**
- **Anstatt eines Objektes wird ein Typ (für diese Objekte) definiert.**
- **Betrachten wir den ADT als (formale) Spezifikation eines Typs,**
  - dann entwerfen wir ihn durch Angabe von
  - ***Typnamen***
  - ***Signaturen*** (Operationen)
    - zum Erzeugen von Objekten, zum Verändern etc.
  - ***Axiome***
    - formulieren den semantischen Zusammenhang der Operationen.
  - ***Vorbedingungen***
    - geben an, in welchem Zustand welche Operationen gültig sind.

Beispiel: ADT

TYPE

Stack [G]

FUNCTIONS

push: Stack[G] x G -> Stack[G]

pop: Stack[G] -> Stack[G]

item: Stack[G] -> G

empty: Stack[G] -> Bool

new: Stack[G]

programmiersprachenunabhängig, formale Formulierung

AXIOMS

For any x: G, s: Stack[G]

item (push(s,x)) = x

pop (push (s,x)) = s

empty(new)

not empty(push (s,x))

PRECONDITIONS

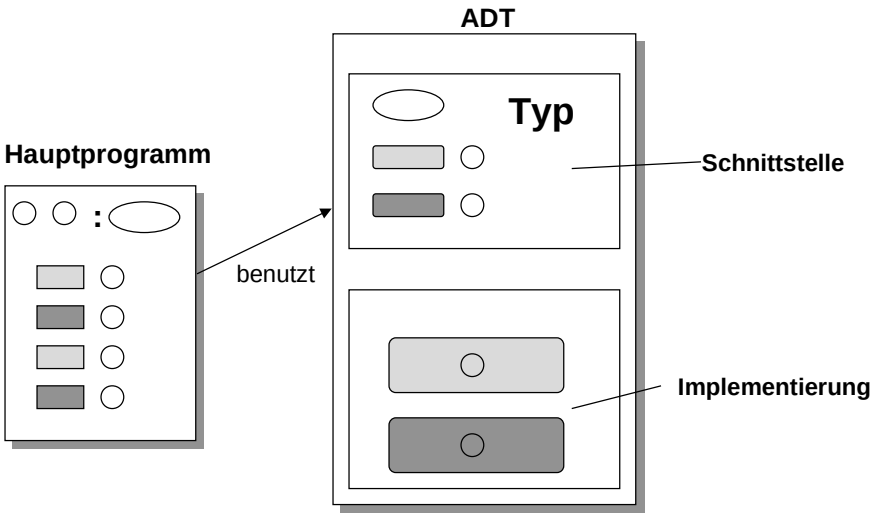
pop (s: Stack[G]) requires not empty (s)

item (s: Stack[G]) requires not empty (s)

Realisierung von ADTs durch Module

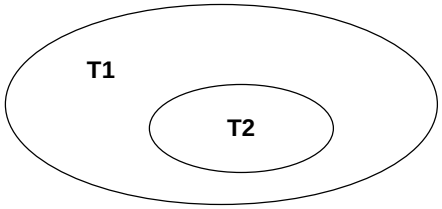
- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Menge gleichartiger Datenstrukturen von ihren allgemeinen Eigenschaften.
- **Merkmale:**
  - Die Beschreibung der gleichartigen Exemplare einer Datenstruktur wird in einem sog. ADT-Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen*** sichtbar, die den allgemeinen Umgang mit ***jedem Exemplar*** der Datenstruktur beschreiben.
  - Ein ***Bezeichner für den Typ*** der Datenstruktur wird exportiert.
  - Die Implementierung der Datenstruktur selbst ist ***verborgen***.
- **Jedes ADT-Modul beschreibt und realisiert eine Menge von Exemplaren der abstrakten Datenstruktur.**
  - Die Exemplare werden von ***ihren Kunden*** verwaltet. Ihre Bearbeitung geschieht nur mit Hilfe der exportierten Operationen des ADT-Moduls.

Realisierungsschema ADT



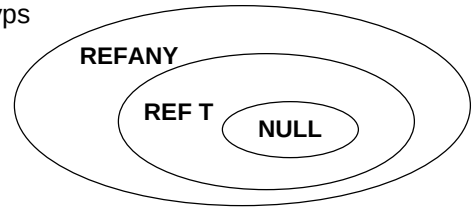
Exkurs: Subtypen in Modula-3

- **Subtypen in Modula-3**
  - Modula-3 kennt das Konzept der ***Subtyp-Bildung***
  - Subtypbeziehung wird durch "***<:***" angezeigt
- **Definition**
  - Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
  - T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
  - Es gilt: ein Typ kann beliebig viele Subtypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Subtypen von Referenztypen

- **Modula-3**
  - definiert zwei vorgegebene Referenztypen
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Interpretation:**
  - jeder Referenztyp in ***Untertyp*** von REFANY
  - NULL ist ***Untertyp*** jedes Referenztyps
    - Der Referenztyp NULL kennt nur den Wert NIL
    - NIL ist Wert jedes Referenztyps



Realisierung eines ADT in Modula-3

- **Forderung:**
  - In der Schnittstelle darf die Typstruktur eines ADT ***nicht sichtbar*** sein.
  - Lediglich der ***Name*** soll bekannt gemacht werden.
- **Realisierung im Modula-3**
  - In der Schnittstelle wird ein ***opakter Typ*** (verdeckter Typ) deklariert
  - Dies geschieht, indem der Typ als ***Untertyp*** zum vordefinierten Typ REFANY deklariert wird.
  - ***Obertyp*** eines opakten Typs muß ein ***Referenztyp*** sein.
  - In der Implementierung des ADTs wird der opake Typ ***offengelegt*** ("enthüllt")
  - Bsp.:

```
TYPE Ordner <: REFANY;
```

Mithilfe des Subtyp-Konzepts und mit opaken Tyoen können die Forderungen umgesetzt werden

Schnittstelle des ADT-Moduls Ordner

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

Nur der Name des ADT ist sichtbar!

Alle Operationen erhalten als ersten Parameter das jeweilige Ordnerobjekt

Erzeugt ein neues Ordnerobjekt und gibt es zurück

Beispiel: ADT Ordner

Angabe des Typnamens

Angabe der Operationen

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

```
IMPORT OrdnerADT; (* Client *)
VAR
  ord1, ord2 : OrdnerADT.Ordner

BEGIN
  ord1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ord1, "Nicht immer sind bequeme ...");

  ord2 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord2, "Musikstuecke");
  OrdnerADT.LegeTextAb (ord2, "Tief im Westen ...");
```

Implementierung des ADT Ordner

```
MODULE OrdnerADT EXPORTS OrdnerADT;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
     Texte = ARRAY Fassungsvermoegen OF TEXT;

REVEAL Ordner = BRANDED REF RECORD
                  ordnerInhalt : Texte;
                  anzahlTexte  : Fassungsvermoegen;
                  beschriftung : TEXT := "";
                END;
```

- **REVEAL-Deklaration**
  - damit wird die ***interne Struktur*** des opakenTyps bekannt gegeben
  - Steht immer in der ***Implementierung*** eines ADTs (information hiding).
  - Der äußere Typ-Konstruktor ***muß*** ein mit einem ***Brandzeichen*** versehener Referenztyp sein
  - Dieses unterscheidet den Typ von anderen ***strukturell gleichen*** Typen.

Ein Blick ins Innere des ADTs

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)=
BEGIN
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT =
VAR t : TEXT;
BEGIN
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  RETURN t;
END EntnehmeText;


PROCEDURE Anlegen () : Ordner =
VAR o : Ordner;
BEGIN
  o := NEW(Ordner);
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    o^.ordnerInhalt[i] := "";
  END;
  o^.anzahlTexte := 0;
  RETURN o;
END Anlegen;
```

Diskussion : ADT-Realisierung in Modula-3

- **Wir können feststellen**
  - Als Typ für einen ADT müssen wir einen opaken Referenztyp wählen.
  - Grund: So kann der Übersetzer für Objekte eines ADTs ausreichend Speicherplatz reservieren, ohne den inneren Aufbau des Typs zu kennen.
- **Konsequenz:**
  - Es können neben den Operationen der Schnittstelle des ADTs auch die Operationen
    - Zuweisung und
    - Vergleich auf Objekten des ADTs durchgeführt werden (Pointer-Zuweisung und Pointer-Vergleich).
  - Diese sollten jedoch nicht genutzt werden!
  - Das Brandzeichen verhindert, daß einem Objekt eines ADT zufälligerweise ein Wert eines strukturell gleichen Typs zugewiesen werden kann.

Zuweisung und Vergleich von ADT-Objekten

```
ordner1 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner1, "Nicht immer sind bequeme Stuehle ...");

ordner2 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner2, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner2, "Nicht immer sind bequeme Stuehle ...");

IF ordner1 = ordner2 THEN
  SIO.PutLine ("1 ordner sind gleich");
ELSE
  ordner2 := ordner1;
  OrdnerADT.Beschrifte (ordner1, "Romane");
END;
IF ordner1 = ordner2 THEN
  SIO.PutLine ("nach Zuweisung sind die Ordner gleich");
END;

SIO.PutLine (OrdnerADT.GibBeschriftung(ordner1));
SIO.PutLine ("----------------------------");
SIO.PutLine (OrdnerADT.GibBeschriftung(ordner2));
```

Vergleich der Referenzen (Adressen

ordner2 zeigt auf die selbe Adresse wie ordner1

Ändert ordner1 und ordner2

Modul als Entwicklungseinheit

- **Ein Modul ist charakterisiert durch eine *Entwurfs-entscheidung*.**
  - (z.B. wir wollen Ordner verarbeiten können)
- **Jedes Modul basiert auf genau einer Entwurfsentscheidung.**
- **Module *verbergen* Implementierungsentscheidungen.**
  - Jedes Modul hat sein Geheimnis.
- **Es werden immer die Entscheidungen**
  - in einem Modul gekapselt, die vielleicht ***revidiert*** werden müssen als andere.
  - z.B. Datenaufbau
- **Der Änderungsaufwand muß möglichst immer auf ein Modul begrenzt werden.**

Zusammenfassung ADT

- **In imperativen Sprachen**
  - mit einem ***Modulkonzept***
  - realisieren wir abstrakte Datentypen mit einem ADT-Modul.
- **Ein ADT-Modul zeigt folgende Eigenschaften:**
  - Das Modul liefert ***Exemplare einer Datenstruktur***, die beim Kunden aufbewahrt werden.
  - Dadurch können ***mehrere Exemplare*** eines ADT-Moduls bearbeitet werden.
  - Die Datenstruktur ist nur als ***Verweis*** auf die interne Repräsentation bekannt.
  - Die Repräsentation selbst ist ***verborgen*** und ***austauschbar***.
  - Die Datenstrukturen des ADT-Moduls können (fast) nur über die ***exportierten Operationen*** des Moduls bearbeitet werden.

Was haben wir gelernt?

- **Modularisierung**
  - Mittel, um ***handhabbare*** Programmeinheiten zu konstruieren
  - Module bestehen aus ***Schnittstelle*** und ***Implementierung***
- **Information Hiding**
  - Ziel:
    - Implementierung wesentlicher Details ist ***nicht*** nach ***außen sichtbar,*** insbesondere anwendbar auf Datenstrukturierung
  - Objektmodul:
    - Es wird in einem Modul ***ein Objekt*** gemäß Information Hiding realisiert, Datenabstraktion für ein Objekt
  - Abstrakter Datentyp:
    - Es wird ein ***geschützter Typ*** realisiert. Objekte des ADTs können nur mit den Operationen des ADTs manipuliert werden, Datenabstraktion für eine „Klasse“ von Objekten

Glossar

- **Information Hiding: Prozeß- oder funktionale Abstraktion, Datenabstraktion, funktionaler Module, Datenobjektmodul (Kapsel), abstrakter Datentyp**
- **opake Datenstruktur, opake (geschützte, abstrakte, geheime) Datentypen**
- **Schnittstellenoperationen eines Datenabstraktionsbausteins: Initialisierung (Löschung), Lesen, Verändern, Sicherheitsoperationen**
- **Realisierung opaker Datentypen durch Verweistypen**
- **Typname, Signatur, Axiome, Vorbedingungen eines ADT**
- **Subtypen in Modula-3 haben Referenzsemantik (keine Zuweisung und keine Gleichheitsabfrage nutzen!)**

Teil IV

Grobstrukturierung eines Programms

- Modularisierung und Module
- vordefinierte Bausteine (Bsp. Dateien)
- Datenabstraktion
- Objektorientierung
- Beispiel-Programmentwicklung

Modularisierung und Module

- **Modulkonzept**
  - Export-Schnittstelle
  - Import-Schnittstelle
- **Modulrumpf**
- **Austausch der Implementierung**
- **Diskussion modularer Programme**

Vorteile modularer Programme

- **Module können von *unterschiedlichen* Personen entwickelt und gepflegt werden.**
  - Software-Entwicklung ist Team-Arbeit!
- **Module können einzeln *getestet* werden.**
  - Test großer Programme ist extrem aufwendig!
- **Module können geordnet zum Gesamtsystem *integriert* werden.**
- **Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung *ersetzt* werden.**
  - z.B. durch eine effizientere Implementierung
- **Module können in verschiedenen Programmen *wiederverwendet* werden (Modul-Bibliothek).**
  - Dies senkt die Kosten für die Entwicklung!

Module

- **neuere imperative Sprachen sehen Module vor**
- **Entstanden aus der Notwendigkeit,**
  - große Programmtexte in für den ***Übersetzer faßliche Einheiten*** zu zerlegen,
  - Modulkonzept ist zum zentralen ***Organisationskonzept*** für Entwürfe und Programmtexte geworden.
- **Module werden hier als**
  - *Konstruktionshilfsmittel* der Sprache eingeführt.
- **Die Diskussion,**
  - wie das Modulkonzept genutzt werden sollte, folgt in einem eigenen Kapitel.

Definition: Modul

- **programmiersprachliche Definition:**
  - Ein Modul ist die ***Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren*** zu einer Einheit. Soll ein Modul von einem anderen benutzt werden, so muß man angeben, welche Teile der Schnittstelle dieses Moduls von ***außen sichtbar*** sein sollen und welche nicht. Grundsätzlich bleibt aber die Implementierung eines Moduls, also die konkrete Realisierung der Datentypen und Prozedurrümpfe, vor allen anderen Modulen ***verborgen***.
- **methodische Definition:**
  - Unter einem Modul verstehen wir eine ***Sammlung von Objekten und Algorithmen*** mit der Eigenschaft, daß ihre Kommunikation mit der Außenwelt nur über eine klar ***definierte Schnittstelle*** erfolgt. Das ***Zusammensetzen*** mehrerer Module zu einer Gesamtlösung darf keine Kenntnis ihres ***inneren Aufbaus*** voraussetzen, und die Korrektheit eines Moduls muß ohne Kenntnis seiner Einbettung in die Gesamtlösung nachprüfbar sein.

Erinnerung: Lebensdauer

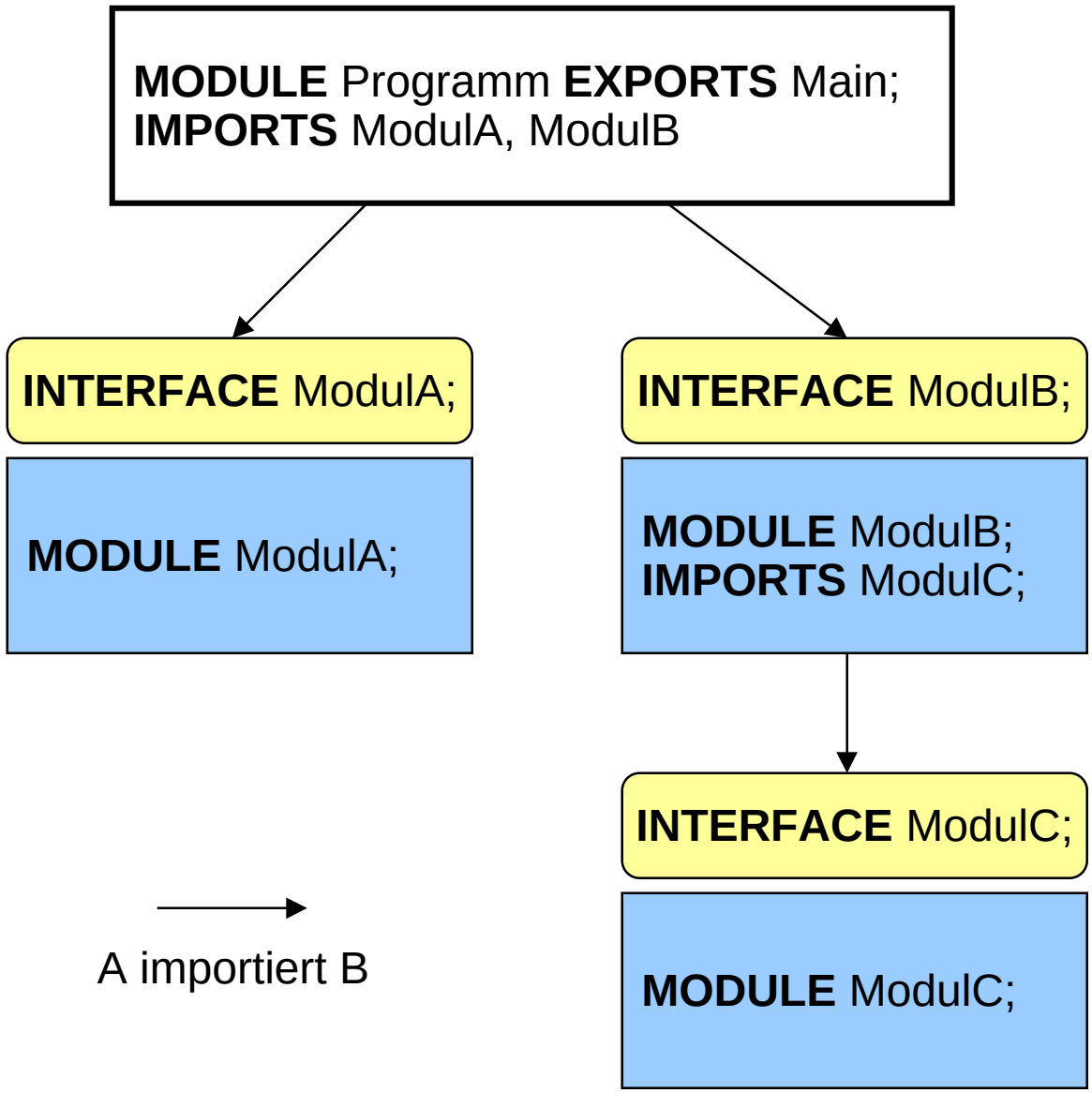
- **Prozedur:**
  - Alle Objekte im Namensraum einer Prozedur existieren nur solange die Prozedur aktiv ist.
  - Bei jedem neuen Aufruf einer Prozedur werden u.a. die lokalen Variablen neu angelegt.
- **Modul:**
  - Module sind ***statisch***, d.h. ihr Namensraum existiert solange das Programm oder die Anwendung ***insgesamt*** aktiv ist.
  - Variablen, die im Deklarationsteil eines Moduls eingeführt werden, haben die gleiche Lebensdauer wie das Modul; sie heißen ***global***.
- **In Modula-3**
  - können Module ***nicht*** ineinander geschachtelt werden!
  - Bei Prozeduren ist dies möglich!

Aufbau von Modula-3 Programmen

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***
- **Modul-Aufbau**
  - ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Schnittstelle*** (interface)
      - definiert, was ein Modul exportiert
      - Exportschnittstelle
    - ***Implementierung*** (body, Rumpf)
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Bisher**
  - bestanden unsere Programme lediglich aus einem Modul, dem Hauptmodul und weiteren Prozeduren

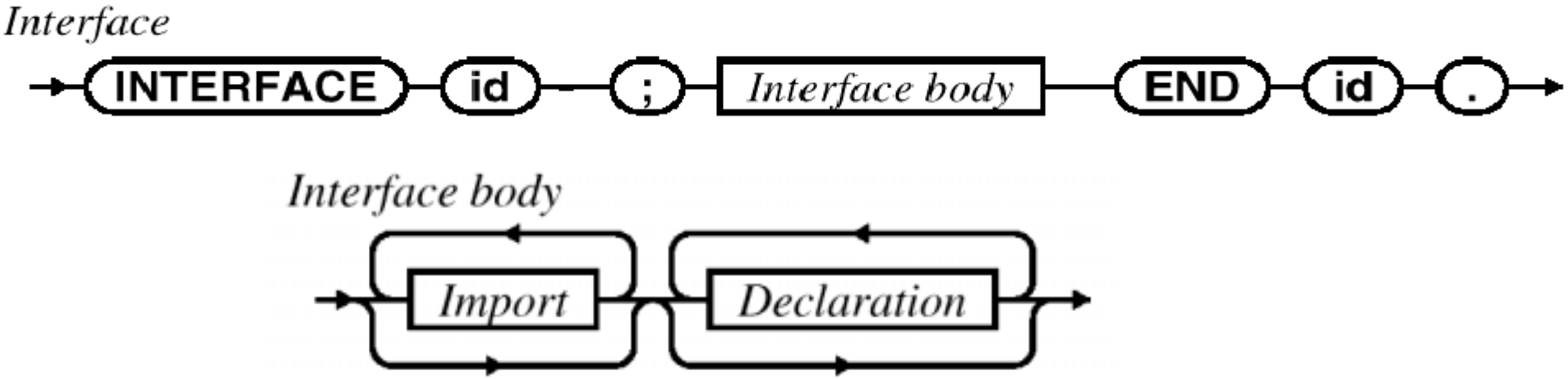
Modul-Hierarchie

- ein Modul kann Elemente anderer Module ***benutzbar machen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar



Die Export-Schnittstelle

- **Wie sieht eine Export-Schnittstelle eines Moduls aus?**



- **Alle in der Schnittstelle deklarierten Objekte werden vom Modul *exportiert***
  - können von importierenden Modulen ***verwendet*** werden.
- **Hinweis**
  - alles was in der Schnittstelle "***versprochen***" wird, muß auch von der Implementierung realisiert werden

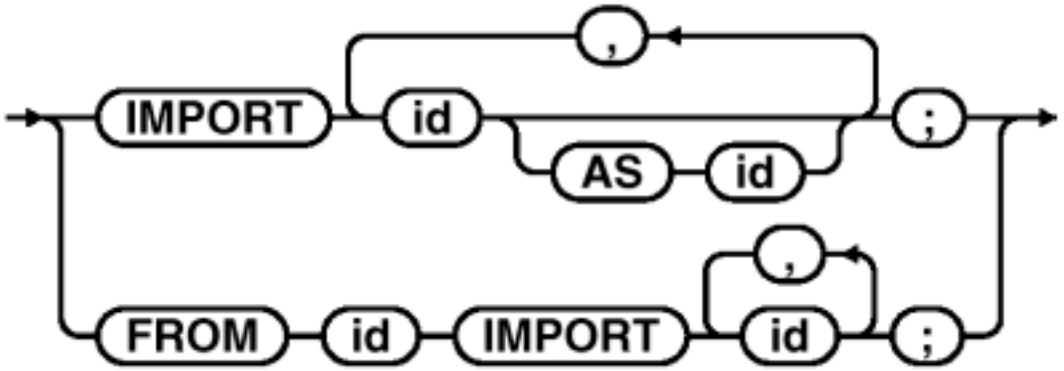
Beispiel: Export-Schnittstelle

```
INTERFACE SIO;

IMPORT Fmt, Rd, Wr, Word;

...

PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};
(* Read next character from stream rd and return it. *)

PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL);
(* Write ch to outputstream wr. *)

PROCEDURE GetText(rd: Reader := NIL; len: CARDINAL): TEXT;
(* Read a sequence of len characters from rd and return them. If there are not
enougth characters return what is there. *)

PROCEDURE PutText(t: TEXT; wr: Writer := NIL);
(* Write character sequence t to outputstream wr. *)

PROCEDURE GetLine(rd: Reader := NIL): TEXT RAISES {Error};
(* Read a full line of text terminated by the next RETURN from
   inputstream rd and return it (without RETURN!). *)
. . .
END SIO.
```

IMPORT-Schnittstelle - 1

- **Um Programmobjekte über Modulgrenzen zu verwenden,**
  - müssen sie ***gezielt angefordert*** werden.
  - Die IMPORT-Klausel dient dazu,
    - die Dienste in einem anderen Modul ***sichtbar*** zu machen.
- **Mögliche IMPORT-Varianten:**
  - importieren ***aller*** Dienste eines Moduls
  - importieren ***aller*** Dienste eines Moduls unter einem ***Alias-Namen***
  - importieren nur der Dienste eines Moduls, die ***tatsächlich*** vom importierenden Modul verwendet werden

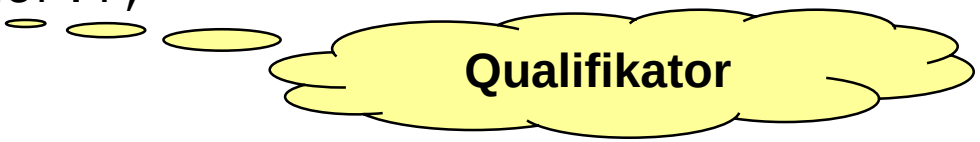
Import



IMPORT-Klausel - 2

- **Beispiele für Importe:**
  - wird nicht selektiv importiert, muß der Modulname als ***Qualifikator*** verwendet werden

```
IMPORT Text;                          (* import aller Deklarationen *)
IMPORT Rd AS Reader;                  (* import mit Alias-Namen *)
FROM SIO IMPORT                       (* selektives Importieren *)
  (*PROCS*) PutLine, PutText, GetChar;
…
VAR eingabestrom : Reader.T;
…
n := Text.Length(t1);
…
PutText("abcdefg");
```



Diskussion: Import-Varianten

Globales Importieren

An ***jeder Stelle*** im Programmtext ist ersichtlich, wo das jeweilige Objekt deklariert ist.

```
Text.Length(t1);   Length(m3);   List.Length(l1);
```

Identisch deklarierte Bezeichner verschiedener Module können verwendet werden.

Zum Teil erheblich ***längere Schreibweise.***

Erst mit einem Werkzeug kann einfach ermittelt werden, was ***tatsächlich*** alles von einem Modul verwendet wird.

Selektives Importieren

- ***kürzere*** Schreibweise
- Es ist alles das, was verwendet wird, auch ***explizit angegeben***
- Dies erhöht die Änderbarkeit

  Es ist nicht direkt ersichtlich, wo her ein Dienst kommt.

Regeln

- **1. Schnittstelle und Implementierung eines Moduls sind in *unterschiedlichen* Dateien (Programmtextdatei) enthalten.**
  - Jede Programmtextdatei bildet eine ***Übersetzungseinheit*** und kann vom Übersetzer getrennt behandelt werden.
- **2. Import**
  - alle Dienste: Qualifikation zeigt Herkunft des Dienstes
  - selektiver Import: Erhöht die Änderungsfreundlichkeit
- **3. Zyklische Importe sind verboten!**
  - A importiert B, B importiert A
  - zyklischer Import ist ein Hinweis auf eine ***schlechte Modularisierung***

Import für Schnittstelle oder für Rumpf

- **Import zur Schnittstellendefinition**
  - z.B. Typ für Deklaration von Formalparameter oder Ergebnis
  - Konstante für Vorberechnung von Parametern

- **Import für Rumpfrealisierung**
  - z.B. Typ für die Realisierung einer modulrumpflokalen Variablen
  - Prozedur/Funktion als Hilfe für die Implementierung einer Schnittstellenoperation oder des Anweisungsteils

Modul-rumpf

Implementierung eines Moduls

- **Regel:**
  - Die Implementierung einer Schnittstelle realisiert ***alle*** in der Schnittstelle ***deklarierten*** Prozeduren (Funktionen).
- **EXPORTS-Klausel**
  - gibt an, welche Schnittstelle ein Modul realisiert

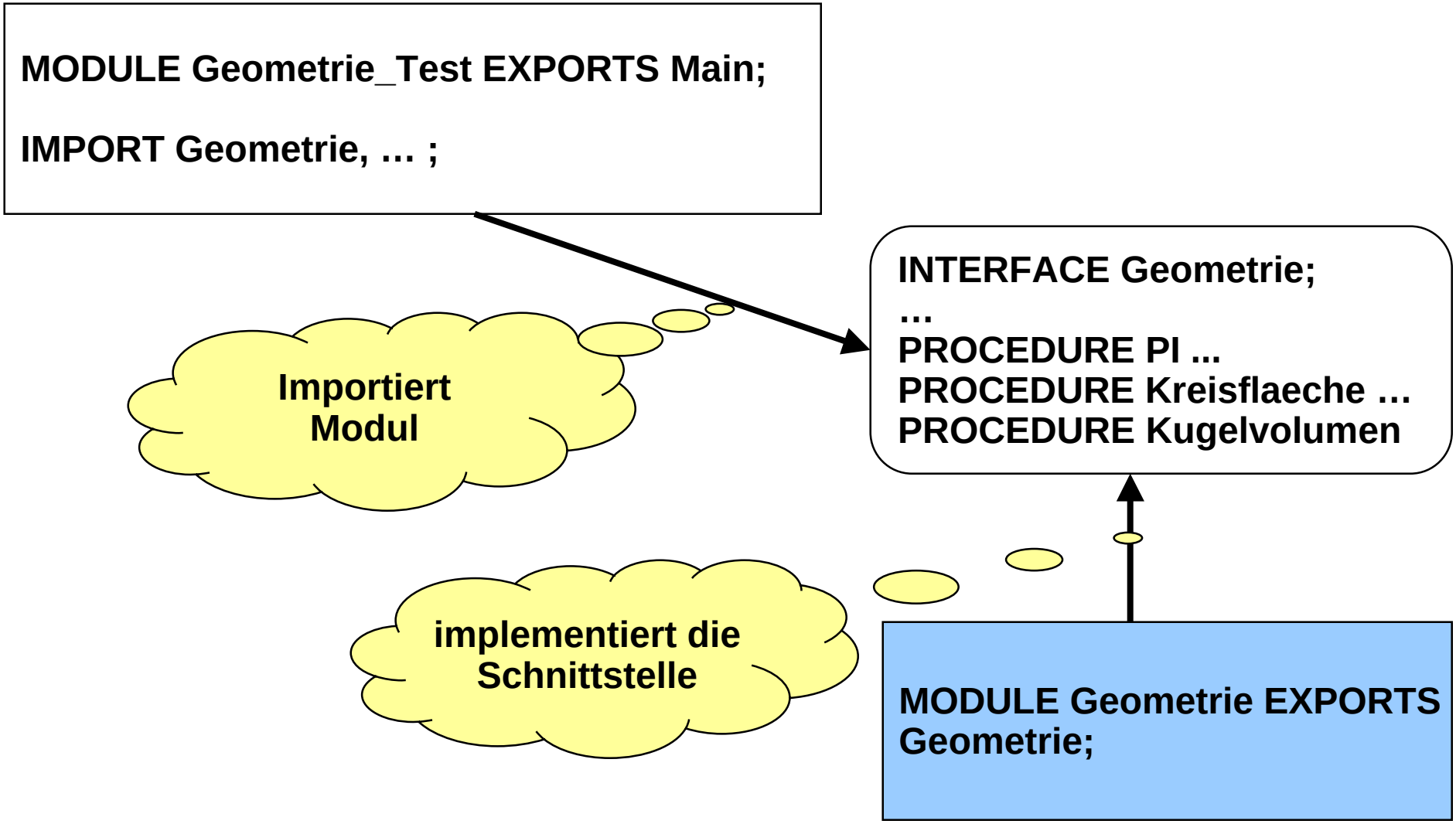
```
MODULE Geometrie EXPORTS Geometrie;
```

  - Fehlt die EXPORTS-Klausel, dann realisiert das Modul eine Schnittstelle ***gleichen Namens.***

```
MODULE Geometrie;
```

- **Modul-Initialisierung**
  - Im ***Block*** eines Moduls wird das Modul initialisiert.
  - Regel: Ein importiertes Modul wird vor dem importierenden Modul initialisiert.

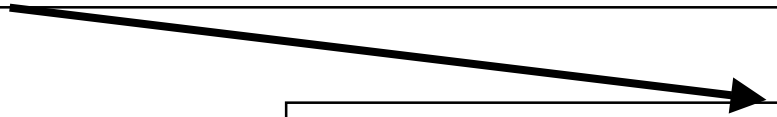
Beispiel


MODULE Geometrie_Test EXPORTS Main;
IMPORT Geometrie, … ;
INTERFACE Geometrie;
…
PROCEDURE PI ...
PROCEDURE Kreisflaeche …
PROCEDURE Kugelvolumen
Importiert Modul
implementiert die Schnittstelle
MODULE Geometrie EXPORTS Geometrie;

Modul-rumpf

Beispiel: Schnittstelle

```
MODULE Geometrie_Test EXPORTS Main;
IMPORT Geometrie, SIO;

VAR radius : REAL;
BEGIN
  SIO.PutText ("Geben Sie bitte einen Radius ein: ");
  radius := SIO.GetReal();
  SIO.PutText ("Kreisflaeche: ");
  SIO.PutReal (Geometrie.Kreisflaeche(radius));
  SIO.Nl();
  SIO.PutText ("Kugelvolumen: ");
  SIO.PutReal (Geometrie.Kugelvolumen(radius));
END Geometrie_Test.
```



```
INTERFACE Geometrie;

PROCEDURE PI () : REAL;
PROCEDURE Kreisflaeche(r :REAL): REAL;
PROCEDURE Kugelvolumen(r : REAL): REAL;

END Geometrie.
```

Modul-rumpf

Beispiel: Implementierung

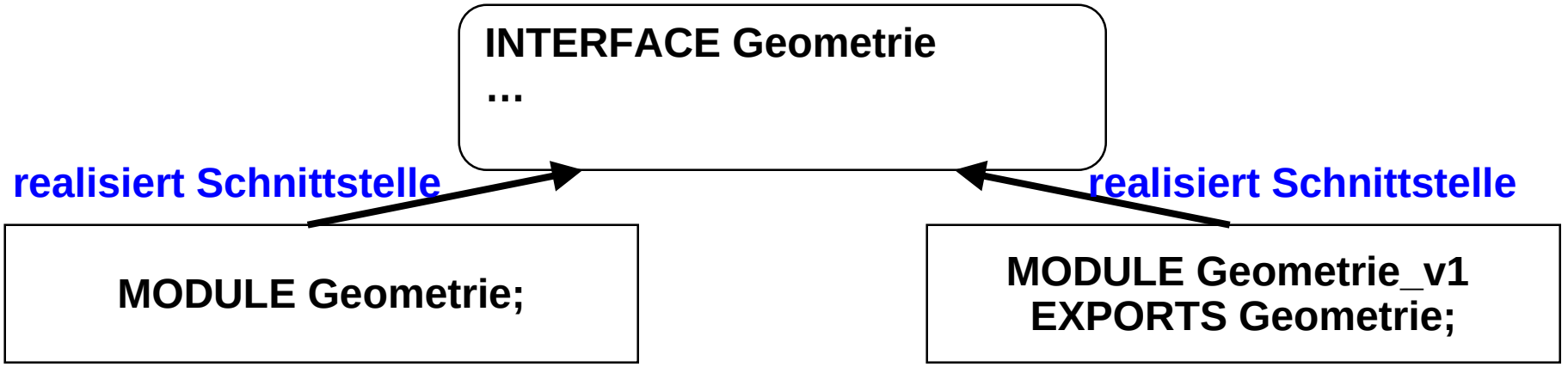
```
MODULE Geometrie EXPORTS Geometrie;

VAR pi : REAL;

PROCEDURE PI () : REAL =
BEGIN
  RETURN pi;
END PI;

PROCEDURE Kreisflaeche(radius :REAL): REAL =
BEGIN
  RETURN (pi * radius * radius);
END Kreisflaeche;

PROCEDURE Kugelvolumen(radius : REAL): REAL =
BEGIN
  RETURN ( (4.0/3.0) * pi * radius * radius * radius );
END Kugelvolumen;

BEGIN
  pi := 3.14147;
END Geometrie.
```

Interne Variable

Realisierung der Prozeduren / Funktionen der Schnittstelle

Initialisierung der Moduls

Zwei Rümpfe für die gleiche Schnittstelle




```
% Makefile ...
…

% module(Geometrie)

interface(Geometrie)
implementation (Geometrie_v1)

implementation (Geometrie_Test)
program (Geometrie_Test)
```

```
MODULE Geometrie_v1 EXPORTS Geometrie;

PROCEDURE PI () : REAL =
BEGIN
  RETURN 3.14147;
END PI;

PROCEDURE Kreisflaeche(r :REAL): REAL = ...

PROCEDURE Kugelvolumen(r : REAL): REAL = ...
BEGIN
END Geometrie_v1.
```

Vorteile modularer Programme

Vorteile (Erinnerung)

- Module von unterschiedlichen Personen entwickelt und gepflegt werden.
- Module können einzeln getestet werden.
- Module können geordnet zum Gesamtsystem integriert werden.
- Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung ersetzt werden.
- Module können in verschiedenen Programmen wiederverwendet werden (Modul-Bibliothek).

Modulkonzept

- **Module sind Sammlungen von Programmobjekten und Algorithmen:**
  - Sie sind keine ***direkt aufrufbaren*** Programmeinheiten (wie Prozeduren).
  - Sie sind eine Einheit für die ***Übersetzung***.
- **Als Sammlung sollen sie *keine* beliebige Anordnung sein**
  - Kriterien für die Zusammenstellung eines Moduls müssen geklärt werden.
  - Module verbergen ***Implementierungen*** d.h. ihren inneren Aufbau und zeigen nur ihre Schnittstelle:
  - Es gibt unterschiedlich ***starke Möglichkeiten*** des Verbergens.
  - Der Aufbau einer Schnittstelle unterliegt bestimmten Kriterien.
- **Module benutzen andere Module:**
  - Das Zusammenspiel verschiedener Module bezeichnet man auch als ***Architektur***.
  - Die ***Import-Beziehungen*** koppeln Module miteinander.

Was haben wir gelernt?

- Module als Sprachkonstrukt moderner imperativer Programmiersprachen
- Modul Schnittstelle - Rumpf, verschiedenen Rümpfe zu einer Schnittstelle
- Vorteile der Modularisierung bzgl. Qualität und Effizienz der Softwareerstellung bzw. bzgl. Qualität des resultierenden Programmsystems
- Modulhierachie über Importbeziehungen
- Implementierung eines Moduls: Realisierung der Dienste und Initialisierung

Glossar

- Exportschnittstelle, Importschnittstelle eines Moduls, Rumpf eines Moduls
- getrennte Übersetzung von Schnittstelle und Rumpf
- programmiersprachliche und methodische Definition eines Moduls
- statische Lebensdauer von Modulvariablen (immer im Rumpf)
- Import für Schnittstelle oder für Rumpf
- Schnittstellenimplementierung, Implementierung der Initialisierung
- schlechter Sprachgebrauch Interface (Module), (Implementation) Module

Teil IV

Grobstrukturierung eines Programms

- Modularisierung und Module
- vordefinierte Bausteine (Bsp. Dateien)
- Datenabstraktion
- Objektorientierung
- Beispiel-Programmentwicklung

Modularisierung und Module

- Modulkonzept
  - Export-Schnittstelle
  - Import-Schnittstelle
- Modulrumpf
- Austausch der Implementierung
- Diskussion modularer Programme

Modul-konzept

Vorteile modularer Programme

- **Module können von *unterschiedlichen* Personen entwickelt und gepflegt werden.**
  - Software-Entwicklung ist Team-Arbeit!
- **Module können einzeln *getestet* werden.**
  - Test großer Programme ist extrem aufwendig!
- **Module können geordnet zum Gesamtsystem *integriert* werden.**
- **Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung *ersetzt* werden.**
  - z.B. durch eine effizientere Implementierung
- **Module können in verschiedenen Programmen *wiederverwendet* werden (Modul-Bibliothek).**
  - Dies senkt die Kosten für die Entwicklung!

Modul-konzept

Module

- **neuere imperative Sprachen sehen Module vor**
- **Entstanden aus der Notwendigkeit,**
  - große Programmtexte in für den ***Übersetzer faßliche Einheiten*** zu zerlegen,
  - Modulkonzept ist zum zentralen ***Organisationskonzept*** für Entwürfe und Programmtexte geworden.
- **Module werden hier als**
  - *Konstruktionshilfsmittel* der Sprache eingeführt.
- **Die Diskussion,**
  - wie das Modulkonzept genutzt werden sollte, folgt in einem eigenen Kapitel.

Definition: Modul

- **programmiersprachliche Definition:**
  - Ein Modul ist die ***Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren*** zu einer Einheit. Soll ein Modul von einem anderen benutzt werden, so muß man angeben, welche Teile der Schnittstelle dieses Moduls von ***außen sichtbar*** sein sollen und welche nicht. Grundsätzlich bleibt aber die Implementierung eines Moduls, also die konkrete Realisierung der Datentypen und Prozedurrümpfe, vor allen anderen Modulen ***verborgen***.
- **methodische Definition:**
  - Unter einem Modul verstehen wir eine ***Sammlung von Objekten und Algorithmen*** mit der Eigenschaft, daß ihre Kommunikation mit der Außenwelt nur über eine klar ***definierte Schnittstelle*** erfolgt. Das ***Zusammensetzen*** mehrerer Module zu einer Gesamtlösung darf keine Kenntnis ihres ***inneren Aufbaus*** voraussetzen, und die Korrektheit eines Moduls muß ohne Kenntnis seiner Einbettung in die Gesamtlösung nachprüfbar sein.

Erinnerung: Lebensdauer

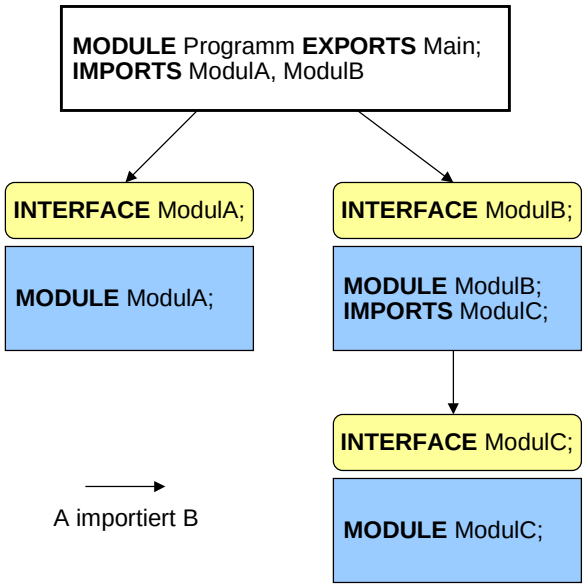
- **Prozedur:**
  - Alle Objekte im Namensraum einer Prozedur existieren nur solange die Prozedur aktiv ist.
  - Bei jedem neuen Aufruf einer Prozedur werden u.a. die lokalen Variablen neu angelegt.
- **Modul:**
  - Module sind ***statisch***, d.h. ihr Namensraum existiert solange das Programm oder die Anwendung ***insgesamt*** aktiv ist.
  - Variablen, die im Deklarationsteil eines Moduls eingeführt werden, haben die gleiche Lebensdauer wie das Modul; sie heißen ***global***.
- **In Modula-3**
  - können Module ***nicht*** ineinander geschachtelt werden!
  - Bei Prozeduren ist dies möglich!

Aufbau von Modula-3 Programmen

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***
- **Modul-Aufbau**
  - ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Schnittstelle*** (interface)
      - definiert, was ein Modul exportiert
      - Exportschnittstelle
    - ***Implementierung*** (body, Rumpf)
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Bisher**
  - bestanden unsere Programme lediglich aus einem Modul, dem Hauptmodul und weiteren Prozeduren

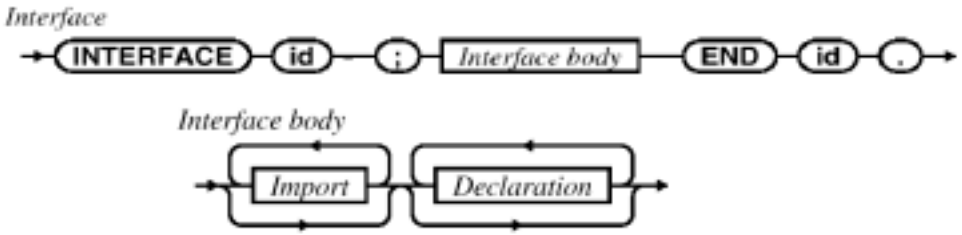
Modul-Hierarchie

- ein Modul kann Elemente anderer Module ***benutzbar machen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar



Die Export-Schnittstelle

- **Wie sieht eine Export-Schnittstelle eines Moduls aus?**



- **Alle in der Schnittstelle deklarierten Objekte werden vom Modul *exportiert***
  - können von importierenden Modulen ***verwendet*** werden.
- **Hinweis**
  - alles was in der Schnittstelle "***versprochen***" wird, muß auch von der Implementierung realisiert werden

Beispiel: Export-Schnittstelle

```
INTERFACE SIO;

IMPORT Fmt, Rd, Wr, Word;

...

PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};
(* Read next character from stream rd and return it. *)

PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL);
(* Write ch to outputstream wr. *)

PROCEDURE GetText(rd: Reader := NIL; len: CARDINAL): TEXT;
(* Read a sequence of len characters from rd and return them. If there are not
enougth characters return what is there. *)

PROCEDURE PutText(t: TEXT; wr: Writer := NIL);
(* Write character sequence t to outputstream wr. *)

PROCEDURE GetLine(rd: Reader := NIL): TEXT RAISES {Error};
(* Read a full line of text terminated by the next RETURN from
   inputstream rd and return it (without RETURN!). *)
. . .
END SIO.
```

IMPORT-Schnittstelle - 1

- **Um Programmobjekte über Modulgrenzen zu verwenden,**
  - müssen sie ***gezielt angefordert*** werden.
  - Die IMPORT-Klausel dient dazu,
    - die Dienste in einem anderen Modul ***sichtbar*** zu machen.
- **Mögliche IMPORT-Varianten:**
  - importieren ***aller*** Dienste eines Moduls
  - importieren ***aller*** Dienste eines Moduls unter einem ***Alias-Namen***
  - importieren nur der Dienste eines Moduls, die ***tatsächlich*** vom importierenden Modul verwendet werden

IMPORT-Klausel - 2

- **Beispiele für Importe:**
  - wird nicht selektiv importiert, muß der Modulname als ***Qualifikator*** verwendet werden

```
IMPORT Text;                          (* import aller Deklarationen *)
IMPORT Rd AS Reader;                  (* import mit Alias-Namen *)
FROM SIO IMPORT                       (* selektives Importieren *)
  (*PROCS*) PutLine, PutText, GetChar;
…
VAR eingabestrom : Reader.T;
…
n := Text.Length(t1);
…
PutText("abcdefg");
```

Diskussion: Import-Varianten

- **Globales Importieren**

  An ***jeder Stelle*** im Programmtext ist ersichtlich, wo das jeweilige Objekt deklariert ist.

  ```
  Text.Length(t1);   Length(m3);   List.Length(l1);
  ```

  Identisch deklarierte Bezeichner verschiedener Module können verwendet werden.

  Zum Teil erheblich ***längere Schreibweise.***

  Erst mit einem Werkzeug kann einfach ermittelt werden, was ***tatsächlich*** alles von einem Modul verwendet wird.

- **Selektives Importieren**
  - ***kürzere*** Schreibweise
  - Es ist alles das, was verwendet wird, auch ***explizit angegeben***
  - Dies erhöht die Änderbarkeit

    Es ist nicht direkt ersichtlich, wo her ein Dienst kommt.

Regeln

- **1. Schnittstelle und Implementierung eines Moduls sind in *unterschiedlichen* Dateien (Programmtextdatei) enthalten.**
  - Jede Programmtextdatei bildet eine ***Übersetzungseinheit*** und kann vom Übersetzer getrennt behandelt werden.
- **2. Import**
  - alle Dienste: Qualifikation zeigt Herkunft des Dienstes
  - selektiver Import: Erhöht die Änderungsfreundlichkeit
- **3. Zyklische Importe sind verboten!**
  - A importiert B, B importiert A
  - zyklischer Import ist ein Hinweis auf eine ***schlechte Modularisierung***

Modul-konzept

Import für Schnittstelle oder für Rumpf

- **Import zur Schnittstellendefinition**
  - z.B. Typ für Deklaration von Formalparameter oder Ergebnis
  - Konstante für Vorberechnung von Parametern
- **Import für Rumpfrealisierung**
  - z.B. Typ für die Realisierung einer modulrumpflokalen Variablen
  - Prozedur/Funktion als Hilfe für die Implementierung einer Schnittstellenoperation oder des Anweisungsteils

Modul-rumpf

Implementierung eines Moduls

- **Regel:**
  - Die Implementierung einer Schnittstelle realisiert ***alle*** in der Schnittstelle ***deklarierten*** Prozeduren (Funktionen).
- **EXPORTS-Klausel**
  - gibt an, welche Schnittstelle ein Modul realisiert

```
MODULE Geometrie EXPORTS Geometrie;
```

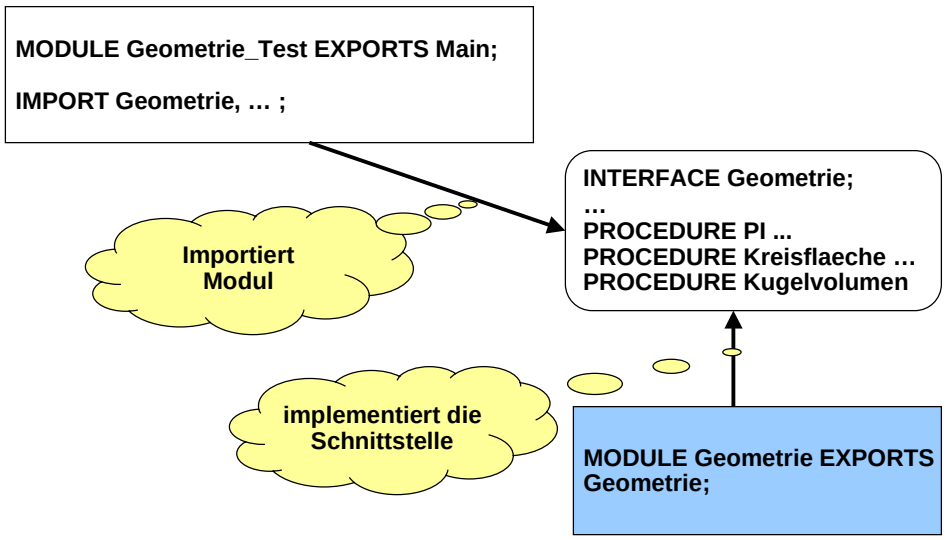
  - Fehlt die EXPORTS-Klausel, dann realisiert das Modul eine Schnittstelle ***gleichen Namens.***

```
MODULE Geometrie;
```

- **Modul-Initialisierung**
  - Im ***Block*** eines Moduls wird das Modul initialisiert.
  - Regel: Ein importiertes Modul wird vor dem importierenden Modul initialisiert.

Modul-
rumpf

Beispiel



Modul-
rumpf

Beispiel: Schnittstelle

```
MODULE Geometrie_Test EXPORTS Main;
IMPORT Geometrie, SIO;

VAR radius : REAL;
BEGIN
  SIO.PutText ("Geben Sie bitte einen Radius ein: ");
  radius := SIO.GetReal();
  SIO.PutText ("Kreisflaeche: ");
  SIO.PutReal (Geometrie.Kreisflaeche(radius));
  SIO.Nl();
  SIO.PutText ("Kugelvolumen: ");
  SIO.PutReal (Geometrie.Kugelvolumen(radius));
END Geometrie_Test.
```

```
INTERFACE Geometrie;

PROCEDURE PI () : REAL;
PROCEDURE Kreisflaeche(r :REAL): REAL;
PROCEDURE Kugelvolumen(r : REAL): REAL;

END Geometrie.
```

Modul-rumpf

Beispiel: Implementierung

```
MODULE Geometrie EXPORTS Geometrie;

VAR pi : REAL;

PROCEDURE PI () : REAL =
BEGIN
  RETURN pi;
END PI;

PROCEDURE Kreisflaeche(radius :REAL): REAL =
BEGIN
  RETURN (pi * radius * radius);
END Kreisflaeche;

PROCEDURE Kugelvolumen(radius : REAL): REAL =
BEGIN
  RETURN ( (4.0/3.0) * pi * radius * radius * radius );
END Kugelvolumen;

BEGIN
  pi := 3.14147;
END Geometrie.
```

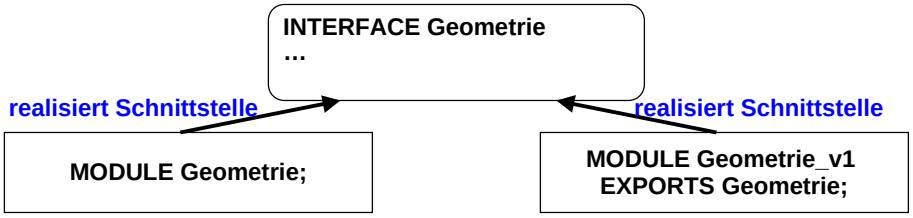
Interne Variable

Realisierung der Prozeduren / Funktionen der Schnittstelle

Initialisierung der Moduls

Austausch der Implementierung

Zwei Rümpfe für die gleiche Schnittstelle




```
% Makefile ...
…

% module(Geometrie)

interface(Geometrie)
implementation (Geometrie_v1)

implementation (Geometrie_Test)
program (Geometrie_Test)
```

```
MODULE Geometrie_v1 EXPORTS Geometrie;

PROCEDURE PI () : REAL =
BEGIN
  RETURN 3.14147;
END PI;

PROCEDURE Kreisflaeche(r :REAL): REAL = ...

PROCEDURE Kugelvolumen(r : REAL): REAL = ...
BEGIN
END Geometrie_v1.
```

Vorteile modularer Programme

- **Vorteile (Erinnerung)**
  - Module von unterschiedlichen Personen entwickelt und gepflegt werden.
  - Module können einzeln getestet werden.
  - Module können geordnet zum Gesamtsystem integriert werden.
  - Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung ersetzt werden.
  - Module können in verschiedenen Programmen wiederverwendet werden (Modul-Bibliothek).

Modulkonzept

- **Module sind Sammlungen von Programmobjekten und Algorithmen:**
  - Sie sind keine ***direkt aufrufbaren*** Programmeinheiten (wie Prozeduren).
  - Sie sind eine Einheit für die ***Übersetzung***.
- **Als Sammlung sollen sie *keine* beliebige Anordnung sein**
  - Kriterien für die Zusammenstellung eines Moduls müssen geklärt werden.
  - Module verbergen ***Implementierungen*** d.h. ihren inneren Aufbau und zeigen nur ihre Schnittstelle:
  - Es gibt unterschiedlich ***starke Möglichkeiten*** des Verbergens.
  - Der Aufbau einer Schnittstelle unterliegt bestimmten Kriterien.
- **Module benutzen andere Module:**
  - Das Zusammenspiel verschiedener Module bezeichnet man auch als ***Architektur***.
  - Die ***Import-Beziehungen*** koppeln Module miteinander.

Was haben wir gelernt?

- Module als Sprachkonstrukt moderner imperativer Programmiersprachen
- Modul Schnittstelle - Rumpf, verschiedenen Rümpfe zu einer Schnittstelle
- Vorteile der Modularisierung bzgl. Qualität und Effizienz der Softwareerstellung bzw. bzgl. Qualität des resultierenden Programmsystems
- Modulhierachie über Importbeziehungen
- Implementierung eines Moduls: Realisierung der Dienste und Initialisierung

Glossar

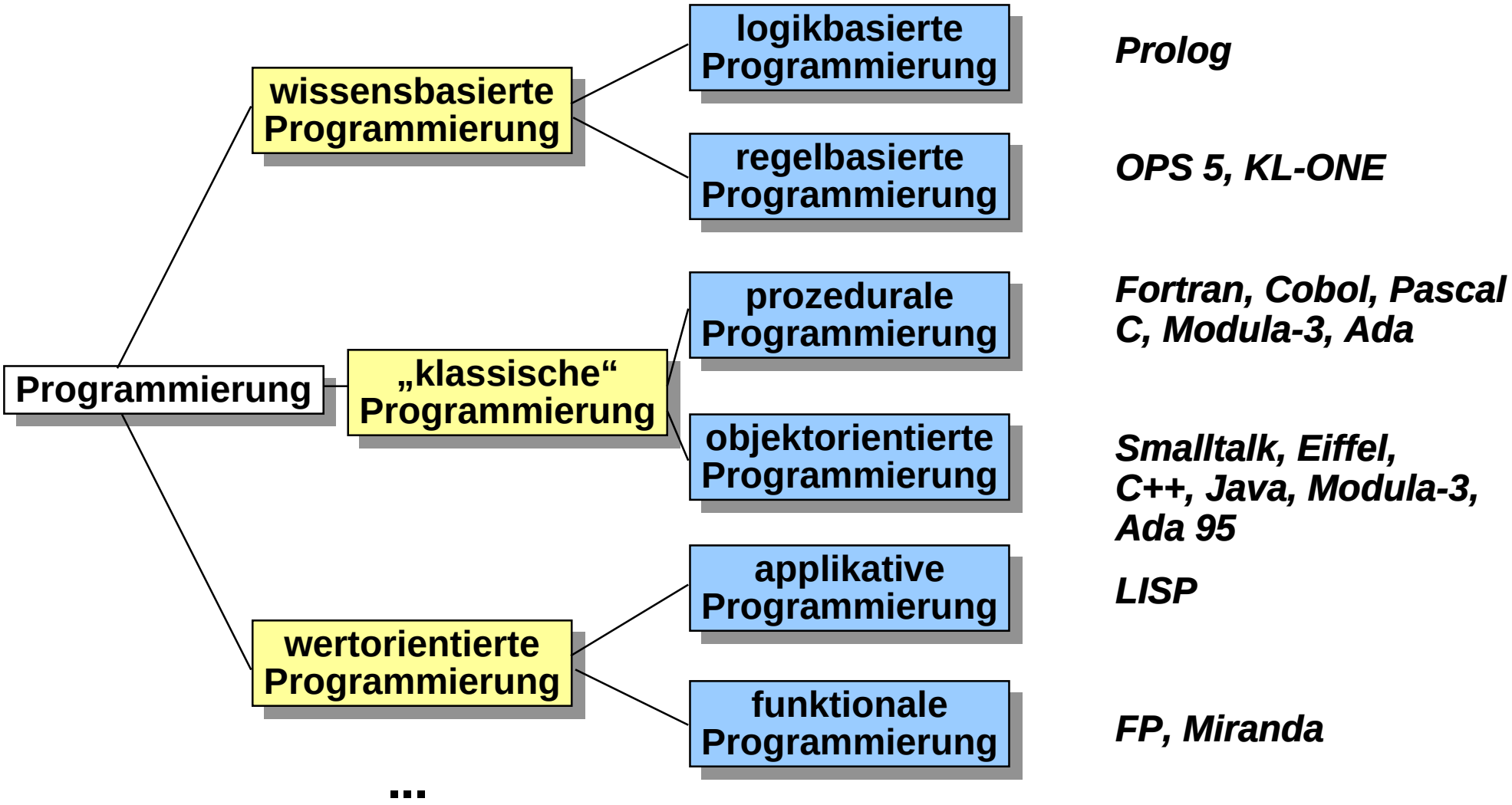
- Exportschnittstelle, Importschnittstelle eines Moduls, Rumpf eines Moduls
- getrennte Übersetzung von Schnittstelle und Rumpf
- programmiersprachliche und methodische Definition eines Moduls
- statische Lebensdauer von Modulvariablen (immer im Rumpf)
- Import für Schnittstelle oder für Rumpf
- Schnittstellenimplementierung, Implementierung der Initialisierung
- schlechter Sprachgebrauch Interface (Module), (Implementation) Module

Objektorientierte Programmierung I

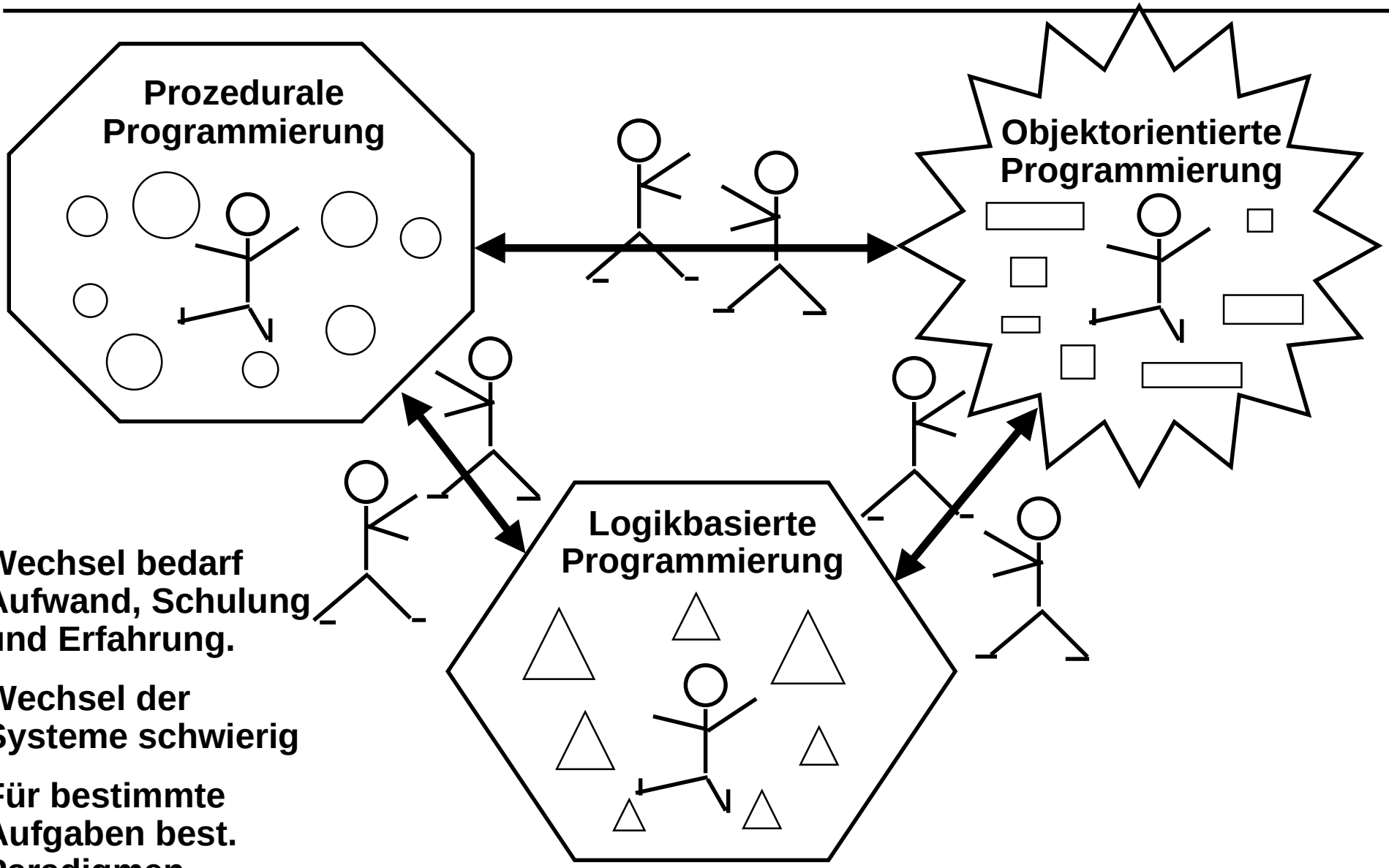
- Verständnis der objektorientierten Programmierung
- Objekte
- Klassen
- Vererbung
- Polymorphismus und dynamisches Binden
- Diskussion

Programmier-Paradigmen

- **Es gibt unterschiedliche Arten der Programmierung:**



Wechsel des Programmier-Paradigmas

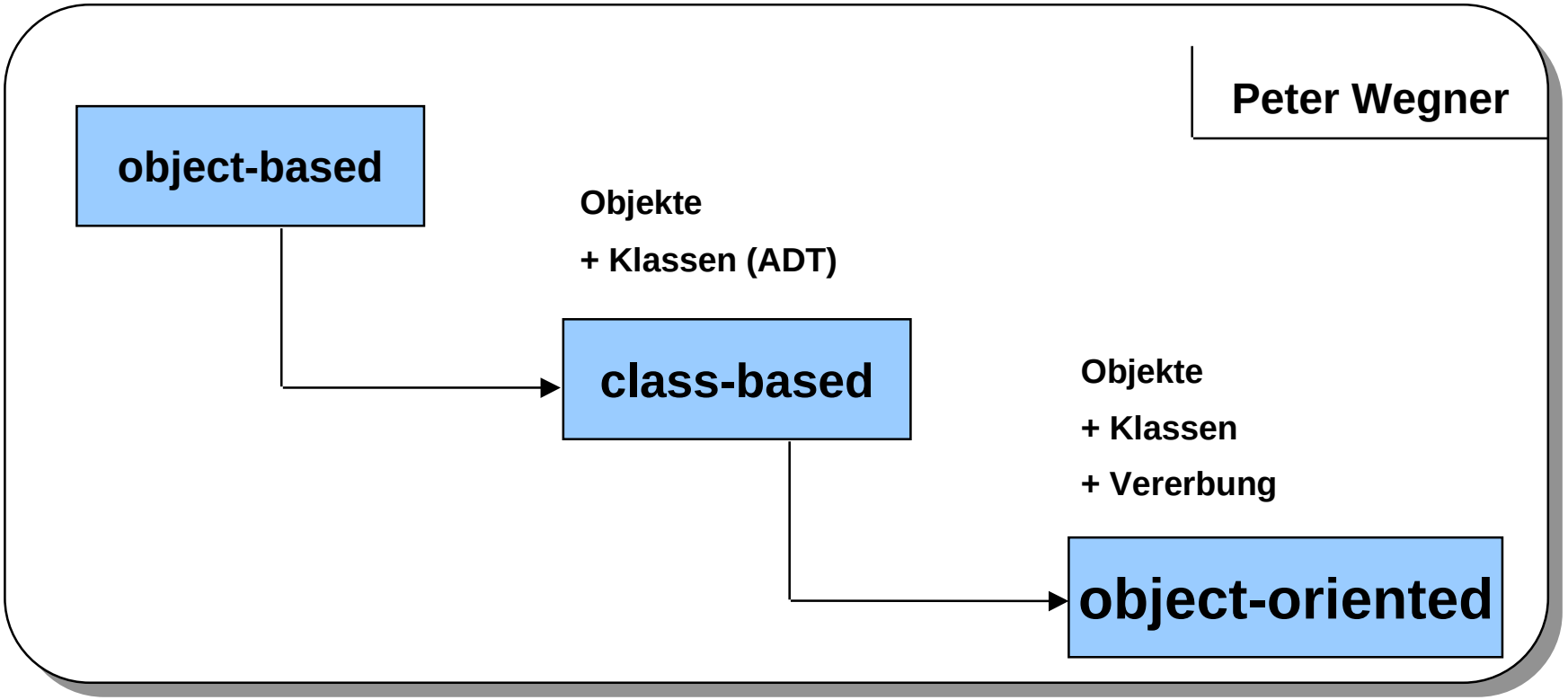


Wechsel bedarf Aufwand, Schulung und Erfahrung.

Wechsel der Systeme schwierig

Für bestimmte Aufgaben best. Paradigmen

Merkmale objektorientierter Systeme/Sprachen



Ada
Modula-2

CLU
(Ada 83)
(Modula-2)

Simula
Smalltalk-80
Eiffel
C++
Ada 95
Java
Modula-3

Klassifikation

Objektbasiert

- Objektbasierte Programmiersprachen bieten die Möglichkeit, **Objekte** im Sinne einer **Datenkapsel** resp. eines **Objektmoduls** zu realsieren.
- Jedes Objekt wir **einzeln** beschrieben und benutzt

Klassenbasiert

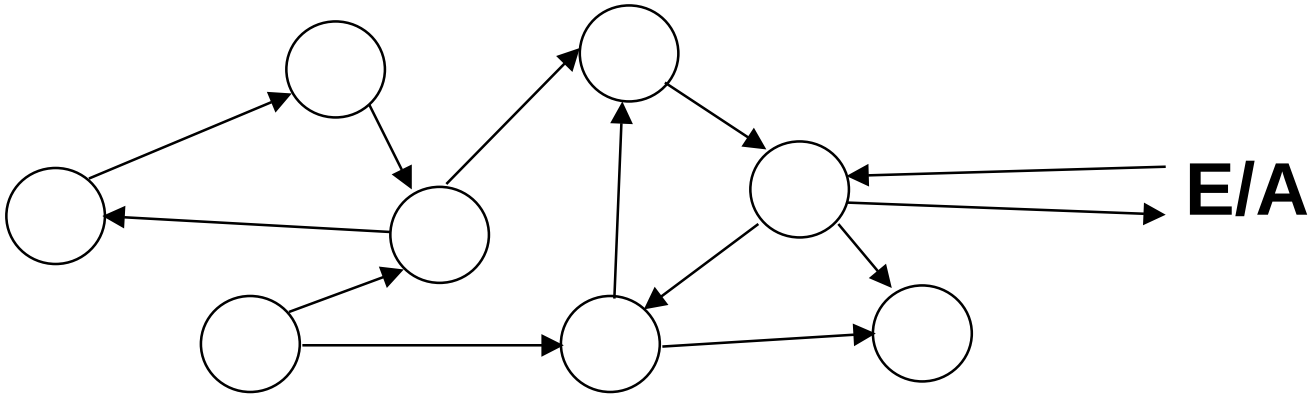
- Klassenbasierte Programmiersprachen bieten die Möglichkeit, Objekte in Forn von **Objekttypen (Klassen)** zu beschreiben.
- Von Objekttypen können beliebig viele **Exemplare** (Objekte des Typs) erzeugt werden.

Objektorientiert

- Objektorientierte Programmiersprachen erlauben, Objekttypen (Klassen) mithilfe der **Vererbungs-Beziehung** zu strukturieren.
- Dadurch lassen sich **Objekttyp-Hierarchien** im Sinne der Spezialisierung resp. Generalisierung modellieren.

Was ist ein rein objektorientiertes System

- **Ein Software-System besteht aus Objekten**
- **Jede Systemaktivität ist die Aktivität eines Objektes**
- **Objekte**
  - haben eine Lebensdauer (werden erzeugt und vernichtet)
  - sind identifizierbar
  - sind aktiv
  - verändern sich während ihrer Lebensdauer
  - senden und empfangen Nachrichten

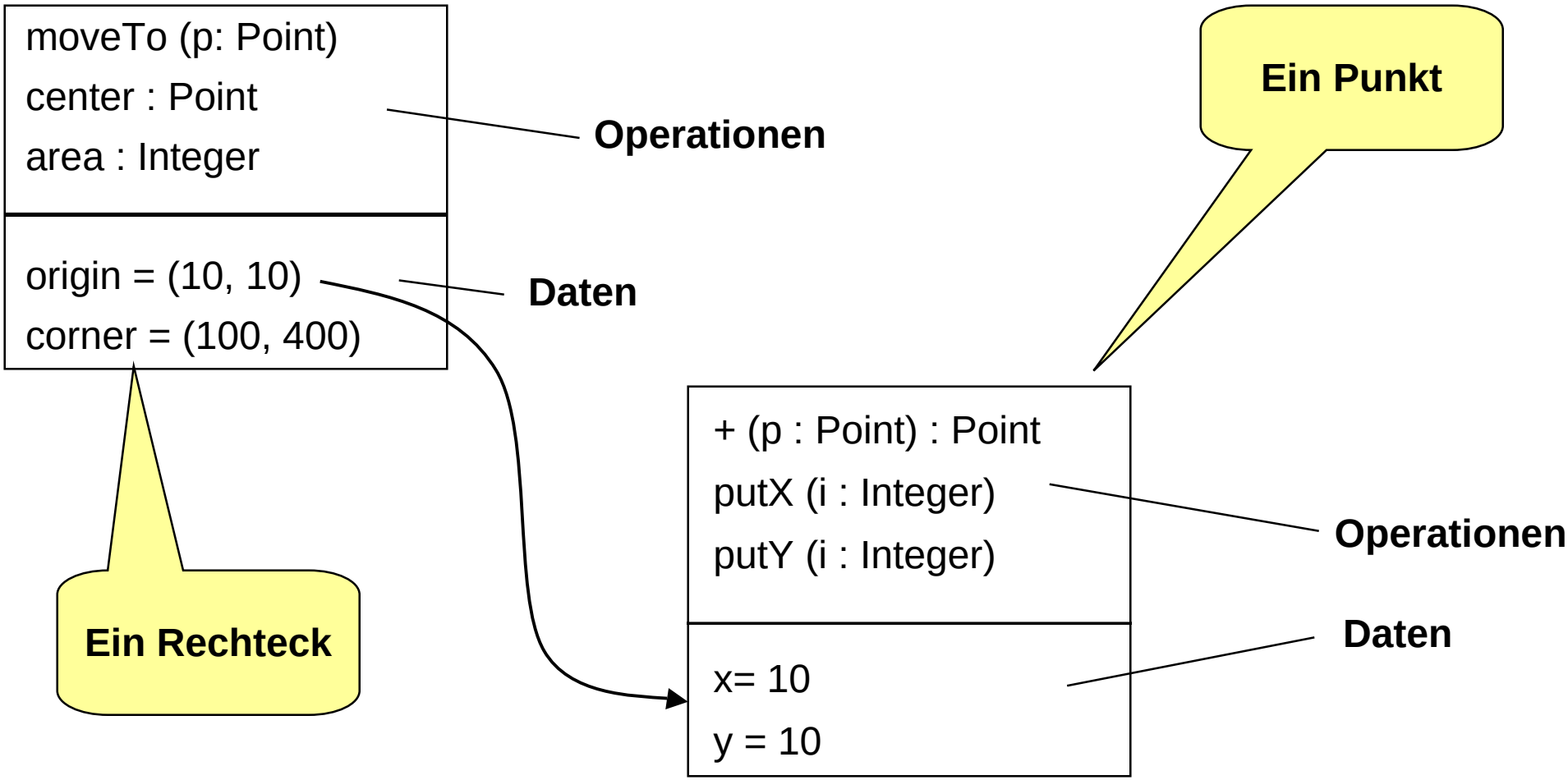


Objekte - die Dynamik des Systems

- **Ein Objekt ist eine Datenkapsel, die aus zwei Teilen besteht**
  - Der Wert der Daten repräsentiert den **Zustand** des Objekts
  - Daten können nur mithilfe von **Operationen** verändert werden (Kapselung)

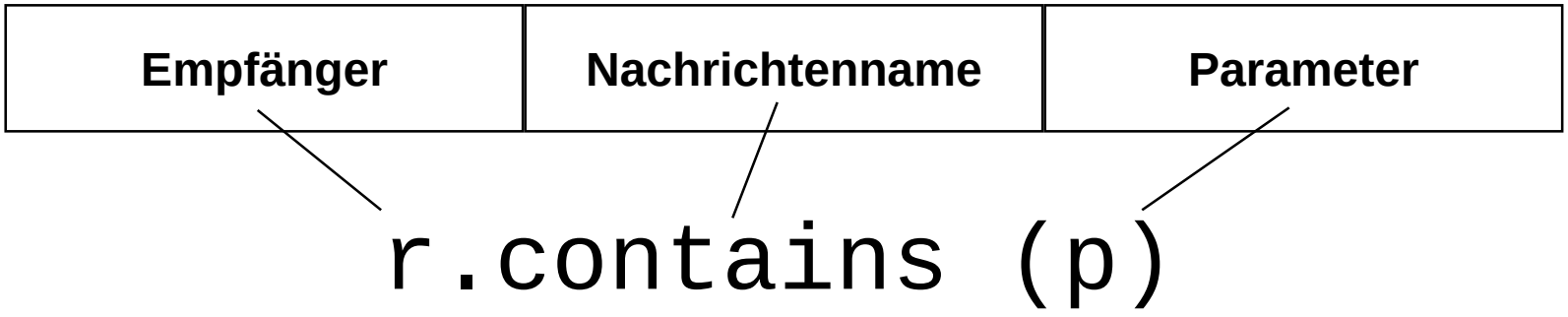
- **Die Aktivität der Objekte ist die Ausführung ihrer Operationen**
  - Eine Operation wird ausgeführt, wenn ein Objekt eine entsprechende **Nachricht** erhält.
  - Der Objektzustand kann sich **verändern**.
  - Mögliche Operationen sind durch den **Objekttyp** bestimmt.
- **Ein Objekt ist ein Exemplar genau einer Klasse.**
- **Objekte existieren nur zur Laufzeit**
  - können aber auch persistent gespeichert werden

Beispiele für Objekte



Nachrichten (Botschaften)

- **Objekte kommunizieren miteinander,**
  - dadurch daß sie **Nachrichten** versenden und Nachrichten empfangen.
- **Objekte reagieren auf Nachrichten,**
  - indem sie eine **Operation** (**Methode, Zugriffsunterprogramm**) ausführen.
- **Der Empfänger einer Nachricht (immer ein Objekt) ist verantwortlich für**
  - **Entschlüsselung** der Nachricht (verstehe ich die Nachricht?)
  - **Wirkung** (was tue ich?)



Klassen - die Statik des Systems

moveTo
origin = (10, 10)
corner = (100, 400)
...

moveTo
origin = (0, 0)
corner = (10, 40)
...

moveTo
origin = (40, 70)
corner = (50, 150)
...

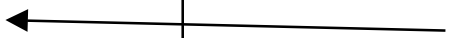
Einzelne Objekte

- **Gleichartige Objekte werden zusammengefaßt**
  - und an einer Stelle beschrieben (Objekttyp oder Klasse)
- **Eine Klasse definiert für ihre Objekte**
  - die **Speicherstruktur**
    - Daten, Attribute, Exemplarvariablen
  - die **Operationen** (Methoden, Routinen),
    - von denen ein Teil exportiert wird (exported, public <-> private)
    - andere werden nur intern benötigt
- **Von einer Klasse können beliebig viele Objekte erzeugt werden**

Beispiel : Klasse Point

```
class Point

feature
  x, y : Integer;

feature
  +: (p : Point) is
      result : Point;
    do
      result.x(x + p.x);
      result.y(x + p.y);
    end;
  x: (i : Integer) is
    do
      x := i;
    end;
  y: (i : Integer) is
    do
      y := i;
    end;
  ...
end -- class Rectangle
```

Exemplarvariablen (int. Verbundkomponente)



exportierte Operationen

Beispiel : Klasse Rechteck

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do ...
    end;

feature {NONE}
  extent : Point
    do
      -- liefert einen Punkt, der die Höhe und
      -- Weite des Rechtecks repräsentiert
    end;
end -- class Rectangle
```

Exemplarvariablen

exportierte Operationen

private Operationen

Klasse - Objekt

- **Eine Klasse entspricht einem ADT**
- **Ein Objekt "entspricht " einem abstarkten Datenobjekt**
  - d.h. die Datenimplementierung ist verborgen.
- **Von einer Klasse sind außerhalb sichtbar:**
  - der Klassenname
  - die exportierten Operationen
- **Jedes Objekt ist ein Exemplar**
  - einer Klasse des Programms
- **Objekte einer Klasse kennen dieselben Operationen**
- **Objekte einer Klasse unterscheiden sich in den Werten ihrer Daten**

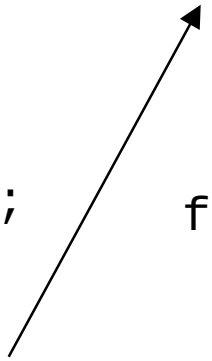
Objekte und Klassen

- Nach außen sichtbar ist

```
class Rectangle
interface
  Create;
  origin : Point;
  corner : Point;
  moveTo (p : Point);
  contains (p : Point)
   Boolean:
end -- class Rectangle
```

Zusammenhang Objekt <-> Klasse

```
r : Rectangle;
p : Point

p.Create;
r.Create;

r.contains (p);
```

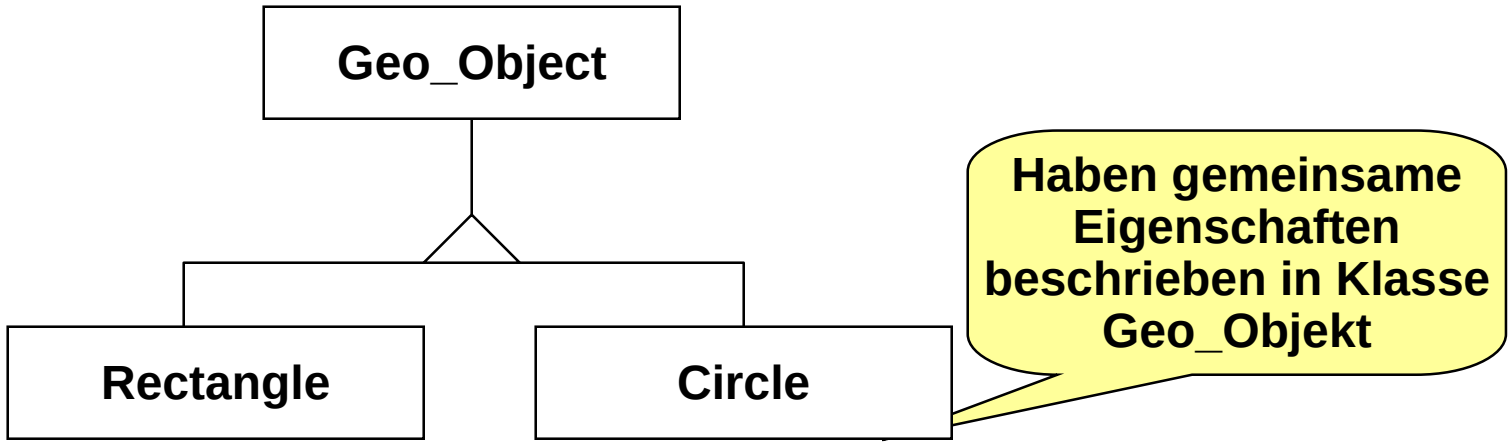


```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do
    end;
feature {NONE}

  extent : Point
    do
      -- liefert ...
    end;
end -- class Rectangle
```

Gemeinsame Eigenschaften, Spezialisierung

- **Gemeinsame Eigenschaften verschiedener Klassen K_1, K_2**
  - werden in einer **eigenen Klasse K** zusammengefaßt und definiert,
  - und anschließend an **K_1**, **K_2** vererbt.

- **Beispiel**



- **Regel:**
  - Eine Klasse A erbt von einer Klasse B genau dann, wenn A eine **Spezialisierung** (Unterbegriff) von B ist. Umgekehrt wird A die **Generalisierung** genannt.

Beispiel 1: Einfachvererbung

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

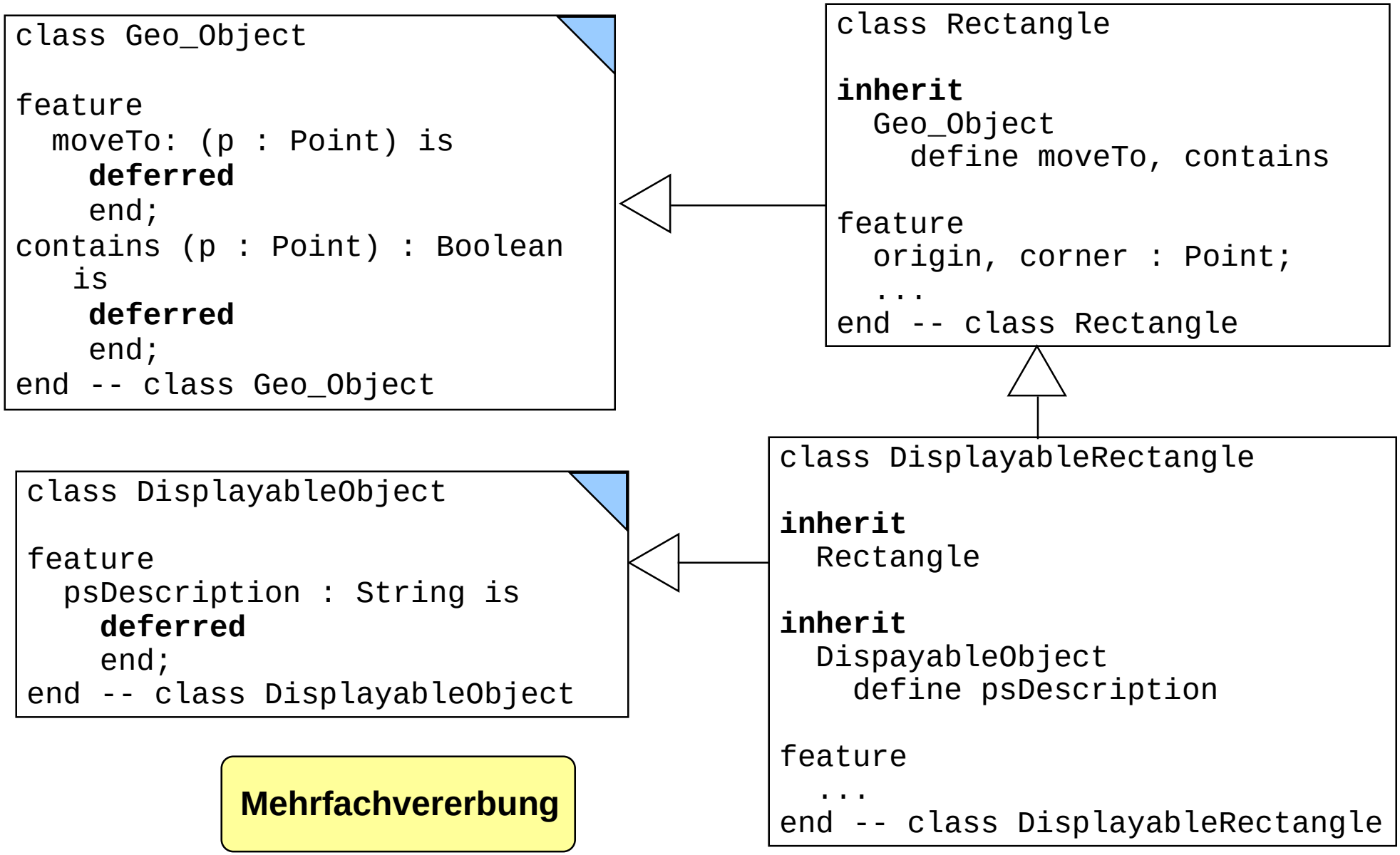
Abstrakte Klasse
Spezifikationsklasse
besitzt keine Objekt

Einfachvererbung

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

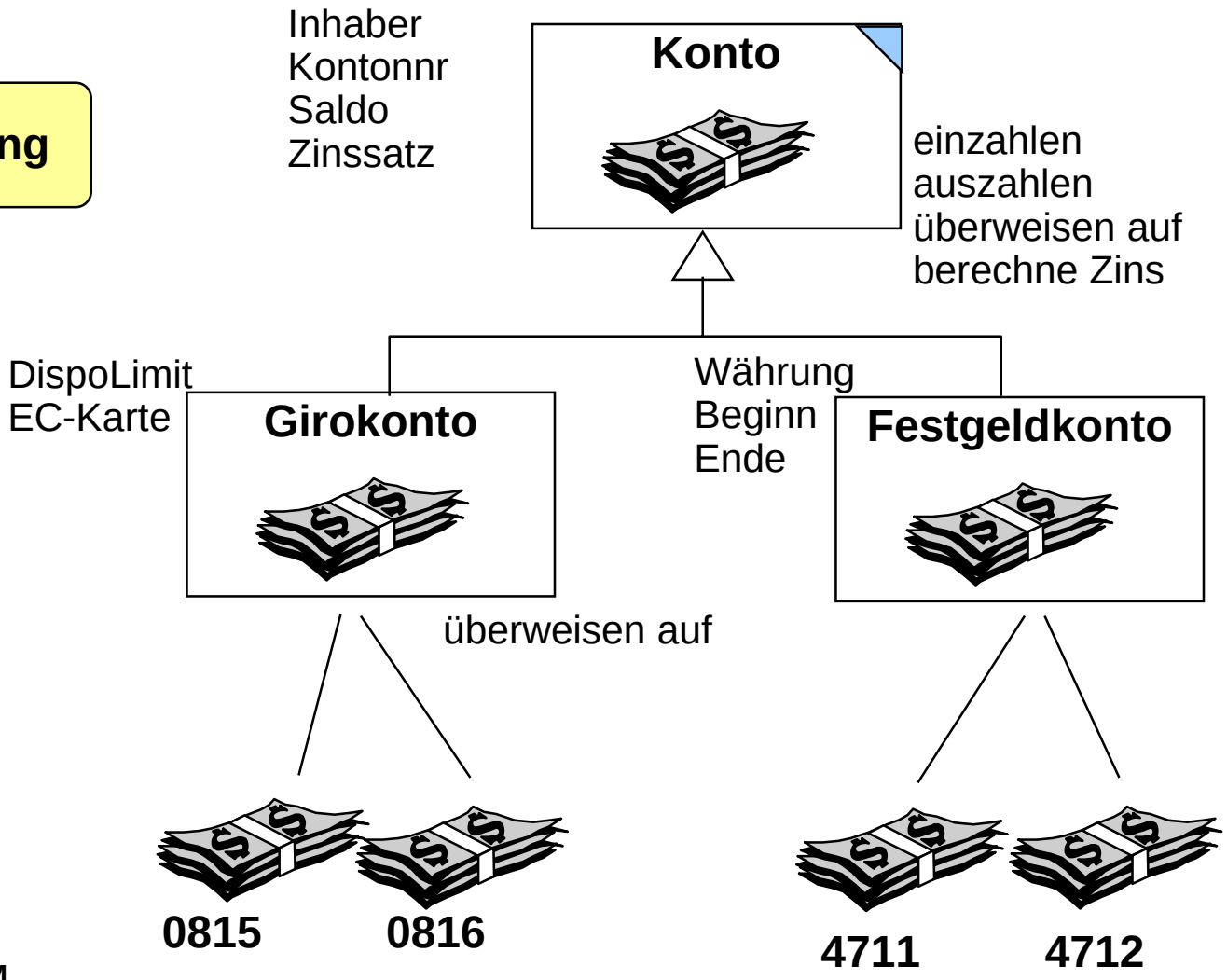
```
class Circle

inherit
  Geo_Object
    define moveTo, contains

feature
  center : Point;
  radius : Integer:
  ...
end -- class Circle
```

Beispiel 1: Mehrfachvererbung



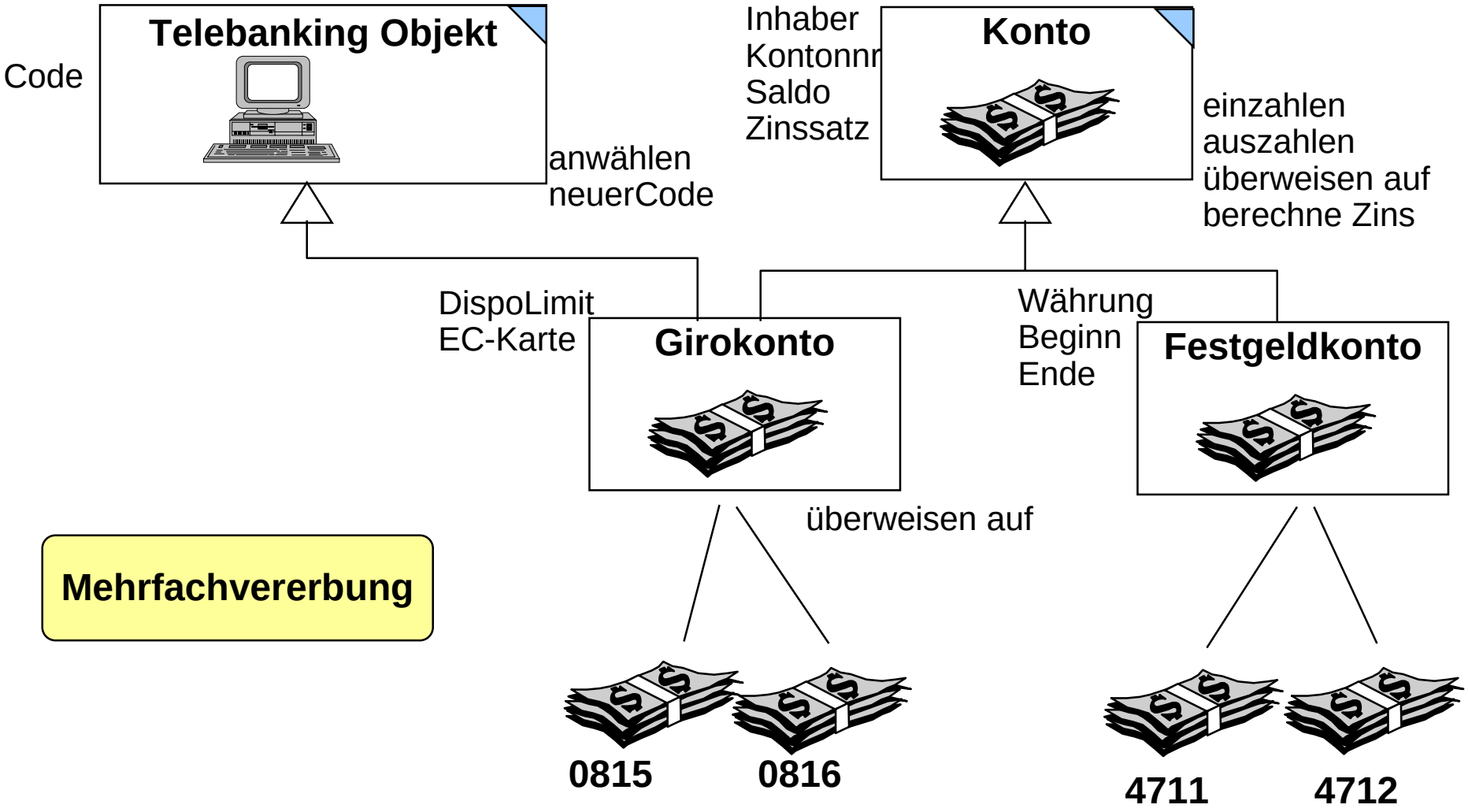
Beispiel 2: Einfachvererbung

Einfachvererbung



Inhaber: Horst Lichter
Kontonr: 0815
Saldo: 1000 DM
Zinssatz: 0,5 %
Dispo-Kredit: 1000 DM
EC-Karte: 3249345

Beispiel 2: Mehrfachvererbung

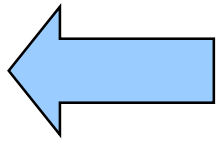


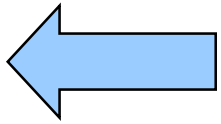
Abstrakte Klassen

- **Eine Klasse ist ein**
  - möglicherweise **nicht vollständig implementierter** ADT.
- **In konkreten Klassen (implementiert, effektiv)**
  - sind alle Operationen **ausführbar.**
- **In abstrakten Klassen (deferred, virtual)**
  - sind einige Operationen ***noch nicht implementiert*** , sondern nur spezifiziert (deklariert). Diese müssen in den konkreten Klassen angegeben werden.
- **Von konkreten Klassen**
  - können **beliebig viele** Objekte (Exemplare) erzeugt werden.
  - new- bzw. create- Nachrichten (Instanziierung)
- **Von abstrakten Klassen**
  - können **keine** Objekte erzeugt werden!

Beschreibungsschema von Klassen

- **Eine Klasse ist definiert durch**
- **ihren Namen**
- **ihre direkten Oberklassen**
  - die erbt_von-Beziehung muß zyklenfrei sein
- **eine Speicherstrukturbeschreibung**
  - erweitert die geerbten Beschreibungen
- **eine Menge von Operationsbeschreibungen**
  - erweitert die geerbten Beschreibungen





Oberklasse <-> Unterklasse

Geerbte Eigenschaften können auf drei Arten in einer Unterklasse modifiziert werden

Erweitern	etwas Neues hinzufügen
Redefinieren	sich ähnlich verhalten, Diff. formulieren
Definieren	etwas Versprochenes realisieren

Erweitern

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo,
    contains

feature
  origin, corner :
   Point;

feature

  height : Integer is
    do ...
    end;
  width : Integer is
    do ...
    end;
  ...
end -- class Rectangle
```

Spezifische Eigenschaften werden hinzugefügt

Definieren

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean
   is
    do
      ...
    end;
  ...
end -- class Rectangle
```

Definieren versprochener, aber noch nicht implementierter Eigenschaften

Redefinieren

```
class DisplayableObject
  feature
  psDescription: String is
    deferred
    end;
end -- class DisplayableObject
```

```
class Rectangle
...
feature
  initialize is do
      origin.Create; corner.Create;
      origin.initialize;
      corner.initialize;
    end
  ...
end -- class Rectangle
```

```
class DisplayableRectangle
inherit
  Rectangle
    rename initialize as initRect
    redefine initialize
...
feature
  color : Color;

feature
  initialize is do
    current.initRect;
    color.create; color.black;
  end;
...
end -- class DisplayableRectangle
```

Die ererbte Implementierung paßt im Kontext der Unterklasse nicht mehr.

Sie wird redefiniert!

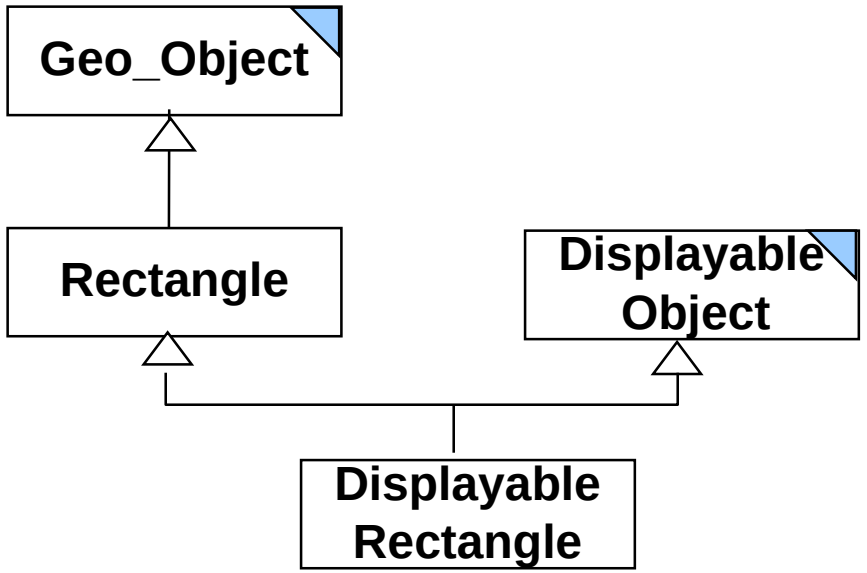
Polymorphismus

- **Allgemein:**
  - "Polymorphismus ist die Fähigkeit von Etwas, von verschiedener Gestalt zu sein"
- **In Programmiersprachen:**
  - Etwas: Variablen, Funktionen, Prozeduren
  - Gestalt: Typ
- **Eine polymorphe "Entität" kann in verschiedenen Kontexten verwendet werden,**
  - die unterschiedliche Typen verlangen.
- **Beispiele**

```
write (100, 3)
write ('a')
```

Beispiel: Polymorphismus - 1

- **Die Vererbung ist *ein* Mechanismus, um Polymorphismus in Programmiersprachen zu realisieren.**



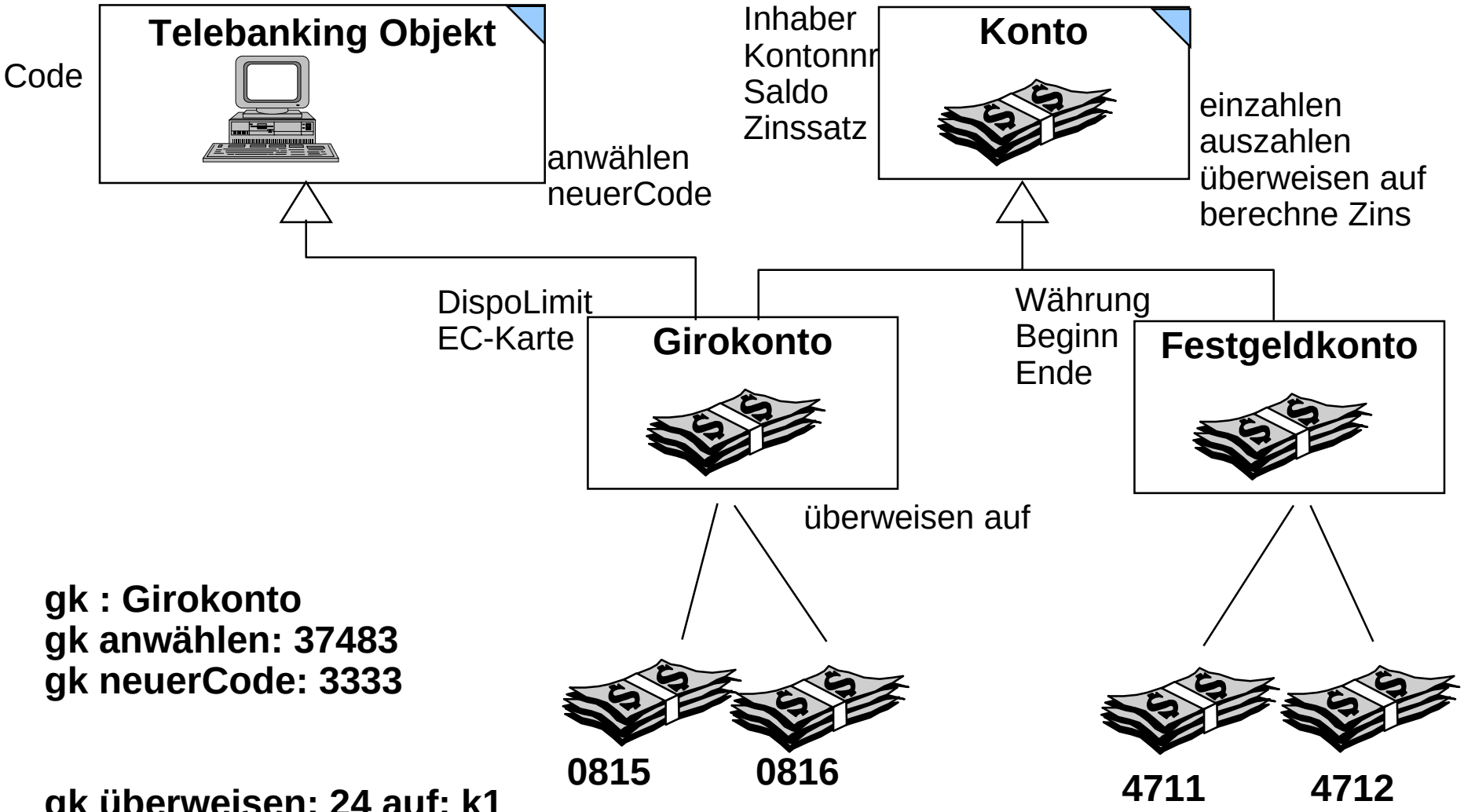
ein DisplayableRectangle **verhält sich wie** ein Rechteck und wie ein DisplayableObject

```
dr : DisplayableRectangle;
dr.Create;


dr.contains (p);
x := dr.center;
dr.moveTo (p);


s := dr.psDescription
```

Rectangle

DisplayableObject

Beispiel: Polymorphismus - 2



gk : Girokonto
gk anwählen: 37483
gk neuerCode: 3333

gk überweisen: 24 auf: k1

Dynamisches Binden: Beispiel 1

```
g : Geo_Object;
r : Rectangle;
c : Circle;

r.Create; c.Create

g := r;
g.contains (p);

g := c;
g.contains (p);
```

```
class Geo_Object

feature
  contains (p : Point) : Boolean is
    deferred
    end;
end -- class Geo_Object
```

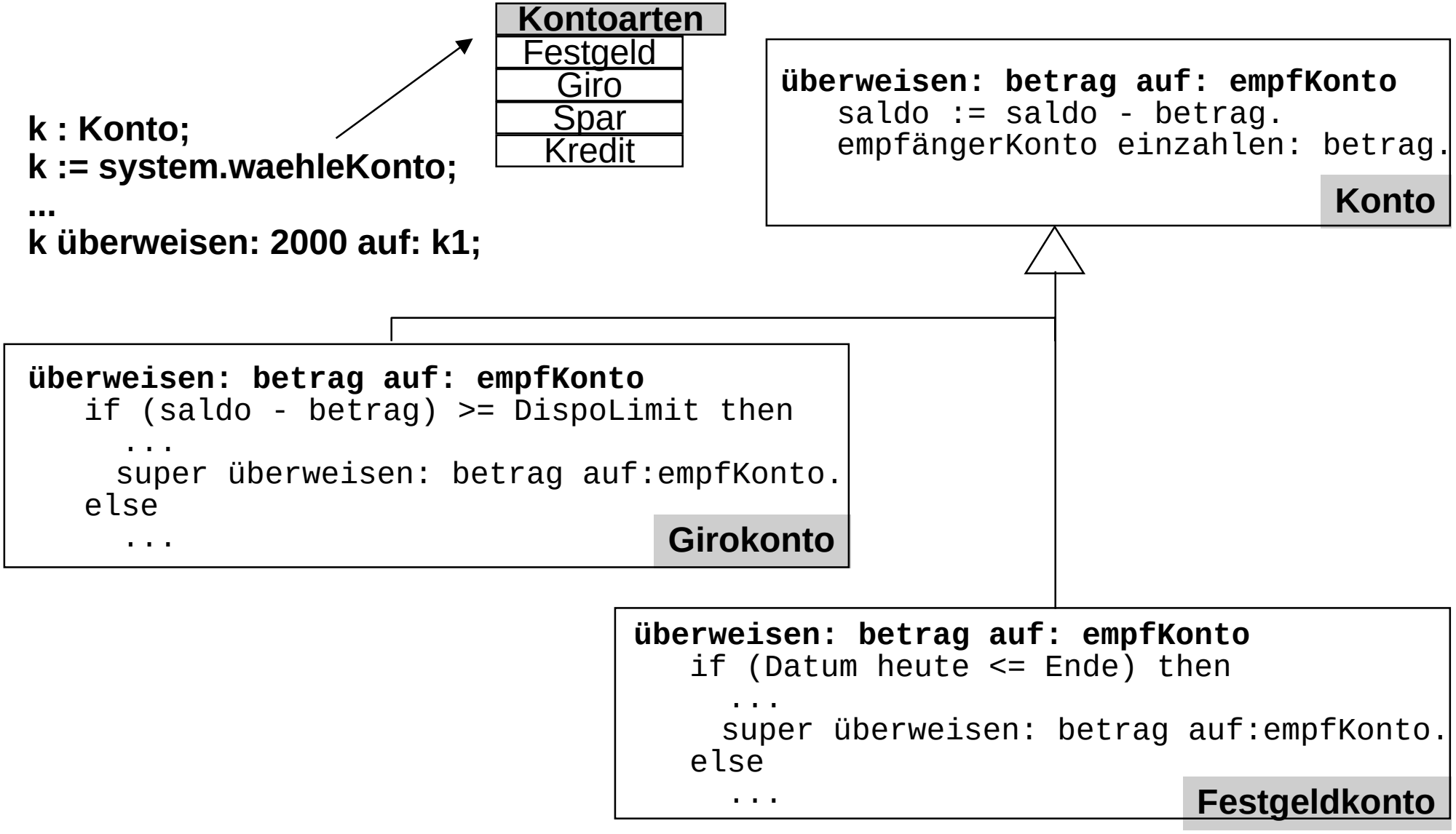
```
class Rectangle

feature
   contains (p : Point) :Boolean is
    ...
end -- class Rectangle
```

```
class Circle

feature
  contains (p : Point) :Boolean is  ...
end -- class Circle
```

Dynamisches Binden heißt:

die **richtige** Implementierung zur **Laufzeit** finden

Dynamisches Binden: Beispiel 2

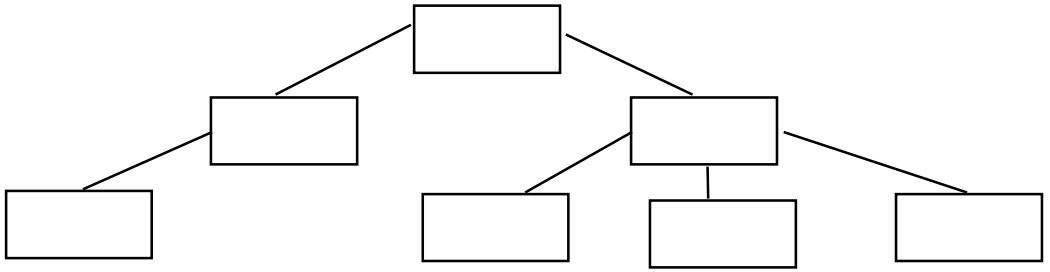


Varianten der Vererbung

Modifikations-möglichkeiten \ **Anzahl der Oberklassen**	eine oder keine	beliebig viele
nur Erweitern und Definieren	***strikte Einfachvererbung***	***strikte Mehrfachvererbung***
Erweitern Redefinieren Definieren	***nicht-strikte Einfachvererbung***	***nicht-strikte Mehrfachvererbung***

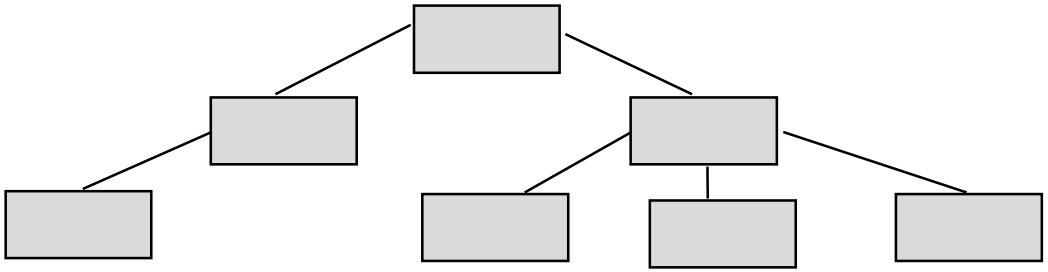
Merkmale objektorientierter Architekturen - 1

Entwurf

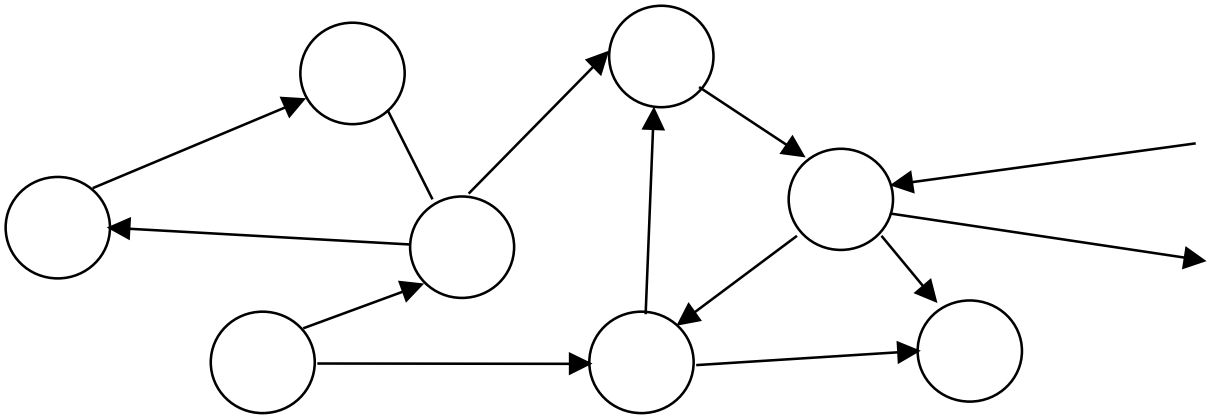


Klassen mit Vererbung

Programm

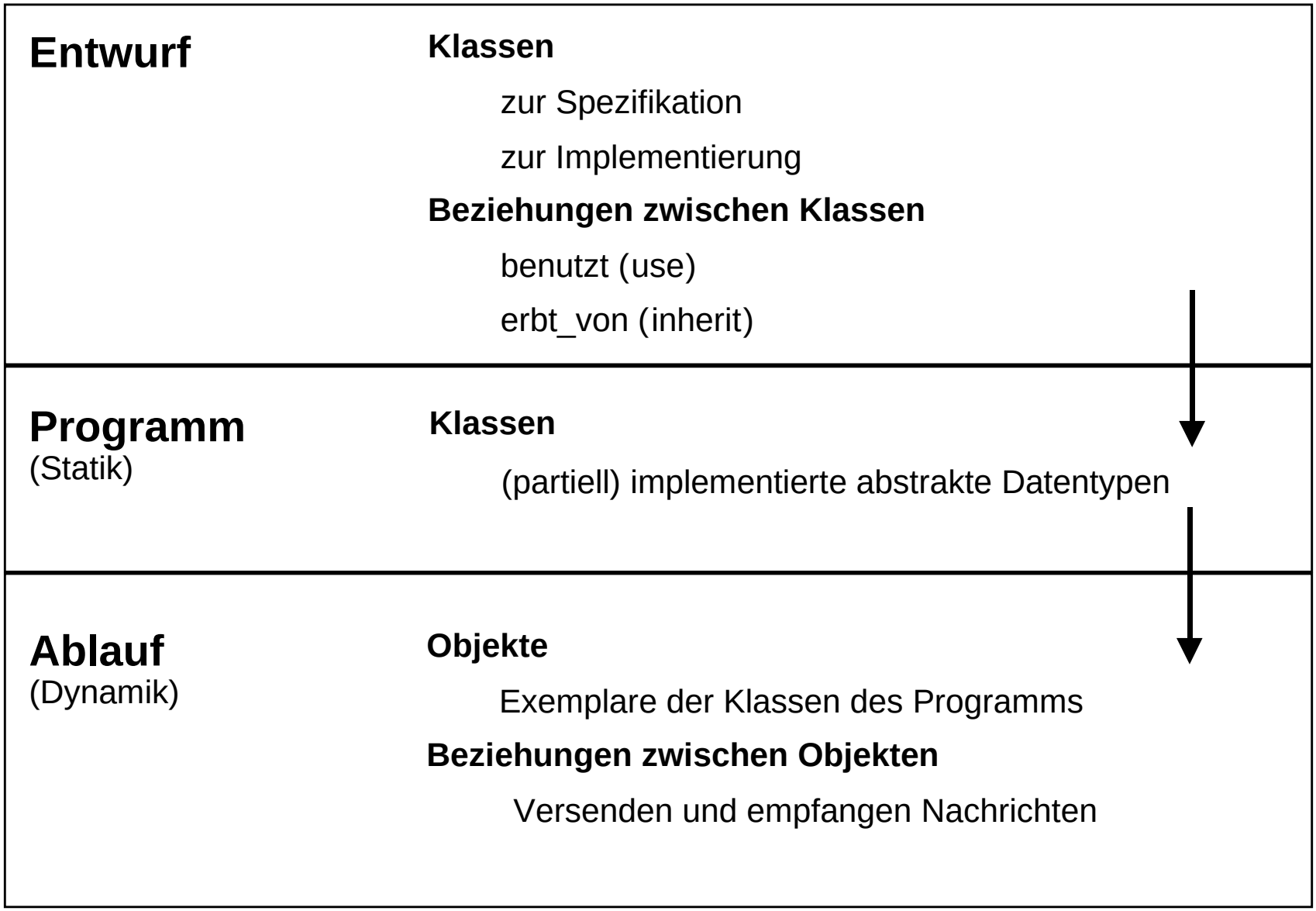


Ablauf



E/A

Merkmale objektorientierter Architekturen - 2



Was haben wir gelernt!

- **Klassifikation**
  - objektbasiert, klassenbasiert
  - objektorientiert
- **Klassen sind (partiell) implementierte ADTs**
  - abstrakte Klassen
  - konkrete Klassen
- **Vererbung**
  - dient dazu, Spezialisierungsbeziehung zwischen Begriffen programmiertechnisch zu realisieren.
  - Unterschied zwischen Einfach- und Mehrfachvererbung
- **Polymorphismus**
  - kann mit Hilfe der Vererbung realisiert werden
  - führt zum dynamischen Binden von Implementierungen

Glossar

- wissenbasierte, wertorientierte, prozedurale, objektorientierte Programmierung
- Objektmodul, ADT, Klasse
- Struktur eines objektorientierten Programms
- Nachrichten, Methoden, Laufzeitzustand eines objektorientierten Systems
- abstrakte Klasse, konkrete Klasse
- Definieren, Redefinieren, Erweitern von Methoden bei einem Spezialisierungsschritt
- Vererbung (Spezialisierung), Verallgemeinerung (Generalisierung)
- Einfach-, Mehrfachvererbung; strikte Vererbung, nichtstrikte Vererbung
- Polymorphismus, Polymorphismus durch Vererbung
- dynamisches Binden (Dispatching)

Objektorientierte Programmierung II: OO in Modula-3

- Objekttypen für Klassen
- Untertypen für opake Klassen
- Vererbung zwischen opaken Klassen
- Diskussion

Wiederholung ADT

- **ADTs in Modula-3**
  - ein ADT ist immer ein **Referenztyp**
  - in der Schnittstelle wird ein **opaker Typ als Untertyp** des vordefinierten Referenztyps REFANY deklariert
- **Ein ADT entspricht dem Konzept einer Klasse.**
- **Eine Exemplar eines ADTs entspricht einem abstrakten Objekt**
- **Mit Hilfe der ADTs**
  - kann eine **klassenbasierte** Programmierung in Modula-3 umgesetzt werden.
- **Zur Objektorientierung fehlen noch**
  - Sprachkonstrukte, um die **Vererbung** zu modellieren.
- **Hierfür**
  - stellt Modula-3 das Konzept der **Objekttypen** bereit.

Objekttypen in Modula-3

- **Mit Hilfe der Objekttypen können in Modula-3 Klassen beschrieben werden.**
- **Ein Objekttyp gibt an**
  - Bezeichner der Klasse (des Objekttyps)
  - Bezeichner der Oberklasse
    - Es kann maximal eine Oberklasse geben

Einfachvererbung

  - Bezeichner der Exemplarvariablen
  - Signaturen der Methoden
  - Bezeichner der Methoden, die in der Klasse redefiniert werden
    - in Modula-3 spricht man von überschreiben (overrides)
- **Ein Objekttyp (Klasse) wird**
  - in einer Modulschnittstelle deklariert
  - die Implementierung enthält den strukturellen Aufbau des Objekttyps (Klasse) und die Realisierung der Methoden.

Beispiel Objekttyp : Schnittstelle

INTERFACE Point

```
TYPE Point = ROOT OBJECT
                x : INTEGER; y : INTEGER;
             METHODS
                getX(): INTEGER          := PointGetX;
                setX (value : INTEGER)   := PointSetX;
                getY (): INTEGER         := PointGetY;
                setY (value : INTEGER)   := PointSetY;
                add (p: Point) : Point   := PointAdd;
                . . .
 ...         END;
```

- **Die Klasse**
  - hat die vordefinierte Klasse **ROOT** als Oberklasse
  - deklariert zwei Exemplarvariablen x und y
- **In der METHODS-Klausel**
  - kann die **Bindung** zwischen Methode und der Prozedur, die sie realisiert, hergestellt werden!
- **Objekttypen sind spezielle Referenztypen**
  - Objekte werden mit der **NEW**-Operation erzeugt.

Beispiel Objekttyp: Rumpf des Moduls, Verwendung der Klasse

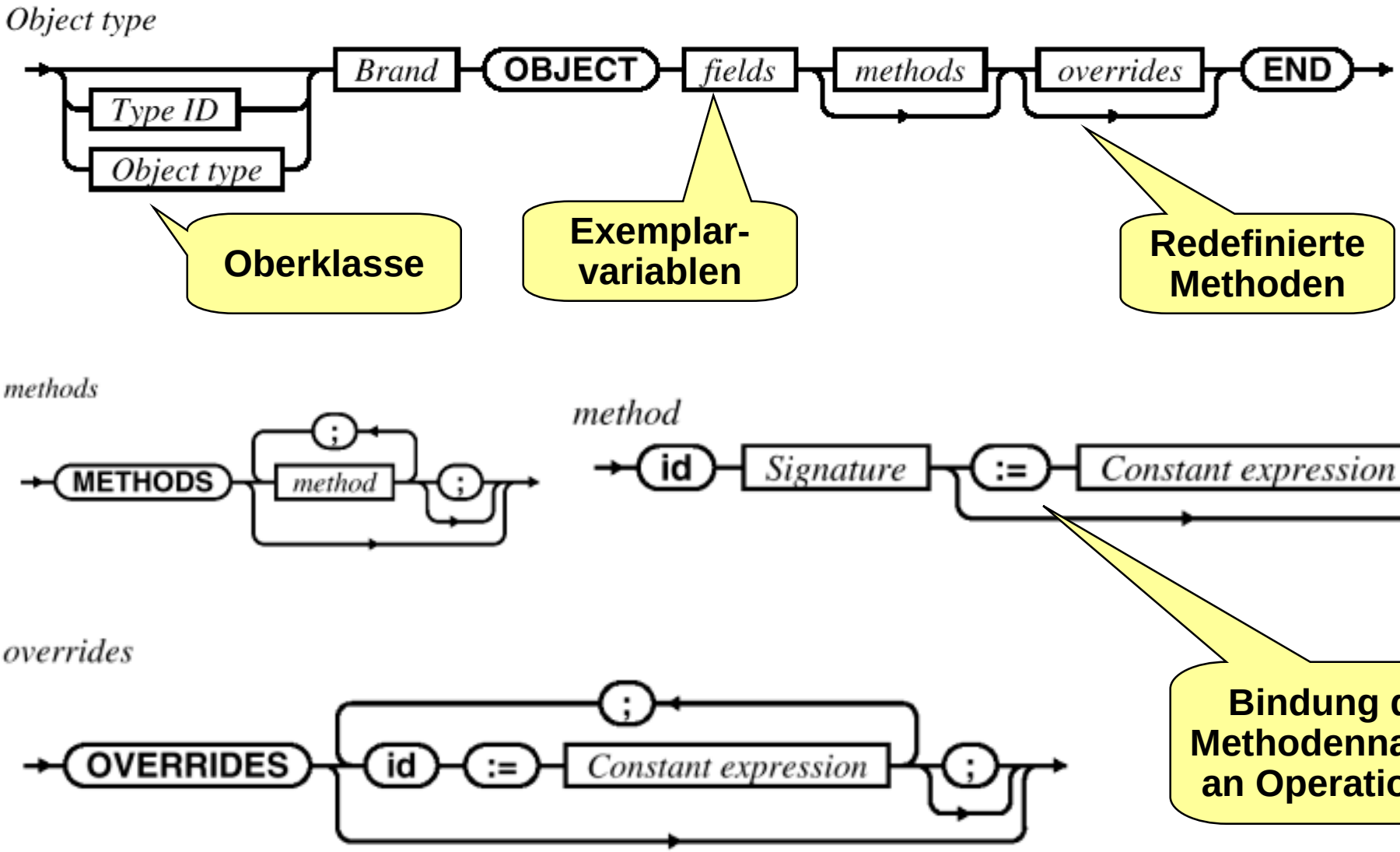
Die Prozeduren, die Methoden realisieren, erwarten als ersten Parameter ein Objekt der Klasse.

```
MODULE Points;

PROCEDURE PointGetX(self : Point): INTEGER =
BEGIN
  RETURN self.x;
END PointGetX;

PROCEDURE PointSetX(self: Point; value : INTEGER) =
BEGIN
  self.x := value;
END PointSetX;
...

PROCEDURE PointAdd (self: Point; p: Point) : Point =
VAR newP : Point;
BEGIN
  newP := NEW(Point);
  newP.setX(self.x + p.getX());
  newP.setY(self.y + p.getY());
  RETURN newP;
END PointAdd;
...
```

Verwendung der Klasse `Point`

```
VAR p1, p2, p3 : Point;
BEGIN
  p1 := NEW(Point);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
END Point_Test.
```

Syntax der Objekttyp-Deklaration



Operationenbindung und Redefinition

```
TYPE Geo_Object =  ROOT OBJECT
       METHODS
        moveTo (p : Point):= NIL ;
        contains (p : Point) : BOOLEAN:= NIL;
       END;
```

abstrakte Klasse

```
TYPE Rectangle = Geo_Object OBJECT
          origin : Point;
          corner : Point;
        METHODS
          area() : INTEGER       := RectArea;
          height () : INTEGER    := RectHeight;
          setOrigin (p : Point) := RectOrigin;
          getOrigin(): Point     := RectGetOrigin;
          …
          asText(): TEXT         := RectAsText;
        OVERRIDES
          moveTo   := RectMoveTo;
          contains := RectContains;
        END;
```

konkrete Klasse

konkrete Klasse

```
TYPE DispRectangle = Rectangle OBJECT
          color : TEXT;
        METHODS
          setColor(c : TEXT) := DispRectSetColor;
          getColor() : TEXT  := DispRectGetColor;
          draw(): TEXT       := DispRectDraw;
        OVERRIDES
          asText := DispRectAsText
        END;
```

Opake Objekttypen

- **Um opake (geschützte) Objekttypen zu deklarieren**
  - verwenden wir die M3-Sprachkonstrukte, die wir zur Realisierung ADTen benutzt haben!
    - **Trennung** von Schnittstelle und Implementierung
      - Pro Objekttp (Klasse) ein Modul
    - **Opake** Typen
    - Konzept der **Untertypen**
    - Konvention der **T-Notation** für Objekttypen
- **Der Name der Klasse wird als opaker Typ deklariert**
  - Deklaration in der Schnittstelle.
  - Ein opaker Typ muß immer ein Untertyp eines Referenz- oder Objekttyps sein.
  - Der Obertyp (ein Objektyp) deklariert die **öffentliche** Schnittstelle
    - D.h. nur die Signaturen der Methoden werden angegeben

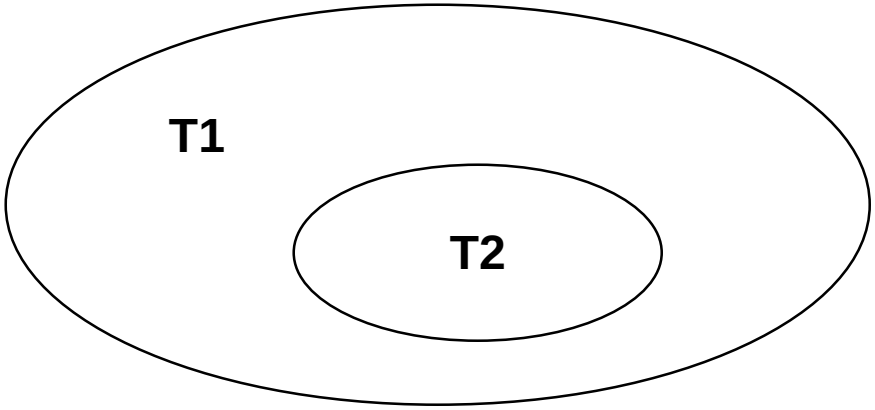
Untertypen in Modula-3

Subtypen in Modula-3

- Modula-3 kennt das Konzept der Konstruktion von ***Untertypen***
- Subtypbeziehung wird durch "***<:***" angezeigt

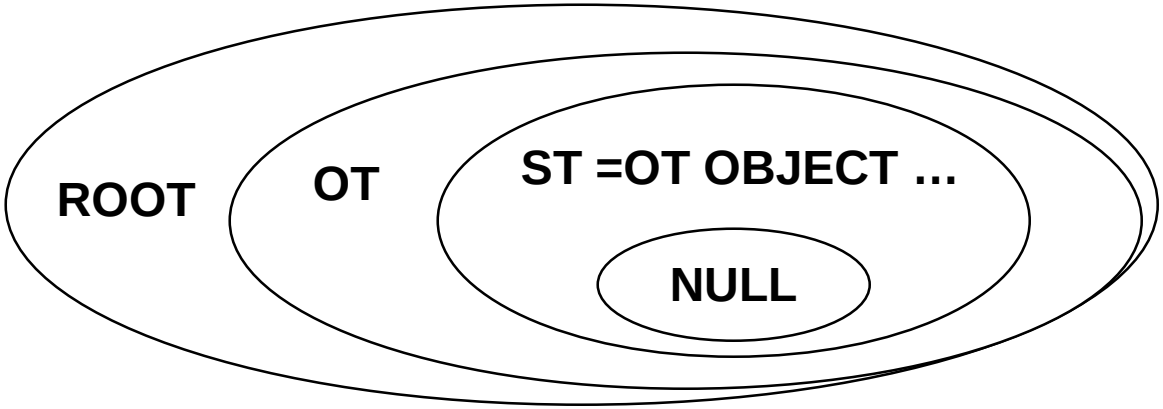
Definition

- Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
- T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
- Es gilt: ein Typ kann beliebig viele Untertypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Untertypen von Referenz- und Objekttypen

- **Modula-3 definiert zwei vorgegebene Referenztypen**
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Modula-3 definiert einen vorgegebenen Objekttyp**
  - ROOT mit folgender Subtypbeziehung
  - `NULL <: ST = OT OBJECT … END <: OT <: ROOT <: REFANY`
- **Interpretation:**
  - jeder Objekttyp ist ***Untertyp*** von ROOT (und damit ein Referenztyp)



Beispiel: opaker Objekttyp, Schnittstelle

```
INTERFACE Point;

TYPE Point <: PublicPoint;

     PublicPoint = ROOT OBJECT
                   METHODS
                     getX(): INTEGER;
                     setX (value : INTEGER);
                     getY (): INTEGER;
                     setY (value : INTEGER);
                     add (p: Point) : Point;
                     minus (p : Point) : Point;
                     asText(): TEXT;
                   END;
END Point.
```

Point ist Untertyp des "öffentlichen Teils von Point"

Dieser Typ wird exportiert und kann benutzt werden!!!!!!

In der Implementierung müssen die Exemplar-variablen und die Operationen, die die Methoden realisieren angegeben werden!

Point <: PublicPoint <: ROOT

Opaker Objekttyp: Rumpf

```
MODULE Point;

REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX;
        getX := PointGetX;
        setY := PointSetY;
        getY := PointGetY;
        add := PointAdd;
        minus := PointMinus;
        asText := PointAsText;
      END;

PROCEDURE PointGetX(self : Point): INTEGER
=
BEGIN
  RETURN self.x;
END PointGetX;
...
END Point.
```

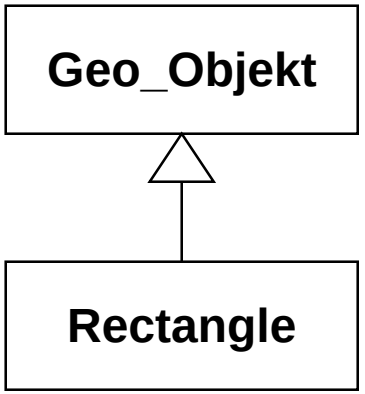
Rumpf der Klasse

- **Point wird als Objekttyp deklariert**
  - PublicPoint ist der **Obertyp**
  - Point <: PublicPoint <: ROOT
- **Point erweitert den Obertyp**
  - um die fehlenden **Instanzvariablen.**
- **Point bindet**
  - an die Methoden entsprechende **Implementierungen.**

Diskussion dieser Realisierung

- **Untertyp-Konzept wird verwendet, um**
  - **Spezialisierungsbeziehung** zwischen (hier offenen) Klassen auszudrücken

```
TYPE Rectangle = Geo_Object OBJECT
         origin : Point;
         corner : Point;
       METHODS
         area() : INTEGER      := RectArea;
         …
         asText(): TEXT        := RectAsText;
       OVERRIDES
         moveTo   := RectMoveTo;
         contains := RectContains;
       END;
```



  - geschützte Objekttypen (Klassen) zu **implementieren**

```
REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX; ...
      END;
```

T-Konvention

T bezeichnet immer den im Modul realisierten Objekttyp (Klasse)

Schnittstelle

```
INTERFACE Point;

TYPE T <: Public;

          Public = ROOT OBJECT
              METHODS
                setX(): INTEGER;
                ...
                add (p: T) : T;
                minus (p : T) : T;
                asText(): TEXT;
              END;
END Point.
```

Verwender

```
MODULE Point_Test  EXPORTS Main;

IMPORT Point,  SIO;
´
VAR p1, p2, p3 : Point.T;
BEGIN
  p1 := NEW(Point.T);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point.T);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
  SIO.PutLine(p3.asText());
END Point_Test.
```

```
MODULE Point;

  REVEAL T = Public BRANDED OBJECT
        x:INTEGER;
        y:INTEGER;
     OVERRIDES
        setX:= PointSetX;
        ...
     END;
  ...
```

Rumpf

Implementierung der Klasse `Point`

```
PROCEDURE AsText (self : T) : TEXT =
BEGIN
  RETURN (Fmt.Int(self.x)  &  "&"  &  Fmt.Int(self.y));
END AsText;

PROCEDURE GetX(self : T): INTEGER =
BEGIN
  RETURN self.x;
END GetX;

PROCEDURE GetY(self : T): INTEGER =
BEGIN
  RETURN self.y;
END GetY;

...

PROCEDURE Add (self: T; p: T) : T =
VAR newP : T;
BEGIN
  newP := NEW(T);
  newP.setX(self.getX() + p.getX());
  newP.setY(self.getY() + p.getY());
  RETURN newP;
END Add;
```

Beispiel: Klassenhierarchie Geo_Objects

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```

Vererbung zwischen opaken Klassen

Implementierung von Rectangle

```
MODULE Rectangle;
IMPORT Point;

REVEAL
  T = Public BRANDED OBJECT
        origin : Point.T;
        corner : Point.T;
      OVERRIDES
        setOrigin := SetOrigin;
        getOrigin := GetOrigin;
        setCorner := SetCorner;
        getCorner := GetCorner;
        area := Area;
        height := Height;

        moveTo   := MoveTo;
        contains := Contains;
        asText   := AsText;
      END;

PROCEDURE AsText(self : T) : TEXT =
BEGIN
  RETURN ("Rect origin: " & self.origin.asText() &
              " corner: " & self.corner.asText());
END AsText;
```

Notwendig, um geschützte Implementierung zu erhalten

Echte Redefinitionen

```
PROCEDURE MoveTo (self : T; p : Point.T)=
VAR newPoint := NEW(Point.T);
BEGIN
  newPoint := self.getCorner();
  self.setCorner(newPoint.add(p));

  newPoint := self.getOrigin();
  self.setOrigin(newPoint.add(p));
END MoveTo;


PROCEDURE Contains (self : T; p : Point.T):
                          BOOLEAN =
BEGIN
  ...
END Contains;
```

Implementierung von `DispRectangle`

```
MODULE DisplayableRectangle;

IMPORT Rectangle;
TYPE SuperClass = Rectangle.T;

REVEAL
  T = Public BRANDED OBJECT
        color : TEXT;
      OVERRIDES
        setColor := SetColor;
        getColor := GetColor;
        draw     := Draw;

        asText := AsText;
END;
```

Echte Redefinition

- **Direkter Aufruf einer Methode der Oberklasse**
  - Sollte im Normalfall **nicht** gemacht werden.
  - Sinnvoll jedoch, wenn **rekursiv redefiniert** wird, d.h. daß die Leistung der redefinierten Methode bei der neuen Implementierung verwendet werden soll.

```
PROCEDURE AsText(self : T) : TEXT =
VAR t : TEXT;
BEGIN
  t := SuperClass.asText(self);
  RETURN (t & " color: " & self.color );
END AsText;
```

Verwenden der gerade redefinierten der Oberklasse

Umgang mit Objekten

```
IMPORT Point,  SIO, Rectangle, DisplayableRectangle;
VAR
   p1, p2, p3 := NEW(Point.T);
   r1         := NEW(DisplayableRectangle.T);

BEGIN
  p1.setX(10);
  p1.setY(10);
  SIO.PutLine(p1.asText());

  p2.setX(100);
  p2.setY(100);
  SIO.PutLine(p2.asText());

  r1.setOrigin(p1);
  r1.setCorner(p2);
  r1.setColor("black");
  SIO.PutLine(r1.asText());

  p3.setX(50);
  p3.setY(50);
  SIO.PutLine(p3.asText());

  r1.moveTo(p3);
  SIO.PutLine(r1.asText());
```

Deklarieren der Variablen und erzeugen der Objekte

Senden von Nachrichten an die erzeugen Objekte

```
10&10
100&100
Rect origin: 10&10 corner: 100&100 color: black
50&50
Rect origin: 60&60 corner: 150&150 color: black
```

Objektorientierung in M3

- **Feststellung:**
  - Es ist **möglich**!
- **Anwendbarkeit und Verbindung mit anderen Paradigmen**
  - Modula-3 ist eine **hybride** Sprache
  - Sie erlaubt **imperative** und **objektorientierte** Programmierung
  - Beides kann bunt durcheinander **gemischt** werden
    - Vorteil: Jedes Konzept an seinem Platz
    - Nachteil: Mischung, unüberlegt angewendet, kann undurchsichtig werden
- **Die Implementierung der objektorientierten Konzepte**
  - basiert auf dem Untertyp-Konzept der Sprache
  - die OO-Konzepte können damit umgesetzt werden
- **Konsequenz**
  - Soll durchgängig objektorientiert entwickelt werden, dann sollte man eine **rein objektorientierte** Sprache verwenden, z.B. Java, Eiffel, Smalltalk!
  - die Vorteile eines hybriden Ansatzes sind nicht mehr gegeben

Was haben wir gelernt?

- Realisierung von Klassen in Modula-3
- Objekttypen zur Formulierung von Klassen in der Schnittstelle von Modulen
- offene Klassen (nicht zu empfehlen, wider Datenabstraktion), opake Klassen
- Unterkonzept für opake Klassen (Information Hiding) und für Vererbung
- Je Klasse ein Modul und eine Schnittstelle
- In Modula-3 kann objektorientiert programmiert werden (etwas umständlich)
- T-Konvention
- Vorteil hybrider Sprachen: OO und objektbasierte, adt-basierte sowie prozedurale Programmierung
- Nachteil: Programme können sehr unübersichtlich sein; reine OO-Programme auch
- Objektorientierte Programmierung schwierig: Struktur ist nur die Klassenhierachie, saubere Klassenhierachien zu modellieren, ist schwer

Glossar

- ADT bzw. Klasse, Realisierung durch ADT-Modul bzw. opake Klasse in Interface mit zugehörigem Rumpf
- Objekttypen von Modula-3, Deklaration in einer Schnittstelle
- Examplarvariable, Signatur von Methoden
- Redefinition und Binden von Prozeduren an die Methoden der Signatur bzw. der redefinierten Methoden
- offene Klassen, opake Klassen
- Schnittstelle für opake Klasse, Details im Rumpf über die REVEAL-Klausel
- Aufruf einer Methode der gleiche Klasse über `self`
- Aufruf einer Methode der Oberklasse über `super`

Objektorientierte Programmierung II: OO in Modula-3

- **Objekttypen für Klassen**
- **Untertypen für opake Klassen**
- **Vererbung zwischen opaken Klassen**
- **Diskussion**

Wiederholung ADT

- **ADTs in Modula-3**
  - ein ADT ist immer ein **Referenztyp**
  - in der Schnittstelle wird ein **opaker Typ als Untertyp** des vordefinierten Referenztyps REFANY deklariert
- **Ein ADT entspricht dem Konzept einer Klasse.**
- **Eine Exemplar eines ADTs entspricht einem abstrakten Objekt**
- **Mit Hilfe der ADTs**
  - kann eine **klassenbasierte** Programmierung in Modula-3 umgesetzt werden.
- **Zur Objektorientierung fehlen noch**
  - Sprachkonstrukte, um die **Vererbung** zu modellieren.
- **Hierfür**
  - stellt Modula-3 das Konzept der **Objekttypen** bereit.

Objekttypen in Modula-3

- **Mit Hilfe der Objekttypen können in Modula-3 Klassen beschrieben werden.**
- **Ein Objekttyp gibt an**
  - Bezeichner der Klasse (des Objekttyps)
  - Bezeichner der Oberklasse
    - Es kann maximal eine Oberklasse geben

    Einfachvererbung
  - Bezeichner der Exemplarvariablen
  - Signaturen der Methoden
  - Bezeichner der Methoden, die in der Klasse redefiniert werden
    - in Modula-3 spricht man von überschreiben (overrides)
- **Ein Objekttyp (Klasse) wird**
  - in einer Modulschnittstelle deklariert
  - die Implementierung enthält den strukturellen Aufbau des Objekttyps (Klasse) und die Realisierung der Methoden.

Beispiel Objekttyp : Schnittstelle

INTERFACE Point

```
TYPE Point = ROOT OBJECT
                x : INTEGER; y : INTEGER;
             METHODS
                getX(): INTEGER          := PointGetX;
                setX (value : INTEGER)   := PointSetX;
                getY (): INTEGER         := PointGetY;
                setY (value : INTEGER)   := PointSetY;
                add (p: Point) : Point   := PointAdd;
                . . .
...          END;
```

- **Die Klasse**
  - hat die vordefinierte Klasse **ROOT** als Oberklasse
  - deklariert zwei Exemplarvariablen x und y
- **In der METHODS-Klausel**
  - kann die **Bindung** zwischen Methode und der Prozedur, die sie realisiert, hergestellt werden!
- **Objekttypen sind spezielle Referenztypen**
  - Objekte werden mit der **NEW**-Operation erzeugt.

Beispiel Objekttyp: Rumpf des Moduls, Verwendung der Klasse

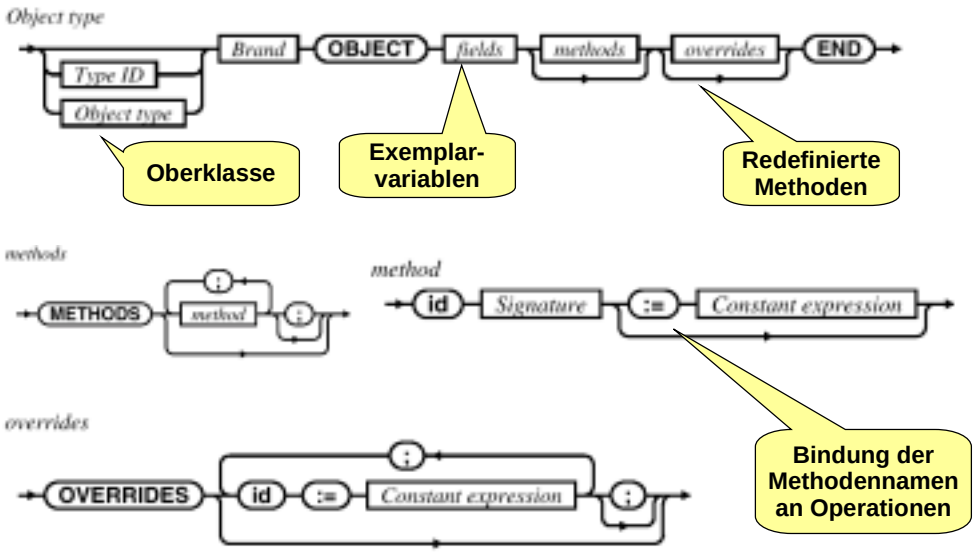
Die Prozeduren, die Methoden realisieren, erwarten als ersten Parameter ein Objekt der Klasse.

```
MODULE Points;

PROCEDURE PointGetX(self : Point): INTEGER =
BEGIN
  RETURN self.x;
END PointGetX;

PROCEDURE PointSetX(self: Point; value : INTEGER) =
BEGIN
  self.x := value;
END PointSetX;
...

PROCEDURE PointAdd (self: Point; p: Point) : Point =
VAR newP : Point;
BEGIN
  newP := NEW(Point);
  newP.setX(self.x + p.getX());
  newP.setY(self.y + p.getY());
  RETURN newP;
END PointAdd;
...
```

Verwendung der Klasse `Point`

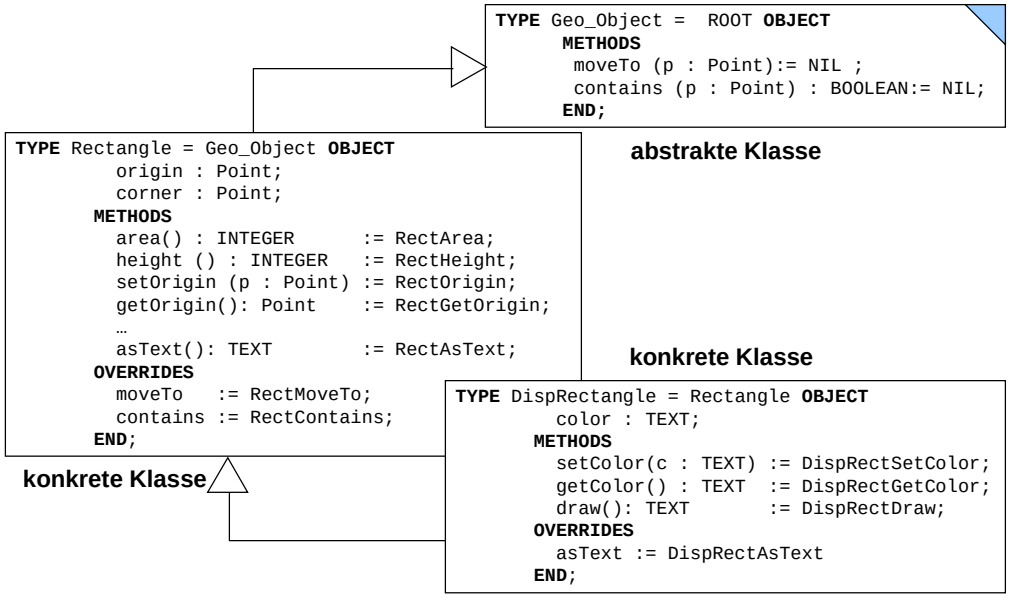
```
VAR p1, p2, p3 : Point;
BEGIN
  p1 := NEW(Point);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
END Point_Test.
```


Syntax der Objekttyp-Deklaration



Objekttypen für Klassen

Operationenbindung und Redefinition



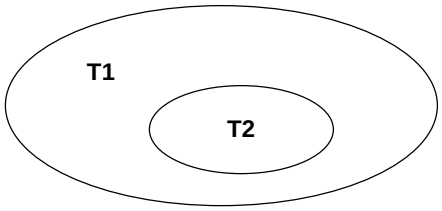
Untertypen für opake Klassen

Opake Objekttypen

- **Um opake (geschützte) Objekttypen zu deklarieren**
  - verwenden wir die M3-Sprachkonstrukte, die wir zur Realisierung ADTen benutzt haben!
    - **Trennung** von Schnittstelle und Implementierung
      - Pro Objekttp (Klasse) ein Modul
    - **Opake** Typen
    - Konzept der **Untertypen**
    - Konvention der **T-Notation** für Objekttypen
- **Der Name der Klasse wird als opaker Typ deklariert**
  - Deklaration in der Schnittstelle.
  - Ein opaker Typ muß immer ein Untertyp eines Referenz- oder Objekttyps sein.
  - Der Obertyp (ein Objektyp) deklariert die **öffentliche** Schnittstelle
    - D.h. nur die Signaturen der Methoden werden angegeben

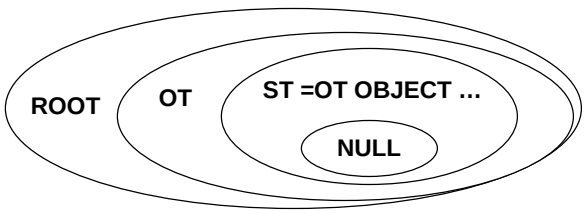
Untertypen in Modula-3

- **Subtypen in Modula-3**
  - Modula-3 kennt das Konzept der Konstruktion von ***Untertypen***
  - Subtypbeziehung wird durch "***<:***" angezeigt
- **Definition**
  - Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
  - T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
  - Es gilt: ein Typ kann beliebig viele Untertypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Untertypen von Referenz- und Objekttypen

- **Modula-3 definiert zwei vorgegebene Referenztypen**
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Modula-3 definiert einen vorgegebenen Objekttyp**
  - ROOT mit folgender Subtypbeziehung
  - `NULL <: ST = OT OBJECT … END <: OT <: ROOT <: REFANY`
- **Interpretation:**
  - jeder Objekttyp ist ***Untertyp*** von ROOT (und damit ein Referenztyp)



Beispiel: opaker Objekttyp, Schnittstelle

```
INTERFACE Point;

TYPE Point <: PublicPoint;

     PublicPoint = ROOT OBJECT
             METHODS
               getX(): INTEGER;
               setX (value : INTEGER);
               getY (): INTEGER;
               setY (value : INTEGER);
               add (p: Point) : Point;
               minus (p : Point) : Point;
               asText(): TEXT;
             END;
END Point.
```

Point ist Untertyp des "öffentlichen Teils von Point"

Dieser Typ wird exportiert und kann benutzt werden!!!!!!

In der Implementierung müssen die Exemplar-variablen und die Operationen, die die Methoden realisieren angegeben werden!

Point <: PublicPoint <: ROOT

Opaker Objekttyp: Rumpf

```
MODULE Point;

REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX;
        getX := PointGetX;
        setY := PointSetY;
        getY := PointGetY;
        add := PointAdd;
        minus := PointMinus;
        asText := PointAsText;
      END;

PROCEDURE PointGetX(self : Point): INTEGER
=
BEGIN
  RETURN self.x;
END PointGetX;
...
END Point.
```

Rumpf der Klasse

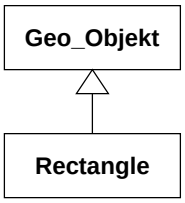
- **Point wird als Objekttyp deklariert**
  - PublicPoint ist der **Obertyp**
  - Point <: PublicPoint <: ROOT
- **Point erweitert den Obertyp**
  - um die fehlenden **Instanzvariablen.**
- **Point bindet**
  - an die Methoden entsprechende **Implementierungen.**

Untertypen für opake Klassen

Diskussion dieser Realisierung

- **Untertyp-Konzept wird verwendet, um**
  - **Spezialisierungsbeziehung** zwischen (hier offenen) Klassen auszudrücken

```
TYPE Rectangle = Geo_Object OBJECT
         origin : Point;
         corner : Point;
       METHODS
         area() : INTEGER      := RectArea;
         …
         asText(): TEXT        := RectAsText;
       OVERRIDES
         moveTo   := RectMoveTo;
         contains := RectContains;
       END;
```



  - geschützte Objekttypen (Klassen) zu **implementieren**

```
REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX; ...
      END;
```


Vererbung zwischen opaken Klassen

T-Konvention

T bezeichnet immer den im Modul realisierten Objekttyp (Klasse)

Schnittstelle

```
INTERFACE Point;

TYPE T <: Public;

          Public = ROOT OBJECT
              METHODS
                setX(): INTEGER;
                ...
                add (p: T) : T;
                minus (p : T) : T;
                asText(): TEXT;
              END;
END Point.
```

Verwender

```
MODULE Point_Test  EXPORTS Main;

IMPORT Point,  SIO;

VAR p1, p2, p3 : Point.T;
BEGIN
  p1 := NEW(Point.T);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point.T);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
  SIO.PutLine(p3.asText());
END Point_Test.
```

```
MODULE Point;

  REVEAL T = Public BRANDED OBJECT
        x:INTEGER;
        y:INTEGER;
     OVERRIDES
        setX:= PointSetX;
        ...
     END;
  ...
```

Rumpf

Implementierung der Klasse Point

```
PROCEDURE AsText (self : T) : TEXT =
BEGIN
  RETURN (Fmt.Int(self.x)  &  "&"  &  Fmt.Int(self.y));
END AsText;

PROCEDURE GetX(self : T): INTEGER =
BEGIN
  RETURN self.x;
END GetX;

PROCEDURE GetY(self : T): INTEGER =
BEGIN
  RETURN self.y;
END GetY;

...

PROCEDURE Add (self: T; p: T) : T =
VAR newP : T;
BEGIN
  newP := NEW(T);
  newP.setX(self.getX() + p.getX());
  newP.setY(self.getY() + p.getY());
  RETURN newP;
END Add;
```


Beispiel: Klassenhierarchie Geo_Objects

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```

Implementierung von Rectangle

```
MODULE Rectangle;
IMPORT Point;

REVEAL
  T = Public BRANDED OBJECT
        origin : Point.T;
        corner : Point.T;
      OVERRIDES
        setOrigin := SetOrigin;
        getOrigin := GetOrigin;
        setCorner := SetCorner;
        getCorner := GetCorner;
        area := Area;
        height := Height;

        moveTo   := MoveTo;
        contains := Contains;
        asText   := AsText;
      END;

PROCEDURE AsText(self : T) : TEXT =
BEGIN
  RETURN ("Rect origin: " & self.origin.asText() &
              " corner: " & self.corner.asText());
END AsText;
```

Notwendig, um geschützte Implementierung zu erhalten

Echte Redefinitionen

```
PROCEDURE MoveTo (self : T; p : Point.T)=
VAR newPoint := NEW(Point.T);
BEGIN
  newPoint := self.getCorner();
  self.setCorner(newPoint.add(p));

  newPoint := self.getOrigin();
  self.setOrigin(newPoint.add(p));
END MoveTo;


PROCEDURE Contains (self : T; p : Point.T):
                    BOOLEAN =
BEGIN
  ...
END Contains;
```


Implementierung von DispRectangle

```
MODULE DisplayableRectangle;

IMPORT Rectangle;
TYPE SuperClass = Rectangle.T;

REVEAL
  T = Public BRANDED OBJECT
        color : TEXT;
      OVERRIDES
        setColor := SetColor;
        getColor := GetColor;
        draw     := Draw;

        asText := AsText;
END;
```

Echte Redefinition

- **Direkter Aufruf einer Methode der Oberklasse**
  - Sollte im Normalfall **nicht** gemacht werden.
  - Sinnvoll jedoch, wenn **rekursiv redefiniert** wird, d.h. daß die Leistung der redefinierten Methode bei der neuen Implementierung verwendet werden soll.

```
PROCEDURE AsText(self : T) : TEXT =
VAR t : TEXT;
BEGIN
  t := SuperClass.asText(self);
  RETURN (t & " color: " & self.color );
END AsText;
```

Verwenden der gerade redefinierten der Oberklasse

Vererbung zwischen opaken Klassen

Umgang mit Objekten

```
IMPORT Point,  SIO, Rectangle, DisplayableRectangle;
VAR
   p1, p2, p3 := NEW(Point.T);
   r1         := NEW(DisplayableRectangle.T);

BEGIN
  p1.setX(10);
  p1.setY(10);
  SIO.PutLine(p1.asText());

  p2.setX(100);
  p2.setY(100);
  SIO.PutLine(p2.asText());

  r1.setOrigin(p1);
  r1.setCorner(p2);
  r1.setColor("black");
  SIO.PutLine(r1.asText());

  p3.setX(50);
  p3.setY(50);
  SIO.PutLine(p3.asText());

  r1.moveTo(p3);
  SIO.PutLine(r1.asText());
```

Deklarieren der Variablen und erzeugen der Objekte

Senden von Nachrichten an die erzeugen Objekte

```
10&10
100&100
Rect origin: 10&10 corner: 100&100 color: black
50&50
Rect origin: 60&60 corner: 150&150 color: black
```


Diskussion

Objektorientierung in M3

- **Feststellung:**
  - Es ist **möglich**!
- **Anwendbarkeit und Verbindung mit anderen Paradigmen**
  - Modula-3 ist eine **hybride** Sprache
  - Sie erlaubt **imperative** und **objektorientierte** Programmierung
  - Beides kann bunt durcheinander **gemischt** werden
    - Vorteil: Jedes Konzept an seinem Platz
    - Nachteil: Mischung, unüberlegt angewendet, kann undurchsichtig werden
- **Die Implementierung der objektorientierten Konzepte**
  - basiert auf dem Untertyp-Konzept der Sprache
  - die OO-Konzepte können damit umgesetzt werden
- **Konsequenz**
  - Soll durchgängig objektorientiert entwickelt werden, dann sollte man eine **rein objektorientierte** Sprache verwenden, z.B. Java, Eiffel, Smalltalk!
  - die Vorteile eines hybriden Ansatzes sind nicht mehr gegeben

Was haben wir gelernt?

- Realisierung von Klassen in Modula-3
- Objekttypen zur Formulierung von Klassen in der Schnittstelle von Modulen
- offene Klassen (nicht zu empfehlen, wider Datenabstraktion), opake Klassen
- Unterkonzept für opake Klassen (Information Hiding) und für Vererbung
- Je Klasse ein Modul und eine Schnittstelle
- In Modula-3 kann objektorientiert programmiert werden (etwas umständlich)
- T-Konvention
- Vorteil hybrider Sprachen: OO und objektbasierte, adt-basierte sowie prozedurale Programmierung
- Nachteil: Programme können sehr unübersichtlich sein; reine OO-Programme auch
- Objektorientierte Programmierung schwierig: Struktur ist nur die Klassenhierachie, saubere Klassenhierachien zu modellieren, ist schwer

Glossar

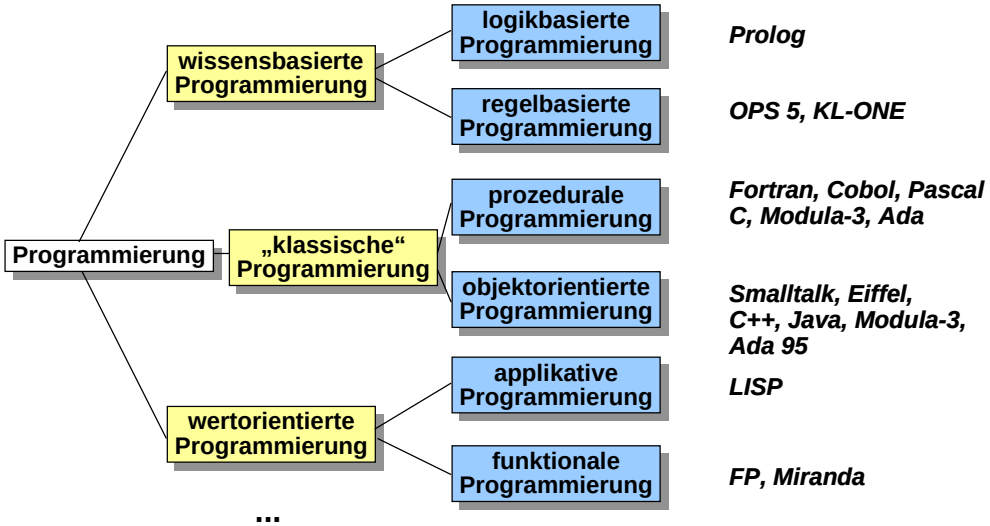
- ADT bzw. Klasse, Realisierung durch ADT-Modul bzw. opake Klasse in Interface mit zugehörigem Rumpf
- Objekttypen von Modula-3, Deklaration in einer Schnittstelle
- Examplarvariable, Signatur von Methoden
- Redefinition und Binden von Prozeduren an die Methoden der Signatur bzw. der redefinierten Methoden
- offene Klassen, opake Klassen
- Schnittstelle für opake Klasse, Details im Rumpf über die REVEAL-Klausel
- Aufruf einer Methode der gleiche Klasse über `self`
- Aufruf einer Methode der Oberklasse über `super`

Objektorientierte Programmierung I

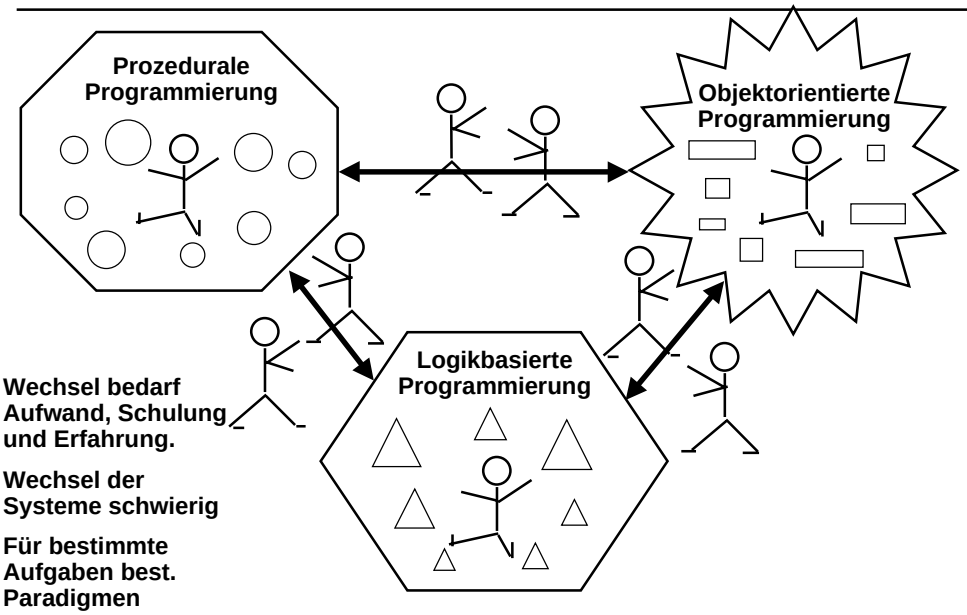
- Verständnis der objektorientierten Programmierung
- Objekte
- Klassen
- Vererbung
- Polymorphismus und dynamisches Binden
- Diskussion

Programmier-Paradigmen

- Es gibt unterschiedliche Arten der Programmierung:



Wechsel des Programmier-Paradigmas

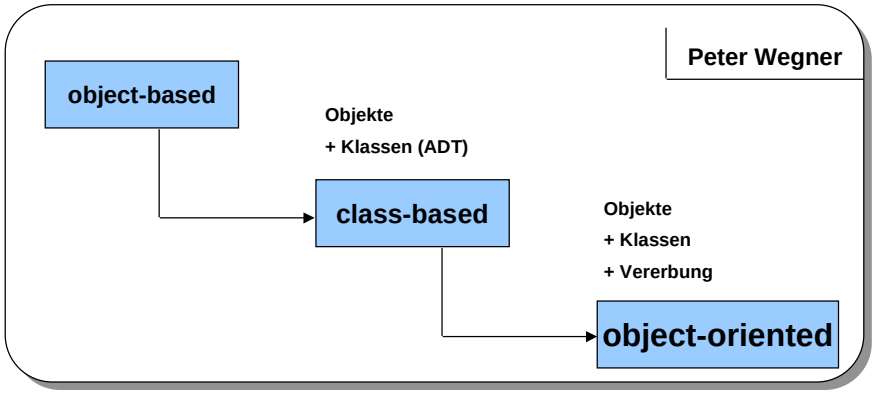


Wechsel bedarf Aufwand, Schulung und Erfahrung.

Wechsel der Systeme schwierig

Für bestimmte Aufgaben best. Paradigmen

Merkmale objektorientierter Systeme/Sprachen



Ada
Modula-2

CLU
(Ada 83)
(Modula-2)

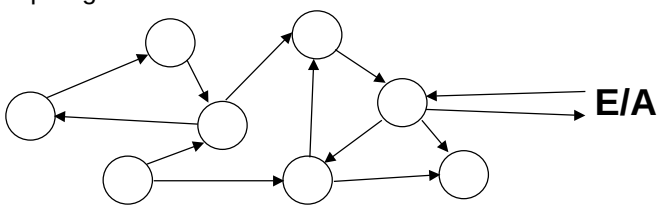
Simula
Smalltalk-80
Eiffel
C++
Ada 95
Java
Modula-3

Klassifikation

- **Objektbasiert**
  - Objektbasierte Programmiersprachen bieten die Möglichkeit, **Objekte** im Sinne einer **Datenkapsel** resp. eines **Objektmoduls** zu realsieren.
  - Jedes Objekt wir **einzeln** beschrieben und benutzt
- **Klassenbasiert**
  - Klassenbasierte Programmiersprachen bieten die Möglichkeit, Objekte in Forn von **Objekttypen (Klassen)** zu beschreiben.
  - Von Objekttypen können beliebig viele **Exemplare** (Objekte des Typs) erzeugt werden.
- **Objektorientiert**
  - Objektorientierte Programmiersprachen erlauben, Objekttypen (Klassen) mithilfe der **Vererbungs-Beziehung** zu strukturieren.
  - Dadurch lassen sich **Objekttyp-Hierarchien** im Sinne der Spezialisierung resp. Generalisierung modellieren.

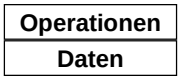
Was ist ein rein objektorientiertes System

- **Ein Software-System besteht aus Objekten**
- **Jede Systemaktivität ist die Aktivität eines Objektes**
- **Objekte**
  - haben eine Lebensdauer (werden erzeugt und vernichtet)
  - sind identifizierbar
  - sind aktiv
  - verändern sich während ihrer Lebensdauer
  - senden und empfangen Nachrichten



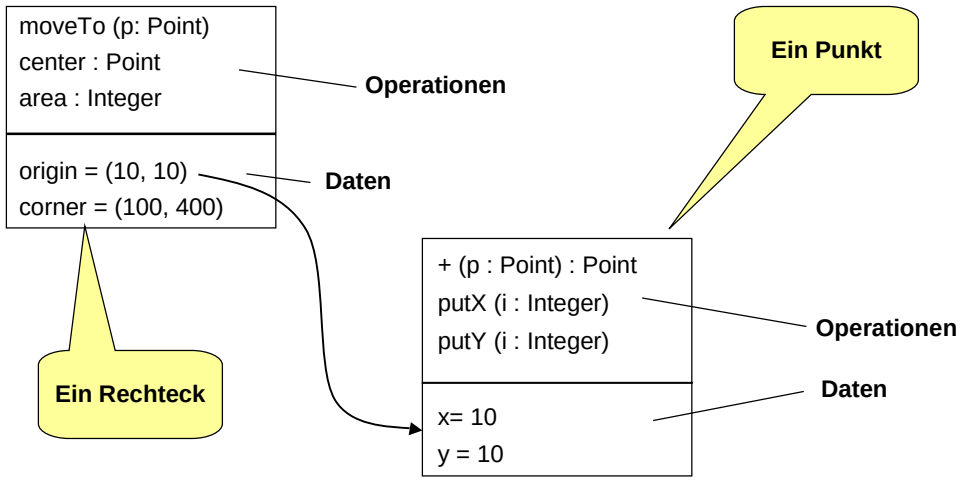
Objekte - die Dynamik des Systems

- **Ein Objekt ist eine Datenkapsel, die aus zwei Teilen besteht**
  - Der Wert der Daten repräsentiert den **Zustand** des Objekts
  - Daten können nur mithilfe von **Operationen** verändert werden (Kapselung)



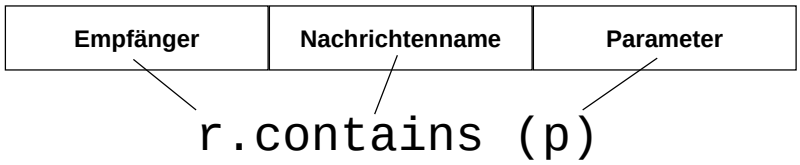
- **Die Aktivität der Objekte ist die Ausführung ihrer Operationen**
  - Eine Operation wird ausgeführt, wenn ein Objekt eine entsprechende **Nachricht** erhält.
  - Der Objektzustand kann sich **verändern**.
  - Mögliche Operationen sind durch den **Objekttyp** bestimmt.
- **Ein Objekt ist ein Exemplar genau einer Klasse.**
- **Objekte existieren nur zur Laufzeit**
  - können aber auch persistent gespeichert werden

Beispiele für Objekte

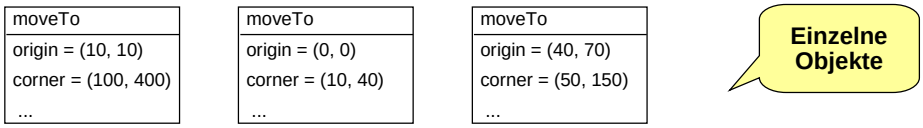


Nachrichten (Botschaften)

- **Objekte kommunizieren miteinander,**
  - dadurch daß sie **Nachrichten** versenden und Nachrichten empfangen.
- **Objekte reagieren auf Nachrichten,**
  - indem sie eine **Operation** (**Methode, Zugriffsunterprogramm**) ausführen.
- **Der Empfänger einer Nachricht (immer ein Objekt) ist verantwortlich für**
  - **Entschlüsselung** der Nachricht (verstehe ich die Nachricht?)
  - **Wirkung** (was tue ich?)



Klassen - die Statik des Systems



- **Gleichartige Objekte werden zusammengefaßt**
  - und an einer Stelle beschrieben (Objekttyp oder Klasse)
- **Eine Klasse definiert für ihre Objekte**
  - die **Speicherstruktur**
    - Daten, Attribute, Exemplarvariablen
  - die **Operationen** (Methoden, Routinen),
    - von denen ein Teil exportiert wird (exported, public <-> private)
    - andere werden nur intern benötigt
- **Von einer Klasse können beliebig viele Objekte erzeugt werden**

Beispiel : Klasse Point

```
class Point

feature
  x, y : Integer;

feature
  +: (p : Point) is
      result : Point;
    do
      result.x(x + p.x);
      result.y(x + p.y);
    end;
  x: (i : Integer) is
    do
      x := i;
    end;
  y: (i : Integer) is
    do
      y := i;
    end;
  ...
end -- class Rectangle
```

Exemplarvariablen (int. Verbundkomponente)

exportierte Operationen

Beispiel : Klasse Rechteck

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do ...
    end;

feature {NONE}
  extent : Point
    do
      -- liefert einen Punkt, der die Höhe und
      -- Weite des Rechtecks repräsentiert
    end;
end -- class Rectangle
```

Exemplarvariablen

exportierte Operationen

private Operationen

Klasse - Objekt

- **Eine Klasse entspricht einem ADT**
- **Ein Objekt "entspricht " einem abstarkten Datenobjekt**
  - d.h. die Datenimplementierung ist verborgen.
- **Von einer Klasse sind außerhalb sichtbar:**
  - der Klassenname
  - die exportierten Operationen
- **Jedes Objekt ist ein Exemplar**
  - einer Klasse des Programms
- **Objekte einer Klasse kennen dieselben Operationen**
- **Objekte einer Klasse unterscheiden sich in den Werten ihrer Daten**

Objekte und Klassen

- Nach außen sichtbar ist

```
class Rectangle
interface
  Create;
  origin : Point;
  corner : Point;
  moveTo (p : Point);
  contains (p : Point)
   Boolean:
end -- class Rectangle
```

Zusammenhang Objekt <-> Klasse

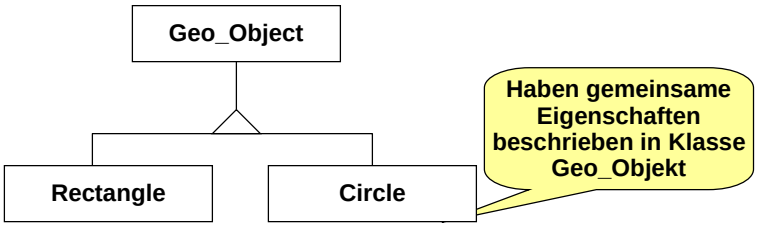
```
r : Rectangle;
p : Point

p.Create;
r.Create;

r.contains (p);
```

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do
    end;
feature {NONE}

  extent : Point
    do
      -- liefert ...
    end;
end -- class Rectangle
```

Gemeinsame Eigenschaften, Spezialisierung

- **Gemeinsame Eigenschaften verschiedener Klassen K_1, K_2**
  - werden in einer **eigenen Klasse K** zusammengefaßt und definiert,
  - und anschließend an K_1, K_2 vererbt.

- **Beispiel**



- **Regel:**
  - Eine Klasse A erbt von einer Klasse B genau dann, wenn A eine **Spezialisierung** (Unterbegriff) von B ist. Umgekehrt wird A die **Generalisierung** genannt.

Beispiel 1: Einfachvererbung

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

```
class Circle

inherit
  Geo_Object
    define moveTo, contains

feature
  center : Point;
  radius : Integer:
  ...
end -- class Circle
```

Abstrakte Klasse Spezifikationsklasse besitzt keine Objekt

Einfachvererbung

Beispiel 1: Mehrfachvererbung

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;
contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

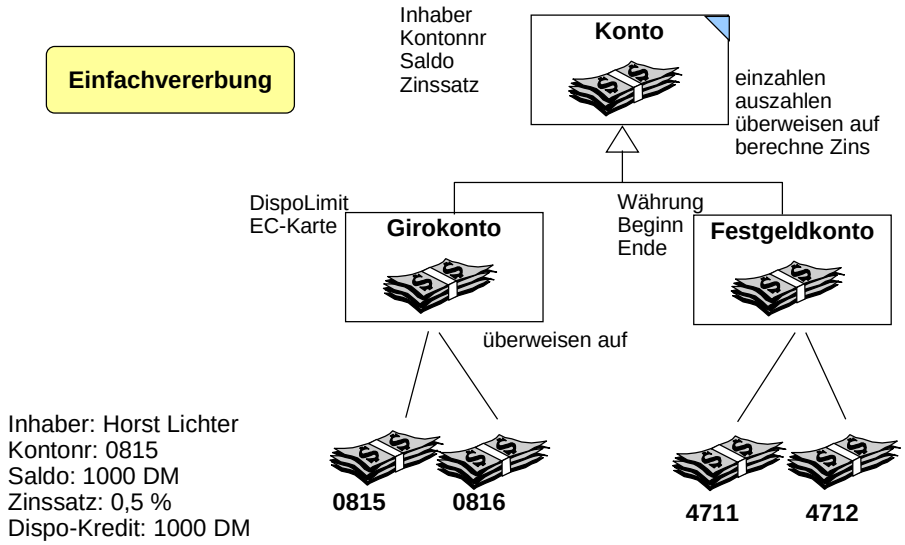
```
class DisplayableObject

feature
  psDescription : String is
    deferred
    end;
end -- class DisplayableObject
```

```
class DisplayableRectangle

inherit
  Rectangle

inherit
  DispayableObject
    define psDescription

feature
  ...
end -- class DisplayableRectangle
```

Mehrfachvererbung

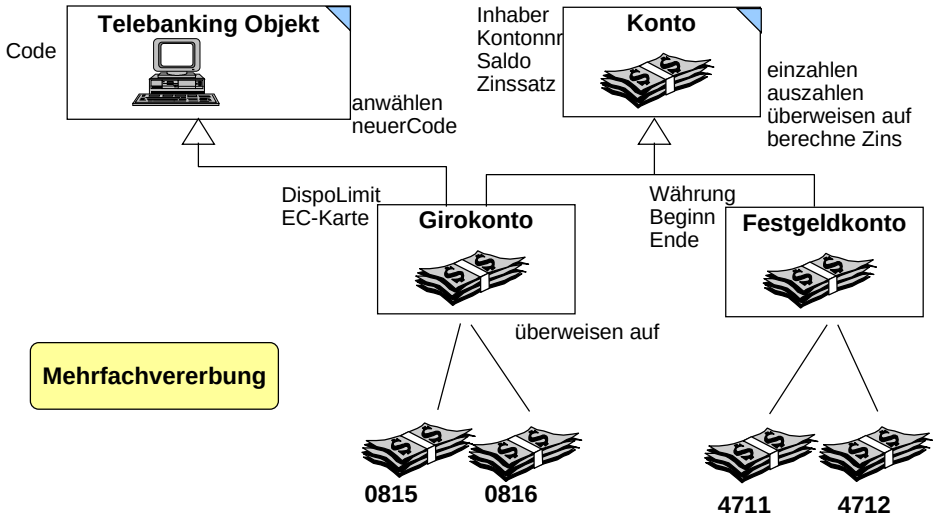
Beispiel 2: Einfachvererbung

Einfachvererbung



Inhaber: Horst Lichter
Kontonr: 0815
Saldo: 1000 DM
Zinssatz: 0,5 %
Dispo-Kredit: 1000 DM
EC-Karte: 3249345

Beispiel 2: Mehrfachvererbung

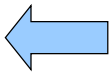


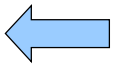
Abstrakte Klassen

- **Eine Klasse ist ein**
  - möglicherweise **nicht vollständig implementierter** ADT.
- **In konkreten Klassen (implementiert, effektiv)**
  - sind alle Operationen **ausführbar.**
- **In abstrakten Klassen (deferred, virtual)**
  - sind einige Operationen ***noch nicht implementiert*** , sondern nur spezifiziert (deklariert). Diese müssen in den konkreten Klassen angegeben werden.
- **Von konkreten Klassen**
  - können **beliebig viele** Objekte (Exemplare) erzeugt werden.
  - new- bzw. create- Nachrichten (Instanziierung)
- **Von abstrakten Klassen**
  - können **keine** Objekte erzeugt werden!

Beschreibungsschema von Klassen

- **Eine Klasse ist definiert durch**
- **ihren Namen**
- **ihre direkten Oberklassen**
  - die erbt_von-Beziehung muß zyklenfrei sein
- **eine Speicherstrukturbeschreibung**
  - erweitert die geerbten Beschreibungen
- **eine Menge von Operationsbeschreibungen**
  - erweitert die geerbten Beschreibungen





Oberklasse <-> Unterklasse

Geerbte Eigenschaften können auf drei Arten in einer Unterklasse modifiziert werden

Erweitern	etwas Neues hinzufügen
Redefinieren	sich ähnlich verhalten, Diff. formulieren
Definieren	etwas Versprochenes realisieren

Erweitern

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo,
    contains

feature
  origin, corner :
   Point;

feature

  height : Integer is
    do ...
    end;
  width : Integer is
    do ...
    end;
  ...
end -- class Rectangle
```

Spezifische Eigenschaften werden hinzugefügt

Definieren

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean
   is
    do
      ...
    end;
  ...
end -- class Rectangle
```

Definieren versprochener, aber noch nicht implementierter Eigenschaften

Redefinieren

```
class DisplayableObject
  feature
  psDescription: String is
    deferred
    end;
end -- class DisplayableObject
```

```
class Rectangle
...
feature
  initialize is do
      origin.Create; corner.Create;
      origin.initialize;
      corner.initialize;
    end
  ...
end -- class Rectangle
```

```
class DisplayableRectangle
inherit
  Rectangle
    rename initialize as initRect
    redefine initialize
...
feature
  color : Color;

feature
  initialize is do
    current.initRect;
    color.create; color.black;
  end;
...
end -- class DisplayableRectangle
```

Die ererbte Implementierung paßt im Kontext der Unterklasse nicht mehr.

Sie wird redefiniert!

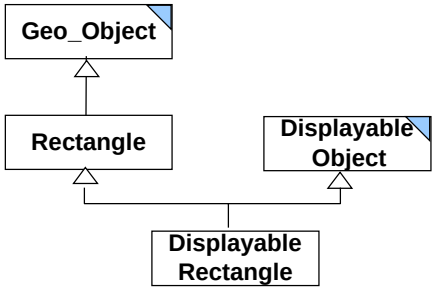
Polymorphismus

- **Allgemein:**
  - "Polymorphismus ist die Fähigkeit von Etwas, von verschiedener Gestalt zu sein"
- **In Programmiersprachen:**
  - Etwas: Variablen, Funktionen, Prozeduren
  - Gestalt: Typ
- **Eine polymorphe "Entität" kann in verschiedenen Kontexten verwendet werden,**
  - die unterschiedliche Typen verlangen.
- **Beispiele**

```
write (100, 3)
write ('a')
```

Beispiel: Polymorphismus - 1

- **Die Vererbung ist *ein* Mechanismus, um Polymorphismus in Programmiersprachen zu realisieren.**



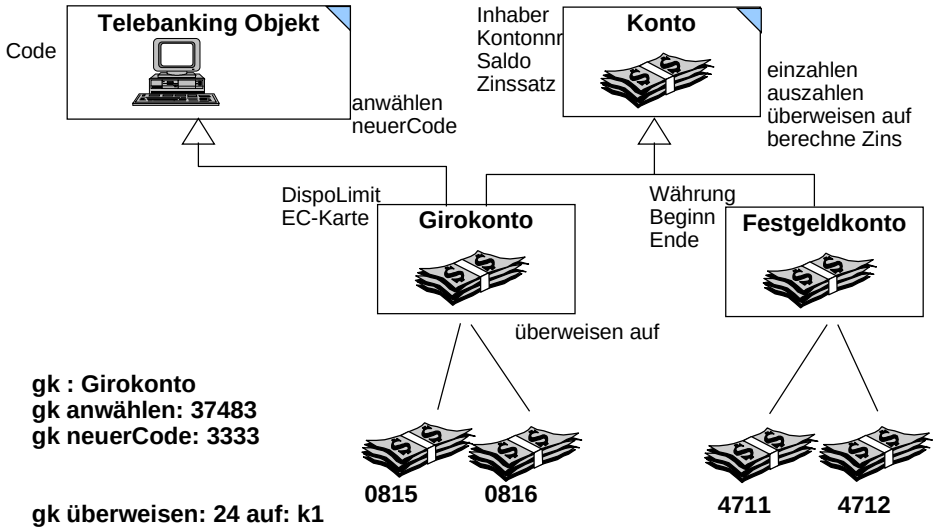
ein DisplayableRectangle **verhält sich wie** ein Rechteck und wie ein DisplayableObject

```
dr : DisplayableRectangle;
dr.Create;

dr.contains (p);
x := dr.center;
dr.moveTo (p);

s := dr.psDescription
```

Rectangle

DisplayableObject

Beispiel: Polymorphismus - 2



gk : Girokonto
gk anwählen: 37483
gk neuerCode: 3333

gk überweisen: 24 auf: k1

Dynamisches Binden: Beispiel 1

```
g : Geo_Object;
r : Rectangle;
c : Circle;

r.Create; c.Create

g := r;
g.contains (p);

g := c;
g.contains (p);
```

```
class Geo_Object

feature
  contains (p : Point) : Boolean is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

feature
   contains (p : Point) :Boolean is
     ...
end -- class Rectangle
```

```
class Circle

feature
  contains (p : Point) :Boolean is  ...
end -- class Circle
```

Dynamisches Binden heißt:

die **richtige** Implementierung zur **Laufzeit** finden

Dynamisches Binden: Beispiel 2

k : Konto;
k := system.waehleKonto;
...
k überweisen: 2000 auf: k1;

Kontoarten
- Festgeld
- Giro
- Spar
- Kredit

```
überweisen: betrag auf: empfKonto
   saldo := saldo - betrag.
   empfängerKonto einzahlen: betrag.
```
Konto

```
überweisen: betrag auf: empfKonto
   if (saldo - betrag) >= DispoLimit then
     ...
     super überweisen: betrag auf:empfKonto.
   else
     ...
```
Girokonto

```
überweisen: betrag auf: empfKonto
   if (Datum heute <= Ende) then
     ...
     super überweisen: betrag auf:empfKonto.
   else
     ...
```
Festgeldkonto

Diskussion

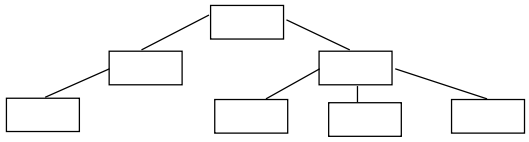
Varianten der Vererbung

Anzahl der Oberklassen / Modifikations-möglichkeiten	eine oder keine	beliebig viele
nur Erweitern und Definieren	***strikte Einfachvererbung***	***strikte Mehrfachvererbung***
Erweitern Redefinieren Definieren	***nicht-strikte Einfachvererbung***	***nicht-strikte Mehrfachvererbung***

Diskussion

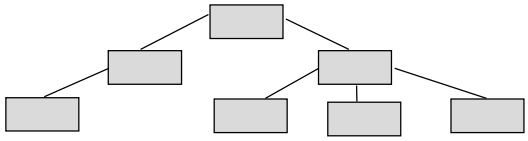
Merkmale objektorientierter Architekturen - 1

Entwurf

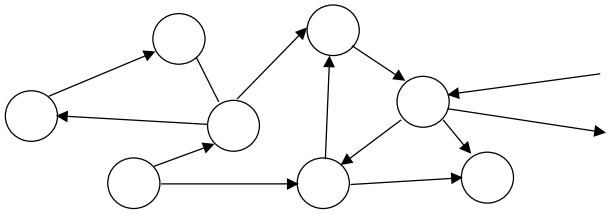


Klassen mit Vererbung

Programm



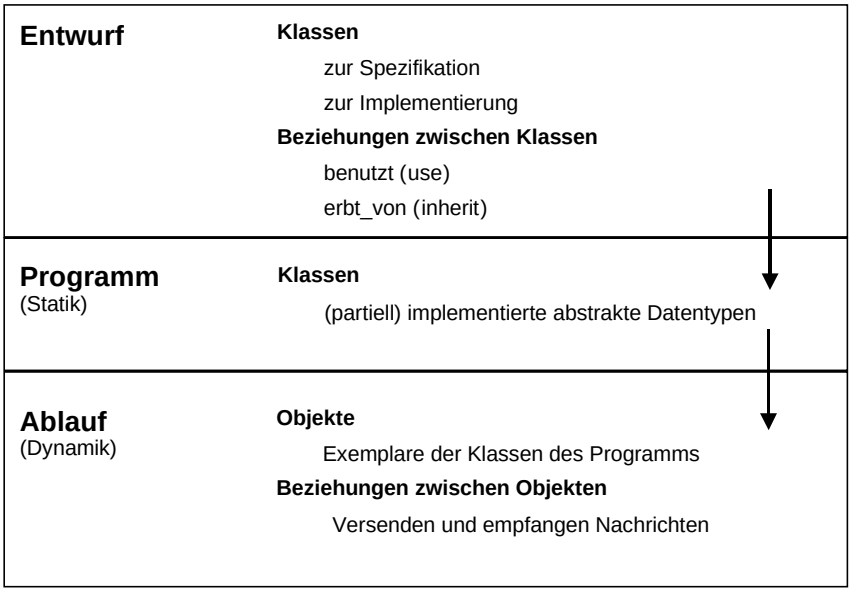
Ablauf



E/A

Diskussion

Merkmale objektorientierter Architekturen - 2



Was haben wir gelernt!

- **Klassifikation**
  - objektbasiert, klassenbasiert
  - objektorientiert
- **Klassen sind (partiell) implementierte ADTs**
  - abstrakte Klassen
  - konkrete Klassen
- **Vererbung**
  - dient dazu, Spezialisierungsbeziehung zwischen Begriffen programmiertechnisch zu realisieren.
  - Unterschied zwischen Einfach- und Mehrfachvererbung
- **Polymorphismus**
  - kann mit Hilfe der Vererbung realisiert werden
  - führt zum dynamischen Binden von Implementierungen

Glossar

- wissenbasierte, wertorientierte, prozedurale, objektorientierte Programmierung
- Objektmodul, ADT, Klasse
- Struktur eines objektorientierten Programms
- Nachrichten, Methoden, Laufzeitzustand eines objektorientierten Systems
- abstrakte Klasse, konkrete Klasse
- Definieren, Redefinieren, Erweitern von Methoden bei einem Spezialisierungsschritt
- Vererbung (Spezialisierung), Verallgemeinerung (Generalisierung)
- Einfach-, Mehrfachvererbung; strikte Vererbung, nichtstrikte Vererbung
- Polymorphismus, Polymorphismus durch Vererbung
- dynamisches Binden (Dispatching)

Vertragsmodell

- Konzept des Vertragsmodells
- Zusicherungen
- Realisierung von Zusicherungen mit Pragmas
- Exkurs Ausnahmebehandlung
- Zusicherungen mittels Ausnahmebehandlung

Erinnerung

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in ***bestimmten Situationen*** (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, stellen wir an der Schnittstelle entsprechende ***Testfunktionen*** (Sicherheitsabfragen) wie `IstLeer` oder `IstVoll` zur Verfügung.

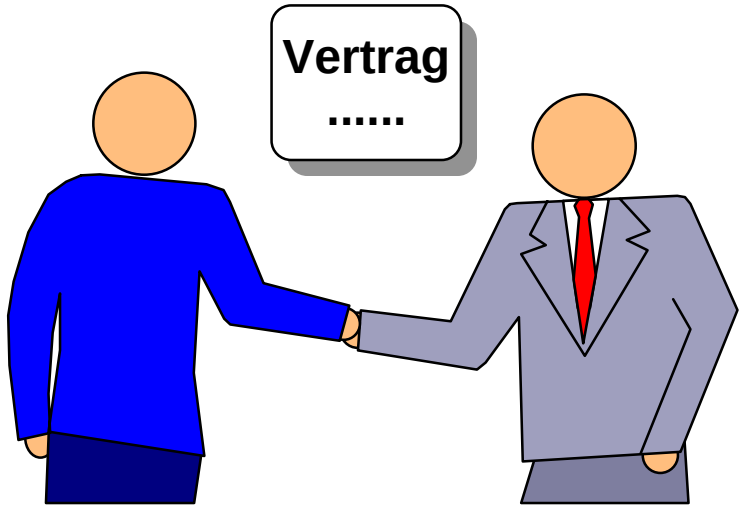
Einsatz der Testfunktionen

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Prüfen, ob die Operation angewedet werden darf.

- **Frage:**
  - Wer soll ***sicherstellen***, daß eine Operation immer richtig angewendet wird?
  - ***Nutzer*** oder ***Anbieter***? oder
  - zur Sicherheit auf ***beiden Seiten***?

Idee des Vertragsmodells



Nutzer (Client)

Anbieter (Server)

- **Vertrag**
  - zwischen Nutzer und Anbieter einer Operation regelt, wer der beiden Partner welche ***Verpflichtungen*** einhalten muß (und welchen ***Nutzen*** er dadurch hat)
- **Operation**
  - arbeitet korrekt, wenn sie vertragsgemäß ***benutzt*** bzw. ***realisiert*** wird.
- **Frage**
  - Wie kann ein solcher Vertrag ***programmtechnisch*** realisiert werden?

Arten und Nutzung

- **Zusicherungen**
  - sind eine Technik, um eine bestimmte Art von Verträgen zwischen Anbieter und Nutzer zu formulieren.

- **Zusicherungen werden formuliert als**
  - ***Vorbedingungen*** für Operationen
  - ***Nachbedingungen*** von Operationen
  - ***Invarianten*** von abstrakten Datentypen

- **Zusicherungen**
  - erhöhen die ***Benutzbarkeit***, indem sie diese formal definieren
  - verbessern die ***Testbarkeit***
  - verbessern die ***Fehlersuche*** (debugging)
  - verlangen vom Entwickler ***abstraktes*** Denken

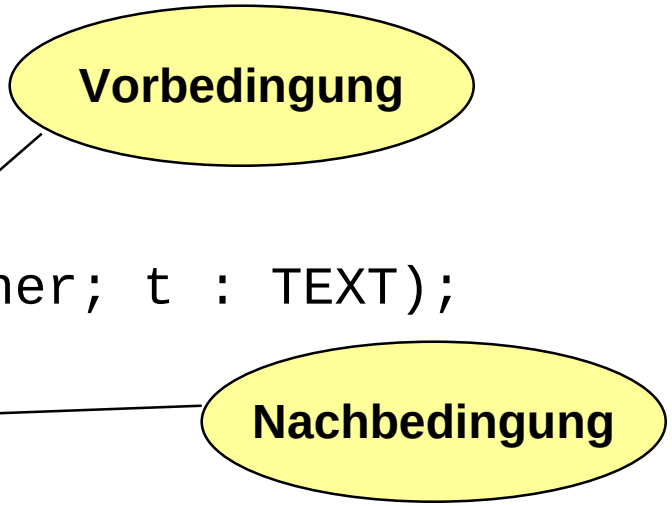
Vor- und Nachbedingungen

- **Vorbedingung**
  - beschreibt eine Bedingung, die der ***Nutzer*** (Aufrufer) einer Operation ***einhalten*** muß, damit die Operation korrekt arbeitet
- **Nachbedingung**
  - beschreibt einen Zustand, der nach dem erfolgreichen Ausführen der Operation vorhanden ist
  - diese garantiert der ***Anbieter***

Die Technik der Zusicherungen sollte insb. bei der Entwicklung von Programmkomponenten (Modulen) eingesetzt werden, die wiederverwendet werden sollen!

Erstes Beispiel

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  requires NOT IstVoll(o)
  ensures  NOT IstLeer(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  requires NOT IstLeer(o)
  ensures  NOT IstVoll(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .

END OrdnerADT.
```



Verpflichtung <-> Nutzen

- **Vorbedingungen sind**
  - Verpflichtungen für den Benutzer
  - Nutzen für den Anbieter
- **Nachbedingungen sind**
  - Nutzen für den Benutzer
  - Verpflichtungen für den Anbieter

Operation `EntnehmeText`

	Verpflichtungen	**Nutzen**
Nutzer	Der Ordner darf nicht leer sein.	Der zuletzt eingegebene Text wird entnommen und zurückgegeben.
Anbieter	Verändere den Ordner so, daß der zuletzt eingegebene Text entnommen wird.	Ist sicher, daß es noch einen Text im Ordner gibt

Invariante

- **Invariante beschreibt Bedingungen,**
  - die erfüllt sind, wenn Objekte eines ADTs ***erzeugt*** werden
  - die während der ***gesamten*** Lebenszeit der Objekte gelten
  - d.h. von allen Operationen ***nicht verletzt*** werden

- **Beispiel: ADT Ordner**
  - zu jedem Zeitpunkt im "Leben" eines Ordnerobjekts gilt:
  - **`anzahlTexte >= 0 AND anzahlTexte <= MaxTexte`**
  - ADT Ordner wird um eine interne Funktion erweitert, die die Invariante prüft.

```
PROCEDURE Invariante (o : Ordner): BOOLEAN =
BEGIN
  RETURN (o^.anzahlTexte >= 0 AND o^.anzahlTexte <= MaxTexte);
END Invariante;
```

ADT Ordner mit Invariante

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  requires NOT IstVoll(o)
  ensures  NOT IstLeer(o)
  ensures  Invariante(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  requires NOT IstLeer(o)
  ensures  NOT IstVoll(o)
  ensures  Invariante(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .
END OrdnerADT.
```

Pragmas in M3

Pragmas in Modula-3

- Pragmas sind Anweisungen an den ***Übersetzer***
- Sie ändern die ***Semantik*** des Programmes nicht
- Die Implementierung eines Pragmas ist ***übersetzerspezifisch***
- Pragmas können ***irgendwo*** im Programmtext auftreten
- Pragmas müssen einer vordefinierte Syntax entsprechen
  - *<* Pragmaname Parameter *>*

Das Pragma ASSERT

- `<* ASSERT AUSDRUCK *>`
- Der AUSDRUCK muß einen Wert vom Typ BOOLEAN liefern
- Der Ausdruck wird zur ***Laufzeit*** ausgewertet
- Ist das Ergebnis des Ausdrucks FALSE
  - wird ein ***Laufzeitfehler*** ausgelöst
  - und das Programm bricht ab!

Zusicherungen mit Pragma ASSERT

```
PROCEDURE EntnehmeText (VAR o: Ordner): TEXT =
VAR t : TEXT;
BEGIN
  <* ASSERT NOT IstLeer(o) *>

  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;

 <* ASSERT NOT IstVoll(o)  *>
 <* ASSERT Invariante(o)   *>

  RETURN t;
END EntnehmeText;
```

Laufzeitfehler

```
VAR ordner1, ordner2 : OrdnerADT.Ordner; t : TEXT;

BEGIN
  ordner1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ordner1, "Nicht immer...");
  t := OrdnerADT.EntnehmeText(ordner1);
  t := OrdnerADT.EntnehmeText(ordner1);
```

Diskussion: Einsatz von ASSERT

- **ASSERT**
  - bietet eine einfache Möglichkeit, Zusicherungen zu implementieren.
- **Nachteile**
  - "brutale" Implementierung: Programmabbruch
  - es wird keine Information über die Verletzung des Vertrages geliefert
  - es gibt keine Möglichkeit, auf die Verletzung des Vertrages zu reagieren
- **Bessere Lösung**
  - Entwerfen eines Moduls zur Prüfung von Zusicherungen

```
PROCEDURE Require (expr : BOOLEAN);
PROCEDURE Ensure  (expr : BOOLEAN);
```

- **Frage**
  - Was soll geschehen, wenn eine Zusicherung verletzt wird?
  - Verletzung: Ausnahmesituation!

Ausnahmen: Bsp. und Def.

- **Beispiel für Ausnahmesituationen**
  - Während des Schreibens einer Datei auf Diskette wird die Diskette entfernt.
  - Kein Plattenplatz mehr verfügbar.
  - Berechnung eines Addition ist größer als LAST(INTEGER).
  - Falsche Daten werden von einer Datei eingelesen.
- **Definition: Ausnahme**
  - IEEE Glossary: "An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception."
  - Ausnahmen sind Programmzustände, die nicht im normalen Programmablauf vorgesehen sind.

Merkmale von Ausnahmen

- **Merkmale**
  - Ausnahmen entstehen zur ***Laufzeit.***
  - Einige Programmiersprachen (Java, Ada, Modula-3) erlauben, benutzerdefinierte ***Ausnahmen*** zu deklarieren und ***Ausnahmebehandlung*** durchzuführen.
  - Beispiel: vorgegebene Ausnahme
    - *SIO.Error*
      Wird von den IO-Modulen erweckt, wenn Datei-Operationen nicht wie intendiert durchgeführt werden können.
- **Deklaration benutzerdefinierter Ausnahmen**
  - EXCEPTION <name>;
  - Wird eine Ausnahme von einer Schnittstelle exportiert, können auch die Klienten diese Ausnahme generieren.
- **Erwecken einer Ausnahme**
  - RAISE-Anweisung

Ausnahme und Ausnahmebehandlung

```
MODULE Ausnahme EXPORTS Main;
IMPORT SIO;
VAR eingabe : INTEGER;
BEGIN
  LOOP
    TRY
      SIO.PutLine ("Geben Sie bitte eine ganze Zahl ein:");
      eingabe := SIO.GetInt();
      EXIT;
    EXCEPT
      SIO.Error => SIO.PutLine ("*** Eingabeformat falsch");
                   line := SIO.GetLine();
    END;
  END;
  . . .
END Ausnahme.
```

Schützt Anweisungen, bzgl. dem Auftreten von Ausnahmen

Ausnahmebehandlung "exception handler"

Weiterleiten von Ausnahmen

- **Idee:**
  - Wenn in einer Prozedur eine Ausnahme nicht behandelt werden soll, dann kann diese Prozedur die Ausnahme an die ***sie rufende Prozedur*** weiterleiten.
  - So können Ausnahmen über ***mehrere Stufen*** weitergeleitet und an der entsprechenden Stelle behandelt werden.
- **Weiterleitung von Ausnahmen**
  - muß bei der Prozedur-Deklaration angegeben werden
  - PROCEDURE <name> signature RAISES {exc1, .. excN}
- **Beispiele:**
  - viele Operationen des Moduls `SIO` leiten die Ausnahme `Error` weiter
  - `PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};`

Kontrollfluß bei Ausnahmen

- Eine Ausnahme tritt in TRY-EXCEPT-Anweisung auf und die Ausnahme wird dort behandelt, dann
  - werden die in dem Ausnahmebehandler für die Ausnahme stehenden Anweisungen durchgeführt,
  - Das Programm wird anschließend nach dem END der TRY-EXCEPT-Anweisung ***fortgeführt***.

- Eine Ausnahme tritt in einem ***ungeschützten*** Bereich einer Prozedur auf und die Ausnahme ist Element der RAISES-Liste der Prozedur, dann,
  - wird die Prozedur abgebrochen und die Ausnahme an die die Prozedur rufende Prozedur ***weitergeleitet***

- Kann eine Ausnahme weder behandelt noch weitergeleitet werden, dann,
  - wird das Programm mit einem Laufzeitfehler ***abgebrochen***

Schnittstelle des Moduls `Assertion`

```
INTERFACE Assertion;

EXCEPTION Violated;
          Terminate;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                  RAISES {Violated};

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                  RAISES {Violated};

PROCEDURE EnableAssertions();
PROCEDURE DisableAssertions();

END Assertion.
```

Ausnahme `Violated` wird generiert, wenn eine Zusicherung verletzt wird

Prüfung der Zusicherungen kann unterdrückt werden

Implementierung Assertion - 1

```
MODULE Assertion;
IMPORT SIO;

VAR enabled : BOOLEAN;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                  RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Precondition violated: " & procName);
    RAISE Violated;
  END;
END Require;

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Postcondition violated: " & procName);
    RAISE Violated;
  END;
END Ensure;
```

Geschützte Variable

Implementierung Assertion - 2

```
PROCEDURE EnableAssertions()=
BEGIN
  enabled := TRUE;
END EnableAssertions;

PROCEDURE DisableAssertions()=
BEGIN
  enabled := FALSE;
END DisableAssertions;


BEGIN
  EnableAssertions();
END Assertion.
```

Ordner-Operationen mit Assertion - 1

```
INTERFACE OrdnerADT;

IMPORT Assertion AS As;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb     (VAR o: Ordner; t : TEXT)
                          RAISES {As.Violated};
PROCEDURE EntnehmeText   (VAR o: Ordner): TEXT
                          RAISES {As.Violated};
PROCEDURE IstVoll        (o: Ordner): BOOLEAN;
PROCEDURE IstLeer        (o: Ordner): BOOLEAN;
PROCEDURE Beschrifte     (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung(o: Ordner): TEXT;
PROCEDURE Anlegen        ( ) : Ordner RAISES {As. Violated};

END OrdnerADT.
```

Zusicherungen mittels Ausnahmebeh.

Ordner-Operationen mit Assertion - 2

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)
                      RAISES {As. Violated} =
BEGIN
  As.Require (NOT IstVoll(o), "OrdnerADT.LegeTextAb");
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
  As.Ensure (NOT IstLeer(o), "OrdnerADT.LegeTextAb");
  As.Ensure (Invariante(o), "OrdnerADT.LegeTextAb");
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT
                        RAISES {As. Violated} =
VAR t : TEXT;
BEGIN
  As.Require (NOT IstLeer(o), "OrdnerADT.EntnehmeText");
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  As.Ensure (NOT IstVoll(o), "OrdnerADT.EntnehmeText");
  As.Ensure (Invariante(o), "OrdnerADT.EntnehmeText");
  RETURN t;
END EntnehmeText;
```

Umgang mit den Ausnahmen - 1

Empfehlung

- Der aufrufende Block klammert alle Anweisungen in einer TRY-EXCEPT-Anweisung.
- Im EXCEPT-Teil werden noch mögliche Abschluß-Operationen durchgeführt (z.B. Schließen von Dateien) und eine entsprechende Meldung ausgegeben und die Ausnahme `Terminate` generiert.

```
PROCEDURE ArbeiteMitOrdner() RAISES {As.Terminate} =
VAR ordner1 : OrdnerADT.Ordner;
BEGIN
  TRY
    ordner1 := OrdnerADT.Anlegen();
    OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
    SIO.PutLine (OrdnerADT.EntnehmeText(ordner1));
    ...
  EXCEPT
    As.Violated =>
      SIO.PutLine ("*** Called from: ArbeiteMitOrdner");
      RAISE As.Terminate;
  END;
END ArbeiteMitOrdner.
```

Umgang mit den Ausnahmen - 2

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Assertion AS As;
IMPORT OrdnerADT,  SIO;

PROCEDURE ArbeiteMitOrdner()RAISES {As.Terminate}=
BEGIN
  ...
END ArbeiteMitOrdner;

BEGIN
  As.EnableAssertions();
  TRY
    ArbeiteMitOrdner();
  EXCEPT
    As.Terminate =>
      SIO.PutLine ("*** Program terminated ");
  END
END Ordner_Test.
```

```
*** Precondition violated: OrdnerADT.EntnehmeText
*** Called from: ArbeiteMitOrdner
*** Program terminated
```

Arbeiten mit Zusicherungen - 1

- **Im Rumpf einer Operation darf die Vorbedingung nicht geprüft werden**
  - widerspricht dem herkömmlichen ***defensiven*** Programmieren
- **Vorteil**
  - schon mittelgroße Systeme enthalten ca. 10 -20% Code, um solche Eingangsprüfungen durchzuführen
  - dabei entstehen komplexe Prüfungen
  - Prüfroutinen sind häufig Ursachen für Fehler

```
PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
END Wurzel ;

PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
  if x < 0 then
    “handle error”
  else
    RETURN (x * x);
  end
END Wurzel ;
```

Arbeiten mit Zusicherungen - 2

- **Zusicherungen sollen nicht verwendet werden, um Spezialfälle zu behandeln**
  - Zusicherungen sind Aussagen über die ***korrekte*** Nutzung.
  - Sollen Spezialfälle behandelt werden, dann werden herkömmliche ***Kontrollanweisungen*** verwendet (IF oder CASE).
- **Beispiel**
  - Wenn `Wurzel` den Fall `x < 0` als Spezialfall behandeln soll, dann ist das zu programmieren.
  - Wenn `x >= 0` die Vorbedingung ist, dann ist ein Aufruf

    ```
    Wurzel (-1);
    ```

    nicht erlaubt, also eine ***nicht vertragskonforme Benutzung***!
- **Wird eine Zusicherung verletzt (zur Laufzeit)**
  - Vorbedingung: Nutzer hat den Vertrag verletzt
  - Nachbedingung: Anbieter hat den Vertrag verletzt

Was haben wir gelernt?

- Vertragsmodell: Vertrag zwischen Nutzer (Client) und Anbieter (Server),
- Konsistenz durch Vor- und Nachbedingungen sowie Invarianten
- Pragmas, Pragma `Assert`
- Ausnahmen: Motivation, Ausnahmedeklaration, Erwecken einer Ausnahme, Behandlung durch Ausnahmebehandler, Weiterreichen einer Ausnahme
- Zusicherungen mit Ausnahmen, allgemeingültiges Ausnahmebehandlungsmodul `Assertion`, Verwendung bei Clienten und bei Server

Glossar

- **Vertragsmodell, Vorbedingungen und Benutzerverpflichtung, Nachbedingungen und Anbieterverpflichtung**
- **Formalisierung von Zusicherungen, Vorbedingungen, Nachbedingungen, Invarianten**
- **ADT-Schnittstelle mit Zusicherung**
- **Pragmas in Modula-3, vordefiniertes Pragma `Assert`, Nutzung für Zusicherungsrealisierung**
- **Ausnahmesituationen, Definition Ausnahme, (vordefinierte und) benutzerdefinierte Ausnahme, Ausnahmedeklaration, Erwecken einer Ausnahme, Ausnahmebehandlung und `TRY-EXCEPT`-Anweisung**
- **Ausnahmebehandlung im Ausnahmebehandler, ungeschützte Ausnahmen und weiterreichen, falls gerufene Prozedur diese Ausnahme im Kopf angibt, Programmabbruch, falls weder behandelt noch weitergeleitet**
- **Ausnahmen für die Realisierung von Zusicherungen**

Vertragsmodell

- **Konzept des Vertragsmodells**
- **Zusicherungen**
- **Realisierung von Zusicherungen mit Pragmas**
- **Exkurs Ausnahmebehandlung**
- **Zusicherungen mittels Ausnahmebehandlung**

Erinnerung

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in ***bestimmten Situationen*** (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, stellen wir an der Schnittstelle entsprechende ***Testfunktionen*** (Sicherheitsabfragen) wie `IstLeer` oder `IstVoll` zur Verfügung.

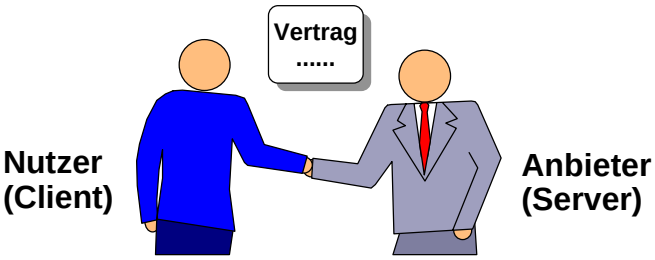
Einsatz der Testfunktionen

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Prüfen, ob die Operation angewedet werden darf.

- **Frage:**
  - Wer soll ***sicherstellen***, daß eine Operation immer richtig angewendet wird?
  - ***Nutzer*** oder ***Anbieter***? oder
  - zur Sicherheit auf ***beiden Seiten***?

Idee des Vertragsmodells



- **Vertrag**
  - zwischen Nutzer und Anbieter einer Operation regelt, wer der beiden Partner welche ***Verpflichtungen*** einhalten muß (und welchen ***Nutzen*** er dadurch hat)
- **Operation**
  - arbeitet korrekt, wenn sie vertragsgemäß ***benutzt*** bzw. ***realisiert*** wird.
- **Frage**
  - Wie kann ein solcher Vertrag ***programmtechnisch*** realisiert werden?

Arten und Nutzung

- **Zusicherungen**
  - sind eine Technik, um eine bestimmte Art von Verträgen zwischen Anbieter und Nutzer zu formulieren.
- **Zusicherungen werden formuliert als**
  - ***Vorbedingungen*** für Operationen
  - ***Nachbedingungen*** von Operationen
  - ***Invarianten*** von abstrakten Datentypen
- **Zusicherungen**
  - erhöhen die ***Benutzbarkeit***, indem sie diese formal definieren
  - verbessern die ***Testbarkeit***
  - verbessern die ***Fehlersuche*** (debugging)
  - verlangen vom Entwickler ***abstraktes*** Denken

Vor- und Nachbedingungen

- **Vorbedingung**
  - beschreibt eine Bedingung, die der ***Nutzer*** (Aufrufer) einer Operation ***einhalten*** muß, damit die Operation korrekt arbeitet
- **Nachbedingung**
  - beschreibt einen Zustand, der nach dem erfolgreichen Ausführen der Operation vorhanden ist
  - diese garantiert der ***Anbieter***

Die Technik der Zusicherungen sollte insb. bei der Entwicklung von Programmkomponenten (Modulen) eingesetzt werden, die wiederver-wendet werden sollen!

Erstes Beispiel

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  requires NOT IstVoll(o)
  ensures  NOT IstLeer(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  requires NOT IstLeer(o)
  ensures  NOT IstVoll(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .

END OrdnerADT.
```

Vorbedingung

Nachbedingung

Verpflichtung <-> Nutzen

- **Vorbedingungen sind**
  - Verpflichtungen für den Benutzer
  - Nutzen für den Anbieter
- **Nachbedingungen sind**
  - Nutzen für den Benutzer
  - Verpflichtungen für den Anbieter

Operation `EntnehmeText`

	Verpflichtungen	Nutzen
Nutzer	Der Ordner darf nicht leer sein.	Der zuletzt eingegebene Text wird entnommen und zurückgegeben.
Anbieter	Verändere den Ordner so, daß der zuletzt eingegebene Text entnommen wird.	Ist sicher, daß es noch einen Text im Ordner gibt

Invariante

- **Invariante beschreibt Bedingungen,**
  - die erfüllt sind, wenn Objekte eines ADTs ***erzeugt*** werden
  - die während der ***gesamten*** Lebenszeit der Objekte gelten
  - d.h. von allen Operationen ***nicht verletzt*** werden

- **Beispiel: ADT Ordner**
  - zu jedem Zeitpunkt im "Leben" eines Ordnerobjekts gilt:
  - **`anzahlTexte >= 0 AND anzahlTexte <= MaxTexte`**
  - ADT Ordner wird um eine interne Funktion erweitert, die die Invariante prüft.

```
PROCEDURE Invariante (o : Ordner): BOOLEAN =
BEGIN
  RETURN (o^.anzahlTexte >= 0 AND o^.anzahlTexte <= MaxTexte);
END Invariante;
```


ADT Ordner mit Invariante

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  requires NOT IstVoll(o)
  ensures  NOT IstLeer(o)
  ensures  Invariante(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  requires NOT IstLeer(o)
  ensures  NOT IstVoll(o)
  ensures  Invariante(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .
END OrdnerADT.
```

Pragmas in M3

Pragmas in Modula-3

- Pragmas sind Anweisungen an den ***Übersetzer***
- Sie ändern die ***Semantik*** des Programmes nicht
- Die Implementierung eines Pragmas ist ***übersetzerspezifisch***
- Pragmas können ***irgendwo*** im Programmtext auftreten
- Pragmas müssen einer vordefinierte Syntax entsprechen
  - *<* Pragmaname Parameter *>*

Das Pragma ASSERT

- `<* ASSERT AUSDRUCK *>`
- Der AUSDRUCK muß einen Wert vom Typ BOOLEAN liefern
- Der Ausdruck wird zur ***Laufzeit*** ausgewertet
- Ist das Ergebnis des Ausdrucks FALSE
  - wird ein ***Laufzeitfehler*** ausgelöst
  - und das Programm bricht ab!

Zusicherungen mit Pragma ASSERT

```
PROCEDURE EntnehmeText (VAR o: Ordner): TEXT =
VAR t : TEXT;
BEGIN
  <* ASSERT NOT IstLeer(o) *>

  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;

 <* ASSERT NOT IstVoll(o)  *>
 <* ASSERT Invariante(o)   *>

  RETURN t;
END EntnehmeText;
```

```
VAR ordner1, ordner2 : OrdnerADT.Ordner; t : TEXT;

BEGIN
  ordner1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ordner1, "Nicht immer...");
  t := OrdnerADT.EntnehmeText(ordner1);
  t := OrdnerADT.EntnehmeText(ordner1);
```

Laufzeitfehler

Realisierung mit Pragmas

Diskussion: Einsatz von ASSERT

- **ASSERT**
  - bietet eine einfache Möglichkeit, Zusicherungen zu implementieren.
- **Nachteile**
  - "brutale" Implementierung: Programmabbruch
  - es wird keine Information über die Verletzung des Vertrages geliefert
  - es gibt keine Möglichkeit, auf die Verletzung des Vertrages zu reagieren
- **Bessere Lösung**
  - Entwerfen eines Moduls zur Prüfung von Zusicherungen

```
PROCEDURE Require (expr : BOOLEAN);
PROCEDURE Ensure  (expr : BOOLEAN);
```

- **Frage**
  - Was soll geschehen, wenn eine Zusicherung verletzt wird?
  - Verletzung: Ausnahmesituation!

Exkurs Ausnahme-behandlung

Ausnahmen: Bsp. und Def.

- **Beispiel für Ausnahmesituationen**
  - Während des Schreibens einer Datei auf Diskette wird die Diskette entfernt.
  - Kein Plattenplatz mehr verfügbar.
  - Berechnung eines Addition ist größer als LAST(INTEGER).
  - Falsche Daten werden von einer Datei eingelesen.
- **Definition: Ausnahme**
  - IEEE Glossary: "An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception."
  - Ausnahmen sind Programmzustände, die nicht im normalen Programmablauf vorgesehen sind.

Merkmale von Ausnahmen

- **Merkmale**
  - Ausnahmen entstehen zur ***Laufzeit.***
  - Einige Programmiersprachen (Java, Ada, Modula-3) erlauben, benutzerdefinierte ***Ausnahmen*** zu deklarieren und ***Ausnahmebehandlung*** durchzuführen.
  - Beispiel: vorgegebene Ausnahme
    - *SIO.Error*
      Wird von den IO-Modulen erweckt, wenn Datei-Operationen nicht wie intendiert durchgeführt werden können.
- **Deklaration benutzerdefinierter Ausnahmen**
  - EXCEPTION <name>;
  - Wird eine Ausnahme von einer Schnittstelle exportiert, können auch die Klienten diese Ausnahme generieren.
- **Erwecken einer Ausnahme**
  - RAISE-Anweisung

Ausnahme und Ausnahmebehandlung

```
MODULE Ausnahme EXPORTS Main;
IMPORT SIO;
VAR eingabe : INTEGER;
BEGIN
  LOOP
    TRY
      SIO.PutLine ("Geben Sie bitte eine ganze Zahl ein:");
      eingabe := SIO.GetInt();
      EXIT;
    EXCEPT
      SIO.Error => SIO.PutLine ("*** Eingabeformat falsch");
                   line := SIO.GetLine();
    END;
  END;
  . . .
END Ausnahme.
```

Schützt Anweisungen, bzgl. dem Auftreten von Ausnahmen

Ausnahmebehandlung "exception handler"

Weiterleiten von Ausnahmen

- **Idee:**
  - Wenn in einer Prozedur eine Ausnahme nicht behandelt werden soll, dann kann diese Prozedur die Ausnahme an die ***sie rufende Prozedur*** weiterleiten.
  - So können Ausnahmen über ***mehrere Stufen*** weitergeleitet und an der entsprechenden Stelle behandelt werden.
- **Weiterleitung von Ausnahmen**
  - muß bei der Prozedur-Deklaration angegeben werden
  - PROCEDURE <name> signature RAISES {exc1, .. excN}
- **Beispiele:**
  - viele Operationen des Moduls `SIO` leiten die Ausnahme `Error` weiter
  - `PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};`

Kontrollfluß bei Ausnahmen

- Eine Ausnahme tritt in TRY-EXCEPT-Anweisung auf und die Ausnahme wird dort behandelt, dann
  - werden die in dem Ausnahmebehandler für die Ausnahme stehenden Anweisungen durchgeführt,
  - Das Programm wird anschließend nach dem END der TRY-EXCEPT-Anweisung ***fortgeführt***.
- Eine Ausnahme tritt in einem ***ungeschützten*** Bereich einer Prozedur auf und die Ausnahme ist Element der RAISES-Liste der Prozedur, dann,
  - wird die Prozedur abgebrochen und die Ausnahme an die die Prozedur rufende Prozedur ***weitergeleitet***
- Kann eine Ausnahme weder behandelt noch weitergeleitet werden, dann,
  - wird das Programm mit einem Laufzeitfehler ***abgebrochen***

Schnittstelle des Moduls `Assertion`

```
INTERFACE Assertion;

EXCEPTION Violated;
          Terminate;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated};

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated};

PROCEDURE EnableAssertions();
PROCEDURE DisableAssertions();

END Assertion.
```

Ausnahme `Violated` wird generiert, wenn eine Zusicherung verletzt wird

Prüfung der Zusicherungen kann unterdrückt werden

Implementierung `Assertion` - 1

```
MODULE Assertion;
IMPORT SIO;

VAR enabled : BOOLEAN;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                  RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Precondition violated: " & procName);
    RAISE Violated;
  END;
END Require;

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Postcondition violated: " & procName);
    RAISE Violated;
  END;
END Ensure;
```

Geschützte Variable

Zusicherungen mittels Ausnahmebeh.

Implementierung Assertion - 2

```
PROCEDURE EnableAssertions()=
BEGIN
  enabled := TRUE;
END EnableAssertions;

PROCEDURE DisableAssertions()=
BEGIN
  enabled := FALSE;
END DisableAssertions;


BEGIN
  EnableAssertions();
END Assertion.
```


Zusicherungen mittels Ausnahmebeh.

Ordner-Operationen mit Assertion - 1

```
INTERFACE OrdnerADT;

IMPORT Assertion AS As;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb     (VAR o: Ordner; t : TEXT)
                          RAISES {As.Violated};
PROCEDURE EntnehmeText   (VAR o: Ordner): TEXT
                          RAISES {As.Violated};
PROCEDURE IstVoll        (o: Ordner): BOOLEAN;
PROCEDURE IstLeer        (o: Ordner): BOOLEAN;
PROCEDURE Beschrifte     (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung(o: Ordner): TEXT;
PROCEDURE Anlegen        ( ) : Ordner RAISES {As. Violated};

END OrdnerADT.
```

Ordner-Operationen mit Assertion - 2

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)
                     RAISES {As. Violated} =
BEGIN
  As.Require (NOT IstVoll(o), "OrdnerADT.LegeTextAb");
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
  As.Ensure (NOT IstLeer(o), "OrdnerADT.LegeTextAb");
  As.Ensure (Invariante(o), "OrdnerADT.LegeTextAb");
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT
                       RAISES {As. Violated} =
VAR t : TEXT;
BEGIN
  As.Require (NOT IstLeer(o), "OrdnerADT.EntnehmeText");
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  As.Ensure (NOT IstVoll(o), "OrdnerADT.EntnehmeText");
  As.Ensure (Invariante(o), "OrdnerADT.EntnehmeText");
  RETURN t;
END EntnehmeText;
```


Umgang mit den Ausnahmen - 1

- **Empfehlung**
  - Der aufrufende Block klammert alle Anweisungen in einer TRY-EXCEPT-Anweisung.
  - Im EXCEPT-Teil werden noch mögliche Abschluß-Operationen durchgeführt (z.B. Schließen von Dateien) und eine entsprechende Meldung ausgegeben und die Ausnahme `Terminate` generiert.

```
PROCEDURE ArbeiteMitOrdner() RAISES {As.Terminate} =
VAR ordner1 : OrdnerADT.Ordner;
BEGIN
  TRY
    ordner1 := OrdnerADT.Anlegen();
    OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
    SIO.PutLine (OrdnerADT.EntnehmeText(ordner1));
    ...
  EXCEPT
    As.Violated =>
      SIO.PutLine ("*** Called from: ArbeiteMitOrdner");
      RAISE As.Terminate;
  END;
END ArbeiteMitOrdner.
```

Umgang mit den Ausnahmen - 2

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Assertion AS As;
IMPORT OrdnerADT,  SIO;

PROCEDURE ArbeiteMitOrdner()RAISES {As.Terminate}=
BEGIN
  ...
END ArbeiteMitOrdner;

BEGIN
  As.EnableAssertions();
  TRY
    ArbeiteMitOrdner();
  EXCEPT
    As.Terminate =>
      SIO.PutLine ("*** Program terminated ");
  END
END Ordner_Test.
```

```
*** Precondition violated: OrdnerADT.EntnehmeText
*** Called from: ArbeiteMitOrdner
*** Program terminated
```


Arbeiten mit Zusicherungen - 1

- **Im Rumpf einer Operation darf die Vorbedingung nicht geprüft werden**
  - widerspricht dem herkömmlichen ***defensiven*** Programmieren
- **Vorteil**
  - schon mittelgroße Systeme enthalten ca. 10 -20% Code, um solche Eingangsprüfungen durchzuführen
  - dabei entstehen komplexe Prüfungen
  - Prüfroutinen sind häufig Ursachen für Fehler

```
PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
END Wurzel ;

PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
  if x < 0 then
    "handle error"
  else
    RETURN (x * x);
  end
END Wurzel ;
```

Zusicherungen mittels Ausnahmebeh.

Arbeiten mit Zusicherungen - 2

- **Zusicherungen sollen nicht verwendet werden, um Spezialfälle zu behandeln**
  - Zusicherungen sind Aussagen über die ***korrekte*** Nutzung.
  - Sollen Spezialfälle behandelt werden, dann werden herkömmliche ***Kontrollanweisungen*** verwendet (IF oder CASE).
- **Beispiel**
  - Wenn `Wurzel` den Fall x < 0 als Spezialfall behandeln soll, dann ist das zu programmieren.
  - Wenn x >= 0 die Vorbedingung ist, dann ist ein Aufruf

    ```
    Wurzel (-1);
    ```

    nicht erlaubt, also eine ***nicht vertragskonforme Benutzung***!
- **Wird eine Zusicherung verletzt (zur Laufzeit)**
  - Vorbedingung: Nutzer hat den Vertrag verletzt
  - Nachbedingung: Anbieter hat den Vertrag verletzt

Was haben wir gelernt?

- **Vertragsmodell: Vertrag zwischen Nutzer (Client) und Anbieter (Server),**
- **Konsistenz durch Vor- und Nachbedingungen sowie Invarianten**
- **Pragmas, Pragma `Assert`**
- **Ausnahmen: Motivation, Ausnahmedeklaration, Erwecken einer Ausnahme, Behandlung durch Ausnahmebehandler, Weiterreichen einer Ausnahme**
- **Zusicherungen mit Ausnahmen, allgemeingültiges Ausnahmebehandlungsmodul `Assertion`, Verwendung bei Clienten und bei Server**

Glossar

- **Vertragsmodell, Vorbedingungen und Benutzerverpflichtung, Nachbedingungen und Anbieterverpflichtung**
- **Formalisierung von Zusicherungen, Vorbedingungen, Nachbedingungen, Invarianten**
- **ADT-Schnittstelle mit Zusicherung**
- **Pragmas in Modula-3, vordefiniertes Pragma `Assert`, Nutzung für Zusicherungsrealisierung**
- **Ausnahmesituationen, Definition Ausnahme, (vordefinierte und) benutzerdefinierte Ausnahme, Ausnahmedeklaration, Erwecken einer Ausnahme, Ausnahmebehandlung und `TRY-EXCEPT`-Anweisung**
- **Ausnahmebehandlung im Ausnahmebehandler, ungeschützte Ausnahmen und weiterreichen, falls gerufene Prozedur diese Ausnahme im Kopf angibt, Programmabbruch, falls weder behandelt noch weitergeleitet**
- **Ausnahmen für die Realisierung von Zusicherungen**

Vordefinierte Bausteine: Beispiel Dateien

- **Dateisystem**
- **Operationen auf Dateien**
  - Lesen
  - Schreiben
- **Dateien in Modula-3**
  - wichtige Datei-Operationen

Dateien: Zweck

- **Verarbeiten von Daten**
  - Daten müssen in vielen Fällen dauerhaft (***persistent)*** gespeichert werden
- **Bisher:**
  - Daten wurden im ***Arbeitsspeicher*** zur Laufzeit des Programms erzeugt
  - Nachdem das Programm beendet ist, sind diese Daten ***verloren***
  - "***flüchtiger***" Speicher
- **Hintergrundspeicher**
  - persistenter Speicher
  - Diskette, CD, Festplatte etc.
- **Frage:**
  - Wie können wir Daten aus dem Arbeitsspeicher in den Hintergrundspeicher bringen und umgekehrt?

Dateisystem

- **Betriebssystem:**
  - stellt ***Dienstleistungen*** zur Verfügung, damit der Umgang mit dem Rechner einfach und mit ***bestimmten Diensten*** möglich ist
  - eine angebotene Dienstleistung des Betriebssystems ist das ***Dateisystem***
- **Dateisystem**
  - erlaubt, den Hintergrundspeicher in ***einzelnen Bereiche*** aufzuteilen
  - diese nennt man ***Dateien*** (file)
  - Dateien können in sogenannten ***Verzeichnissen*** gruppiert werden (directory)
  - jede Datei hat einen ***Namen***
  - aus der Sicht des Rechners ist eine Datei eine Folge von ***Informationseinheiten*** (Bytes)
  - ein Verzeichnis verwaltet für alle seine Dateien den Namen und die Position, wo die Dateien physisch auf dem Hintergrundspeicher liegen

Dateien: Zweck und Formen

- **Dateien können**
  - angelegt und gelöscht werden
  - gelesen und geschrieben werden
- **Dabei gibt es zwei Dateiformen:**
  - ***Sequentiell***
    - Daten werden von Anfang bis zum Schluß der Datei ***nacheinander*** gelesen (geschrieben).
    - Es gibt keine Möglichkeit einen Ausschnitt der Datei zu ***überspringen***
  - ***Direkt***
    - Es kann eine ***Position*** angegeben werden, ab der gelesen (geschrieben) werden soll
    - Wechseln der Position ist ***beliebig*** möglich
- **Das Betriebssystem**
  - kennt für jede Datei einen Zähler, der die aktuelle Position kennzeichnet.

Arbeiten mit Dateien

Programmiersysteme

- bieten in der Regel Möglichkeiten und Mechanismen, um vom Programm aus ***Dateioperationen*** ausführen zu können
- dabei werden i.d.R. nicht die Funktionen des Betriebssystems benutzt
- Programmiersysteme stellen eine ***abstrakte Schnittstelle*** zum Dateisystem zur Verfügung

Wichtige Dateioperationen

- Datei ***öffnen***
  - öffnen zum Lesen
  - öffnen zum Schreiben (von vorne)
  - öffnen zum Schreiben (neue Zeichen werden hinten angehängt)
- neue Datei ***erzeugen***
- ***lesen*** und ***schreiben***
- Abfragen des ***Dateiendes*** (end of file, EOF)
- Datei ***schließen***

Sprachumgebung, E/A-Strom

- **Sprachumgebung von Modula-3**
  - stellt Module zur Verfügung, um Dateien manipulieren zu können
- **Ein- / Ausgabestrom**
  - Mit einem Ein- / Ausgabestrom kann auf eine Datei zugegriffen werden
  - In der Standardbibliothek sind diese definiert
    - `reader, writer`
  - Ein Ein- / Ausgabestrom kann bei seiner Initialisierung mit einer Datei verbunden werden

```
IMPORT IO, Rd, Wr;
VAR eingabestrom : Rd.T;
    ausgabestrom : Wr.T;
…
eingabestrom := IO.OpenRead ("eingabe.txt");
ausgabestrom := IO.OpenWrite("ausgabe.txt");
```

Beispiel: Kopieren einer Datei

```
IMPORT SIO,IO;
IMPORT Wr AS Writer, Rd AS Reader;

PROCEDURE KopiereDatei (original, kopie : TEXT)=
VAR original_datei : Reader.T;
    kopie_datei    : Writer.T;
    zeile : TEXT;
BEGIN
  original_datei := IO.OpenRead (orginal);
  kopie_datei := IO.OpenWrite(kopie);
  WHILE NOT IO.EOF(original_datei) DO
    zeile := SIO.GetLine(original_datei);
    SIO.PutLine(zeile, kopie_datei);
  END;
  Reader.Close(original_datei);
  Writer.Close(kopie_datei);
END KopiereDatei;
```

Wichtige Dateioperationen in M3

Modul: IO

- PROCEDURE **EOF** (rd: Rd.T := NIL): BOOLEAN;
  - *Return TRUE iff rd is at end-of-file.*
- PROCEDURE **OpenRead** (f: TEXT): Rd.T;
  - *Open the file name f for reading and return a reader on its contents. If the file doesn't exist or is not readable, return NIL.*
- PROCEDURE **OpenWrite** (f: TEXT): Wr.T;
  - *Open the file named f for writing and return a writer on its contents. If the file does not exist it will be created. If the process does not have the authority to modify or create the file, return NIL.*

Modul: Rd und Wr

- PROCEDURE **Close** (rd: T) RAISES {Failure, Alerted};
  - *Release any resources associated with rd and set closed(rd) := TRUE.*
- PROCEDURE **Close** (wr: T) RAISES {Failure, Alerted};
  - *Flush wr, release any resources associated with wr, and set closed(wr) := TRUE.*

Was haben wir gelernt?

- Dateien für die persistente Speicherung
- Dateisysteme als Teil des Betriebssystems
- Dateiformen (sequentielle / direkte Dateien), Modi (Lese-, Schreibdateien)
- interne Dateien, externe Dateien und Bindung / Auflösung der Bindung beim Öffnen oder Schließen
- Anschluß von Dateien an das Programmiersystem über vordefinierte E/A-Bausteine
- Dateioperationen für Dateiformen, Eingabe- oder Ausgabedateien

Glossar

- Hintergrundspeicherarten
- flüchtiger, persistenter Speicher
- Betriebssystem, Dateisystem, Anschluß von seiten des Programmiersystems
- Datei (file), Verzeichnis (directory)
- externe, interne Dateien, Namen für beides
- Dateiformen, Eingabe-/Ausgabedatei
- Dateioperationen, Öffnen, Schließen, Lesen, Schreiben, Positionierung, Abfrage Dateiende
- Einschränkung der Operationen bei bestimmten Dateiformen, Ein- oder Ausgabedateien

Vordefinierte Bausteine: Beispiel Dateien

- **Dateisystem**
- **Operationen auf Dateien**
  - Lesen
  - Schreiben
- **Dateien in Modula-3**
  - wichtige Datei-Operationen

Datei-system

Dateien: Zweck

- **Verarbeiten von Daten**
  - Daten müssen in vielen Fällen dauerhaft (***persistent)*** gespeichert werden
- **Bisher:**
  - Daten wurden im ***Arbeitsspeicher*** zur Laufzeit des Programms erzeugt
  - Nachdem das Programm beendet ist, sind diese Daten ***verloren***
  - "***flüchtiger***" Speicher
- **Hintergrundspeicher**
  - persistenter Speicher
  - Diskette, CD, Festplatte etc.
- **Frage:**
  - Wie können wir Daten aus dem Arbeitsspeicher in den Hintergrundspeicher bringen und umgekehrt?

Datei-system

Dateisystem

- **Betriebssystem:**
  - stellt ***Dienstleistungen*** zur Verfügung, damit der Umgang mit dem Rechner einfach und mit ***bestimmten Diensten*** möglich ist
  - eine angebotene Dienstleistung des Betriebssystems ist das ***Dateisystem***
- **Dateisystem**
  - erlaubt, den Hintergrundspeicher in ***einzelnen Bereiche*** aufzuteilen
  - diese nennt man ***Dateien*** (file)
  - Dateien können in sogenannten ***Verzeichnissen*** gruppiert werden (directory)
  - jede Datei hat einen ***Namen***
  - aus der Sicht des Rechners ist eine Datei eine Folge von ***Informationseinheiten*** (Bytes)
  - ein Verzeichnis verwaltet für alle seine Dateien den Namen und die Position, wo die Dateien physisch auf dem Hintergrundspeicher liegen

Operationen auf Dateien

Dateien: Zweck und Formen

- **Dateien können**
  - angelegt und gelöscht werden
  - gelesen und geschrieben werden
- **Dabei gibt es zwei Dateiformen:**
  - ***Sequentiell***
    - Daten werden von Anfang bis zum Schluß der Datei ***nacheinander*** gelesen (geschrieben).
    - Es gibt keine Möglichkeit einen Ausschnitt der Datei zu ***überspringen***
  - ***Direkt***
    - Es kann eine ***Position*** angegeben werden, ab der gelesen (geschrieben) werden soll
    - Wechseln der Position ist ***beliebig*** möglich
- **Das Betriebssystem**
  - kennt für jede Datei einen Zähler, der die aktuelle Position kennzeichnet.

Operationen auf Dateien

Arbeiten mit Dateien

- **Programmiersysteme**
  - bieten in der Regel Möglichkeiten und Mechanismen, um vom Programm aus ***Dateioperationen*** ausführen zu können
  - dabei werden i.d.R. nicht die Funktionen des Betriebssystems benutzt
  - Programmiersysteme stellen eine ***abstrakte Schnittstelle*** zum Dateisystem zur Verfügung
- **Wichtige Dateioperationen**
  - Datei ***öffnen***
    - öffnen zum Lesen
    - öffnen zum Schreiben (von vorne)
    - öffnen zum Schreiben (neue Zeichen werden hinten angehängt)
  - neue Datei ***erzeugen***
  - ***lesen*** und ***schreiben***
  - Abfragen des ***Dateiendes*** (end of file, EOF)
  - Datei ***schließen***

Dateien in Modula-3

Sprachumgebung, E/A-Strom

- **Sprachumgebung von Modula-3**
  - stellt Module zur Verfügung, um Dateien manipulieren zu können
- **Ein- / Ausgabestrom**
  - Mit einem Ein- / Ausgabestrom kann auf eine Datei zugegriffen werden
  - In der Standardbibliothek sind diese definiert
    - `reader, writer`
  - Ein Ein- / Ausgabestrom kann bei seiner Initialisierung mit einer Datei verbunden werden

```
IMPORT IO, Rd, Wr;
VAR eingabestrom : Rd.T;
    ausgabestrom : Wr.T;
…
eingabestrom := IO.OpenRead ("eingabe.txt");
ausgabestrom := IO.OpenWrite("ausgabe.txt");
```

Beispiel: Kopieren einer Datei

```
IMPORT SIO,IO;
IMPORT Wr AS Writer, Rd AS Reader;

PROCEDURE KopiereDatei (original, kopie : TEXT)=
VAR original_datei : Reader.T;
    kopie_datei    : Writer.T;
    zeile : TEXT;
BEGIN
  original_datei := IO.OpenRead (orginal);
  kopie_datei := IO.OpenWrite(kopie);
  WHILE NOT IO.EOF(original_datei) DO
    zeile := SIO.GetLine(original_datei);
    SIO.PutLine(zeile, kopie_datei);
  END;
  Reader.Close(original_datei);
  Writer.Close(kopie_datei);
END KopiereDatei;
```


Wichtige Dateioperationen in M3

- **Modul: IO**
  - PROCEDURE **EOF** (rd: Rd.T := NIL): BOOLEAN;
    - *Return TRUE iff rd is at end-of-file.*
  - PROCEDURE **OpenRead** (f: TEXT): Rd.T;
    - *Open the file name f for reading and return a reader on its contents. If the file doesn't exist or is not readable, return NIL.*
  - PROCEDURE **OpenWrite** (f: TEXT): Wr.T;
    - *Open the file named f for writing and return a writer on its contents. If the file does not exist it will be created. If the process does not have the authority to modify or create the file, return NIL.*
- **Modul: Rd und Wr**
  - PROCEDURE **Close** (rd: T) RAISES {Failure, Alerted};
    - *Release any resources associated with rd and set closed(rd) := TRUE.*
  - PROCEDURE **Close** (wr: T) RAISES {Failure, Alerted};
    - *Flush wr, release any resources associated with wr, and set closed(wr) := TRUE.*

Was haben wir gelernt?

- Dateien für die persistente Speicherung
- Dateisysteme als Teil des Betriebssystems
- Dateiformen (sequentielle / direkte Dateien), Modi (Lese-, Schreibdateien)
- interne Dateien, externe Dateien und Bindung / Auflösung der Bindung beim Öffnen oder Schließen
- Anschluß von Dateien an das Programmiersystem über vordefinierte E/A-Bausteine
- Dateioperationen für Dateiformen, Eingabe- oder Ausgabedateien

Glossar

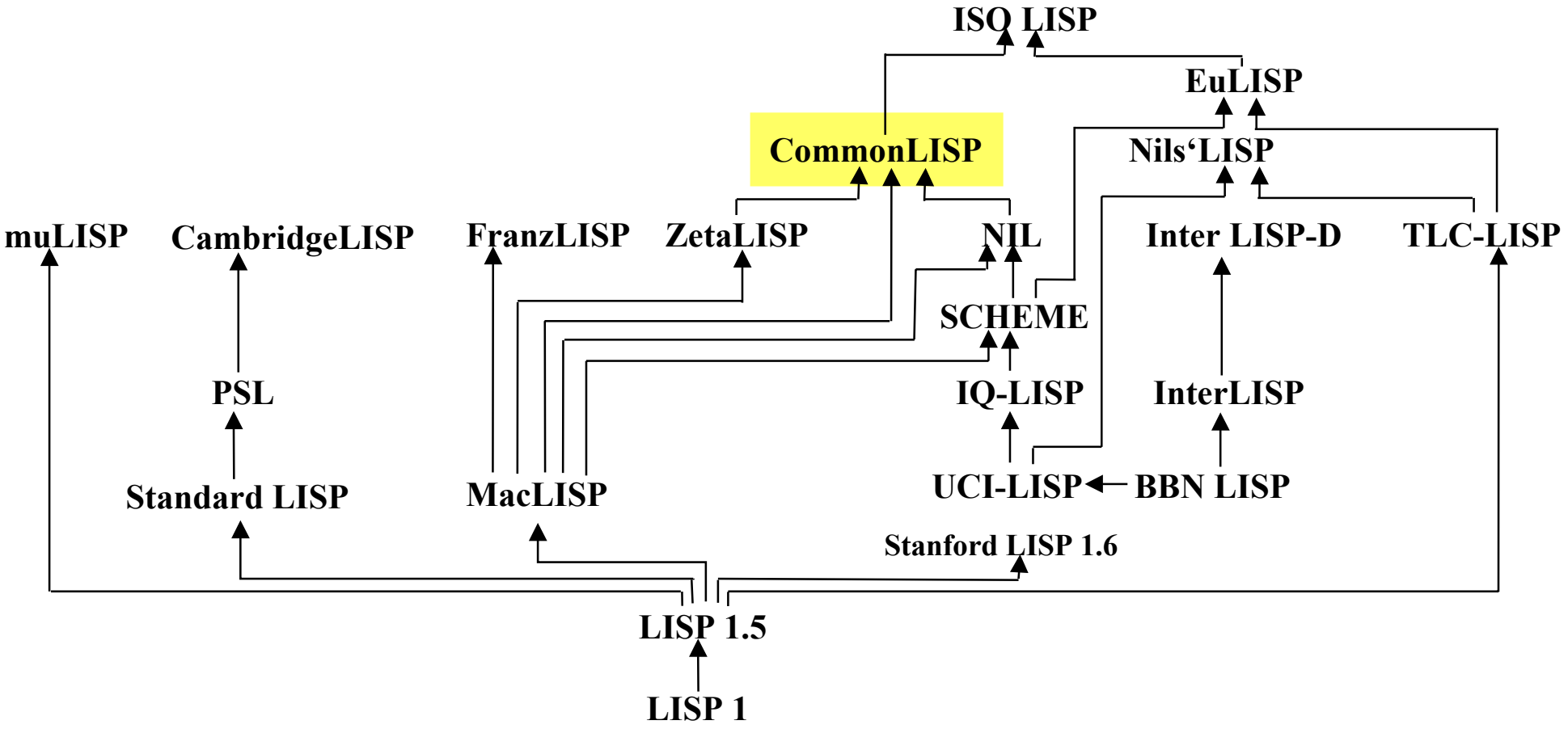
- Hintergrundspeicherarten
- flüchtiger, persistenter Speicher
- Betriebssystem, Dateisystem, Anschluß von seiten des Programmiersystems
- Datei (file), Verzeichnis (directory)
- externe, interne Dateien, Namen für beides
- Dateiformen, Eingabe-/Ausgabedatei
- Dateioperationen, Öffnen, Schließen, Lesen, Schreiben, Positionierung, Abfrage Dateiende
- Einschränkung der Operationen bei bestimmten Dateiformen, Ein- oder Ausgabedateien

Exkurs 2: Lisp (applikative Programmierung)

- Allgemeines
- Listen für Daten und Programme
- Wertzuweisungen, -ermittlung, -interpretation
- Listenverarbeitung
- Systemfunktionen, Ausdrücke
- Funktionen und Rekursion
- Abrundung

Historie

- Lisp 58 - so alt wie FORTRAN (numerische Anwendungen)
  Mc Cartny
  List Procesing
- Vielzahl von Dialekten: hier COMMON - Lisp - Ausschnitt

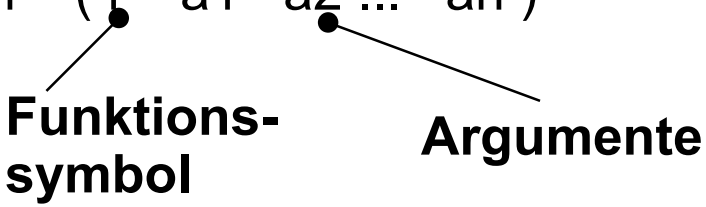


Charakterisierung

- **Vielfältiger Einsatz in der KI**
  - Wissensverarbeitung
  - Bildverarbeitung / Bildverstehen
  - Übersetzung natürlicher Sprachen
  - Automatisches Beweisen
  - Symbolische Algebra etc.
- **LISP interaktive Arbeitsweise: Interpreter**
- **LISP Programme und Daten haben dieselbe Form LISP-Programm kann andere Programme als Daten benutzen**
- **Literatur:**
  - Anderson/Corbett/Reisner: Essential LISP
  - Winston/Horn: LISP
  - Stoyan/Görz: LISP, eine Einführung in die Programmierung
  - Mayer: Programmieren in COMMON Lisp

LISP-Programm

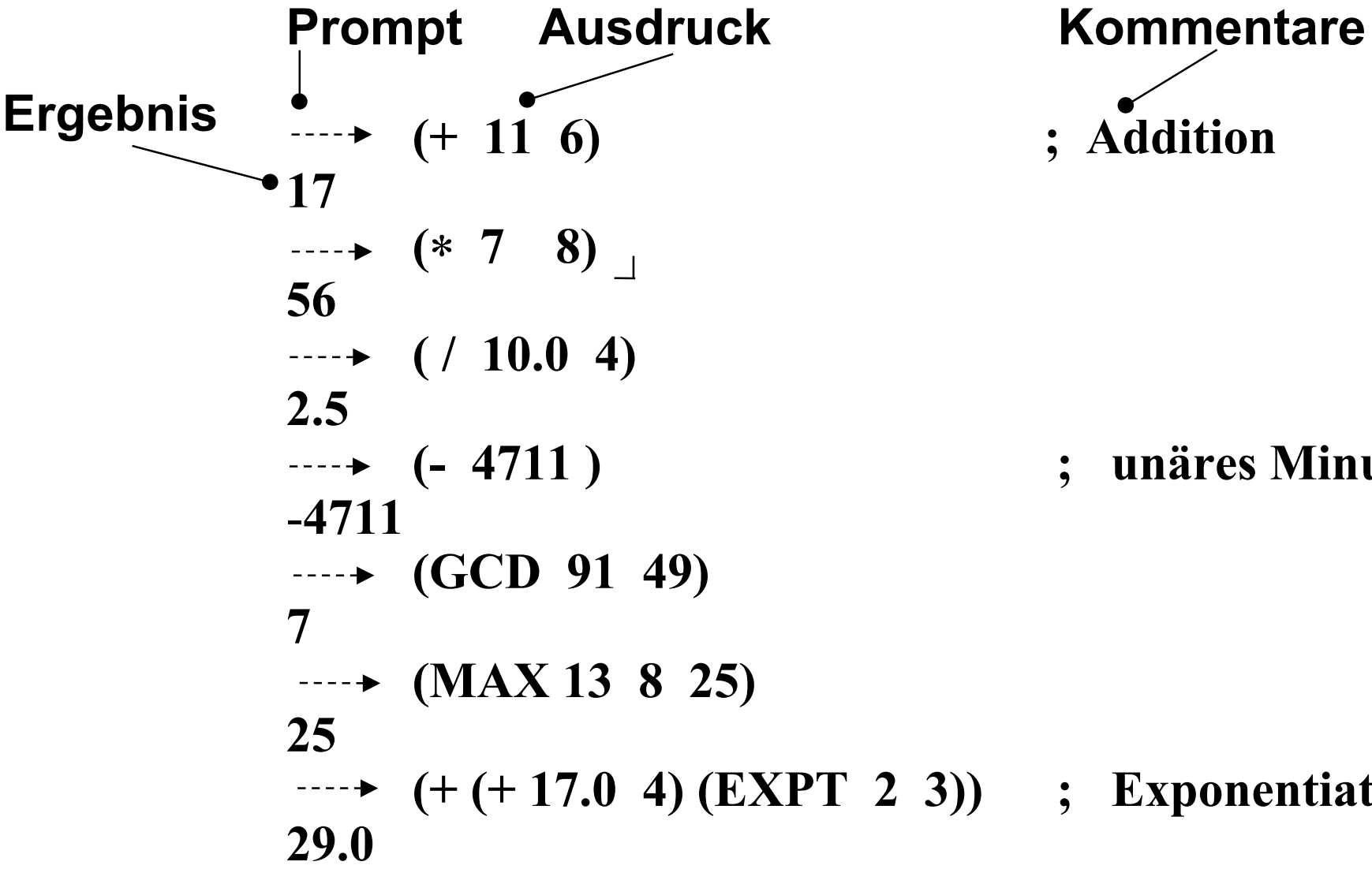
- **LISP – Programm:** **Folge von Lisp-Ausdrücken die nacheinander ausgewertet werden**
- **Ausdrücke**
  - Atome mit Wert
  - oder Listen (f a1 a2 ... an)



Wert ist Wert der Funktion für diese Argumente

  - Präfixnotation, klammerlos

Beispiele für Ausdrücke



Atom, Liste, Form

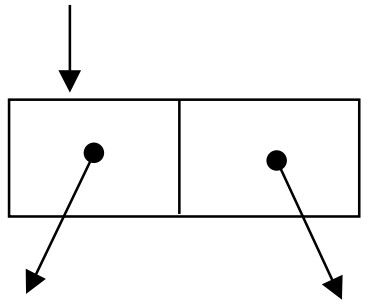
- **Atome**
  - Zahlenliterale
  - Symbole (Literale, Konstante, Variablen)
    T, NIL, ANTON, ARGUMENT-1, X_17
  - Zeichenketten(literale) „a b c d“ „~ A“
- **Liste**

  (listelem listelem ... listelem)

  ( )
  NIL
  } leere Liste
- **Ausdrücke: Atome oder Listen**
  auswertbarer Ausdruck: Form

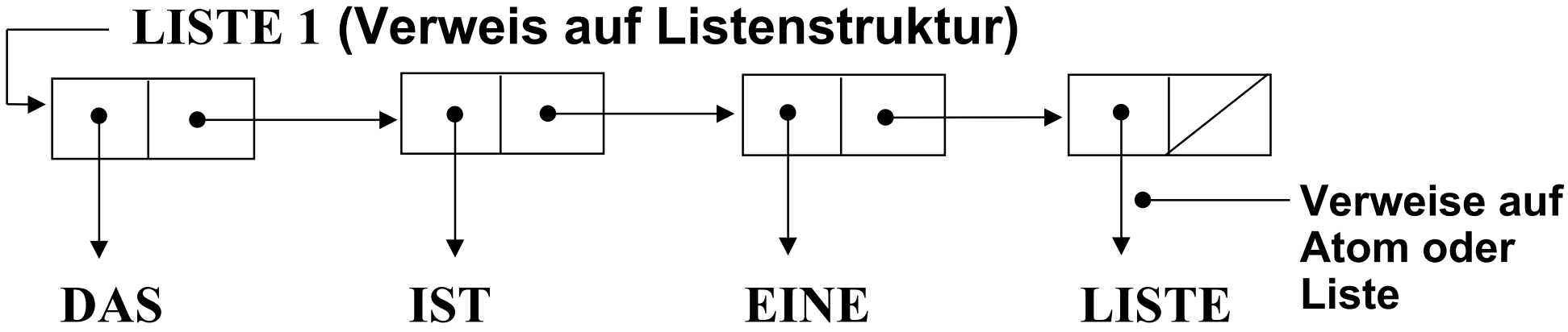
Interne Listendarstellung

- **Nehmen o. B. d. A LISP – Wörter folgende Struktur an: CONS-Zellen**



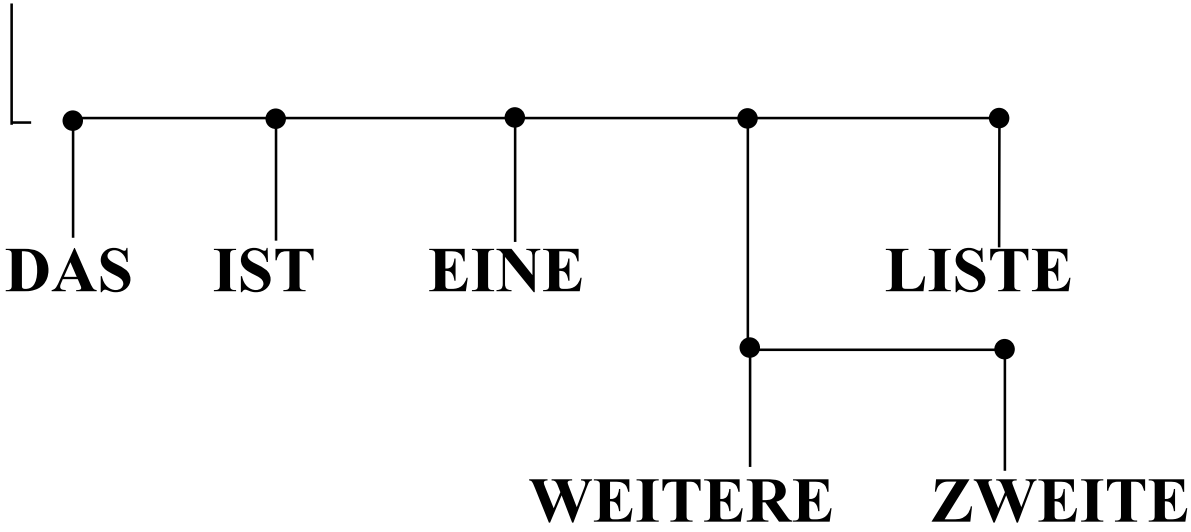
2 Inhalte
jeweils Adressen
(Zeiger)

- **Liste: Folge von CONS-Zellen, die über rechten Zeiger verknüpft sind**
- **(DAS IST EINE LISTE) sei Wert des Atoms LISTE 1**



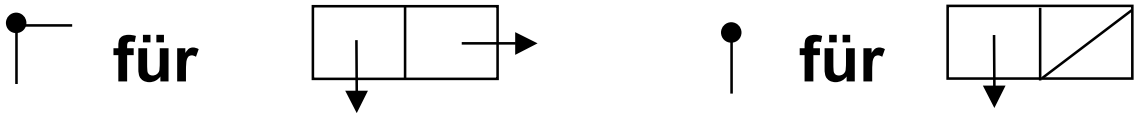
Liste für Daten

- **(DAS IST EINE (WEITERE ZWEITE) LISTE)**
  abgekürzt notiert als



Liste für Daten

- **Abkürzungen / Bedeutung**



für **für**

Hinuntersteigen von listelem zu Wert: Atom, Liste

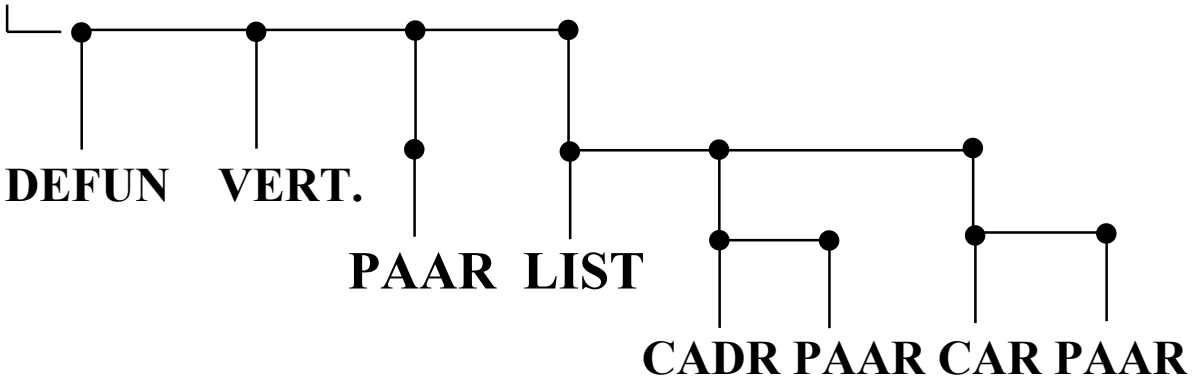
Übergang zu nächstem Listelement

Liste für Funktion

- (DEFUN VERTAUSCHE (PAAR))

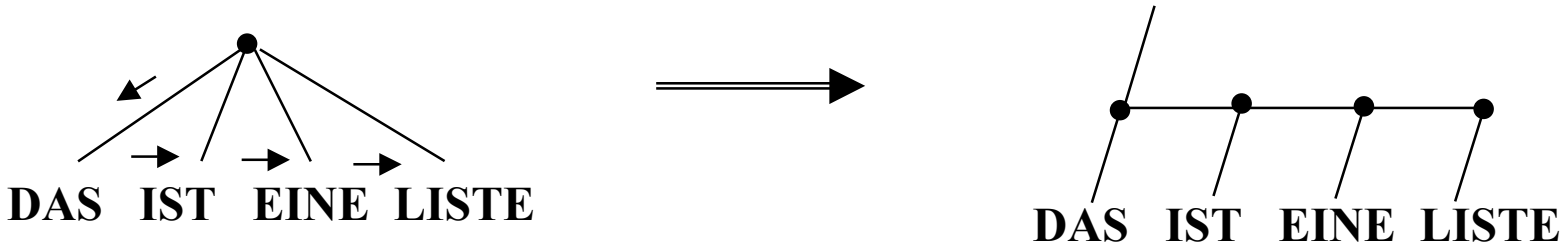
Liste für Funktion

(LIST (CADR PAAR) (CAR PAAR)))



- **Lisp - Liste und allgemeine Liste**

Lisp – Liste ist erster Sohn – nächster Bruder– Darstellung (Binärbaum-Darstellung) eines n-ären Baums (→ Datenstrukturen): Wert ins linke Zeigerfeld



Auswertung, Wert, Wertzuweisung

- **Form: auswertbarer Ausdruck**
  ist insb. Liste aus Listenelementen

Listenelemente: Zahlenliterale, Zeichenliterale haben Wert.
Wie erhalten die anderen Listenelemente einen Wert?
Wie wird die Liste ausgewertet?

- **Wertzuweisung an symbolischen Atome durch Systemfunktion SETQ**

```
(SETQ  L  '( A  B ))
```

Quote-Zeichen für (Quote (A B))
Wert ist Ausdruck selbst = Liste

Funktion mit Seiteneffekt:
Liste wird ausgerichtet
L verweist darauf

Beispiele für Zuweisungen - 1

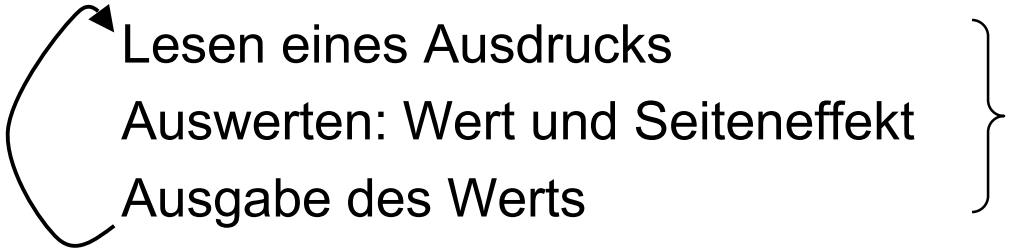
```
--->(SETQ   L      '(A  B))
(A  B)
--->L
(A  B)


--->(SETQ  X   (+  3  2  1))
6
--->(SETQ  X   '(+ 3  2  1))
(+  3  2  1)
--->(SETQ  X   (+ 3  2  1)   Y  ( -  9  X))
3
                                ; X hat Wert 6
                                ; Y   3
```

Beispiele für Zuweisungen - 2

```
----► (SETQ   A '   ( A  B ' C  D))
      ( A  B   (QUOTE  C)   D)          ; QUOTE in der Angabe
                                        ; nicht abgekürzt
                                        ; Ouotierung von Literalen
                                        ; ohne Bedeutung

----► X
6                                       ; Wert von X wird ermittelt

                                        ; und ausgegeben

----► ( +   '7   '4)                    ; Zahlenliterale haben
11                                      ; sich  selbst als Wert
```

Ausführung eines Programms

- **Nacheinanderauswertung der Ausdrücke durch Interpreter**



- **Seiteneffekte**
  - Zustand z: Gesamtheit von Zuordnungen von Atomen
  - nach Ausdrucksermittlung, neuer Zustand z'
  - nicht der Zustand der Laufzeit-Datenstruktur!

- **Änderungen von Zuordnungen nur über Seiteneffekte**

Interpreter als Systemfunktion

- **anwendbar auf Form**
  explizit durch EVAL

- **Wert eines Ausdrucks**
  - Atom: Literal: hat Wert, keine Auswertung
    Symbol: Wert durch Zuordnung im Speicher

  - Liste: Quote-Ausdruck (QUOTE B): B ist Wert
    Funktionsaufruf (FB ARG_1 ... ARG_n): auswerten
    normale Form
    Auswertung der Funktion auf die ausgewerteten Argumente
    spezielle Form (z.B. SETQ)

Beispiele und Motivation für Compilation

```
----►  (SETQ  A   'B)          ;  Zuweisung des Symbols und nicht des
B                              ;  Werts von B an A
----►  (SETQ  B   'C)
C
----►  A
B
----►  B
C
----►  (EVAL  A)
C
```

Ausw. von A
Erg. B
damit EVAL B
Erg. C

Interpr. EVAL auf Listen
vorab Auswertung der Elemente, wieder Listen etc.
hoher Zeit- und Speicherbedarf, da jedes Mal erneut ausgeführt wird

Compiler !

Motivation und Operationen

- **Lisp Symbolverarbeitung**

  abgebildet auf Listenverarbeitung

  Grundmaschinerie jedes LISP-Systems

  Notwendigkeit entsprechender Listenoperationen

- **Listenoperationen**
  - Aufbau
  - Zerlegung
  - Erweiterung
  - Spiegelung
  - etc.

Listenaufbau mit CONS

- (CONS Ausdruck Ausdruck)

  Wert: $Listenel_1$ $Liste_1$

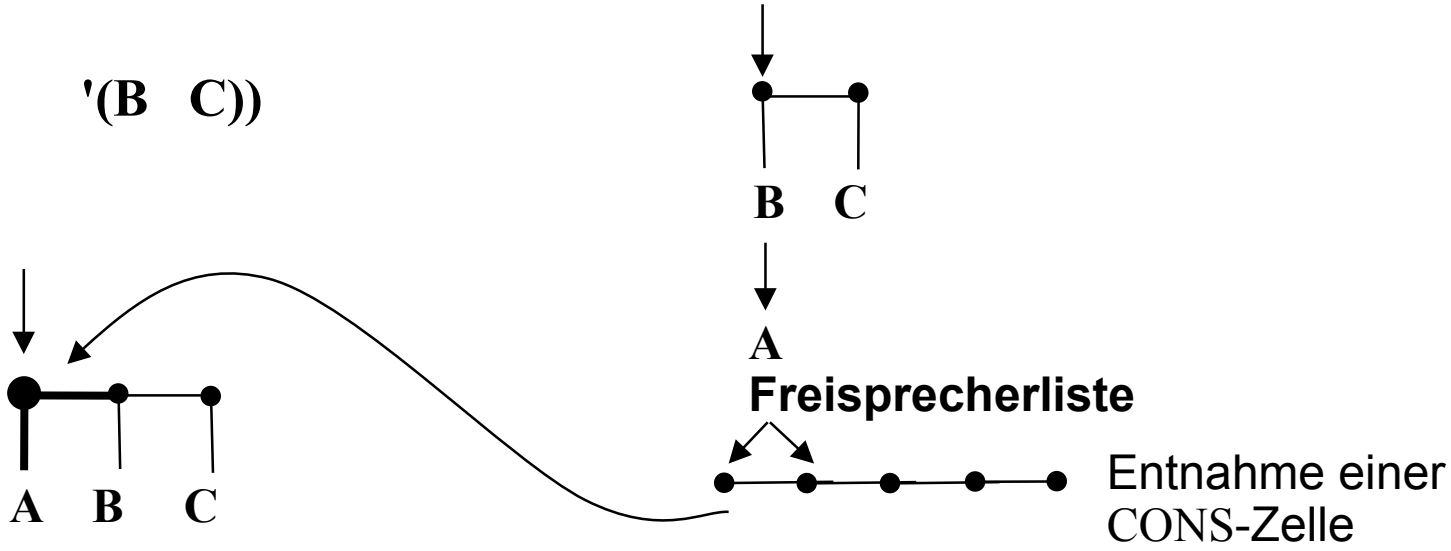
  Ergebnis: die Liste, die durch Anhängen von $Listenel_1$ am vorderen Ende von $Liste_1$ entsteht

  kein Seiteneffekt (vgl. Def. Seiteneffekt)
  Änderung der Zuordnungen symbolische Namen→ Werte, Strukturen

- Beispiel:

  ----→ (CONS 'A '(B C))

  (A B C)



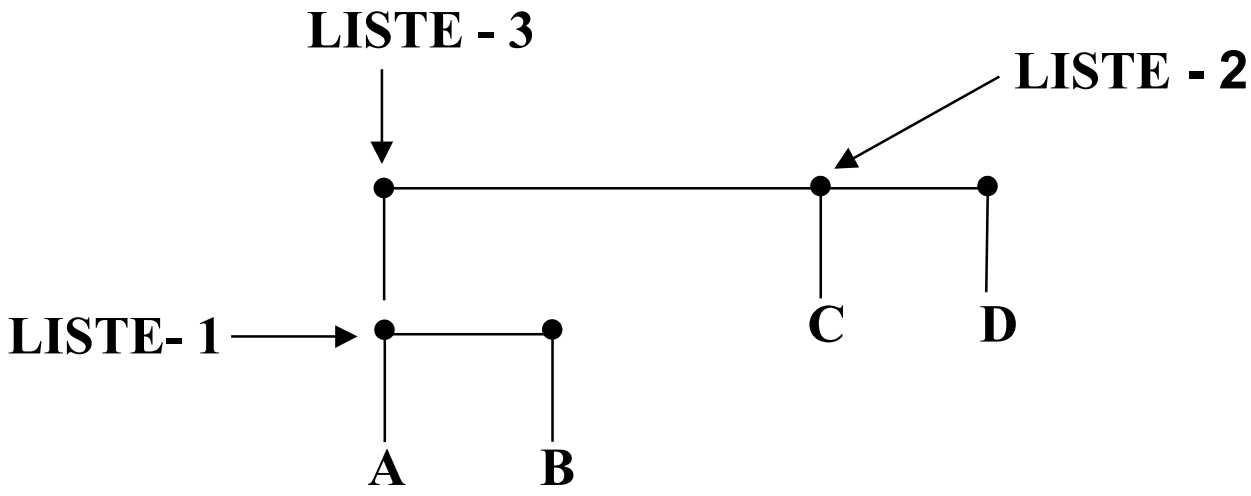
Weitere Beispiele mit CONS

----► (CONS '(A B) '(C D))
((A B) C D)

----► (SETQ LISTE – 1 '(A B))
(A B)

----► (SETQ LISTE – 2 '(C D))
(C D)

----► (SETQ LISTE – 3 (CONS LISTE – 1 LISTE – 2))
((A B) C D)



Zerlegen von Listen mit CAR, CDR

- Namen CAR, CDR erste Implementation: historisch

  CAR liefert erstes Element, falls L nicht leer sonst ()
  CDR liefert Rest als Liste, leere Liste falls geg. Liste leer war oder nur aus einem Element bestand

  } keine Seiten-effekte

- Beispiele:

```
----►  (CAR  '( (A B)  C))
(A B)

----►  (CDR  '((A B)  C))
(C)

----►  (SETQ  SPRUCH  '(WISSEN IST MACHT))
(WISSEN IST MACHT)

----►  (CAR  SPRUCH)
WISSEN                          ; Listenel nicht Liste

----►  (CDR  SPRUCH)
(IST MACHT)

----►  (CAR  'SPRUCH)           ; Wert des Arguments ist Atom nicht Liste
ERROR

----►  (CDR  'SPRUCH)
ERROR
```

Weitere Beispiele mit CONS, CAR, CDR

```
----→  (CDR  '(CAR  '((A B)  C)))      ; Quotierung schützt '(CAR...)
((QUOTE  ((A B)  C)))                  ; vor Auswertung

----→  (CAR  (CONS  'A  '(B C)))       ; Listen man. auf Programme
A                                      ; als Listen

----→  (CDR  (CONS  'A  '(B C)))
(B C)

----→  (CAR   (+  17  4))              ; Wert von (+ 17  4) ist Atom
ERROR

----→  (CAR  '(+  17  4))
+

----→  (CDR  '(+  17  4))
(17  4)
```

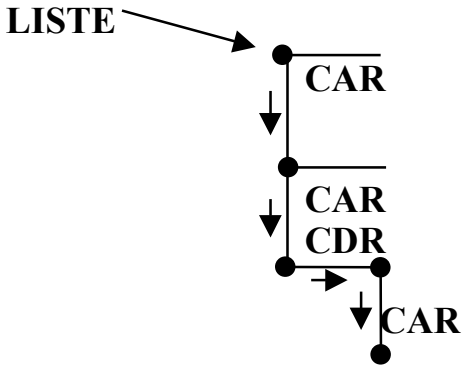
Zusammengesetzte Zerlegung: Zugriffspfade

- **Erweiterungen durch zusammengesetzte Zugriffsoperationen**

(CAR (CDR (CAR (CAR LISTE))))

abgek. zu (CADAAR LISTE)

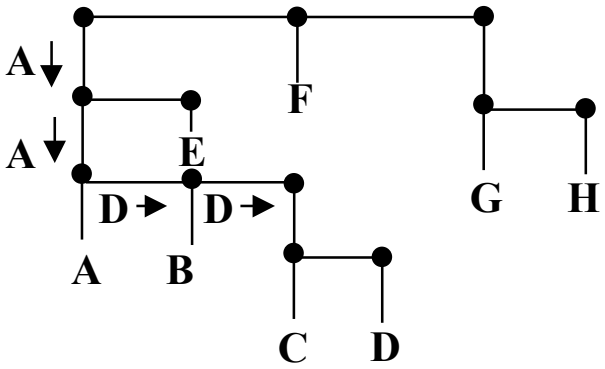
bis zur Tiefe 4, Lesen von rechts nach links



- **Beispiel:**

(C D D A A R '(((A B (C D)) E) F (G H)))

((C D))



Zusammenhang

- (CONS 'Π '($_1$ $_2$ ... $_n$)) Wert (Π $_1$ $_2$ ... $_n$)

  (CAR '(Π $_1$ $_2$ ... $_n$)) Wert Π

  (CDR '(Π $_1$ $_2$ ... $_n$)) Wert ($_1$ ... $_n$)

- **bequemere Listenoperationen**

  alle mit Hilfe von CONS, CAR, CDR implementierbar

Zusammenfügen durch APPEND

- **(APPEND L1 L2 .... Ln)** **Li hat als Wert eine Liste**
- **Wert ist Liste, durch Aneinanderfügen der Elemente von Li gewonnen wird.**
- **Beispiele:**

```
--► (APPEND   '(A B)  '(C D))
(ABCD)

--► (SETQ   L   '(A B))
(A B)

--► (CONS   L L)
((A B)   A B)

--► (APPEND   L  L)
(A B    A B)
```

Zusammenfügen durch LIST

- **(LIST Ausdr-1 Ausdr-2 ... Ausdr-n)**
- **Wert: Liste, die die Werte der Argumente in der geg. Reihenfolge zu Liste zusammenfasst**
- **Beispiele:**

--->(LIST '(A B) '(C D))

((A B) C D))

--->(LIST L L)

((A B) (AB))

Weitere Beispiele

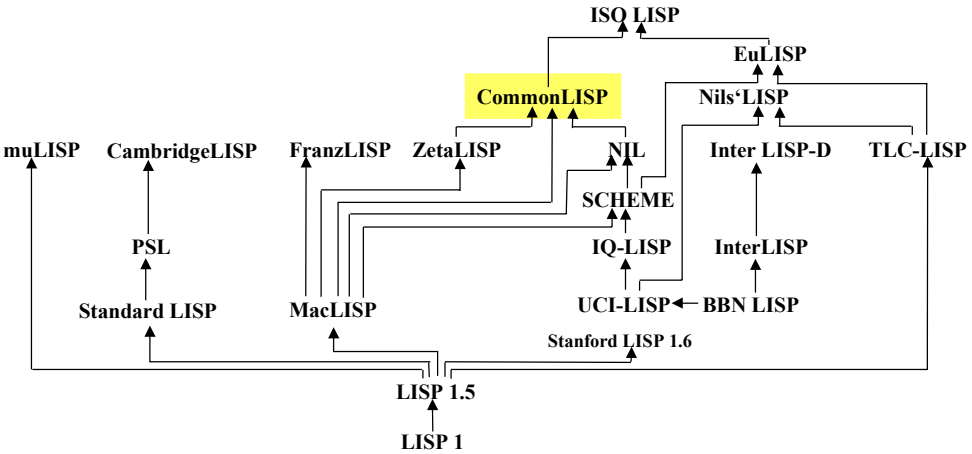
```
-----> ( CONS  'L  L)
( L  A  B)

-----> (APPEND   'L  L)                    ; 'L keine Liste
ERROR

-----> (LIST  'L  L)
(L  (A B))

-----> (APPEND   '(C )   '( )   L   '(D E))
(C A B D E)

-----> (LIST   L   L  '(C D ))
-----> (( A B  (AB)  (C D))

-----> (APPEND    '()   '() )
NIL

-----> (LIST    ' ()  '() )
( NIL  NIL)
```

Exkurs 2: Lisp
(applikative Programmierung)

- Allgemeines
- Listen für Daten und Programme
- Wertzuweisungen, -ermittlung, -interpretation
- Listenverarbeitung
- Systemfunktionen, Ausdrücke
- Funktionen und Rekursion
- Abrundung

Historie

- Lisp 58 - Mc Cartny
  List Procesing
  so alt wie FORTRAN (numerische Anwendungen)
- Vielzahl von Dialekten: hier COMMON - Lisp - Ausschnitt



Allgemeines

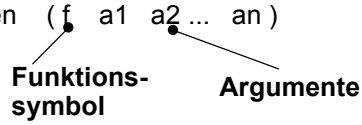
Charakterisierung

- **Vielfältiger Einsatz in der KI**
  - **Wissensverarbeitung**
  - **Bildverarbeitung / Bildverstehen**
  - **Übersetzung natürlicher Sprachen**
  - **Automatisches Beweisen**
  - **Symbolische Algebra etc.**
- **LISP interaktive Arbeitsweise: Interpreter**
- **LISP Programme und Daten haben dieselbe Form LISP-Programm kann andere Programme als Daten benutzen**
- **Literatur:**
  - **Anderson/Corbett/Reisner: Essential LISP**
  - **Winston/Horn: LISP**
  - **Stoyan/Görz: LISP, eine Einführung in die Programmierung**
  - **Mayer: Programmieren in COMMON Lisp**

Listen für Daten und Programme

LISP-Programm

- **LISP – Programm:** **Folge von Lisp-Ausdrücken die nacheinander ausgewertet werden**
- **Ausdrücke**
  - Atome mit Wert
  - oder Listen (f a1 a2 ... an)



Wert ist Wert der Funktion für diese Argumente

  - Präfixnotation, klammerlos

Beispiele für Ausdrücke

Prompt Ausdruck Kommentare

Ergebnis

```
----> (+ 11 6)                          ; Addition
17
----> (* 7 8)
56
----> ( / 10.0 4)
2.5
----> (- 4711 )                         ; unäres Minus
-4711
----> (GCD 91 49)
7
----> (MAX 13 8 25)
25
----> (+ (+ 17.0 4) (EXPT 2 3))         ; Exponentiation
29.0
```


Atom, Liste, Form

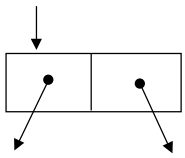
- **Atome**
  - Zahlenliterale
  - Symbole (Literale, Konstante, Variablen)
    T, NIL, ANTON, ARGUMENT-1, X_17
  - Zeichenketten(literale) „a b c d“ „~ A“
- **Liste**

  (listelem listelem ... listelem)

  ( ) } leere Liste
  NIL }
- **Ausdrücke: Atome oder Listen**
  auswertbarer Ausdruck: Form

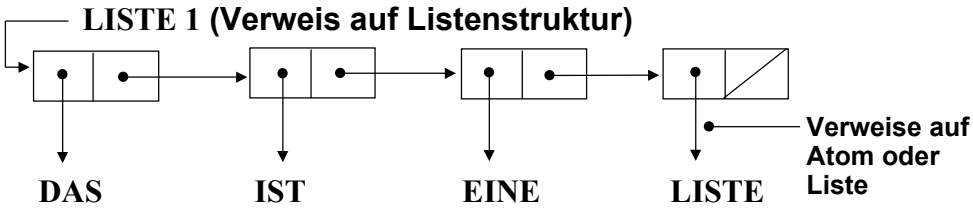
Interne Listendarstellung

- Nehmen o. B. d. A
  LISP – Wörter folgende Struktur.
  an: CONS-Zellen



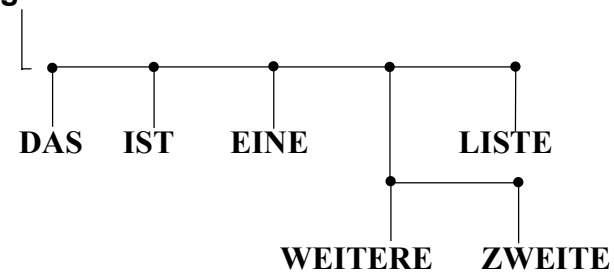
2 Inhalte
jeweils Adressen
(Zeiger)

- Liste: Folge von CONS-Zellen, die über rechten Zeiger verknüpft sind
- (DAS IST EINE LISTE) sei Wert des Atoms LISTE 1



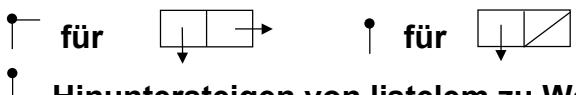
Liste für Daten

- (DAS IST EINE (WEITERE ZWEITE) LISTE)
  abgekürzt notiert als



Liste für Daten

- Abkürzungen / Bedeutung



Hinuntersteigen von listelem zu Wert: Atom, Liste

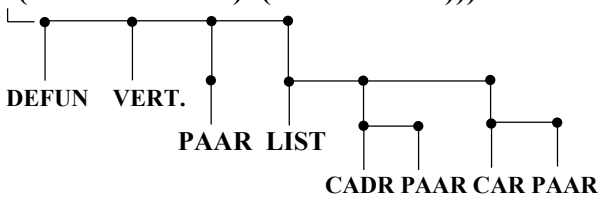
Übergang zu nächstem Listelement

Liste für Funktion

- (DEFUN VERTAUSCHE (PAAR))

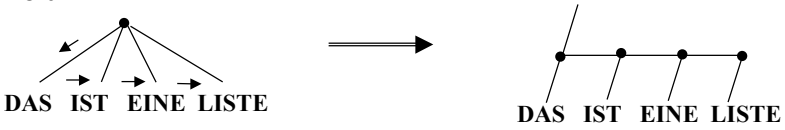
Liste für Funktion

(LIST (CADR PAAR) (CAR PAAR)))



- **Lisp - Liste und allgemeine Liste**

Lisp – Liste ist erster Sohn – nächster Bruder– Darstellung (Binärbaum-Darstellung) eines n-ären Baums (→ Datenstrukturen): Wert ins linke Zeigerfeld



Auswertung, Wert, Wertzuweisung

- **Form: auswertbarer Ausdruck**
  ist insb. Liste aus Listenelementen
  Listenelemente: Zahlenliterale, Zeichenliterale haben Wert.
  Wie erhalten die anderen Listenelemente einen Wert?
  Wie wird die Liste ausgewertet?

- **Wertzuweisung an symbolischen Atome durch Systemfunktion SETQ**

(SETQ L '(A B))

Quote-Zeichen für (Quote (A B))
Wert ist Ausdruck selbst = Liste

Funktion mit Seiteneffekt:
Liste wird ausgerichtet
L verweist darauf

Beispiele für Zuweisungen - 1

```
--->(SETQ   L     '(A  B))
(A  B)
 --->L
(A  B)


 --->(SETQ  X   (+  3  2  1))
6
 --->(SETQ  X   '(+ 3  2  1))
(+  3  2  1)
 --->(SETQ  X   (+ 3  2  1)   Y  ( -  9  X))
3
                              ; X hat Wert 6
                              ; Y   3
```


Beispiele für Zuweisungen - 2

```
----> (SETQ    A  '    ( A  B ' C  D))
      ( A  B    (QUOTE  C)   D)          ; QUOTE in der Angabe
                                         ; nicht abgekürzt
                                         ; Ouotierung von Literalen
                                         ; ohne Bedeutung

----> X
6                                        ; Wert von X wird ermittelt

                                         ; und ausgegeben

----> ( +   '7   '4)                     ; Zahlenliterale haben
11                                       ; sich  selbst als Wert
```

Ausführung eines Programms

- **Nacheinanderauswertung der Ausdrücke durch Interpreter**

  Lesen eines Ausdrucks
  Auswerten: Wert und Seiteneffekt
  Ausgabe des Werts

  } Top Level (-Schleife)

- **Seiteneffekte**
  - Zustand z: Gesamtheit von Zuordnungen von Atomen
  - nach Ausdrucksermittlung, neuer Zustand z'
  - nicht der Zustand der Laufzeit-Datenstruktur!

- **Änderungen von Zuordnungen nur über Seiteneffekte**

Interpreter als Systemfunktion

- **anwendbar auf Form**
  explizit durch EVAL

- **Wert eines Ausdrucks**
  - Atom: Literal: hat Wert, keine Auswertung
    Symbol: Wert durch Zuordnung im Speicher
  - Liste: Quote-Ausdruck (QUOTE B): B ist Wert
    Funktionsaufruf (FB ARG_1 ... ARG_n): auswerten
    normale Form
    Auswertung der Funktion auf die ausgewerteten Argumente
    spezielle Form (z.B. SETQ)

Beispiele und Motivation für Compilation

```
----> (SETQ  A   'B)          ;  Zuweisung des Symbols und nicht des
B                             ;  Werts von B an A
----> (SETQ  B   'C)
C
----> A
B
----> B
C
----> (EVAL  A)
C
```

Ausw. von A
Erg. B
damit EVAL B
Erg. C

Interpr. EVAL auf Listen
vorab Auswertung der Elemente, wieder Listen etc.
hoher Zeit- und Speicherbedarf, da jedes Mal erneut ausgeführt wird

Compiler !

Motivation und Operationen

- **Lisp Symbolverarbeitung**
  - abgebildet auf Listenverarbeitung
  - Grundmaschinerie jedes LISP-Systems
  - Notwendigkeit entsprechender Listenoperationen
- **Listenoperationen**
  - Aufbau
  - Zerlegung
  - Erweiterung
  - Spiegelung
  - etc.

Listenaufbau mit CONS

- (CONS Ausdruck Ausdruck)

  Wert: $Listenel_1$ $Liste_1$

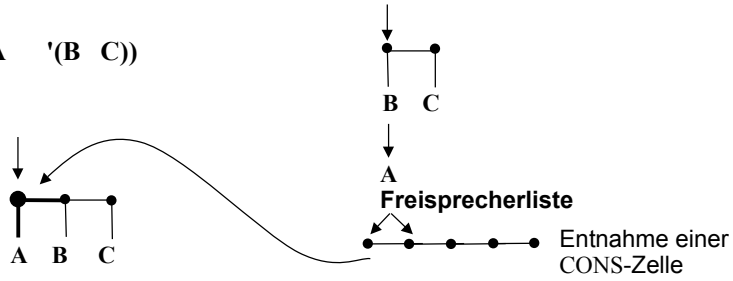
  Ergebnis: die Liste, die durch Anhängen von $Listenel_1$ am vorderen Ende von $Liste_1$ entsteht

  kein Seiteneffekt (vgl. Def. Seiteneffekt)
  Änderung der Zuordnungen symbolische Namen→ Werte, Strukturen

- Beispiel:

  ----→ (CONS 'A '(B C))

  (A B C)



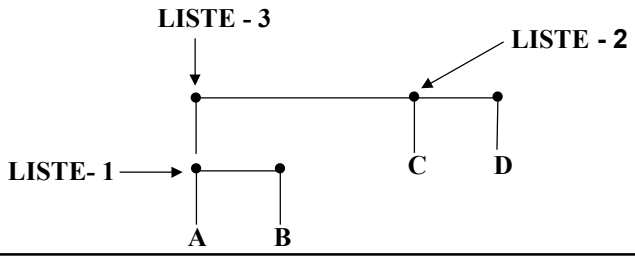
Weitere Beispiele mit CONS

----→ (CONS '(A B) '(C D))
((A B) C D)

----→ (SETQ LISTE – 1 '(A B))
(A B)

----→ (SETQ LISTE – 2 '(C D))
(C D)

----→ (SETQ LISTE – 3 (CONS LISTE – 1 LISTE – 2))
((A B) C D)



Zerlegen von Listen mit CAR, CDR

- Namen CAR, CDR erste Implementation: historisch

  CAR liefert erstes Element, falls L nicht leer sonst ()
  CDR liefert Rest als Liste, leere Liste falls geg. Liste leer war oder nur aus einem Element bestand

  } keine Seiten-effekte

- Beispiele:

```
----→ (CAR '( (A B) C))
(A B)
----→ (CDR '((A B) C))
(C)
----→ (SETQ SPRUCH '(WISSEN IST MACHT))
(WISSEN IST MACHT)
----→ (CAR SPRUCH)
WISSEN                    ; Listenel nicht Liste
----→ (CDR SPRUCH)
(IST MACHT)
----→ (CAR 'SPRUCH)       ; Wert des Arguments ist Atom nicht Liste
ERROR
----→ (CDR 'SPRUCH)
ERROR
```


Weitere Beispiele mit CONS, CAR, CDR

```
----→ (CDR '(CAR '((A B) C)))     ; Quotierung schützt '(CAR...)
((QUOTE ((A B) C)))               ; vor Auswertung

----→ (CAR (CONS 'A '(B C)))      ; Listen man. auf Programme
A                                 ; als Listen

----→ (CDR (CONS 'A '(B C)))
(B C)

----→ (CAR (+ 17 4))              ; Wert von (+ 17 4) ist Atom
ERROR

----→ (CAR '(+ 17 4))
+

----→ (CDR '(+ 17 4))
(17 4)
```

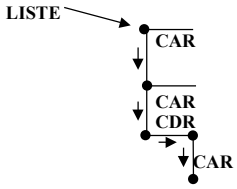
Zusammengesetzte Zerlegung: Zugriffspfade

- **Erweiterungen durch zusammengesetzte Zugriffsoperationen**

  (CAR (CDR (CAR (CAR LISTE))))

  abgek. zu (CADAAR LISTE)

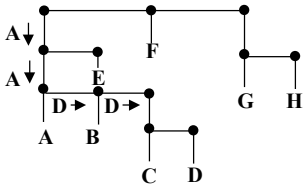
  bis zur Tiefe 4, Lesen von rechts nach links



- **Beispiel:**

  (C D D A A R '(((A B (C D)) E) F (G H)))

  ((C D))



Zusammenhang

- (CONS 'Π '(1 2 … n)) Wert (Π 1 2 … n)

  (CAR '(Π 1 2 … n)) Wert Π

  (CDR '(Π 1 2 … n)) Wert (1 … n)

- **bequemere Listenoperationen**

  alle mit Hilfe von CONS, CAR, CDR implementierbar

Listen verarbeitung

Zusammenfügen durch APPEND

- **(APPEND L1 L2 .... Ln)** **Li hat als Wert eine Liste**
- **Wert ist Liste, durch Aneinanderfügen der Elemente von Li gewonnen wird.**
- **Beispiele:**

--→ (APPEND '(A B) '(C D))
(ABCD)

--→ (SETQ L '(A B))
(A B)

--→ (CONS L L)
((A B) A B)

--→ (APPEND L L)
(A B A B)

Listen verarbeitung

Zusammenfügen durch LIST

- **(LIST Ausdr-1 Ausdr-2 ... Ausdr-n)**
- **Wert: Liste, die die Werte der Argumente in der geg. Reihenfolge zu Liste zusammenfasst**
- **Beispiele:**

--→(LIST '(A B) '(C D))
((A B) C D))

--→(LIST L L)
((A B) (AB))

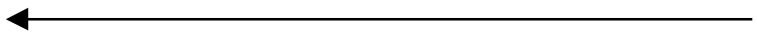
Weitere Beispiele

```
-----> ( CONS  'L  L)
( L  A  B)

-----> (APPEND   'L  L)              ; 'L keine Liste
ERROR

-----> (LIST  'L  L)
(L  (A B))

-----> (APPEND   '(C )   '( )   L   '(D E))
(C A B D E)

-----> (LIST   L   L  '(C D ))
-----> (( A B  (AB)  (C D))

-----> (APPEND    '()   '() )
NIL

-----> (LIST    ' ()  '() )
( NIL  NIL)
```

Speicherbereinigung

- **CONS, APPEND, LIST brauchen Speicherzellen von der Freispeicherliste. Der Freispeicher ist natürlich endlich**
- **viele Speicherstrukturen werden kurz nach ihrem Aufbau nicht mehr benötigt**
- **Beispiel:**

```
(SETQ  WASTE  '(A B C D))
(A B C D)
```



```
(SETQ  WASTE   (LIST   'X  'Y))
(X   Y)
```

Struktur der Liste WASTE im alten Zustand ist Null (Garbage)

Speicherbereinigung (Garbage Collection) bei allen interpreter-orientieren Sprachen

Weitere Funktonen zur Listenmanipulation

- **(REVERSE Liste)**

  Wert: Liste, die auf oberstem Niveau die Reihenfolge der Elemente invertiert, die Elemente bleiben unverändert

  --->(REVERSE '(A B))
  (B A)

  --->(REVERSE '((A B) (C D)))
  ((C D) (A B))

  --->(REVERSE (APPEND '(A B) '(C D)))
  (D C B A)

- **(LAST Liste1)**

  Wert: Liste, die das letzte Element von Liste1 als einziges Element enthält

  --->(LAST '(((A B) (C D)) (A B)))
  ((A B))

Prädikate

- **Programmteile abhängig von Bedingungen**

  Bedingungen: Prädikate durch Junktoren verknüpfen

  Prädikat: Wert $\begin{matrix}\text{wahr}\\\text{falsch}\end{matrix}$ sprachseitig festgelegt $\begin{matrix}\text{T}\\\text{NIL}\end{matrix}$ oder irgendein Wert

- **Strukturprädikate für Atome, CONS-Paare, Listen**

  ist Atom
  ist leer
  ist Lisp-Ausdruck
  Übereinstimmung von Listen
  Listenauskunft, z.B. Länge

Strukturprädikate für Atome

- **(ATOM Ausdruck)**

  Wert: T, falls der Wert von Ausdruck nicht mit CONS aufgebaut werden kann, NIL sonst

- **(CONSP Ausdruck)**

  Wert: T, falls der Wert von Ausdruck mit CONS aufgebaut werden kann, NIL sonst

  damit Präzisierung des Begriffs „Atom“

- **Beispiel:**

```
--->(ATOM  'NIL)
 T
--->(ATOM  T)
 T
--->(ATOM  'ZWILLINGE)
 T
--->(SETQ  ZWILLINGE  '(MAX  MORITZ))
(MAX  MORITZ)
--->(ATOM ZWILLINGE)
NIL
```

```
--->(CONSP  ZWILLINGE)
T
--->(CONSP  '(MAX  MORITZ)
T
--->(ATOM  '(+  17  4))
NIL
--->(CONSP  '(A  B  C))
T
--->(CONSP  '17)
NIL
```

Strukturprädikate für Ausdrücke

- **(NULL Ausdruck)**

; Wert: T, falls Ausdruck als Wert die leere Liste hat, NIL sonst

----► (NULL '())
T
----► (NULL NIL)
T
----► (NULL 0)
NIL
----► (NULL (CDDR '(A B)))
T

- **(LISTP Ausdruck)**

; Wert: T, falls der Wert von Ausdruck das Prädikat CONSP erfüllt oder NIL ist, NIL sonst

----► (LISTP 'T)
NIL
----► (LISTP 'NIL)
T
----► (LISTP '( ))
T

Strukturprädikate für Listen

- **Übereinstimmung von LISP-Strukturen**

  Vergleich bzgl. interner Speicherstrukturen : EQ
  Vergleich bzgl. externer Darstellung : EQUAL

- **(EQ Ausdr1 Ausdr2)**

  Wert T, falls dem Wert der beiden Argumente dieselbe Speicherstruktur zugrundeliegt, NIL sonst

- **(EQUAL Ausdr1 Ausdr2)**

  Wert T, falls der Wert der beiden Argumente die gleiche externe Repräsentation besitzt, NIL sonst

Beispiele zu Listenvergleich

- ------► (SETQ L1 (LIST 'A 'B 'C))
  (A B C)

  ------► (SETQ L2 (LIST 'A 'B 'C))
  (A B C)

  ------► (SETQ L3 L2)
  (A B C)

- ------► (EQ 'A 'A)
  T

  ------► (EQ 'A 'B)
  NIL

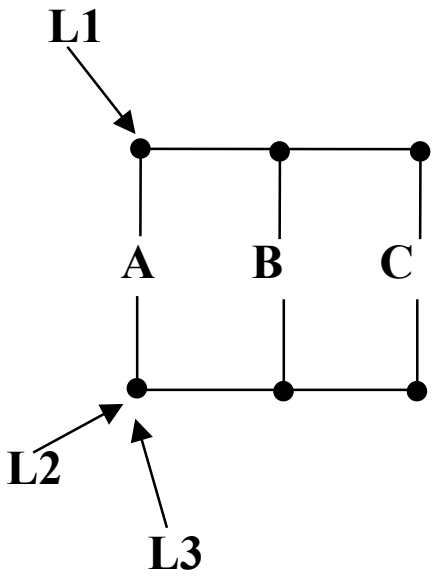
  ------► (EQ L1 L2)
  NIL

  ------► (EQ L2 L3)
  T

  ------► (EQUAL L1 L2)
  T

  ------► (EQUAL L2 L3)
  T

  ------► (EQ (CAR L1) (CAR L2))
  T



haben die gleiche Struktur, sind aber nicht die gleichen Speicherstrukturen

haben beide das Atom A als Wert; das hat eindeutige Speicherrepräsentation

Leere Liste

- **Bemerkung**

  NIL und leere Liste -darg. ( )- sind äquivalent

- **Beispiel**

```
------> (EQ   NIL '( ))
        T                ; leere Liste bei Angabe stets durch NIL

        (ATOM   '( ))
------> T

                         ; Sei  a Ausdruck mit Wert als Liste
        (ATOM a)
------> T                ; g. d. w.   Liste leer
```

NIL ist der einzige Ausdruck der gleichzeitig Liste und Atom ist.

Listenauskunftsfunktionen

- **(MEMBER Ausdruck Liste)**

  Wert: Liste, falls in der Liste (auf oberstem Niveau) ein Element mit dem Wert des Ausdrucks auftaucht; die Restliste ab erstem Auftauchen dieses Elements (incl.) NIL sonst

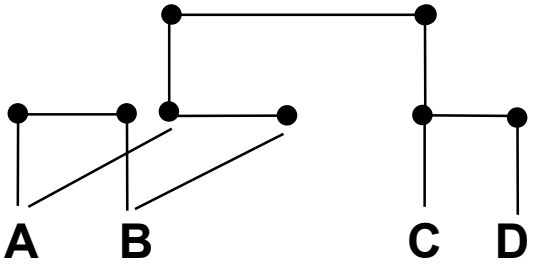
- **Beispiele:**

```
-----►(SETQ  FUENF-X  '(X1 X2 X3 X4 X5))
(X1 X2 X3 X4 X5)

-----►(MEMBER  'X6  FUENF-X)
NIL

-----►(MEMBER  'X3  FUENF-X)
(X3 X4 X5)

-----►(SETQ  A 'X4)
X4

-----►(MEMBER A  FUENF-X)
(X4 X5)
```

Weitere Beispiele - 1

```
-------►(MEMBER   A  '(( X1 X2)   (X3 X4 X5)))
NIL                                  ; X4 nicht El. der zweielementigen Liste

-------►(MEMBER   A  (CDR  '(( X1 X2)   (X3 X4 X5))))
NIL                                  ; X4 nicht El. von ((X3  X4 X5))

-------►(MEMBER   A  (CADR  '(( X1 X2)   (X3 X4 X5))))
(X4 X5)                              ; X4 ist El. von (X3 X4 X5)
```

Bemerkung 1:
MEMBER verwendet üblicherweise EQ bei Strukturen!
Problem, wenn erstes Argument eine Struktur ist.




```
-------►(MEMBER    '(A B)   '((A B) (C D)))
NIL
```

Erstes El. der zweiten Liste hat zwar dieselbe Struktur aber nicht dieselbe Speicherstruktur

Weitere Beispiele - 2

- **Abhilfe**
  vorschreiben von EQUAL Strukturvergleich

```
(MEMBER '(A B)   '((A B) (C D)) : Test  'EQUAL)
((A B) (C D))
```

Systemprädikat kann auch selbstdefiniertes Prädikat sein

- **MEMBER ist kein Prädikat, liefert im positiven Fall Liste, kann aber als Prädikat verwendet werden, da für wahr irgendein Wert ≠ NIL genommen wird**

- **Probleme:**

```
--->(EQ   T   (MEMBER   'Y   '(X  Y  Z)))
NIL
--->(EQUAL  T   (MEMBER  'Y   '(X  Y  Z)))
NIL
```

Längenbestimmung

- **(LENGTH Liste)**

  Wert: Anzahl der Elemente der Liste

- **Beispiel:**

```
------> (SETQ  L  '(A B))
(A B)
------> (LENGTH   L)
2
------> (LENGTH '((A B)  (C D)))
2
------> (LENGTH   (APPEND   L '(A B)))
4
------> (LENGTH   LIST   L   '((A B)))
2
------> (LENGTH   (MEMBER   'X3   '(X1 X2 X3 X4 X5)))
3
```

Überprüfung/Vergleich von Atomen

Wertvergleiche

- Zahlen ganze Zahlen
  Gleitpunktzahlen verschiedener Genauigkeit
  komplexe Zahlen
- Ergebnis von EQ nicht def., da Zahlen des gleichen Typs und mit gleichem Wert nicht nicht notwendigerweise im gleichem Speicherbereich untergebracht sind.
- Vergleich mit EQUAL liefert für Zahlen des gleichen Typs und Werts passendes Ergebnis

Beispiele:

```
---►(EQ   2.0  2)              ; Interndarstellung i.a. verschieden
NIL
---►(EQ   2.13  2.13)          ; nicht def.
NIL oder T                     ; systemabhängig
---►(EQUAL 2.0  2)             ; untersch.Typ
NIL
---►(EQUAL   4.5  (+ 2.3  2.2))
T
---►(EQUAL  4.5  '(+ 2.3  2.2)) )
NIL
```

Resumeé : EQ ungeeignet
EQUAL bedingt geeignet

Speicherbereinigung

- **CONS, APPEND, LIST brauchen Speicherzellen von der Freispeicherliste. Der Freispeicher ist natürlich endlich**
- **viele Speicherstrukturen werden kurz nach ihrem Aufbau nicht mehr benötigt**
- **Beispiel:**

```
(SETQ  WASTE  '(A B C D))
(A B C D)
```



```
(SETQ  WASTE   (LIST   'X  'Y))
(X   Y)
```

Struktur der Liste WASTE im alten Zustand ist Null (Garbage)

Speicherbereinigung (Garbage Collection) bei allen interpreter-orientieren Sprachen

Weitere Funktonen zur Listenmanipulation

- **(REVERSE Liste)**

  Wert: Liste, die auf oberstem Niveau die Reihenfolge der Elemente invertiert, die Elemente bleiben unverändert

```
--->(REVERSE   '(A B))
(B A)

--->(REVERSE   '((A B)  (C D)))
((C D)  (A B))

--->(REVERSE   (APPEND  '(A B)  '(C D)))
(D C B A)
```

- **(LAST Liste1)**

  Wert: Liste, die das letzte Element von Liste1 als einziges Element enthält

```
--->(LAST  '(((A B)  (C D))  (A B)))
((A B))
```

Prädikate

- **Programmteile abhängig von Bedingungen**

  Bedingungen: Prädikate durch Junktoren verknüpfen

  Prädikat: Wert wahr / falsch **sprachseitig festgelegt** T / NIL **oder irgendein Wert**

- **Strukturprädikate für Atome, CONS-Paare, Listen**

  ist Atom
  ist leer
  ist Lisp-Ausdruck
  Übereinstimmung von Listen
  Listenauskunft, z.B. Länge

Strukturprädikate für Atome

- **(ATOM Ausdruck)**

  Wert: T, falls der Wert von Ausdruck nicht mit CONS aufgebaut werden kann, NIL sonst

- **(CONSP Ausdruck)**

  Wert: T, falls der Wert von Ausdruck mit CONS aufgebaut werden kann, NIL sonst

  damit Präzisierung des Begriffs „Atom“

- **Beispiel:**

```
--->(ATOM 'NIL)
 T
--->(ATOM T)
 T
--->(ATOM 'ZWILLINGE)
 T
--->(SETQ ZWILLINGE '(MAX MORITZ))
(MAX MORITZ)
--->(ATOM ZWILLINGE)
NIL
```

```
--->(CONSP ZWILLINGE)
T
--->(CONSP '(MAX MORITZ)
T
--->(ATOM '(+ 17 4))
NIL
--->(CONSP '(A B C))
T
--->(CONSP '17)
NIL
```

Strukturprädikate für Ausdrücke

- **(NULL Ausdruck)**

; Wert: T, falls Ausdruck als Wert die leere Liste hat, NIL sonst

----→ (NULL '())
T

----→ (NULL NIL)
T

----→ (NULL 0)
NIL

----→ (NULL (CDDR '(A B)))
T

- **(LISTP Ausdruck)**

; Wert: T, falls der Wert von Ausdruck das Prädikat CONSP erfüllt oder NIL ist, NIL sonst

----→ (LISTP 'T)
NIL

----→ (LISTP 'NIL)
T

----→ (LISTP '( ))
T

Strukturprädikate für Listen

- **Übereinstimmung von LISP-Strukturen**

  Vergleich bzgl. interner Speicherstrukturen : EQ
  Vergleich bzgl. externer Darstellung : EQUAL

- **(EQ Ausdr1 Ausdr2)**

  Wert T, falls dem Wert der beiden Argumente dieselbe Speicherstruktur zugrundeliegt, NIL sonst

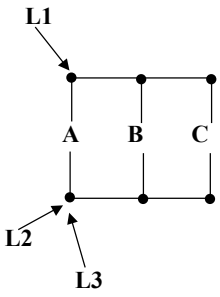
- **(EQUAL Ausdr1 Ausdr2)**

  Wert T, falls der Wert der beiden Argumente die gleiche externe Repräsentation besitzt, NIL sonst

Beispiele zu Listenvergleich

- ------► (SETQ L1 (LIST 'A 'B 'C))
  (A B C)
  ------► (SETQ L2 (LIST 'A 'B 'C))
  (A B C)
  ------► (SETQ L3 L2)
  (A B C)

- ------► (EQ 'A 'A)
  T
  ------► (EQ 'A 'B)
  NIL
  ------► (EQ L1 L2)
  NIL
  ------► (EQ L2 L3)
  T
  ------► (EQUAL L1 L2)
  T
  ------► (EQUAL L2 L3)
  T
  ------► (EQ (CAR L1) (CAR L2))
  T



haben die gleiche Struktur, sind aber nicht die gleichen Speicherstrukturen

haben beide das Atom A als Wert; das hat eindeutige Speicherrepräsentation

Leere Liste

- **Bemerkung**

  NIL und leere Liste -darg. ( )- sind äquivalent

- **Beispiel**

------►(EQ NIL '( ))
T ; leere Liste bei Angabe stets durch NIL

(ATOM '( ))
------►T

; Sei a Ausdruck mit Wert als Liste
(ATOM a)
------►T ; g. d. w. Liste leer

NIL ist der einzige Ausdruck der gleichzeitig Liste und Atom ist.

Listenauskunftsfunktionen

- **(MEMBER Ausdruck Liste)**

  Wert: Liste, falls in der Liste (auf oberstem Niveau) ein Element mit dem Wert des Ausdrucks auftaucht; die Restliste ab erstem Auftauchen dieses Elements (incl.) NIL sonst

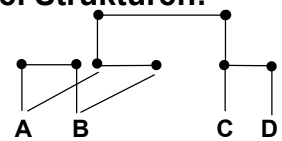
- **Beispiele:**

```
------>(SETQ  FUENF-X  '(X1 X2 X3 X4 X5))
(X1 X2 X3 X4 X5)

------>(MEMBER  'X6  FUENF-X)
NIL

------>(MEMBER  'X3  FUENF-X)
(X3 X4 X5)

------>(SETQ  A 'X4)
X4

------>(MEMBER A  FUENF-X)
(X4 X5)
```


Weitere Beispiele - 1

```
-------->(MEMBER   A  '(( X1 X2)  (X3 X4 X5)))
NIL                              ; X4 nicht El. der zweielementigen Liste

-------->(MEMBER   A  (CDR  '(( X1 X2)  (X3 X4 X5))))
NIL                              ; X4 nicht El. von ((X3  X4 X5))

-------->(MEMBER   A  (CADR  '(( X1 X2)  (X3 X4 X5))))
(X4 X5)                          ; X4 ist El. von (X3 X4 X5)
```

Bemerkung 1:
MEMBER verwendet üblicherweise EQ bei Strukturen!
Problem, wenn erstes Argument eine Struktur ist.




```
-------->(MEMBER    '(A B)   '((A B) (C D)))
NIL
```

Erstes El. der zweiten Liste hat zwar dieselbe Struktur aber nicht dieselbe Speicherstruktur

Weitere Beispiele - 2

- **Abhilfe**
  vorschreiben von EQUAL Strukturvergleich

  (MEMBER '(A B) '((A B) (C D)) : Test 'EQUAL)
  ((A B) (C D))

  Systemprädikat kann auch selbstdefiniertes Prädikat sein

- **MEMBER ist kein Prädikat, liefert im positiven Fall Liste, kann aber als Prädikat verwendet werden, da für wahr irgendein Wert ≠ NIL genommen wird**

- **Probleme:**

```
--->(EQ T (MEMBER 'Y '(X Y Z)))
NIL
--->(EQUAL T (MEMBER 'Y '(X Y Z)))
NIL
```


Längenbestimmung

- **(LENGTH Liste)**

  Wert: Anzahl der Elemente der Liste

- **Beispiel:**

```
------> (SETQ L '(A B))
(A B)
------> (LENGTH L)
2
------> (LENGTH '((A B) (C D)))
2
------> (LENGTH (APPEND L '(A B)))
4
------> (LENGTH LIST L '((A B)))
2
------> (LENGTH (MEMBER 'X3 '(X1 X2 X3 X4 X5)))
3
```

Überprüfung/Vergleich von Atomen

Wertvergleiche

- Zahlen ganze Zahlen
  Gleitpunktzahlen verschiedener Genauigkeit
  komplexe Zahlen
- Ergebnis von EQ nicht def., da Zahlen des gleichen Typs und mit gleichem Wert nicht nicht notwendigerweise im gleichem Speicherbereich untergebracht sind.
- Vergleich mit EQUAL liefert für Zahlen des gleichen Typs und Werts passendes Ergebnis

Beispiele:

```
---►(EQ   2.0  2)              ; Interndarstellung i.a. verschieden
 NIL
---►(EQ   2.13  2.13)          ; nicht def.
 NIL oder T                    ; systemabhängig
---►(EQUAL 2.0  2)             ; untersch.Typ
 NIL
---►(EQUAL   4.5  (+ 2.3  2.2))
 T
---►(EQUAL  4.5  '(+ 2.3  2.2)) )
 NIL
```

Resumeé : EQ ungeeignet
EQUAL bedingt geeignet

Spezielle Prädikate und Beispiele - 1

Spezielle Prädikate

(= Ausdr1 Ausdr2 ... AusdrN) alle Werte Zahlen sind gleich
(< Ausdr1 Ausdr2 ... AusdrN) „ „ „ aufsteigend geordnet
(> Ausdr1 Ausdr2 ... AusdrN) „ „ „ absteigend geordnet
sonst NIL

Aufruf mit einem Argument, Ergebnis T

=,<, > haben keine Seiteneffekte, die Auswertung der Ausdrücke kann welche produzieren

Beispiele:

```
---► (=  2.0  2.00  2)      ; Typen müssen nicht notwendigerweise übereinstimmen
 T

---► (<  -1.8   0   2.3  4)
 T

---► (>  17)
 T

---► (=  17.0  (SETQ  X ( +  13  4)
 T
```

Weitere Prädikate zu Vergleich

weitere Prädikate

```
( /=   a1   a2   ...   an)
(<=    a1   a2   ...   an)
(>=    a1   a2   ...   an)
```

} **Semantik analog zu oben**

Artvergleiche

- **(NUMBERP Ausdr)**

  Wert T, falls Wert von Ausdruck numerisch ist, d.h. externe Zahlendarstellung besitzt, NIL sonst

```
-->(NUMBERP  'X)
NIL
-->(NUMBERP  (+  17  4))
T
```

- **(SYMBOLP Ausdr)**

  Wert T, falls Wert von Ausdruck ein Symbol ist
  NIL sonst

- **(STRINGP Ausdr)**

  Wert T, falls Wert von Ausdruck Zeichenkette als Wert hat
  NIL sonst

- **Beispiele:**

```
----> (SYMBOLP  3.0)        ----> (SYMBOLP  'AB)
NIL                          T
----> (STRINGP  'AB)        ---->(STRINGP  "A0")
NIL                          T
```

Typüberprüfungen

- **Kennen inzwischen „Typen“ (Strukturierungen) von LISP-Objekten**
  - Atome, CONS-Paare, Listen, Zahlen, Symbole, Strings
  - ATOM CONSP LIST NUMBER SYMBOL STRINGP

    Prädikate

  COMMON LISP kennt viele weitere „Datentypen“, Benutzer kann selbst welche definieren

- **Zu jedem Datentyp gehört ein „Typ“-Spezifikator:**

  ATOM CONS NUMBER SYMBOL STRING

- **Ist kein Prädikat, kann aber im allg. als „Typvergleichs“-Prädikat verwendet werden**

  (TYPEP LISP-Objekt Typspez.)

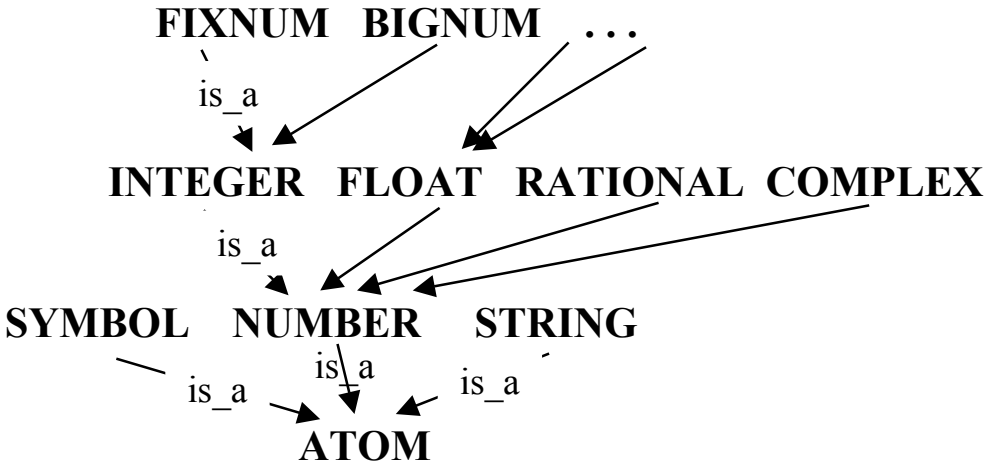
Typüberprüfung - Beispiele

- --▸ (NUMBERP 3.0)
  T

  --▸ (TYPEP 3.0 NUMBER)
  T

  --▸ (TYPEP 3.0 SYMBOL)
  NIL

- Datentypen sind nicht notwendigerweise disjunkt



- --▸ (TYPEP -3.06 INTEGER)
  NIL

  --▸ (TYPEP -3.06 SINGLE-FLOAT)
  T

Komplexe Prädikate

- **Zusammengesetzt aus vordef. Prädikaten (Struktur- und Wertevergleich) und Junktoren AND, OR, NOT: spez. Formen**

- **(AND A1 A2 . . . An)**

  Wert: Anw. A_i
  NIL, falls ein A_i NIL, sonst Wert An
  n=0 Wert T

- **(OR A1 A2 . . . An)**

  Wert: Anw. A_i
  falls ein A_i Wert ≠ NIL, dieser Wert
  NIL sonst
  n=0 Wert NIL

- **(NOT A)**

  Wert: NIL, falls Wert von A ≠ NIL
  T sonst

n≥0

Auswertung von li nach re

Kurzschlußauswertung:

Gefahr Seiteneffekte, nicht durchgeführt

Beispiele - 1

```
------► (AND (NUMBERP 3) (SETQ ACHTUNG 'SEITENEFF) (NULL 'T))
NIL

------► (AND (NUMBERP 3) (NULL 'T) (SETQ ACHTUNG 'K- SEITENEFF))
NIL

------► (AND)
T

------► (AND (NULL '() ) (MEMBER 'X3 '(X1 X2 X3 X4 )))
(X3 X4)

------► (OR (MEMBER 'X6 '(X1 X2 X3 )) (NULL '0))
NIL

------► (OR (MEMBER 'X3 '(X1 X2 X3 )) (NULL 'NIL))
(X3)

------► (OR (NULL 'NIL) (MEMBER 'X1 '(X1 X2 X3 )))
T

------► (OR)
NIL
```

Weitere Beispiele / Bemerkung

- **--→ (NOT 'NIL)**
  T

  --→ (NOT 'X) **; gleiches Ergebnis für bel. Ausdruck**
  NIL **; mit Wert ≠ NIL**

- **Bemerkung:**
  - AND, OR, NOT haben keine Seiteneffekte
    die Auswertung der Argumente kann solche bewirken
  - AND, OR, aufgrund der besonderen Regeln zur Auswertung: spezielle Formen

Bedingte Anweisungen

- **Spezielle Form COND für bedingte „Anweisungen“**
  d.h. Ausdrücke, durch Auswertung von Bedingungen abhängt

  (COND (T1 Programm-Stück1)

  (T1 PS-2)
  (T1 PS-n)) mit n≥ 1

  ...

  PS-i sind (f1 f2 ... fm), fi Formen, m≥ 0

  (Ti PS-i) heißen „Klauseln“

- **Auswertung:**

  Auswertung des Tests in der vorgebenen Reihenfolge bis ein Test-i -Wert ≠ NIL ergibt. In diesem Fall Auswertung des Rests der Klausel, Gesamtformauswertung beendet.

  Wert des PS-i ist Wert der COND-Form.

  Sind die Werte aller Teste NIL, so ist der Wert der Bedingung NIL.

  Ist das Programmstück der zutreffenden Klausel leer, so ist der Wert der Klausel der Wert des Tests.

- **Seiteneffekt:**

  keine durch COND, aber durch Auswertung der T-i und PS-i möglich:

Spezielle Prädikate und Beispiele - 1

- **Spezielle Prädikate**

  (= Ausdr1 Ausdr2 ... AusdrN) alle Werte Zahlen sind gleich
  (< Ausdr1 Ausdr2 ... AusdrN) „ „ „ aufsteigend geordnet
  (> Ausdr1 Ausdr2 ... AusdrN) „ „ „ absteigend geordnet
  sonst NIL

- **Aufruf mit einem Argument, Ergebnis T**

  =,<, > haben keine Seiteneffekte, die Auswertung der Ausdrücke kann welche produzieren

- **Beispiele:**

```
---➤ (=  2.0  2.00  2)      ; Typen müssen nicht notwendigerweise übereinstimmen
 T
---➤ (< -1.8  0  2.3  4)
 T
---➤ (> 17)
 T
---➤ (= 17.0 (SETQ X ( + 13 4)
 T
```


Weitere Prädikate zu Vergleich

weitere Prädikate

(/= a1 a2 ... an)
(<= a1 a2 ... an)
(>= a1 a2 ... an)

} **Semantik analog zu oben**

Artvergleiche

- **(NUMBERP Ausdr)**

  Wert T, falls Wert von Ausdruck numerisch ist, d.h. externe Zahlendarstellung besitzt, NIL sonst

```
-->(NUMBERP 'X)
NIL
-->(NUMBERP (+ 17 4))
T
```

- **(SYMBOLP Ausdr)**

  Wert T, falls Wert von Ausdruck ein Symbol ist
  NIL sonst

- **(STRINGP Ausdr)**

  Wert T, falls Wert von Ausdruck Zeichenkette als Wert hat
  NIL sonst

- **Beispiele:**

```
----> (SYMBOLP 3.0)        ----> (SYMBOLP 'AB)
NIL                        T
----> (STRINGP 'AB)        ----> (STRINGP "A0")
NIL                        T
```


Typüberprüfungen

- **Kennen inzwischen „Typen" (Strukturierungen) von LISP-Objekten**
  - Atome, CONS-Paare, Listen, Zahlen, Symbole, Strings
  - ATOM CONSP LIST NUMBER SYMBOL STRINGP
    Prädikate

  COMMON LISP kennt viele weitere „Datentypen", Benutzer kann selbst welche definieren

- **Zu jedem Datentyp gehört ein „Typ"-Spezifikator:**

  ATOM CONS NUMBER SYMBOL STRING

- **Ist kein Prädikat, kann aber im allg. als „Typvergleichs"-Prädikat verwendet werden**

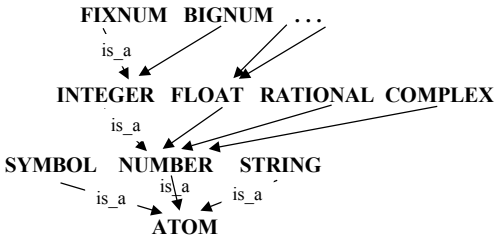
  (TYPEP LISP-Objekt Typspez.)

Typüberprüfung - Beispiele

- --▸ (NUMBERP 3.0)
  T

  --▸ (TYPEP 3.0 NUMBER)
  T

  --▸ (TYPEP 3.0 SYMBOL)
  NIL
- Datentypen sind nicht notwendigerweise disjunkt



- --▸ (TYPEP -3.06 INTEGER)
  NIL

  --▸ (TYPEP -3.06 SINGLE-FLOAT)
  T

Komplexe Prädikate

- Zusammengesetzt aus vordef. Prädikaten (Struktur- und Wertevergleich) und Junktoren AND, OR, NOT: spez. Formen
- (AND A1 A2 . . . An)
  Wert: Anw. A_i
  NIL, falls ein A_i NIL, sonst Wert An
  n=0 Wert T
- (OR A1 A2 . . . An)
  Wert: Anw. A_i
  falls ein A_i Wert ≠ NIL, dieser Wert
  NIL sonst
  n=0 Wert NIL
- (NOT A)
  Wert: NIL, falls Wert von A ≠ NIL
  T sonst

n≥0

Auswertung von li nach re

Kurzschlußauswertung:
Gefahr Seiteneffekte, nicht durchgeführt

Beispiele - 1

```
------► (AND (NUMBERP 3) (SETQ ACHTUNG 'SEITENEFF) (NULL 'T))
NIL

------► (AND (NUMBERP 3) (NULL 'T) (SETQ ACHTUNG 'K- SEITENEFF))
NIL

------► (AND)
T

------► (AND (NULL '() ) (MEMBER 'X3 '(X1 X2 X3 X4 )))
(X3 X4)

------► (OR (MEMBER 'X6 '(X1 X2 X3 )) (NULL '0))
NIL

------► (OR (MEMBER 'X3 '(X1 X2 X3 )) (NULL 'NIL))
(X3)

------► (OR (NULL 'NIL) (MEMBER 'X1 '(X1 X2 X3 )))
T

------► (OR)
NIL
```


Weitere Beispiele / Bemerkung

- --► (NOT 'NIL)
  T

  --► (NOT 'X) ; gleiches Ergebnis für bel. Ausdruck
  NIL ; mit Wert ≠ NIL

- **Bemerkung:**
  - AND, OR, NOT haben keine Seiteneffekte
    die Auswertung der Argumente kann solche bewirken
  - AND, OR, aufgrund der besonderen Regeln zur Auswertung: spezielle Formen

Bedingte Anweisungen

- **Spezielle Form COND für bedingte „Anweisungen“**
  d.h. Ausdrücke, durch Auswertung von Bedingungen abhängt

  (COND (T1 Programm-Stück1)

  (T1 PS-2)
  (T1 PS-n)) mit n≥ 1

  ...

  PS-i sind (f1 f2 ... fm), fi Formen, m≥ 0

  (Ti PS-i) heißen „Klauseln“

- **Auswertung:**

  Auswertung des Tests in der vorgebenen Reihenfolge bis ein Test-i -Wert ≠ NIL ergibt. In diesem Fall Auswertung des Rests der Klausel, Gesamtformauswertung beendet.

  Wert des PS-i ist Wert der COND-Form.

  Sind die Werte aller Teste NIL, so ist der Wert der Bedingung NIL.

  Ist das Programmstück der zutreffenden Klausel leer, so ist der Wert der Klausel der Wert des Tests.

- **Seiteneffekt:**

  keine durch COND, aber durch Auswertung der T-i und PS-i möglich:

Beispiele / Ausführung

- **Formen:**

```
(COND    ((MEMBER  EL LISTE)  LISTE)
         (T  (CONS  EL  LISTE) )
)
(COND    ((NULL  L)  NIL)
         ((EQUAL  EL  (CAR  L))  L)
         (T  (MEMBER  EL (CDR  L)))
 )
```

- **Ausführung:**

```
--->(SETQ  L  '(AB))
(AB)
--->(COND  (( MEMBER  (LAST  '((AB) (CD)))  L)   'FALSCH)
           (( < (LENGTH    (APPEND  '(L)  L)) 4)
                           (REVERSE  '(G I T H C I R))))
(RICHTIG)
```

Funktionen: Definition und Aufruf

Einordnung

Systemfunktionen (bisher) — spezielle Funktionen (Formen)
benutzerdef. Funktionen - normale Funktionen (Formen)
benannte
anonyme Funktionen

benannte Funktionen

- **Definition (Deklaration, Einführung)**

  (DEFUN FName [Parlist] [Rumpf])

  n-Liste v. Symbolen — Folge von Formen

  Auswertung: Wert Funktionsname
  Seiteneffekt Zuordnung FName → Rumpf

- **Aufruf**

  (FName a_1 a_2 .. .. a_n) a_i Ausdrücke

  Auswertung: a_1, a_2 ... ausgewertet
  Bindung am Formalparameter
  Auswertung des Rumpfs
  Wert der letzten Form ist Wert, falls 0 Wert NIL
  Bindung aufgelöst

Funktionen: Beispiele

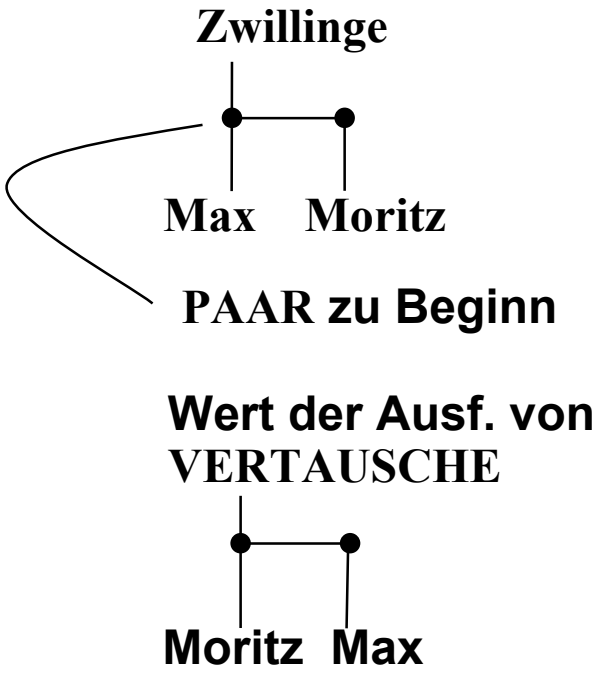
Beispiel:

```
--->(DEFUN  VERTAUSCHE   (PAAR)
                (LIST (CADR  PAAR) (CAR  PAAR)))
VERTAUSCHE
--->(SETQ  ZWILLINGE  '(MAX   MORITZ))
(MAX  MORITZ)
--->(VERTAUSCHE  ZWILLINGE)
(MORITZ MAX)
```

Bemerkung:

- Bindung (dynamisch) nur während Funktionsauswertung, aber s.u.
- neue Funktionsdef. an bereits ex. Namen durch DEFUN:

  Systemfunktionen überschreibbar !
- DEFUN ist spezielle Form: keiner ihrer Argumente wird ausgewertet



Unbenannte Funktionen, Lambda-Ausdrücke

- **(LAMBDA [Parlist] [Rumpf]) analog zu DEFUN**

  Definiert namenlose Funktion, Lambda-Ausdruck hat keinen Wert
  Auswertbarer Ausdruck: zusammen mit Argumenten

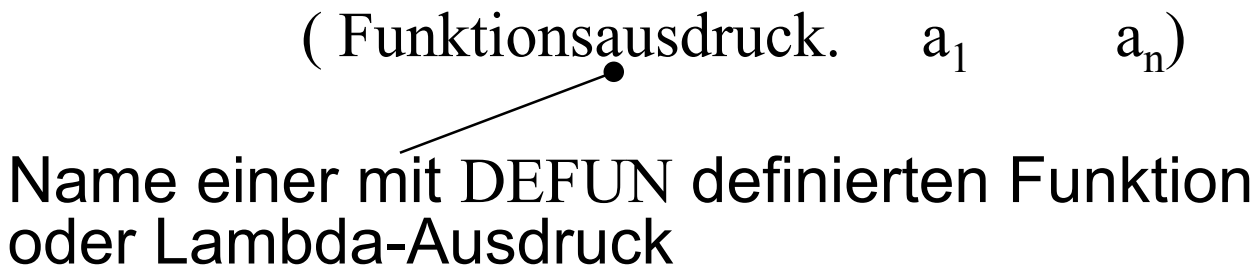
  (Lambda-Ausdruck a1 a2 . . . an) **Auswertung wie bei DEFUN**

- **Beispiel:**

```
-----►(LAMBDA (PAAR) (LIST (CADR PAAR) (CAR PAAR)))

      ; ist nicht auswertbar, jedoch

-----►(( LAMBDA (PAAR) (LIST (CADR PAAR) (CAR PAAR)))
                    '(FRAU MANN))
(MANN FRAU)
```

Bemerkungen: DEFUN, LAMBDA

- **DEFUN führt Namen ein**
  Aufruf und Definition beliebig weit auseinander
  ansonsten Ausführung gleich
  LAMBDA-Funktion nur an Stelle der Definition aufrufbar
- **LAMBDA historisch bedingt → Lambda-Kalkul (CHURCH 41)**
  besser AFUN für Anonymous Function
- **Auswertung eines Funktionsaufrufs heißt Lambda-Konversion**
- **Überall, wo Funktionsaufrufe stehen dürfen, dürfen auch Lambda-Ausdrücke stehen**



Beispiele für Funktionsdeklarationen

- **Hinzunahme eines Objekts zu einer Liste, falls noch nicht enthalten**

```
(DEFUN  OUR-ADJOIN  (OBJEKT  LISTE)
    (COND  ((MEMBER  OBJEKT  LISTE)  LISTE)
           (T  (CONS  OBJEKT  LISTE))))
```

- **Erweiterung OUR-ADJOIN, so daß Reihenfolge der Argumente beliebig sein darf:**

```
(DEFUN  ANY-ORDER-ADJOIN  (ARG-1 ARG-2)
    (COND  ((LISTP  ARG-1)
            (OUR-ADJOIN  ARG-2  ARG-1))
          (T  (OUR-ADJOIN  ARG-1  ARG-2))
))
```

oder für den Rumpf

```
    (SETQ  OBJEKT ARG-1
           LISTE    ARG-2)
    (COND  ((LISTP  ARG-1)   (SETQ  OBJEKT  ARG-2
                                    LISTE    ARG-1)))
    (OUR –ADJOIN  ARG-1  ARG-2)
```

Rekursion: Beispiel

- **Namensgebung**

  direkt rekursiv - rekursiv
  indirekt rekursiv - korekursiv

- **Länge einer Liste:**

```
(DEFUN  OUR-LENGTH  (L)
   (COND  (( NULL  L )  0 )       ; einf. Fall zuerst
          (T  ( +1  (OUR-LENGTH  (CDR  L ))))))
```

  Laufzeitkeller auf der Listenhalde

- **Anzahl der Atome eines Ausdrucks:**

```
(DEFUN  COUNT-ATOMS  (L)
   (COND  (( NULL  L ) 0 )
          (( ATOM L ) 1 )
          (T  ( +  (COUNT-ATOMS  ( CAR  L))
                   (COUNT-ATOMS  ( CDR  L ))))))
```

Wann Rekursion

- **elegante Lösungen ergeben sich aufgrund:**
  - rekursiver Definitionen, denen rekursiver Aufbau zugrundeliegender Listen entspricht
- **Da dies die einzigen Strukturen sind, ergeben sich hier naheliegender Weise mehr rekursive Lösungen**

  Liste: erstes Element, Restliste

  Liste: Folge von Elementen (Atom oder Liste)
- **z.B. rekursive Definition von REVERSE**

  leere Liste $\rightarrow$ **leere Liste**

  Liste $\rightarrow$ **(REVERSE (CDR Liste)) + hinten alten Kopf**

Parameterübergabe

- Parameterübergabe bei Funktionsaufruf
- ------> (SETQ FREI 3)

  3

  ------> (DEFUN DEMO (GEBUNDEN)
  (SETQ GEBUNDEN (+ GEBUNDEN 10))
  (+ GEBUNDEN FREI))

- DEMO

  ------> (DEMO FREI)

  16

  ------> FREI

  3

Call by Value für Parameter-übergabe

Freie und gebundene Variable

- **freie und gebundene Variable**

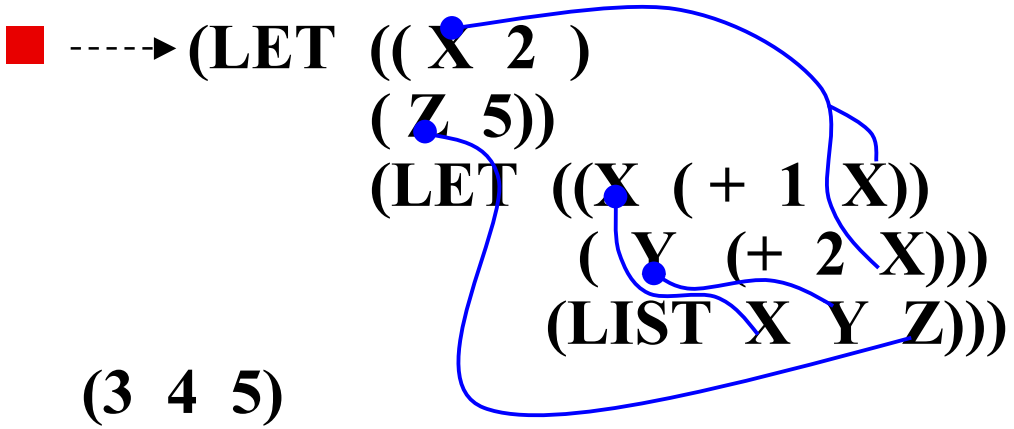
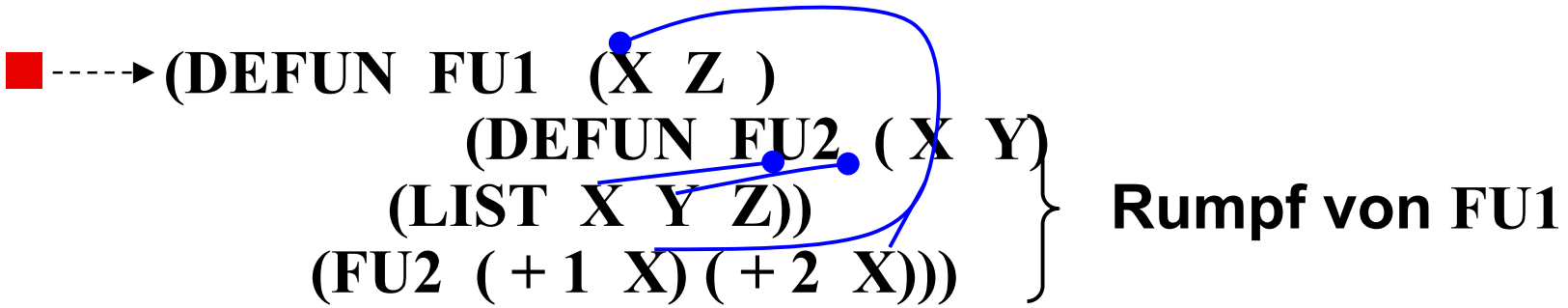
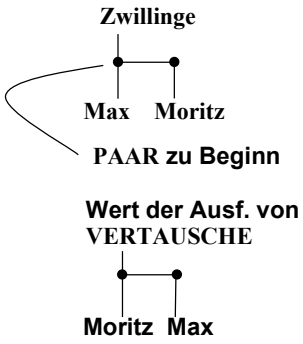
  „gebundene“ oder „lokale“ Variable, wenn sie in Parameterliste auftauchen (geb. Umbennung ändert nichts)

  } **in Bezug auf best. Funktions-definition**

  frei, wenn sie auftauchen, aber nicht in Parameterliste

- ```
  -->(SETQ FREI  3)
  3
  -->(DEFUN  DAMO (GEBUNDEN)
            (SETQ  GEBUNDEN  ( + GEBUNDEN  10))
            ( + FREI  (SETQ FREI  GEBUNDEN )))  ; Seiteneffekt: Vorsicht


  DAMO
  -->(DAMO  FREI)
  16

  -->FREI
  13
  ```
  ```

Namengleichheit freier und gebundener Variablen

```
→(SETQ FREI  3)
3
→(DEFUN  DEMO1  (FREI)
         (SETQ  FREI ( + FREI  10) )
         ( + FREI  FREI ))

DEMO1
→(DEMO1   2)
24
→FREI
3
```

- Zuerst Bindung FREI an 2, dann Bindung FREI an 12. Nach Auswertung von DEMO1 alte Bindung wieder: Während der Auswertung von DEMO1 ist die alte Bindung verdeckt

Bindung und Umgebung

- Wertzuweisung
  Parameterbindung (zeitweilig)
  } Änderungen bestehender Zuordnungen
- Gesamtheit aller zugängl. Zuordnungen: aktuelle Umgebung
  Start: globale oder Top-Level-Umgebung

  globaler Wert eines Atoms: Wert auf Top-Level

  Wertzuweisung: Umgebung geändert
  Funktionsausführung: neue Umgebung eingerichtet für Funktionsaufruf
- Definitionsumgebung einer Funktion: Umgebung z. Zt. der Abarbeitung der Definition

  Aufrufumgebung der Funktion: Umgebung z. Zt. der Auswertung

Beispiele: Bindung / Umgebung

```
⟶(DEFUN SUCESSOR (N)
                (+ 1 N))
SUCESSOR
⟶(SUCESSOR 17)
18
⟶(SETQ N 10)
10
⟶(DEFUN FUNNY-SUCC (N)
          (SETQ A (+ 1 N))
          (AUX-SUCC))
FUNNY-SUCC
⟶(DEFUN AUX-SUCC ( )
          (+ 1 N))
AUX-SUCC
⟶(FUNNY-SUCC 17)
11
⟶A
18
```

Bindung $N \rightarrow 17$ (s.u.) gilt nicht während der Auswertung von AUX-SUCC: dort ist N eine freie Variable

Bindung aus Def. Umg. von AUX-SUCC

COMMON-LISP : „statischer“ Gültigkeitsbereich (Bindung)

alte LISP-Dialekte: dynamische Bindung

Let-Ausdrücke und neue Bindungen

- bisher lokale Variable nur aus der Parameterliste einer Funktion
  jetzt lokale Variable auch in LET-Ausdrücken (entspr. in etwa Block)

- (LET ((Par-1 Ausdr-1)
  ⋮
  (Par-n Ausdr-n))
  Rumpf)

  } lok. Bindungen, mit denen der folg. Rumpf auszuwerten ist

  ; Rumpf Formen $f_1 \ldots fm$, $m \geq 0$

- Auswertung Ausdr-i, Bindung an Par-i „parallel“
  danach Auswertung des Rumpfs
  Wert des LET-Ausdrucks ist Wert der letzten Form, NIL für m=0
  anschließend Aufhebung der Bindungen
  keine Seiteneffekte durch LET aber durch Auswertung von Ausdr-i bzw. fj

LET vs. LAMBDA

- LET-Ausdruck übersichtlicher als der entsprechende Lambda-Ausdruck
  Im LISP-System werden LET-Ausdrücke auf LAMBDA-Ausdrücke zurückgeführt

- ```
  (( LAMBDA   (Par-1  . . . Par-n)
              Rumpf)
              Ausdr-1 . . . Ausdr-n)
  ```

  Bedeutung von LET-Ausdr. als funktionale Argumente (s.u.)

- ```
  ----►(SETQ   X   2)
  2

  ----►(LET  (( X   ( +  1  X))
             (Y  (+  2  X )))
             (LIST  X  Y))
  (3  4 )
  ```

Schachtelung und Gültigkeitsbereich

- **Schachtelung von LET-Ausdrücken bzw. Funktionen und Gültigkeitsbereich**



```
----> (LET  (( X  2  )
            ( Z  5))
            (LET  ((X  ( +  1  X))
                   ( Y   (+  2  X)))
                  (LIST  X  Y  Z)))
      (3  4  5)
```



```
----> (DEFUN  FU1   (X  Z  )
                (DEFUN  FU2  ( X  Y)
                  (LIST  X  Y  Z))
                (FU2  ( + 1  X) ( + 2  X)))
      FU1
----> (FU1  2  5)
      (3  4  5)
```

} **Rumpf von FU1**

Funktionale Argumente

- **LISP-Funktionen haben die gleiche Gestalt wie LISP-Daten, können wie Daten behandelt und verarbeitet werden, insbesondere können Ausdrücke Funktionsbeschreibungen als Wert haben**

- **(SETQ FN (COND (C < X Y) 'ADJOIN)**
  (T 'MEMBER)))

  Auswertung durch FUNCALL oder APPLY

Aufruf mit verschiedenen Funktionen

- **(FUNCALL Funkt-Obj $a_1 \ldots a_n$)**

Ausdruck mit Funktionsbeschreibung als Wert

Argumente für Funktionsaufruf

keine Seiten-effekte durch FUNCALL aber durch Ausw. von fO, a_i möglich

⟶(SETQ L '(A B C) X 3 Y 5)

5

⟶(LET ((FN (COND ((< X Y) 'ADJOIN)
(T 'MEMBER))))
(FUNCALL FN 'D L))

(D A B C)

⟶(FUNCALL 'ADJOIN 'D L)

(D A B C)

Auswertung durch APPLY

- **Auswertung fO durch APPLY**

  flexibler: Argumente der Funktion einzeln oder zusammen in einer Liste

  (APPLY fO a_1 . . . a_n a_{n+1})

  wie oben **Wert** **Liste**

  falls n=0 alle Argumente in

- **Beispiel:**

  ⟶(APPLY '+ (1 2 3 4 5 6)) ; Standardfall
  21

  ⟶(APPLY '+ 1 2 3 '(4 5 6))
  21

Funktionale Objekte

- **Mathematik: funktional: Objekt mit Wertebereich Funktion**
  bisher aufgetretene fOe einfach
  fOe häufig Aufrufe von SYMBOL-Funktion oder FUNCTION

- **(SYMBOL-FUNCTION Symb) ; ermittelt die Symbol zugeordnete Funktion**

```
(SETQ   FN  (COND  (( <  X   Y)   'ADJOIN)
                   (T        'MEMBER )))
(SYMBOL – FUNCTION    'FN)
```

- **(FUNCTION Lambda-Ausdr)**

  Wert sog. „Closure“ des Lambda-Ausdrucks:
  die durch Lambda-Ausdruck darg. Funktion zusammen mit der Bindung der freien Variablen entsprechend Gültigkeitsbereichsregeln

```
--->(DEFUN  ADDER  (X)  (FUNCTION  (LAMBDA  (Y)  (+  X  Y))))
ADDER
```

Was haben wir gelernt?

- COMMON-LISP Ausschnitt für wertorientierte/applikative Programmierung
- LISP-Programm: Folge von auswertbaren Ausdrücken (Formen) mit oder ohne Seiteneffekt (Zuordnung von Variablen zu Atomen oder Listen)
- Interne Listenstruktur als spezielle Haldenstruktur, Auswertung durch rekursiven Abstieg (Compilation!), Gargabe Collection nötig, Listen in leftmost-son-right-sibling-Darstellung
- Auswertung oder Quotierung
- Listenoperationen: Aufbau mit CONS, Zusammenfügen APPEND, LIST
  Zerlegen mit CAR, CDR, abgekürzte
  Zusammenfassung wie CADDR
  Spiegelung mit REVERSE
  Zugriff auf Teile mit LAST, MEMBER
- Strukturprädikate ATOM, CONSP, NULL, LISTP, EQ, EQUAL, LENGTH, <, =, etc.
  Artvergleiche NUMBERP, SYMPOLP, STRINGP, TYPEP, n-äre Junktoren AND, OR, unäres NOT
- Bedingte „Anweisungen“ COND
- Funktionen: Definition und Aufruf, unbenannte Funktionen mit LAMBDA-Ausdrücken, Rekursionshandhabung, Parameterübergabe, freie/gebundene Variable, Bindung und Umgebung, LET-Ausdrücke, Schachtelung, Gültigkeitsbereich, funktionale Argumente, funktionale Objekte

Glossar

- Atome, Listen, Funktionen, Funktionssymbole, Argumente
- Variable als Verweise auf Atome oder Listen, Listenaufbau aus CONS-Zellen
- Seiteneffekte als Änderung der Variablenzuordnung mit SETQ, Top-Level-Schleife, Interpretation als Systemfunktion, Compilation äquivalenten Codes wegen Effizienz
- Listenfunktionen, Listenprädikate (Unterscheidung nicht sauber), Strukturprädikat für Atome, Ausdrücke, Listen, Bedingungen, Klauseln
- DEFUN, LAMBDA, LET, APPLY
- Namensgleichheit, freie/gebundene Variable, Bindung, Umgebung, Schachtelung, Gültigkeit
- funktionale Argumente, Bindung verschiedener Funktionen

Beispiele / Ausführung

- **Formen:**

```
(COND    ((MEMBER EL LISTE) LISTE)
         (T (CONS EL LISTE) )
)
(COND    ((NULL L) NIL)
         ((EQUAL EL (CAR L)) L)
         (T (MEMBER EL (CDR L)))
 )
```

- **Ausführung:**

```
--->(SETQ L '(AB))
(AB)
--->(COND (( MEMBER (LAST '((AB) (CD))) L)  'FALSCH)
          (( < (LENGTH  (APPEND '(L) L)) 4)
                        (REVERSE '(G I T H C I R))))
(RICHTIG)
```


Funktionen: Definition und Aufruf

- **Einordnung**

  Systemfunktionen (bisher) — spezielle Funktionen (Formen)
  benutzerdef. Funktionen — normale Funktionen (Formen)
  - benannte
  - anonyme Funktionen

- **benannte Funktionen**
  - **Definition (Deklaration, Einführung)**

    (DEFUN FName [Parlist] [Rumpf])
    Parlist: n-Liste v. Symbolen; Rumpf: Folge von Formen

    Auswertung: Wert Funktionsname
    Seiteneffekt Zuordnung FName → Rumpf
  - **Aufruf**

    (FName a_1 a_2 .. .. a_n) a_i Ausdrücke

    Auswertung: a_1, a_2 ... ausgewertet
    Bindung am Formalparameter
    Auswertung des Rumpfs
    Wert der letzten Form ist Wert, falls 0 Wert NIL
    Bindung aufgelöst

Funktionen: Beispiele

- **Beispiel:**

  --→ **(DEFUN VERTAUSCHE (PAAR)**
  (LIST (CADR PAAR) (CAR PAAR)))
  VERTAUSCHE

  --→ **(SETQ ZWILLINGE '(MAX MORITZ))**
  (MAX MORITZ)

  --→ **(VERTAUSCHE ZWILLINGE)**
  (MORITZ MAX)

- **Bemerkung:**
  - Bindung (dynamisch) nur während Funktionsauswertung, aber s.u.
  - neue Funktionsdef. an bereits ex. Namen durch DEFUN:
    Systemfunktionen überschreibbar !
  - DEFUN ist spezielle Form: keiner ihrer Argumente wird ausgewertet



Unbenannte Funktionen, Lambda-Ausdrücke

- **(LAMBDA [Parlist] [Rumpf]) analog zu DEFUN**

  Definiert namenlose Funktion, Lambda-Ausdruck hat keinen Wert
  Auswertbarer Ausdruck: zusammen mit Argumenten

  (Lambda-Ausdruck a1 a2 . . . an) **Auswertung wie bei DEFUN**

- **Beispiel:**

  -----→ **(LAMBDA (PAAR) (LIST (CADR PAAR) (CAR PAAR)))**

  ; ist nicht auswertbar, jedoch

  -----→ **((LAMBDA (PAAR) (LIST (CADR PAAR) (CAR PAAR)))**
  '(FRAU MANN))
  (MANN FRAU)

Bemerkungen: DEFUN, LAMBDA

- **DEFUN führt Namen ein**
  Aufruf und Definition beliebig weit auseinander
  ansonsten Ausführung gleich
  LAMBDA-Funktion nur an Stelle der Definition aufrufbar
- **LAMBDA historisch bedingt → Lambda-Kalkul (CHURCH 41)**
  besser AFUN für Anonymous Function
- **Auswertung eines Funktionsaufrufs heißt Lambda-Konversion**
- **Überall, wo Funktionsaufrufe stehen dürfen, dürfen auch Lambda-Ausdrücke stehen**

(Funktionsausdruck. a_1 a_n)

Name einer mit DEFUN definierten Funktion
oder Lambda-Ausdruck

Beispiele für Funktionsdeklarationen

- **Hinzunahme eines Objekts zu einer Liste, falls noch nicht enthalten**

```
(DEFUN OUR-ADJOIN (OBJEKT LISTE)
   (COND ((MEMBER OBJEKT LISTE) LISTE)
         (T (CONS OBJEKT LISTE))))
```

- **Erweiterung OUR-ADJOIN, so daß Reihenfolge der Argumente beliebig sein darf:**

```
(DEFUN ANY-ORDER-ADJOIN (ARG-1 ARG-2)
   (COND ((LISTP ARG-1)
          (OUR-ADJOIN ARG-2 ARG-1))
         (T (OUR-ADJOIN ARG-1 ARG-2))
))
```

oder für den Rumpf

```
   (SETQ OBJEKT ARG-1
         LISTE  ARG-2)
   (COND ((LISTP ARG-1)  (SETQ OBJEKT ARG-2
                               LISTE  ARG-1)))
   (OUR –ADJOIN ARG-1 ARG-2)
```

Rekursion: Beispiel

- **Namensgebung**
  direkt rekursiv - rekursiv
  indirekt rekursiv - korekursiv

- **Länge einer Liste:**

```
(DEFUN  OUR-LENGTH  (L)
   (COND  (( NULL  L )  0 )       ; einf. Fall zuerst
          (T  ( +1  (OUR-LENGTH  (CDR  L ))))))
```

  Laufzeitkeller auf der Listenhalde

- **Anzahl der Atome eines Ausdrucks:**

```
(DEFUN  COUNT-ATOMS  (L)
   (COND  (( NULL  L ) 0 )
          (( ATOM L ) 1 )
          (T  ( +  (COUNT-ATOMS  ( CAR  L))
                   (COUNT-ATOMS  ( CDR  L ))))))
```


Wann Rekursion

- **elegante Lösungen ergeben sich aufgrund:**
  - rekursiver Definitionen, denen rekursiver Aufbau zugrundeliegender Listen entspricht

- **Da dies die einzigen Strukturen sind, ergeben sich hier naheliegender Weise mehr rekursive Lösungen**

  Liste: erstes Element, Restliste

  Liste: Folge von Elementen (Atom oder Liste)

- **z.B. rekursive Definition von REVERSE**

  leere Liste → **leere Liste**

  Liste → **(REVERSE (CDR Liste)) + hinten alten Kopf**

Parameterübergabe

- Parameterübergabe bei Funktionsaufruf
- ------→(SETQ FREI 3)

  3

  ------→(DEFUN DEMO (GEBUNDEN)
  (SETQ GEBUNDEN (+ GEBUNDEN 10))
  (+ GEBUNDEN FREI))

  Call by Value für Parameter-übergabe

- DEMO

  ------→(DEMO FREI)

  16

  ------→FREI

  3

Freie und gebundene Variable

- freie und gebundene Variable

  „gebundene“ oder „lokale“ Variable, wenn sie in Parameterliste auftauchen (geb. Umbennung ändert nichts)

  } in Bezug auf best. Funktions-definition

  frei, wenn sie auftauchen, aber nicht in Parameterliste

- --→(SETQ FREI 3)
  3
  --→ (DEFUN DAMO (GEBUNDEN)
  (SETQ GEBUNDEN (+ GEBUNDEN 10))
  (+ FREI (SETQ FREI GEBUNDEN))) ; Seiteneffekt: Vorsicht

  DAMO
  --→ (DAMO FREI)
  16

  --→ FREI
  13

Abrundung

Namengleichheit freier und gebundener Variablen

- →(SETQ FREI 3)
  3
  →(DEFUN DEMO1 (FREI)
  (SETQ FREI (+ FREI 10))
  (+ FREI FREI))

  DEMO1
  →(DEMO1 2)
  24
  →FREI
  3

- Zuerst Bindung FREI an 2, dann Bindung FREI an 12. Nach Auswertung von DEMO1 alte Bindung wieder: Während der Auswertung von DEMO1 ist die alte Bindung verdeckt

Abrundung

Bindung und Umgebung

- Wertzuweisung, Parameterbindung (zeitweilig) } Änderungen bestehender Zuordnungen
- Gesamtheit aller zugängl. Zuordnungen: aktuelle Umgebung
  Start: globale oder Top-Level-Umgebung

  globaler Wert eines Atoms: Wert auf Top-Level

  Wertzuweisung: Umgebung geändert
  Funktionsausführung: neue Umgebung eingerichtet für Funktionsaufruf
- Definitionsumgebung einer Funktion: Umgebung z. Zt. der Abarbeitung der Definition

  Aufrufumgebung der Funktion: Umgebung z. Zt. der Auswertung

Abrundung

Beispiele: Bindung / Umgebung

```
⟶(DEFUN SUCESSOR (N)
        (+ 1 N))
SUCESSOR
⟶(SUCESSOR 17)
18
⟶(SETQ N 10)
10
⟶(DEFUN FUNNY-SUCC (N)
        (SETQ A (+ 1 N))
        (AUX-SUCC))
FUNNY-SUCC
⟶(DEFUN AUX-SUCC ( )
        (+ 1 N))
AUX-SUCC
⟶(FUNNY-SUCC 17)
11
⟶A
18
```

Bindung N → 17 (s.u.) gilt nicht während der Auswertung von AUX-SUCC: dort ist N eine freie Variable

Bindung aus Def. Umg. von AUX-SUCC

COMMON-LISP : „statischer“ Gültigkeitsbereich (Bindung)

alte LISP-Dialekte: dynamische Bindung

Abrundung

Let-Ausdrücke und neue Bindungen

- bisher lokale Variable nur aus der Parameterliste einer Funktion
  jetzt lokale Variable auch in LET-Ausdrücken (entspr. in etwa Block)
- (LET ((Par-1 Ausdr-1) … (Par-n Ausdr-n)) Rumpf) ; Rumpf Formen $f_1 \ldots fm$, $m \geq 0$

  } lok. Bindungen, mit denen der folg. Rumpf auszuwerten ist
- Auswertung Ausdr-i, Bindung an Par-i „parallel“
  danach Auswertung des Rumpfs
  Wert des LET-Ausdrucks ist Wert der letzten Form, NIL für m=0
  anschließend Aufhebung der Bindungen
  keine Seiteneffekte durch LET aber durch Auswertung von Ausdr-i bzw. fj

Abrundung

LET vs. LAMBDA

- LET-Ausdruck übersichtlicher als der entsprechende Lambda-Ausdruck
  Im LISP-System werden LET-Ausdrücke auf LAMBDA-Ausdrücke zurückgeführt
- ((LAMBDA (Par-1 . . . Par-n)
  Rumpf)
  Ausdr-1 . . . Ausdr-n)

  Bedeutung von LET-Ausdr. als funktionale Argumente (s.u.)
- ----→(SETQ X 2)
  2

  ----→(LET ((X (+ 1 X))
  (Y (+ 2 X)))
  (LIST X Y))
  (3 4)

Abrundung

Schachtelung und Gültigkeitsbereich

- Schachtelung von LET-Ausdrücken bzw. Funktionen und Gültigkeitsbereich
- ----→(LET ((X 2)
  (Z 5))
  (LET ((X (+ 1 X))
  (Y (+ 2 X)))
  (LIST X Y Z)))
  (3 4 5)

- ----→(DEFUN FU1 (X Z)
  (DEFUN FU2 (X Y)
  (LIST X Y Z))
  (FU2 (+ 1 X) (+ 2 X)))

  } Rumpf von FU1

  FU1
  ----→ (FU1 2 5)
  (3 4 5)

Funktionale Argumente

- LISP-Funktionen haben die gleiche Gestalt wie LISP-Daten, können wie Daten behandelt und verarbeitet werden, insbesondere können Ausdrücke Funktionsbeschreibungen als Wert haben

- ```
  (SETQ FN (COND (C < X Y)  'ADJOIN)
                 (T   'MEMBER)))
  ```

  Auswertung durch FUNCALL oder APPLY

Aufruf mit verschiedenen Funktionen

- (FUNCALL Funkt-Obj $\underbrace{a_1 \ldots a_n}$)

  Argumente für Funktionsaufruf

  Ausdruck mit Funktionsbeschreibung als Wert

keine Seiteneffekte durch FUNCALL aber durch Ausw. von fO, a_i möglich

```
→(SETQ L '(A B C) X 3 Y 5)
5
→(LET ((FN (COND ((< X Y) 'ADJOIN)
                 (T 'MEMBER))))
      (FUNCALL FN 'D L))
(D A B C)
→(FUNCALL 'ADJOIN 'D L)
(D A B C)
```
```

Auswertung durch APPLY

- **Auswertung fO durch APPLY**

  flexibler: Argumente der Funktion einzeln oder zusammen in einer Liste

  (APPLY fO a_1 . . . a_n a_{n+1})

  wie oben **Wert** **Liste**

  falls n=0 alle Argumente in

- **Beispiel:**

  →(APPLY '+ (1 2 3 4 5 6)) ; Standardfall
  21

  →(APPLY '+ 1 2 3 '(4 5 6))
  21

Funktionale Objekte

- **Mathematik: funktional: Objekt mit Wertebereich Funktion**
  bisher aufgetretene fOe einfach
  fOe häufig Aufrufe von SYMBOL-Funktion oder FUNCTION
- **(SYMBOL-FUNCTION Symb) ; ermittelt die Symbol zugeordnete Funktion**

  (SETQ FN (COND ((< X Y) 'ADJOIN)
  (T 'MEMBER)))
  (SYMBOL – FUNCTION 'FN)
- **(FUNCTION Lambda-Ausdr)**
  Wert sog. „Closure“ des Lambda-Ausdrucks:
  die durch Lambda-Ausdruck darg. Funktion zusammen mit der Bindung der freien Variablen entsprechend Gültigkeitsbereichsregeln

  ---▸(DEFUN ADDER (X) (FUNCTION (LAMBDA (Y) (+ X Y))))
  ADDER

Was haben wir gelernt?

- COMMON-LISP Ausschnitt für wertorientierte/applikative Programmierung
- LISP-Programm: Folge von auswertbaren Ausdrücken (Formen) mit oder ohne Seiteneffekt (Zuordnung von Variablen zu Atomen oder Listen)
- Interne Listenstruktur als spezielle Haldenstruktur, Auswertung durch rekursiven Abstieg (Compilation!), Gargabe Collection nötig, Listen in leftmost-son-right-sibling-Darstellung
- Auswertung oder Quotierung
- Listenoperationen: Aufbau mit CONS, Zusammenfügen APPEND, LIST
  Zerlegen mit CAR, CDR, abgekürzte
  Zusammenfassung wie CADDR
  Spiegelung mit REVERSE
  Zugriff auf Teile mit LAST, MEMBER
- Strukturprädikate ATOM, CONSP, NULL, LISTP, EQ, EQUAL, LENGTH, <, =, etc. Artvergleiche NUMBERP, SYMPOLP, STRINGP, TYPEP, n-äre Junktoren AND, OR, unäres NOT
- Bedingte „Anweisungen“ COND
- Funktionen: Definition und Aufruf, unbenannte Funktionen mit LAMBDA-Ausdrücken, Rekursionshandhabung, Parameterübergabe, freie/gebundene Variable, Bindung und Umgebung, LET-Ausdrücke, Schachtelung, Gültigkeitsbereich, funktionale Argumente, funktionale Objekte

Glossar

- Atome, Listen, Funktionen, Funktionssymbole, Argumente
- Variable als Verweise auf Atome oder Listen, Listenaufbau aus CONS-Zellen
- Seiteneffekte als Änderung der Variablenzuordnung mit SETQ, Top-Level-Schleife, Interpretation als Systemfunktion, Compilation äquivalenten Codes wegen Effizienz
- Listenfunktionen, Listenprädikate (Unterscheidung nicht sauber), Strukturprädikat für Atome, Ausdrücke, Listen, Bedingungen, Klauseln
- DEFUN, LAMBDA, LET, APPLY
- Namensgleichheit, freie/gebundene Variable, Bindung, Umgebung, Schachtelung, Gültigkeit
- funktionale Argumente, Bindung verschiedener Funktionen

Teil V

Andere Paradigmen

- **Exkurs 1: PROLOG**
  Logikprogrammierung
- **Exkurs 2: LISP**
  Applikative Programmierung

Exkurs 1: PROLOG (Logikprogrammierung)

- Allgemeines
- Grundkonzepte
- Konzepte und Terminologie

PROLOG als deklarative Sprache

- **imperativ vs. deklarativ**
  - imperative Programme (Programmiersprachen) incl. funktional, objektorientiert:
    Programmierer gibt an, wie eine Lösung aussehen soll,
  - deklarative Programme (Programmiersprachen):
    Programmierer gibt an, was er haben will

- **PROLOG (Programmierung in Logic) 1972, Kowalski & Colmerauer**
  - Bedeutung für spezielle Probleme (KI)
  - Schub: 5th-Generation-Projekt (Japan): Ernüchterung
  - PROLOG, neben LISP und der Datenbankanfragesprache SQL wichtigste nichtimperative Sprache

Erläuterung und Literatur

- **Literatur**

  Sterling / Shapiro
  Clocksin / Mellish } **Literaturverzeichnis**

- **hier nur kurzer Abriß**

- **induktive Erläuterung**
  - **Beispiele**
  - **Terminologie und allgemeine Prinzipien später**

Fakten und einfache Fragen

- **Basisfakten (Aussagen über konkrete Objekte und Beziehungen)**

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.
```

Relation(enbezeichner), Objektnamen

null-, ein-, zwei-, und dreistellige Relationen

- **Fragen (normale Eingaben des Programms) Hinschreibung systemspezifisch (z.B.: „hat (Frieda,Buch)?“)**

```
?- hat(frieda,buch).
ja
```

- **Ausführung für einfaches Beispiel: Nachsehen in Faktenbasis; gibt es Fakt, Ausgabe ja, sonst nein**

Weitere einfache Fragen

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.
```

```
?- mag (hans,frieda).
ja
?-mag (frieda,hans).
nein
?-mag (hans,geld).
nein
?-liebt (hans,frieda).
nein
?- feineswetter
ja
```

Relationen sind gerichtet!

Fragen mit Variablen

- **Variable für Objekte: „Was mag Hans?“ anstelle von „Mag Hans Frieda?“**

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda,nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.

?- mag(hans,X).
X=fisch;
X=frieda;
nein
```

- **Ausführung (interaktiv)**
  - Suche nach einem „passenden“ Fakt
  - Wünscht Bediener Fortsetzung: Eingabe „ ; “
  - existiert kein weiteres Fakt: Ausgabe: nein
  - PROLOG-System sucht gemäß Auflistungsreihenfolge

Konjunktiv verknüpfte Fragen und Backtracking

- **„Existiert ein Objekt X, so daß Hans X mag und X ein Buch hat?“**

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.

?- mag(hans,X); hat(X,buch).
X=frieda;
nein
```

- **Backtracking: zwei Teilfragen sind passend zu beantworten**
  - erste Teilantwort Substitution α / fisch aus mag (hans,fisch)
  - Teilfrage 2 mit gleicher Substitution „?-hat (fisch,buch)“ Antwort negativ, Substitution wird aufgehoben
  - Suche nach neuer Substitution für erste Teilfrage: X / frieda Teilfrage 2 nun „?-mag(hans,frieda)“ erfolgreich
  - Eingabe „ ; “ weiteres Backtracking ohne Erfolg

Weitere Beispiele für Fragen und Backtracking

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.
```

```
?-mag (X,Y)
X=hans, Y=fisch;
X=hans, Y=frieda;
nein

?-mag (X,X).
nein
```

- **Fragen beziehen sich auf Existenz einer Lösung:**
  - Gefunden: in Ausgabe angegeben
  - Weitere Lösungen explizit anfordern

Universelle Fakten

Variablen in Fakten

mag(alfred,X).	„Alfred mag alles“
mag(X,X).	„Jedes mag sich selbst“
mag(X,Y)	„Jedes mag alles“

Anmerkungen

- Gleiche Variable in gleichem Fakt: Gleichheit
- Gleiche Variable in verschiedenen Fakten: Nichtgleichheit
  - somit für dieses Faktum mag (Z,W). gleichbedeutend
  - ebenso Frage „?-mag (alfred,X).“ gleichb. zu „?-mag(alfred,V)“
- Variable in Fragen existenzquantifiziert:

  „?-mag (alfred,X).“ „existiert etwas, das Alfred mag“
- Variablen in Fakten allquantifiziert:

  mag (alfred,X). „Alfred mag alles“

Regeln und abgeleitete Fakten

Fakten und Regeln

```
hat(frieda,buch).
mag (hans,fisch).
mag (hans,X)  :- mag (X,fisch).
hat (frieda,nocheinbuch).
mag (hans,X)  :-
        hat (X,buch), spielt (hans,X,tennis),feineswetter.
spielt (hans,frieda,tennis).
feineswetter.
nett(X)  :- mag (Y,X).
```

abgeleitet Fakten

?-nett(frieda).
ja

?- mag (hans,X).
X=fisch
Y=hans
X=frieda
nein

?-mag (hans,hans)
ja

Anmerkung:

- Variablen im Regelkopf (linke Seite von „:-“) allquantifiziert
- Variablen im Regelrumpf (rechte Seite von „:-“) existenzquantifiziert

Definitionen nach Sterling/Shapiro: Programm, Regel, Prädikat

- Ein **Logikprogramm** ist eine endliche Regelmenge.
- Statt Regel sagt man auch **Horn-Klausel** oder bei Eindeutigkeit auch einfach nur **Klausel.**
- Eine **Regel** hat die Form.

  $A <- B_1, B_2 , ... B_n.$ mit $n >= 0$

  A ist der **Regelkopf** und die B_i's sind der **Regelrumpf.**
  Eine Regel mit $n=0$ wird **Fakt** genannt. A und B_i's werden auch Ziele genannt.
- Ein **Ziel** hat die Form eines Prädikats (Prädikatenlogik 1. Stufe)

  $pred (t_1, t_2, ..., t_m)$ mit $m >=0$

  wobei $pred$ Prädikatnamen, m Stelligkeit und die tj Argumente.
  Prädikate beschreiben Objekte und Beziehungen zwischen diesen Objekten.
- Argumente für Prädikate sind **Terme**; induktiv definiert:
  - eine Konstante ist ein Term,
  - eine Variable ist ein Term,
  - sind $t_1, t_2, ..., t_k$ Terme und ist f ein Funktionssymbol, so ist auch $f (t_1, t_2, ..., t_k)$ ein Term.

Semantik von Regeln und Programmen

- **Konventionen:**
  - hans, h1 Konstante,
  - X, Haus Variable,
  - _ anonyme Variable,
  - mag Prädikatname, plus Funktionssymbol
- **Bedeutung von Regel A <- B1, B2 , ..., Bn:**
  - B1 & B2 & ... Bn - >A,
  - umgangssprachlich „Wenn B1 gilt und B2 gilt und ... und Bn gilt, dann gilt auch A“.
- **Prozedurale Interpretation einer Regel:**
  - „Um A zu beantworten, beantworte B1 und B2 und ... und Bn“.
  - Diese Interpretation ist die Grundlage von PROLOG–Systemen.
- **Eine Frage definiert ein Ziel oder eine Konjunktion von Zielen. Die Ausführung des Programms hat zu zeigen, ob alle Ziele erfüllt werden. Ein Ziel ist erfüllt, wenn:**
  - entweder ein „passendes“ Fakt existiert,
  - oder ein „passender“ Regelkopf existiert und jedes Ziel im Regelrumpf erfüllt wird.

Substitution

- Eine Substitution ist eine endliche Menge (ggfs. {}) von Paaren der Form $X=t$, wobei X eine Variable ist und t ein Term, so daß keine zwei Paare dieselbe Variable als linke Seite haben. Für einen Term t und seine Substitution $S = \{X_1 = t_1 , ... X_n = t_n\}$ bezeichnet tS das Ergebnis der simultanen Ersetzung aller Vorkommen aller X_i durch t_i $(1<=i<=n)$.
  tS heißt Instanz von t.
- Ein Ziel und ein Fakt (ein Regelkopf) „passen“, wenn sie eine gemeinsame Instanz besitzen.
- Eine Instanz heißt Grund-Instanz, wenn sie keine Variablen enthält.
- Die Berechnung eines PROLOG-Programms liefert, ausgehend von der Frage, die Substitutionen, für die alle Ziele, die zur Frage gehören, erfüllt sind.

Nichtdeterminismus, Reihenfolge

- **Idealerweise erfolgt die Evaluierung der Ziele auf der rechten Seite einer Regel als auch die Auswahl von passenden Fakten und Regelköpfen nichtdeterministisch. In einem realen System ist dies nicht der Fall, so daß also jeweils eine Reihenfolge festzulegen ist.**

- **PROLOG verwendet als Verarbeitungsstrategie die sogenannte SLD-Resolution (SLD für *S*elect a literal, using *L*inear startegy, restricted to *D*efinite clauses).**

  Dies bedeutet:

  - Ziele auf der rechten Seite einer Regel werden von links nach rechts ausgewertet.
  - Passende Fakten bzw. Regelköpfe werden in der vom Programmierer vorgegebenen Reihenfolge aufgesucht.

- **Die Folge aller Regeln (und damit auch Fakten), deren Köpfe dieselben Prädikatnamen und dieselbe Anzahl von Argumenten haben, wird auch Prozedur genannt. Die Reihenfolge innerhalb einer Prozedur ist für die Verarbeitung in PROLOG von erheblicher Bedeutung, während die Reihenfolge zwischen verschiedenen Prozeduren keine Rolle spielt.**

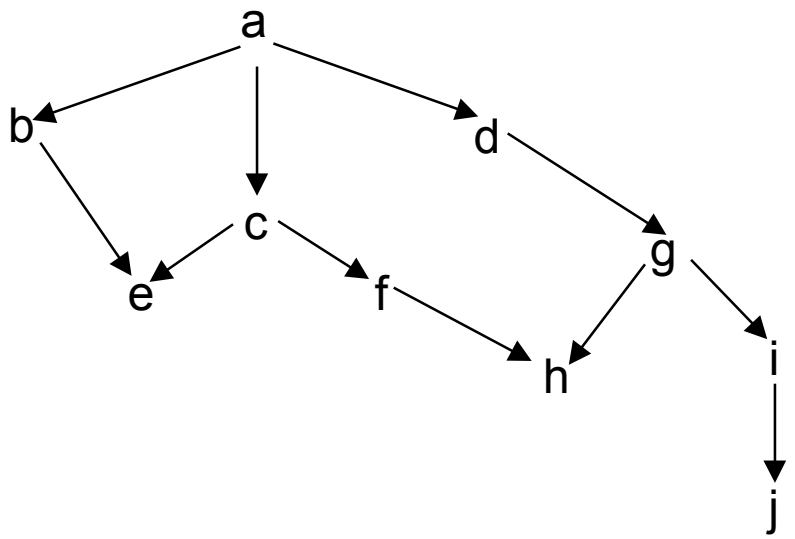
Beispiel

- **Gegeben sei ein gerichteter azyklischer Graph, dessen Knoten eindeutig benannt sind.**

  Gesucht sei ein PROLOG –Programm, das feststellt, ob zwischen zwei Knoten ein Weg existiert.

- **Lösung (mit möglicher Graphenbeschreibung):**

```
Weg (X,Y) :- kante (X,Y).
Weg (X,Z) :-kante (X,Y), weg (Y,Z).
kante (a,b).
kante (a,c).
kante (a,d).
kante (c,e).
kante (c,f).
kante (d,g).
kante (g,h).
kante (g,i).
kante (i,j).
kante (b,e).
kante (f,h).
```



- **Man beachte, daß die Vertauschung der beiden weg-Regeln zu einem fehlerhaften Programm führt.**

Datenstrukturen

- **Terme sind die einzigen Datenstrukturen in PROLOG. Terme induktiv definiert:**
  - Konstanten und Variablen sind Terme,
  - Ist f ein Funktionsymbol und sind t_1 , ..., t_n Terme, so ist auch $f(t_1, ..., t_n)$ ein Term.

- **Terme haben somit die Struktur eines Baums, bei dem die Blattknoten Konstanten oder Variablen sind und alle anderen Knoten Funktionssymbole.**

- **Terme in einem PROLOG-Programm können beliebig tief geschachtelt werden.**

Beispiel Datenstruktur: Terme und Fragen

```
buch('The Art of Prolog',author (sterling,shapiro)).
buch('Programming in Prolog',author(clocksin,mellish)).
buch('Foundations of LogicProgramming',author(lloyd)).

?-buch(X,Y).
X= 'The Art of Prolog' Y=author (sterling, shapiro);
X= 'Programming in Prolog' Y=author(clocksin,mellish);
X= 'Foundations of LogicProgramming' Y=author(lloyd);
nein

?-buch(X,author(Y,Z)).
X= 'The Art of Prolog' Y=sterling Z=shapiro;
X= 'Programming in Prolog' Y=clocksin Z=mellish;
nein
```

Unifikation

- **Nach Einführung von Termen mit Funktionssymbolen ist ein erweiterter Mechanismus zum Auffinden „passender“ Regelköpfe zu definieren.**
- **Ein Unifikator zweier Terme ist eine Substitution, die die beiden Terme identisch macht, also eine Instanz erzeugt, die Instanz beider Terme ist. In diesem Fall sagt man, daß die beiden Terme unifizierbar sind.**
- **Beispiel:**

p(X,	f(g(c,d),	Y),	Z,	W)
p(g(c,d),	f(X,	b),	b,	V)

haben als Unifikator die Substitution

X=g(c,d), Y=b, Z=b, W=V

Allgemeinster Unifikator

- **Der allgemeinste Unifikator für zwei Terme ist ein Unifikator, der die allgemeinste Instanz erzeugt. Term t_1 ist allgemeiner als Term t_2, wenn t_2 eine Instanz von t_1 ist.**
- **Wenn zwei Terme unifizierbar sind, dann existiert ein eindeutiger allgemeinster Unifikator, ggfs. in alphabetischen Varianten (= Umbenennung von Variablen).**
- **Beispiel:**

  p(X, f(g(c,d), Y), Z, W)
  p(g(c,d), f(X, b), b, V)

  haben als allgemeinster Unifikator die Substitution

  X=g(c,d), Y=b, Z=b, W=V,

  womit sich als gemeinsame Instanz

  p(g(c,d),f(g(c,d),b),b,V)

  bzw. die Namensvariante

  p(g(c,d),f(g(c,d),b),b,W) ergibt.

Unifikationsalgorithmus mit 'occurs check'

- **Unifikation beinhaltet den 'occurs check', d.h. Variable X wird mit Term t nur dann unifiziert, wenn X in t nicht vorkommt. Damit wird eine unendliche Unifikation verhindert. PROLOG-Systeme verzichten zwecks Effizienz meist auf die Überprüfung.**

- **Unifikationsalgorithmus : Schnittstelle**

  Eingabe: Zwei zu unifizierende Terme T_1, T_2

  Ausgabe: der allgemeinste Unifikator U oder Fehler

Unifikationsalgorithmus: Rumpf

Initialisiere die Substitution U als leer, den Stack mit Gleichung $T_1=T_2$ und Fehler auf `false`.

```
WHILE Stack nicht leer & nicht Fehler DO
      Nehme X=Y von Stack (POP)
      CASE
            X ist Variable und kommt nicht in Y vor:
                  Substituiere Y für X im Stack und in U,
                  füge X=Y in U hinzu
            Y ist Variable und kommt nicht in X vor:
                  Substituiere X für Y im Stack und in U,
                  füge Y=X hinzu
            X und Y sind identische Konstanten oder Variablen:
                  Weiter
            X ist f(X1 , ..., Xn) und Y ist f(Y1 , ..., Yn)
            mit irgendeinem Funktionssymbol und n > 0:
                  Lege Xi= Yi für i=1, .., n auf den Stack (PUSH)
            ELSE:
                  Fehler:=true
If Fehler THEN output fehler ELSE output U END
```

Was haben wir gelernt

- PROLOG als Beispiel einer logischen, deklarativen Programmiersprache (nicht-imperativ)
- PROLOG als interpretative Sprache mit Backtracking
- Programmstrukturierung in Fakten und Regeln, Programmausführung durch Frage für eine Lösung;
- weitere Lösungen explizit anfordern
- Aussagen über bestimmte Objekte: einfache Fakten,
  Allgemeine Aussagen: universelle Fakten,
  Folgerungen i.a. nicht für bestimmte Objekte: Regelköpfe und Rümpfe enthalten Variable
- PROLOG-Programmen liegt eine Spezialform der Prädikatenlogik zugrunde: Hornlogik (Logik mit Hornklauseln)
- SLD-Resolution und Abhängigkeit der Programmausführung von der Reihenfolge von Regeln und Fakten
- Eignung von PROLOG für „Problemlösungen mit Probieren“ (vgl. Wege in Graphen)
- Unifikation als Kern der Ableitung einer Lösung

Glossar

- deklarativ, logisch, nichtimperativ
- Objekte (Konstanten), Variable, null-, ein-, zwei-, ... –stellige Relationen (gerichtet)
- Einfache Fakten, universelle Fakten, abgeleitete Fakten
- Existenzquantifizierung, Allquantifizierung:
  Variablen in Fragen existenzquantifiziert,
  in Fakten allquantifiziert,
  Variablen in Regelkopf allquantifiziert, in Regelkopf existanzquantifiziert
- Konstante, Variable, Funktionssymbol, Term, Prädikat, Ziel, Regel (Hornklausel), Spezialfall Fakt, Prozedur
- Substitution, gemeinsame Instanz, Grundinstanz
- Unifikation, allgemeinster Unifikator, occurs check, Unifikationsalgorithmus

Teil V

Andere Paradigmen

- **Exkurs 1: PROLOG**
  Logikprogrammierung
- **Exkurs 2: LISP**
  Applikative Programmierung

Exkurs 1: PROLOG (Logikprogrammierung)

- **Allgemeines**
- **Grundkonzepte**
- **Konzepte und Terminologie**

PROLOG als deklarative Sprache

- **imperativ vs. deklarativ**
  - imperative Programme (Programmiersprachen) incl. funktional, objektorientiert:
    Programmierer gibt an, wie eine Lösung aussehen soll,
  - deklarative Programme (Programmiersprachen):
    Programmierer gibt an, was er haben will

- **PROLOG (Programmierung in Logic) 1972, Kowalski & Colmerauer**
  - Bedeutung für spezielle Probleme (KI)
  - Schub: 5th-Generation-Projekt (Japan): Ernüchterung
  - PROLOG, neben LISP und der Datenbankanfragesprache SQL wichtigste nichtimperative Sprache

Erläuterung und Literatur

- **Literatur**

  Sterling / Shapiro
  Clocksin / Mellish } Literaturverzeichnis

- **hier nur kurzer Abriß**

- **induktive Erläuterung**
  - Beispiele
  - Terminologie und allgemeine Prinzipien später

Fakten und einfache Fragen

- **Basisfakten (Aussagen über konkrete Objekte und Beziehungen)**

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.
```

Relation(enbezeichner), Objektnamen

null-, ein-, zwei-, und dreistellige Relationen

- **Fragen (normale Eingaben des Programms) Hinschreibung systemspezifisch (z.B.: „hat (Frieda,Buch)?“)**

```
?- hat(frieda,buch).
ja
```

- **Ausführung für einfaches Beispiel: Nachsehen in Faktenbasis; gibt es Fakt, Ausgabe ja, sonst nein**

Weitere einfache Fragen

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.
```

```
?- mag (hans,frieda).
ja
?-mag (frieda,hans).
nein
?-mag (hans,geld).
nein
?-liebt (hans,frieda).
nein
?- feineswetter
ja
```

Relationen sind gerichtet!

Fragen mit Variablen

- **Variable für Objekte: „Was mag Hans?“ anstelle von „Mag Hans Frieda?“**

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda,nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.

?- mag(hans,X).
X=fisch;
X=frieda;
nein
```

- **Ausführung (interaktiv)**
  - Suche nach einem „passenden“ Fakt
  - Wünscht Bediener Fortsetzung: Eingabe „ ; “
  - existiert kein weiteres Fakt: Ausgabe: nein
  - PROLOG-System sucht gemäß Auflistungsreihenfolge

Konjunktiv verknüpfte Fragen und Backtracking

- **„Existiert ein Objekt X, so daß Hans X mag und X ein Buch hat?“**

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.

?- mag(hans,X); hat(X,buch).
X=frieda;
nein
```

- **Backtracking: zwei Teilfragen sind passend zu beantworten**
  - erste Teilantwort Substitution α / fisch aus mag (hans,fisch)
  - Teilfrage 2 mit gleicher Substitution „?-hat (fisch,buch)“ Antwort negativ, Substitution wird aufgehoben
  - Suche nach neuer Substitution für erste Teilfrage: X / frieda Teilfrage 2 nun „?-mag(hans,frieda)“ erfolgreich
  - Eingabe „ ; “ weiteres Backtracking ohne Erfolg

Weitere Beispiele für Fragen und Backtracking

```
hat (frieda,buch).
mag (hans,fisch).
mag (hans,frieda).
hat (frieda, nocheinbuch).
mensch (hans).
mensch (frieda).
spielt (hans,frieda,tennis).
feineswetter.
```

```
?-mag (X,Y)
X=hans, Y=fisch;
X=hans, Y=frieda;
nein

?-mag (X,X).
nein
```

- **Fragen beziehen sich auf Existenz einer Lösung:**
  - Gefunden: in Ausgabe angegeben
  - Weitere Lösungen explizit anfordern

Universelle Fakten

- **Variablen in Fakten**

mag(alfred,X).	„Alfred mag alles“
mag(X,X).	„Jedes mag sich selbst“
mag(X,Y)	„Jedes mag alles“

- **Anmerkungen**
  - Gleiche Variable in gleichem Fakt: Gleichheit
  - Gleiche Variable in verschiedenen Fakten: Nichtgleichheit
    - somit für dieses Faktum mag (Z,W). gleichbedeutend
    - ebenso Frage „?-mag (alfred,X).“ gleichb. zu „?-mag(alfred,V)“
  - Variable in Fragen existenzquantifiziert:
    „?-mag (alfred,X).“ „existiert etwas, das Alfred mag“
  - Variablen in Fakten allquantifiziert:
    mag (alfred,X). „Alfred mag alles“

Regeln und abgeleitete Fakten

- **Fakten und Regeln**

```
hat(frieda,buch).
mag (hans,fisch).
mag (hans,X)  :- mag (X,fisch).
hat (frieda,nocheinbuch).
mag (hans,X)  :-
        hat (X,buch), spielt (hans,X,tennis),feineswetter.
spielt (hans,frieda,tennis).
feineswetter.
nett(X)  :- mag (Y,X).
```

abgeleitet Fakten

```
?-nett(frieda).
ja

?- mag (hans,X).
X=fisch
Y=hans
X=frieda
nein

?-mag (hans,hans)
ja
```

- **Anmerkung:**
  - Variablen im Regelkopf (linke Seite von „:-“) allquantifiziert
  - Variablen im Regelrumpf (rechte Seite von „:-“) existenzquantifiziert

Definitionen nach Sterling/Shapiro: Programm, Regel, Prädikat

- Ein **Logikprogramm** ist eine endliche Regelmenge.
- Statt Regel sagt man auch **Horn-Klausel** oder bei Eindeutigkeit auch einfach nur **Klausel.**
- Eine **Regel** hat die Form.

  $A <- B_1, B_2 , ... B_n.$ mit $n >= 0$

  A ist der **Regelkopf** und die B_i's sind der **Regelrumpf.**
  Eine Regel mit $n=0$ wird **Fakt** genannt. A und B_i's werden auch Ziele genannt.
- Ein **Ziel** hat die Form eines Prädikats (Prädikatenlogik 1. Stufe)

  $pred (t_1, t_2, ..., t_m)$ mit $m >=0$

  wobei $pred$ Prädikatnamen, m Stelligkeit und die tj Argumente.
  Prädikate beschreiben Objekte und Beziehungen zwischen diesen Objekten.
- Argumente für Prädikate sind **Terme**; induktiv definiert:
  - eine Konstante ist ein Term,
  - eine Variable ist ein Term,
  - sind $t_1, t_2, ..., t_k$ Terme und ist f ein Funktionssymbol, so ist auch $f (t_1, t_2, ..., t_k)$ ein Term.

Semantik von Regeln und Programmen

- **Konventionen:**
  - hans, h1 Konstante,
  - X, Haus Variable,
  - _ anonyme Variable,
  - mag Prädikatname, plus Funktionssymbol
- **Bedeutung von Regel A <- B1, B2 , ..., Bn:**
  - B1 & B2 & ... Bn - >A,
  - umgangssprachlich „Wenn B1 gilt und B2 gilt und ... und Bn gilt, dann gilt auch A“.
- **Prozedurale Interpretation einer Regel:**
  - „Um A zu beantworten, beantworte B1 und B2 und ... und Bn“.
  - Diese Interpretation ist die Grundlage von PROLOG–Systemen.
- **Eine Frage definiert ein Ziel oder eine Konjunktion von Zielen. Die Ausführung des Programms hat zu zeigen, ob alle Ziele erfüllt werden. Ein Ziel ist erfüllt, wenn:**
  - entweder ein „passendes“ Fakt existiert,
  - oder ein „passender“ Regelkopf existiert und jedes Ziel im Regelrumpf erfüllt wird.

Substitution

- **Eine Substitution ist eine endliche Menge (ggfs. {}) von Paaren der Form X=t, wobei X eine Variable ist und t ein Term, so daß keine zwei Paare dieselbe Variable als linke Seite haben. Für einen Term t und seine Substitution $S = \{X_1 = t_1 , ... X_n = t_n\}$ bezeichnet tS das Ergebnis der simultanen Ersetzung aller Vorkommen aller X_i durch t_i ($1 <= i <= n$). tS heißt Instanz von t.**
- **Ein Ziel und ein Fakt (ein Regelkopf) „passen“, wenn sie eine gemeinsame Instanz besitzen.**
- **Eine Instanz heißt Grund-Instanz, wenn sie keine Variablen enthält.**
- **Die Berechnung eines PROLOG-Programms liefert, ausgehend von der Frage, die Substitutionen, für die alle Ziele, die zur Frage gehören, erfüllt sind.**

Nichtdeterminismus, Reihenfolge

- **Idealerweise erfolgt die Evaluierung der Ziele auf der rechten Seite einer Regel als auch die Auswahl von passenden Fakten und Regelköpfen nichtdeterministisch. In einem realen System ist dies nicht der Fall, so daß also jeweils eine Reihenfolge festzulegen ist.**
- **PROLOG verwendet als Verarbeitungsstrategie die sogenannte SLD-Resolution (SLD für *S*elect a literal, using *L*inear startegy, restricted to *D*efinite clauses).**

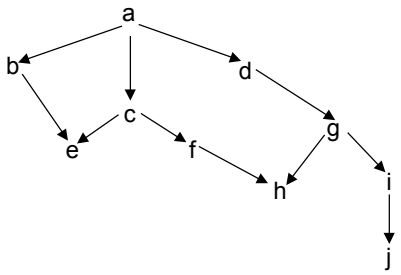
  Dies bedeutet:
  - Ziele auf der rechten Seite einer Regel werden von links nach rechts ausgewertet.
  - Passende Fakten bzw. Regelköpfe werden in der vom Programmierer vorgegebenen Reihenfolge aufgesucht.
- **Die Folge aller Regeln (und damit auch Fakten), deren Köpfe dieselben Prädikatnamen und dieselbe Anzahl von Argumenten haben, wird auch Prozedur genannt. Die Reihenfolge innerhalb einer Prozedur ist für die Verarbeitung in PROLOG von erheblicher Bedeutung, während die Reihenfolge zwischen verschiedenen Prozeduren keine Rolle spielt.**

Beispiel

- **Gegeben sei ein gerichteter azyklischer Graph, dessen Knoten eindeutig benannt sind.**

  Gesucht sei ein PROLOG –Programm, das feststellt, ob zwischen zwei Knoten ein Weg existiert.
- **Lösung (mit möglicher Graphenbeschreibung):**

```
Weg (X,Y) :- kante (X,Y).
Weg (X,Z) :-kante (X,Y), weg (Y,Z).
kante (a,b).
kante (a,c).
kante (a,d).
kante (c,e).
kante (c,f).
kante (d,g).
kante (g,h).
kante (g,i).
kante (i,j).
kante (b,e).
kante (f,h).
```



- **Man beachte, daß die Vertauschung der beiden weg-Regeln zu einem fehlerhaften Programm führt.**

Datenstrukturen

- **Terme sind die einzigen Datenstrukturen in PROLOG. Terme induktiv definiert:**
  - Konstanten und Variablen sind Terme,
  - Ist f ein Funktionsymbol und sind t_1 , ..., t_n Terme, so ist auch $f(t_1, ..., t_n)$ ein Term.
- **Terme haben somit die Struktur eines Baums, bei dem die Blattknoten Konstanten oder Variablen sind und alle anderen Knoten Funktionssymbole.**
- **Terme in einem PROLOG-Programm können beliebig tief geschachtelt werden.**

Beispiel Datenstruktur: Terme und Fragen

```
buch('The Art of Prolog',author (sterling,shapiro)).
buch('Programming in Prolog',author(clocksin,mellish)).
buch('Foundations of LogicProgramming',author(lloyd)).

?-buch(X,Y).
X= 'The Art of Prolog' Y=author (sterling, shapiro);
X= 'Programming in Prolog' Y=author(clocksin,mellish);
X= 'Foundations of LogicProgramming' Y=author(lloyd);
nein

?-buch(X,author(Y,Z)).
X= 'The Art of Prolog' Y=sterling Z=shapiro;
X= 'Programming in Prolog' Y=clocksin Z=mellish;
nein
```

Unifikation

- **Nach Einführung von Termen mit Funktionssymbolen ist ein erweiterter Mechanismus zum Auffinden „passender“ Regelköpfe zu definieren.**
- **Ein Unifikator zweier Terme ist eine Substitution, die die beiden Terme identisch macht, also eine Instanz erzeugt, die Instanz beider Terme ist. In diesem Fall sagt man, daß die beiden Terme unifizierbar sind.**
- **Beispiel:**

  p(X, f(g(c,d), Y), Z, W)
  p(g(c,d), f(X, b), b, V)

  haben als Unifikator die Substitution

  X=g(c,d), Y=b, Z=b, W=V

Allgemeinster Unifikator

- **Der allgemeinste Unifikator für zwei Terme ist ein Unifikator, der die allgemeinste Instanz erzeugt. Term t_1 ist allgemeiner als Term t_2, wenn t_2 eine Instanz von t_1 ist.**
- **Wenn zwei Terme unifizierbar sind, dann existiert ein eindeutiger allgemeinster Unifikator, ggfs. in alphabetischen Varianten (= Umbenennung von Variablen).**
- **Beispiel:**

  p(X, f(g(c,d), Y), Z, W)
  p(g(c,d), f(X, b), b, V)

  haben als allgemeinster Unifikator die Substitution

  X=g(c,d), Y=b, Z=b, W=V,

  womit sich als gemeinsame Instanz

  p(g(c,d),f(g(c,d),b),b,V)

  bzw. die Namensvariante

  p(g(c,d),f(g(c,d),b),b,W) ergibt.

Unifikationsalgorithmus mit 'occurs check'

- **Unifikation beinhaltet den 'occurs check', d.h. Variable X wird mit Term t nur dann unifiziert, wenn X in t nicht vorkommt. Damit wird eine unendliche Unifikation verhindert. PROLOG-Systeme verzichten zwecks Effizienz meist auf die Überprüfung.**
- **Unifikationsalgorithmus : Schnittstelle**

  Eingabe: Zwei zu unifizierende Terme T_1, T_2

  Ausgabe: der allgemeinste Unifikator U oder Fehler

Unifikationsalgorithmus: Rumpf

Initialisiere die Substitution U als leer, den Stack mit Gleichung $T_1=T_2$ und Fehler auf `false`.

```
WHILE Stack nicht leer & nicht Fehler DO
      Nehme X=Y von Stack (POP)
      CASE
            X ist Variable und kommt nicht in Y vor:
                  Substituiere Y für X im Stack und in U,
                  füge X=Y in U hinzu
            Y ist Variable und kommt nicht in X vor:
                  Substituiere X für Y im Stack und in U,
                  füge Y=X hinzu
            X und Y sind identische Konstanten oder Variablen:
                  Weiter
            X ist f(X1 , ..., Xn) und Y ist f(Y1 , ..., Yn)
            mit irgendeinem Funktionssymbol und n > 0:
                  Lege Xi= Yi für i=1, .., n auf den Stack (PUSH)
            ELSE:
                  Fehler:=true
If Fehler THEN output fehler ELSE output U END
```

Was haben wir gelernt

- PROLOG als Beispiel einer logischen, deklarativen Programmiersprache (nicht-imperativ)
- PROLOG als interpretative Sprache mit Backtracking
- Programmstrukturierung in Fakten und Regeln, Programmausführung durch Frage für eine Lösung;
- weitere Lösungen explizit anfordern
- Aussagen über bestimmte Objekte: einfache Fakten,
  Allgemeine Aussagen: universelle Fakten,
  Folgerungen i.a. nicht für bestimmte Objekte: Regelköpfe und Rümpfe enthalten Variable
- PROLOG-Programmen liegt eine Spezialform der Prädikatenlogik zugrunde: Hornlogik (Logik mit Hornklauseln)
- SLD-Resolution und Abhängigkeit der Programmausführung von der Reihenfolge von Regeln und Fakten
- Eignung von PROLOG für „Problemlösungen mit Probieren“ (vgl. Wege in Graphen)
- Unifikation als Kern der Ableitung einer Lösung

Glossar

- deklarativ, logisch, nichtimperativ
- Objekte (Konstanten), Variable, null-, ein-, zwei-, ... –stellige Relationen (gerichtet)
- Einfache Fakten, universelle Fakten, abgeleitete Fakten
- Existenzquantifizierung, Allquantifizierung:
  Variablen in Fragen existenzquantifiziert,
  in Fakten allquantifiziert,
  Variablen in Regelkopf allquantifiziert, in Regelkopf existanzquantifiziert
- Konstante, Variable, Funktionssymbol, Term, Prädikat, Ziel, Regel (Hornklausel), Spezialfall Fakt, Prozedur
- Substitution, gemeinsame Instanz, Grundinstanz
- Unifikation, allgemeinster Unifikator, occurs check, Unifikationsalgorithmus

Teil VI

Zusammenfassung

- **Rückblick, Ausblick**

Rückblick, Ausblick

- Zusammenfassung Inhalt
- Vorlesungsziele - Erfüllung
- Weitere Veranstaltungen
- Vorbereitung

Teil I - Einführung

- **Übersicht**
  - Inhalt, Ziele
  - Orientierung
  - Literatur
- **Informatik-Grundlagen**
  - Klärung "Informatik"
  - Geschichte der Informatik
  - Algorithmus
  - Software, Programm, Programmentwicklung
  - Von-Neumann-Rechner
- **Programmiersprachen-Grundlagen**
  - Syntax (und Semantik)
  - Grammatik-Notationen
  - Programmiersprachen: Allgemeines
  - Warum arbeiten wir mit Modula-3
- **Erste Programmbeispiele**
  - Vorschau: Deklarationen, Anweisungen, Ausdrücke, Datentypen
  - Vordefinierte Datentypen
  - Beispielprogramm aus dem Vorkurs

- **Was machen wir in der Vorlesung?**
- **Wie ordnet sich das ein?**
- **Programmiersprache: Grundbegriffe**
- **Aufsatzpunkt Vorkurs**

Teil II – Programmiersprachenkonstrukte und Programmierstile

- **"Funktionale" Programmierung**
  - Funktionen, Parameter
  - Vernetzung von Funktionen
  - Bedingungen in funktionalen Programmen
  - Rekursion
- **Imperative Programmierung**
  - Konzepte der imperativen Programmierung
  - Variable und Wertzuweisungen
  - Prozeduren
  - rekursive Prozeduren
  - Parameterübergabe
  - Gültigkeitsbereich und Lebensdauer
- **Kontrollstrukturen**
  - Ablaufkontrolle
  - Fallunterscheidungen
  - Wiederholungsanweisungen
- **Datentypen I (statisch)**
  - Datentypen: Allgemeines
  - Skalare benutzerdefinierte Datentypen
  - Zusammengesetzte benutzerdefinierte Datentypen
- **Datentypen II (dynamisch)**
  - Dynamische Datentypen
  - Anwendungsbeispiele
  - Prozedurtyp
- **Zusammenfassung**
  - Ablaufkontrolle
  - Datenstrukturierung
  - Entsprechung Kontroll- und Datenstrukturen
  - Strenge Typisierung
  - Typäquivalenz

- **Funktionalle vs. imperative Programmierung**
- **Programmieren-im-Kleinen - Konstrukte imperativer Sprachen**
- **Zusammenhänge, Details**
- **Voraussetzung für kleine Programme/Innenleben von Bausteinen**

Teil III - Methodisches Programmieren im Kleinen

- **Allgemeines und Beispiel 1**
  - Entwickeln
  - Verbessern
  - Effizienzbetrachtung
  - Dokumentieren
  - Test
- **Beispiel 2**
  - Partielle Korrektheit
  - Termination
- **Beispiel 3**
  - Entscheidungstabellen
  - Programmcode für ETn

- **Entwicklung mit ...**
- **Verbesserung, Effizienz**
  ($\rightarrow$ Datenstruktur-Vorlesung)
- **Prüfung durch Test und Verifikation**
- **Spezialprobleme: andere Vorgehensweise (ET)**

- **Modularisierung und Module Modulkonzept**
  - Modulrumpf
  - Austausch der Implementierung
  - Diskussion modularer Programme
- **vordefinierte Bausteine: Beispiel Dateien**
  - Dateisystem
  - Operationen auf Dateien
  - Dateien in Modula-3
- **Datenabstraktion**
  - Prozeß- und Datenabstraktion
  - Objektmodule (Datenkapselung)
  - Abstrakte Datentypen
- **Vertragsmodell**
  - Konzept des Vertragsmodells
  - Zusicherungen
  - Realisierung von Zusicherungen mit Pragmas
  - Exkurs Ausnahmebehandlung
  - Zusicherungen mittels Ausnahmebehandlung
- **Objektorientierte Programmierung I**
  - Verständnis der objektorientierten Programmierung
  - Objekte
  - Klassen
  - Vererbung
  - Polymorphismus und dynamisches Binden
  - Diskussion
- **Objektorientierte Programmierung II: OO in Modula-3**
  - Objekttypen für Klassen
  - Untertypen für opake Klassen
  - Vererbung zwischen opaken Klassen
  - Diskussion
- **Beispiel einer Programmentwicklung**
  - Aufgabenstellung
  - Prozedurale Lösung
  - Objekt- und klassenbasierte Lösung
  - Lösung mit Objektorientierung

- **Modularisierung: PS-Ebene**
- **Datenabstraktion Vertragsmodelle: Methodikebene**
- **Objektorientierung: neue Gedankenwelt Klassifikation, Ähnlichkeiten gemeinsame Handhabung verschiedenartiger Objekte**
- **Beispiel: Was bringt das Ganze?**

Zusammen-
fassung

Teil IV – Andere Paradigmen

- **Exkurs 1: PROLOG (Logikprogrammierung)**
  - Allgemeines
  - Grundkonzepte
  - Konzepte und Terminologie
- **Exkurs 2: LISP (Applikative Programmierung)**
  - Allgemeines
  - Listen für Daten und Programme
  - Wertzuweisungen, -ermittlung, -interpretation
  - Listenverarbeitung
  - Systemfunktionen Ausdrücke
  - Funktionen und Rekursion
  - Abrundung

- **Völlig andere gedankliche Hilfsmittel und Ergebnisse**
- **Logisch, deklarativ: PROLOG**
- **Wertorientiert, applikativ: LISP**
- **Spezielle Probleme: Probieren, Backtracking: PROLOG**
  KI, selbstmodifizierende Programme: LISP

Was ist Programmierung?

- **Bandbreite der Programmierung**

  In Formalismus: endlicher Automat, Turing-Maschine

  imperativ / prozedural:

  Objekt-, klassenbasiert, Objektorientiert

  } **klassische Welt neuer Ausprägung**

  logisch deklarativ: PROLOG

  applikativ: LISP

  }
  - **jede Lösung sieht strukturell anders aus**
  - **für jedes Problem das richtige Arbeitsmittel**

- **Programmierung: Programme und Maschinen**
  - **Modula 3-Programme und Modula 3- Maschine**
  - **Prolog-Programme und Prolog-Maschine**
  - **Lisp-Programme und Lisp-Maschine**
  - **Eiffel-Programme und Eiffel-Maschine (Kap. OO I)**

Saubere Programmierung: Systematik

- Teil III der Vorlesung für klassische / moderne Welt hybrider Sprachen auf Pik-Ebene
  Teil IV Beispiel für Gesamtstrukturebenen
- Entwicklung mit:
  - schrittweiser Verfeinerung
  - Bezeichnerwahl
  - Kommentar
  - Bedienungsschnittstellengestaltung
- Verbesserung
- Prüfung
- Dokumentation
- Klare Gesamtstruktur
- Elementarste Standardkenntnisse: Suchen/Sortieren, Datenstrukturen
  (→ Vorlesung Datenstrukturen)

Was sind Programmiersprachen?

- Klassische Welt statisch und streng typisierte Sprachen
  Kontrollstrukturen, Datentypenkonstruktoren, Prozeduren → Pascal-Teil von Modula-3, vgl. Teil III
- Klassische Welt in moderner Ausprägung (hybride Sprachen im Gegensatz zu reinen OO-Programmiersprachen
  Module, Verwendung von Modulen für funktionalle Abstraktion und Datenabstraktion → am Beispiel Modula 3
  Strukturierung der Gesamtstruktur: Architektur (Teil IV)
- Reine OO-Welt Eiffel (Kap. OO I): nur angerissen in Teil IV
- Übertragung der Erkenntnisse in weit genutzte Sprachen C, C++, COBOL, FORTRAN
- Interpreterorientierte Sprachen, wie PROLOG, LISP
- Insgesamt 4 (völlig) verschiedene Sprachfamilien

Saubere Begriffe über Programmiersprache

- **Syntax, Semantik, Pragmatik**
- **Ebenen der Syntaxbeschreibung**
- **Syntaxnotationen**
- **Konzepte**
  - Kontrollstrukturen, Datentypkonstruktoren, Ausdrücke, Gesamtstruktur in klassischen Sprachen
  - Atome, Terme, ... in logischen Sprachen
  - Atome, Listen, Funktionen in applikativen Sprachen

Abstraktion

- Wir sprechen über Programme in einer Sprache und nicht über ein konkretes Programm (vgl. Teil III, Teil IV)
- Wir sprechen über verschiedene Grobstrukturen zu einem Problem (Bsp. Teil IV)
- Wir sprechen über Konstrukte einer Art von Sprachen und nicht nur über die konkrete Syntax einer Sprache
- Wir sprechen über verschiedene Klassen von Programmiersprachen
- Wir sprechen über Programmierung und verlassen dabei die Programmiersprachen: endlicher Automat, Turing Maschine
  (→ Vorlesung Berechenbarkeit und Komplexität
  - Die Vorlesung und auch das ganze Studium handelt von Abstraktion, Mechanismen, Systemen, die im Idealfall formalisierbar sind
  - dies ist keine Fortsetzung der Pascal- oder Basic-Programmierung der Schule
  - Verstehen steht im Vordergrund, nicht handwerkliches Können
  - Jede Sprache erschließt sich aber nur durch Übung

Grundstudium

- Datenstrukturen
- Rechnerstrukturen
- Systemprogrammierung
- Berechenbarkeit / Komplexität
- Automatentheorie/Formale Sprachen
- Proseminar
- Softwarepraktikum

Hauptstudium

- vergl. Darstellung höherer Programmiersprachen (insbesondere Ada 95)
- effiziente Algorithmen
- theor. Konzepte für Programmierung / Programmiersprachen
- andere Programmiersprachen (Lisp, Smalltalk, Prolog, Ada)
- Softwaretechnik
- nebenläufige Programmierung / hardwarenahe Programmierung
- Semantik / Verifikation
- Compilerbau
- Datenbanksysteme
- Betriebssysteme
- Computergrafik
- Mustererkennung
- Hochleistungsrechnen
- Kommunikation / verteilte Systeme
- Praktika

und beliebige Anwendungsprogrammierung

Scheinklausur: Programmierung

Erstklausur 21.02. wie Wiederholungsklausur nach Pfingsten

- 3-std. 6 Aufgaben
- Nachbereiten Vorlesung
- Durcharbeiten Übungen
- Einüben der Prüfungssituation,
  Zeitablauf, keine Unterlagen
- etwa Hälfte DA, OO (incl. PiK) mit Verwendung solcher Bausteine
- Grundlagen: Syntax, Semantik
- Spezifika von Modula-3
- ein oder zwei Aufgaben PROLOG, LISP, Verifikation, funktionale Programmierung

Ganze Vorlesung kommt vor!

Vordiploms-Klausur/Wünsche

voraussichtlich 07.09. bzw. Wiederholung Frühjahr 2001

- zwei Halbklausuren, eine davon Programmierung
- Stoffvolumen, Schwierigkeitsgrad etwa wie Scheinklausur, abgestimmt auf die Klausurdauer
- Ratschläge wie oben

Viel Erfolg!

Vielleicht sehen wir uns wieder beim Proseminar, Softwarepraktikum

Sicher im Hauptstudium!

Teil VI

Zusammenfassung

- Rückblick, Ausblick

Rückblick, Ausblick

- Zusammenfassung Inhalt
- Vorlesungsziele - Erfüllung
- Weitere Veranstaltungen
- Vorbereitung

Zusammen-fassung

Teil I - Einführung

- **Übersicht**
  - Inhalt, Ziele
  - Orientierung
  - Literatur
- **Informatik-Grundlagen**
  - Klärung "Informatik"
  - Geschichte der Informatik
  - Algorithmus
  - Software, Programm, Programmentwicklung
  - Von-Neumann-Rechner
- **Programmiersprachen-Grundlagen**
  - Syntax (und Semantik)
  - Grammatik-Notationen
  - Programmiersprachen: Allgemeines
  - Warum arbeiten wir mit Modula-3
- **Erste Programmbeispiele**
  - Vorschau: Deklarationen, Anweisungen, Ausdrücke, Datentypen
  - Vordefinierte Datentypen
  - Beispielprogramm aus dem Vorkurs

- **Was machen wir in der Vorlesung?**
- **Wie ordnet sich das ein?**
- **Programmiersprache: Grundbegriffe**
- **Aufsatzpunkt Vorkurs**

Zusammen-fassung

Teil II – Programmiersprachenkonstrukte und Programmierstile

- **"Funktionale" Programmierung**
  - Funktionen, Parameter
  - Vernetzung von Funktionen
  - Bedingungen in funktionalen Programmen
  - Rekursion
- **Imperative Programmierung**
  - Konzepte der imperativen Programmierung
  - Variable und Wertzuweisungen
  - Prozeduren
  - rekursive Prozeduren
  - Parameterübergabe
  - Gültigkeitsbereich und Lebensdauer
- **Kontrollstrukturen**
  - Ablaufkontrolle
  - Fallunterscheidungen
  - Wiederholungsanweisungen
- **Datentypen I (statisch)**
  - Datentypen: Allgemeines
  - Skalare benutzerdefinierte Datentypen
  - Zusammengesetzte benutzerdefinierte Datentypen
- **Datentypen II (dynamisch)**
  - Dynamische Datentypen
  - Anwendungsbeispiele
  - Prozedurtyp
- **Zusammenfassung**
  - Ablaufkontrolle
  - Datenstrukturierung
  - Entsprechung Kontroll- und Datenstrukturen
  - Strenge Typisierung
  - Typäquivalenz

- **Funktionalle vs. imperative Programmierung**
- **Programmieren-im-Kleinen - Konstrukte imperativer Sprachen**
- **Zusammenhänge, Details**
- **Voraussetzung für kleine Programme/Innenleben von Bausteinen**

Zusammen-fassung

Teil III - Methodisches Programmieren im Kleinen

- **Allgemeines und Beispiel 1**
  - Entwickeln
  - Verbessern
  - Effizienzbetrachtung
  - Dokumentieren
  - Test
- **Beispiel 2**
  - Partielle Korrektheit
  - Termination
- **Beispiel 3**
  - Entscheidungstabellen
  - Programmcode für ETn

- **Entwicklung mit ...**
- **Verbesserung, Effizienz (→ Datenstruktur-Vorlesung)**
- **Prüfung durch Test und Verifikation**
- **Spezialprobleme: andere Vorgehensweise (ET)**

Zusammen-fassung

Teil IV - Programmstrukturierung

- **Modularisierung und Module Modulkonzept**
  - Modulrumpf
  - Austausch der Implementierung
  - Diskussion modularer Programme
- **vordefinierte Bausteine: Beispiel Dateien**
  - Dateisystem
  - Operationen auf Dateien
  - Dateien in Modula-3
- **Datenabstraktion**
  - Prozeß- und Datenabstraktion
  - Objektmodule (Datenkapselung)
  - Abstrakte Datentypen
- **Vertragsmodell**
  - Konzept des Vertragsmodells
  - Zusicherungen
  - Realisierung von Zusicherungen mit Pragmas
  - Exkurs Ausnahmebehandlung
  - Zusicherungen mittels Ausnahmebehandlung
- **Objektorientierte Programmierung I**
  - Verständnis der objektorientierten Programmierung
  - Objekte
  - Klassen
  - Vererbung
  - Polymorphismus und dynamisches Binden
  - Diskussion
- **Objektorientierte Programmierung II: OO in Modula-3**
  - Objekttypen für Klassen
  - Untertypen für opake Klassen
  - Vererbung zwischen opaken Klassen
  - Diskussion
- **Beispiel einer Programmentwicklung**
  - Aufgabenstellung
  - Prozedurale Lösung
  - Objekt- und klassenbasierte Lösung
  - Lösung mit Objektorientierung
- **Modularisierung: PS-Ebene**
- **Datenabstraktion Vertragsmodelle: Methodikebene**
- **Objektorientierung: neue Gedankenwelt Klassifikation, Ähnlichkeiten gemeinsame Handhabung verschiedenartiger Objekte**
- **Beispiel: Was bringt das Ganze?**

Zusammen-fassung

Teil IV – Andere Paradigmen

- **Exkurs 1: PROLOG (Logikprogrammierung)**
  - Allgemeines
  - Grundkonzepte
  - Konzepte und Terminologie
- **Exkurs 2: LISP (Applikative Programmierung)**
  - Allgemeines
  - Listen für Daten und Programme
  - Wertzuweisungen, -ermittlung, -interpretation
  - Listenverarbeitung
  - Systemfunktionen Ausdrücke
  - Funktionen und Rekursion
  - Abrundung

- **Völlig andere gedankliche Hilfsmittel und Ergebnisse**
- **Logisch, deklarativ: PROLOG**
- **Wertorientiert, applikativ: LISP**
- **Spezielle Probleme: Probieren, Backtracking: PROLOG**
  KI, selbstmodifizierende Programme: LISP

Vorlesungsziele, Erfüllung

Was ist Programmierung?

- **Bandbreite der Programmierung**

  In Formalismus: endlicher Automat, Turing-Maschine

  imperativ / prozedural:

  Objekt-, klassenbasiert, Objektorientiert

  } **klassische Welt neuer Ausprägung**

  logisch deklarativ: PROLOG

  applikativ: LISP

  } **jede Lösung sieht strukturell anders aus**
  für jedes Problem das richtige Arbeitsmittel

- **Programmierung: Programme und Maschinen**
  - **Modula 3-Programme und Modula 3- Maschine**
  - **Prolog-Programme und Prolog-Maschine**
  - **Lisp-Programme und Lisp-Maschine**
  - **Eiffel-Programme und Eiffel-Maschine (Kap. OO I)**

Saubere Programmierung: Systematik

- Teil III der Vorlesung für klassische / moderne Welt hybrider Sprachen
  auf Pik-Ebene
  Teil IV Beispiel für Gesamtstrukturebenen
- Entwicklung mit:
  - schrittweiser Verfeinerung
  - Bezeichnerwahl
  - Kommentar
  - Bedienungsschnittstellengestaltung
- Verbesserung
- Prüfung
- Dokumentation
- Klare Gesamtstruktur
- Elementarste Standardkenntnisse: Suchen/Sortieren, Datenstrukturen
  (→ Vorlesung Datenstrukturen)

Was sind Programmiersprachen?

- Klassische Welt statisch und streng typisierte Sprachen
  Kontrollstrukturen, Datentypenkonstruktoren, Prozeduren → Pascal-
  Teil von Modula-3, vgl. Teil III
- Klassische Welt in moderner Ausprägung (hybride Sprachen im
  Gegensatz zu reinen OO-Programmiersprachen
  Module, Verwendung von Modulen für funktionalle Abstraktion und
  Datenabstraktion → am Beispiel Modula 3
  Strukturierung der Gesamtstruktur: Architektur (Teil IV)
- Reine OO-Welt Eiffel (Kap. OO I): nur angerissen in Teil IV
- Übertragung der Erkenntnisse in weit genutzte Sprachen C, C++,
  COBOL, FORTRAN
- Interpreterorientierte Sprachen, wie PROLOG, LISP
- Insgesamt 4 (völlig) verschiedene Sprachfamilien

Saubere Begriffe über Programmiersprache

- **Syntax, Semantik, Pragmatik**
- **Ebenen der Syntaxbeschreibung**
- **Syntaxnotationen**
- **Konzepte**
  - Kontrollstrukturen, Datentypkonstruktoren, Ausdrücke, Gesamtstruktur in klassischen Sprachen
  - Atome, Terme, ... in logischen Sprachen
  - Atome, Listen, Funktionen in applikativen Sprachen

Abstraktion

- **Wir sprechen über Programme in einer Sprache und nicht über ein konkretes Programm (vgl. Teil III, Teil IV)**
- **Wir sprechen über verschiedene Grobstrukturen zu einem Problem (Bsp. Teil IV)**
- **Wir sprechen über Konstrukte einer Art von Sprachen und nicht nur über die konkrete Syntax einer Sprache**
- **Wir sprechen über verschiedene Klassen von Programmiersprachen**
- **Wir sprechen über Programmierung und verlassen dabei die Programmiersprachen: endlicher Automat, Turing Maschine (→ Vorlesung Berechenbarkeit und Komplexität**
  - **Die Vorlesung und auch das ganze Studium handelt von Abstraktion, Mechanismen, Systemen, die im Idealfall formalisierbar sind**
  - **dies ist keine Fortsetzung der Pascal- oder Basic-Programmierung der Schule**
  - **Verstehen steht im Vordergrund, nicht handwerkliches Können**
  - **Jede Sprache erschließt sich aber nur durch Übung**

Grundstudium

- Datenstrukturen
- Rechnerstrukturen
- Systemprogrammierung
- Berechenbarkeit / Komplexität
- Automatentheorie/Formale Sprachen
- Proseminar
- Softwarepraktikum

Hauptstudium

- vergl. Darstellung höherer Programmiersprachen (insbesondere Ada 95)
- effiziente Algorithmen
- theor. Konzepte für Programmierung / Programmiersprachen
- andere Programmiersprachen (Lisp, Smalltalk, Prolog, Ada)
- Softwaretechnik
- nebenläufige Programmierung / hardwarenahe Programmierung
- Semantik / Verifikation
- Compilerbau
- Datenbanksysteme
- Betriebssysteme
- Computergrafik
- Mustererkennung
- Hochleistungsrechnen
- Kommunikation / verteilte Systeme
- Praktika

und beliebige Anwendungsprogrammierung

Vorbereitung auf Klausuren

Scheinklausur: Programmierung

Erstklausur 21.02. wie Wiederholungsklausur nach Pfingsten

- 3-std. 6 Aufgaben
- Nachbereiten Vorlesung
- Durcharbeiten Übungen
- Einüben der Prüfungssituation, Zeitablauf, keine Unterlagen
- etwa Hälfte DA, OO (incl. PiK) mit Verwendung solcher Bausteine
- Grundlagen: Syntax, Semantik
- Spezifika von Modula-3
- ein oder zwei Aufgaben PROLOG, LISP, Verifikation, funktionale Programmierung

Ganze Vorlesung kommt vor!

Vorbereitung auf Klausuren

Vordiploms-Klausur/Wünsche

voraussichtlich 07.09. bzw. Wiederholung Frühjahr 2001

- zwei Halbklausuren, eine davon Programmierung
- Stoffvolumen, Schwierigkeitsgrad etwa wie Scheinklausur, abgestimmt auf die Klausurdauer
- Ratschläge wie oben

Viel Erfolg!

Vielleicht sehen wir uns wieder beim Proseminar, Softwarepraktikum

Sicher im Hauptstudium!