4.5. Datentypen

* bisher: einfache DO, vor allem Standardtypen;
d.h. Wertemengen für mögliche DO, die i. allg. einfach auf Rechenanlage realisierbar;

Zahlen	logische Werte	Zeichen
real	boolean	char
integer		

Für alle sind zulässige Grundoperationen die
(1) Zuweisung „ : = “ und die
(2) Gleichheitsprüfung „ = “ .

In der Regel existieren noch Typtransformationsfunktionen!

* strukturierte Datentypen:

jeweils charakterisiert durch 3 Dinge:
(I) Wertemenge (strukturbildende Regeln);
(II) Zugriff auf Komponenten (Selektion);
(III) (zulässige) Operationen;

4.5.1 Der Feldtyp

{in der Regel als ARRAY: Pascal, Oberon, Modula-2}

* Motivation → 2 Beispiele

(i) Menge reeller Meßwerte $a_1,...,a_n$ kann als Vektor $x = (a_1,a_2,...,a_n)$ verstanden werden;

(ii) 5 Hersteller, 8 Verbraucher; jew. Entfernung in km;
→ diese Daten in übersichtlicher Form als Matrix

$$M = \begin{pmatrix} e_{11} & e_{12} & ... & e_{18} \\ e_{21} & e_{22} & ... & e_{28} \\ ... & ... & ... & ... \\ e_{51} & e_{52} & ... & e_{58} \end{pmatrix}$$

mit e_{ij} – Anzahl der km von Hersteller i zu Verbraucher j,
$i = 1(1)5$; $j = 1(1)8$;

Datenmengen mit folgenden Eigenschaften:

(1) bestehen aus Komponenten/Elementen gleichen Typs (sog. Grund- oder Basistyp);
(2) unter einem Namen zusammengefaßt (Vektor *x*, Matrix M);
(3) Anzahl der Elemente beliebig, aber fest vorgegeben;
(4) über gemeinsamen Namen und entspr. Index ist jedes Element direkt bezeichnet und somit auch ansprechbar (a_1, a_2; e_{25}, e_{43});

Datentyp mit diesen Eigenschaften → Feldtyp oder ARRAY-Typ;

Vektor: 1-dim. Feld; Matrix: 2-dim. Feld; allg: n-dim. Feld;

* formale Beschreibung:

Sei A - beliebige Menge von Objekten (Basistyp);

$$A^N = \underbrace{A \times ... \times A}_{N-fach}$$

$x \in A^N$: $x = (a_1, a_2, ..., a_n)$ mit $a_i \in A$, $i = 1(1)N$.

Def 4.1: Jedes $x \in A$ heißt Feld oder Sequenz von N Elementen aus A.

Bem: Die Bildung des Kreuzproduktes ist eine Strukturierungsoperation. Dabei wird *x* als Vektor interpretiert.

Operationen:

Die wesentlich angebotene Operation ist die Selektion →

Zu belieb. Stuktur A^N existieren N Operatoren [*i*] mit der Eig.
$[i] : A^N \rightarrow A$, $i = 1(1)N$.

Definiert durch: $\boldsymbol{x[i] = a_i}$ für $x = (a_1, a_2, ..., a_n)$ mit $a_i \in A$, $i = 1(1)N$.

[*i*] ist ein sogenannter Postfix-Operator; er wählt die *i*-te Komponente.
⇒ $x = (x[1], x[2], ..., x[N])$;

i heißt **Indexausdruck**; $x[i]$ heißt **indizierte Variable**;

* in Pascal

I sei ein Indextyp; lin. geordn. Menge von Elementen;
T sei ein beliebiger Typ, der Basistyp;

Typedeklaration: TYPE *feldtyp* = ARRAY[I] of T;

Variablendeklaration: VAR *f* : *feldtyp*;
oder: VAR *f* : ARRAY[I] of T;

⇒ *f* – eine Folge von Elementen aus T; ihre Länge ist durch die Kardinalität von I festgelegt;

Bem.:

- Indextyp I kann ein einfacher Typ sein, ausgenommen REAL;
- Wert des Indexausdruckes *i* muß in I sein;
- Operationen:
  - (i) mit indiz. Var. alle Operationen, die auf Basistyp anwendbar;
  - (ii) Feld als Einheit: unter best. Bedingungen Wertzuweisung;

für Turbo Pascal 6.0, 7.0 gilt:

```
Bsp.1:    type feld = array[1..5] of integer;
          var  x : feld;
               y : feld;
          → Zuweisung y := x ist möglich;

Bsp.2:    type feld = array[1..5] of integer;
          var  x : array[1..5] of integer;
               y : feld;
          → Zuweisung y := x ist nicht möglich;
          (Typen sind für Compiler nicht vereinbar)

Bsp.3:    var  x : array[1..5] of integer;
               y : array[1..5] of integer;
          → Zuweisung y := x ist nicht möglich;
          (Typen sind für Compiler nicht vereinbar)

Bsp.4:    var  x, y : array[1..5] of integer;
          → Zuweisung y := x ist möglich;
```

Bsp. 1)

VAR *w* : ARRAY[1..10] OF REAL;
VAR *m* : ARRAY[1..5, 1..8] OF INTEGER;

Bsp. 2)

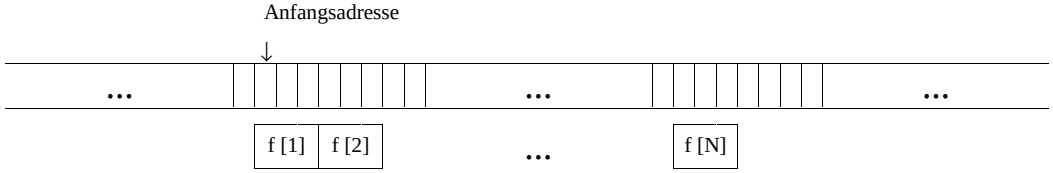
TYPE *zeile* = ARRAY[1..60] OF CHAR;
TYPE *seite* = ARRAY[1..25] OF *zeile*; } Vorteil: Möglichkeit Typen
oder } mehrfach zu verwenden
→
TYPE *seite* = ARRAY[1..60, 1..25] OF CHAR; } ganze Zeile ist ansprechbar
VAR *z* : *zeile*; *s* : *seite*;

wenn noch VAR *i, j* : INTEGER; so bezeichnet

z[*i*] eine Variable vom Typ CHAR, wenn $1 \le i \le 60$;
s[*j*] eine Variable vom Typ *zeile*, wenn $1 \le j \le 25$;
s[*i*,*j*] eine Variable vom Typ CHAR, wenn $1 \le i \le 60$ und $1 \le j \le 25$;

Speicherung:

- durch das Aneinanderreihen der Repräsentationen des Basistyps, d.h. beginnend mit einer Anfangsadresse wird bei Vereinbarung einer Feldvariablen *f* entsprechend Speicherplatz zugeordnet (im HS):



→ Ergebnis: direkter Zugriff auf jede Komponente

4.5.2 Der Recordtyp

* bisher: einfache DO und Array als strukturiertes DO

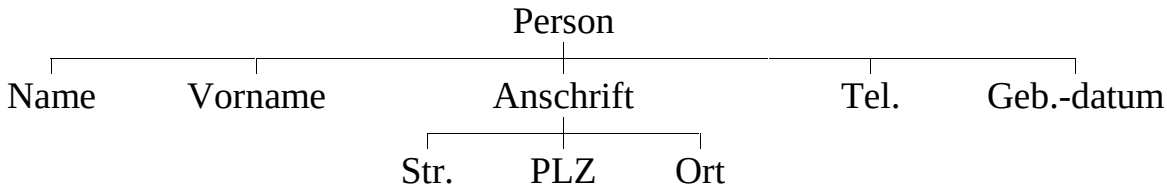
ARRAY: Möglichkeit der Zusammenfassung von Datenelementen des gleichen Datentyps;

→ oft Elemente völlig unterschiedlichen Datentyps zusammenzufassen, z. B.
Person: Name, Vorname, Str., PLZ, Ort, Tel., Geburtsdatum, ...;

Diese Daten werden meistens gemeinsam verarbeitet (Karteikarte), da sie eine zusammenhängende Information darstellen.

⇒ Name: Verbund (RECORD)

* RECORD-Typ: · logische Zusammenfassung von Komponenten, die nicht den gleichen Typ haben müssen;
· Komponenten können strukturierte DT sein
⇒ beliebige hierarchische Strukturen abbildbar;



* formale Beschreibung:

Seien $A_1,...,A_N$ endl. Mengen von Objekten;

$A = \overset{N}{\underset{i=1}{\times}} A_i$ - Wertebereich für Objekte $x = (a_1,...,a_n)$ mit $a_i \in A_i$, $i = 1(1)N$.

Def. 4.2:

Ein $x \in A$ heißt **Satz** (Verbund) von N Komponenten aus A_1 bis A_N.

*** Deklaration** (in Pascal, u.a.): im TYPE-Vereinbarungsteil;

```
TYPE recordtyp = RECORD
                   s1  : t1;
                   s2  : t2;
                     ...
                   sN  : tN
                 END;
```

Bem.: ▪ *recordtyp, s1, s2,...,sN* – Bezeichner
- *t1, t2,...,tN* – Typnamen für einf. oder zus.-gesetzte Datentypen;
- einzelne Komponenten auch als Felder bez.;
  entspr. Bezeichner → Feldnamen; (≠ ARRAY)
- auch Schachtelung möglich, d.h. schon definierte Records;

Bsp.:

```
TYPE    person = RECORD
              name, vorname    : STRING[16];
              anschrift        : RECORD
                                    plz          : INTEGER;
                                    ort, str     : STRING[20]
                                 END;
              tel              : STRING[12];
              gebdat           : RECORD
                                    tag          : 1..31;
                                    monat        : (Jan, Feb,...,Dez);
                                    jahr         : 1900..1999
                                 END;
                   END;
```

nach entspr. Typvereinbar. nun auch Variablen vom RECORD-Typ → VAR r : recordtyp;

VAR *x, y, meier : person*;

→ entspr. Var. vom Typ *person* als Datenobjekt (x, person, wert) mit Wert ∈ $W(t_1)\times...\times W(t_N)$;

Bsp.:

```
VAR     freund : person;
VAR     amor : ARRAY [1..100] OF person;
```

⇒ modelliert Tabelle mit fester Zeilenzahl (100) und „Spalten“ name, vorname, ..., gebdat;

Bem.: Typ-Vereinbarung kann auch umgangen werden:

```
VAR     freund : RECORD
                    name, vorname : ...
                    ...
                 END;
```

Nachteil: Datentyp *person* nun nicht mehr da!!!

Bsp.:

```
TYPE    datum = RECORD
            tag          : 1..31;
            monat        : 1..12;
            jahr         : 1900..1999
        END;
        student = RECORD
            name, vorname    : STRING[16];
            gebdat           : datum;
            stand            : (ledig, verhe, verwi, gesch);
            geschl           : (m, w);
            noten            : ARRAY [1..20] OF 1..5
        END;

VAR     x, y, meier   : student;
        semester : ARRAY[1..250] OF student;
```

*** Operationen:**

(I) Zu ∀ Komponenten *i* gehört ein Komponentenbezeichner $s_i : A \rightarrow A_i$, $i = 1(1)N$.
Der Bezeichner s_i (Postfixoperator) ist der Basismenge A_i zugeordnet; er wird als Selektionsoperator genutzt:

für $x \in A$ gilt $x.s_i = a_i$, $i = 1(1)N$.

Bsp.: ⇒ Mit Vereinbarung der Variablen *meier* werden zugleich die Komponentenvariablen *meier.name*, *meier.vorname*, ... , *meier.noten* vereinbart.

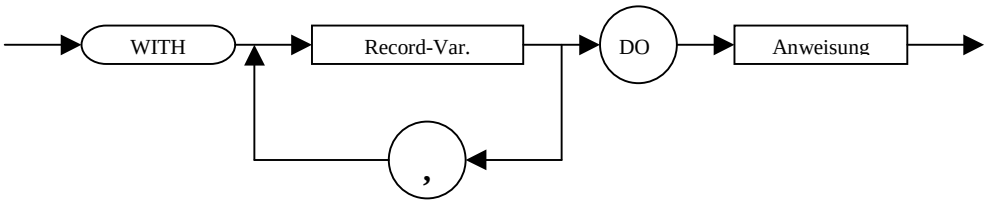
Selektion: – Zugriff auf diese Komponenten über den Selektionsoperator. (Punktoperator)

Bem.: Komponentenvariablen wie gewöhnliche Variablen (typentspr.) nutzbar.

z. B. →
```
meier.geschl := w ;
meier.noten[3] := 2 ;
meier.gebdat.jahr := 1975 ;
```

⇒ Selektionsoperator bewirkt Abstieg auf jeweils tiefer liegende Ebene!

(II) Zur Vereinfachung der Behandlung von Records gibt es, z. B. in Pascal und Modula-2 die sog. WITH – Anweisung:




```
WITH meier DO BEGIN
                    name := ´MEIER´;
                    vorname := ´MAX´;
                    gebdat.tag := 1;
                    gebdat.monat := 1;
                    gebdat.jahr := 1975;
                    . . .
                END;
```

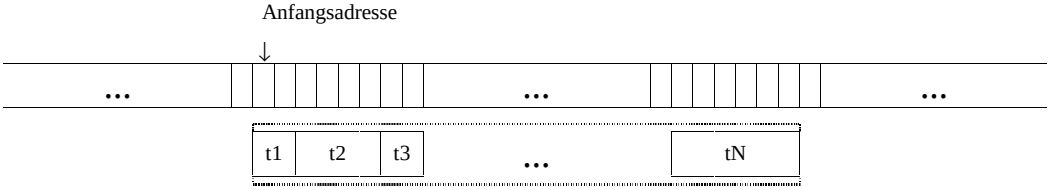
⇒ Innerhalb des von einer WITH – Anweisung umschlossenen Blocks können die Komponentenvariablen ohne Selektionsoperator benutzt werden.

(III) Operationen als Einheit:

a) Wertzuweisung ⇒ *rec1* := *rec2*;
b) als Parameter in Unterprogrammen;
c) als Konstantenvereinbarung ⇒

```
TYPE datum = RECORD
                tag     : 1..31;
                monat   : 1..12;
                jahr    : 0..99
             END;
CONST gebmeier : datum = (tag:1; monat:1; jahr:75);
```

*** Speicherung:** wie bei Feldern durch Aneinanderreihen der Repräsentationen der Grundtypen;



→ Ergebnis: direkter Zugriff auf jede Komponente (in konst. Zeit);

*** Erweiterung:**

Vereinigung von Elementen unterschiedlicher. Datentypen kann zu speicher-ineffizienten Record–Strukturen führen, z. B. wenn Komponenten existieren, die nur für einige der Datenobjekte gültige Werte annehmen

→ z. B.: Lebensmittel mit unterschiedl. Eig. → Fettgehalt, Alkoholgehalt, gesalzen; nun Streichfett, Bier, Fisch; nicht jeder hat alles;

für solche Strukturvarianten ⇒ **Record mit Variantenteil**;

* Formalisierung:

Seien T_1, ..., T_N endl. Mengen mit $T_i \cap T_j = \emptyset$, $i \neq j$.
Sei $T = T_1 \cup T_2 \cup ... \cup T_m$. Nun sei $a \in T$, dann ∃ genau ein k mit $a \in T_k$.

Sei *t* ein Typ mit W(*t*) = T und *x* ein DO vom Typ *t* : (*x*, *t*, wert).
⇒ Der Wert von *x* gehört während seiner Lebenszeit zu exakt einem Typ t_k mit $W(t_k) = T_k$, der sich allerdings ändern kann.

⇒ folg. Typvereinbarung für Record mit Variantenteil:

TYPE *recordtyp* = RECORD

{gemeinsamer Teil}

CASE *mark* : *marktype* OF

comp1 : $(s11 : T_{11}; ...; s1m_1 : T_{1m_1})$;

!! $\{T_{ij} \in \{T_1, ..., T_m\}\}$ *comp2* : $(s21 : T_{21}; ...; s2m_2 : T_{2m_2})$;

. . .

compN : $(sN1 : T_{N1}; ...; sNm_N : T_{Nm_N})$;

END;

Wirkung: In Abh. des Wertes der festen Komponente (Auswahlkomponente) *mark* des Records wird der akt. Datentyp bestimmt bzw. *compN* ausgewählt und realisiert.

Beachte:

(1) Der Typ eines Records mit Variantenteil ist wertgesteuert.
(2) Aus $T_i \cap T_j = \emptyset \Rightarrow s_{ik} \neq s_{jl}$ für ∀ i, j, k, l, d.h. Disjunktheit wird auf die Namen der Komponenten transformiert.
(3) Eine Variable kann mehrere Markierungskonstanten haben.
(4) Verschachtelung möglich.
(5) Gemeinsamer Teil muß vor dem Variantenteil stehen.

Bem.: Für *mark* nur die Typen INTEGER, BOOLEAN, CHAR und der Aufzählungs- / Unterbereichstyp!

Bsp.:

```
TYPE    form = (gerade, kreis);
        punkt = RECORD
                  x, y  : REAL
                END;
        linie = RECORD
                  CASE art : form OF
                    gerade : (a, b : punkt);
                    kreis : (m : punkt ;  r : REAL)
                  END;
VAR     l : linie;
```

4.5.3 Mengen (sets)

Sei B – beliebige endl. Menge; $\mathbf{P}(B) := \{A : A \subseteq B\}$ – Potenzmenge von B.

Klar: $\mathbf{P}(B) \neq \emptyset$, da $\emptyset, B \in \mathbf{P}(B)$.

Def. 4.3:

a) Sei *t* ein Datentyp mit endl. Wertebereich T = W(*t*). Dann wird mit
TYPE *settype* = SET OF *t*
ein Datentyp vom Typ Menge definiert.

b) *t* heißt dabei Basistyp.

c) Die Wertemenge von *settype* ist durch $\mathbf{P}(T)$ gegeben.

Wenn nun VAR *x* : *settype* benutzt wird, so kann *x* ein Wert aus $\mathbf{P}(T)$ zugeordnet werden.

Klar: $|\mathbf{P}(B)| < \infty$, da $|T| < \infty$.

Bsp.:

```
TYPE    student = RECORD
                    name, vorname     : STRING[20];
                    matrikelnr        : 1..5000;
                    semester          : 1..20;
                    . . .
                  END;

TYPE  s = SET OF student;
```

VAR *sem1* : *s*;
→ Menge aller Studenten im 1. Semester, also Menge von Records

Operationen:

(i) Mengenoperationen: versch. Operatoren →
a) Vereinigung + , Durchschnitt * , Differenz – ;
b) Test auf Enthaltensein → Operator IN (Mengenselektor);
IN prüft die Existenz eines Elementes aus T in einer Variablen vom Typ *t* , d. h.

IN: $T \times t \rightarrow \{TRUE, FALSE\}$ mit

$$\text{IN}(e, x) = : e \text{ IN } x := \begin{cases} TRUE, \text{ falls } e \in |x|; \\ FALSE \text{ sonst}; \end{cases}$$

dabei x – entspr. Set-Variable;
$|x|$ – Menge der aktuell enthaltenen Elemente;

(ii) Wertzuweisung:
Variablen vom Typ Menge kann mit Hilfe von Mengen-Konstruktoren eine best. Menge zugewiesen werden;
in Pascal z. B. zwei Konstruktormöglichkeiten:
a) Auflisten einzelner Elemente [e_1, e_2, e_3, e_4];
b) Angabe eines Unterbereiches [e_2 .. e_4];
z. B.:

```
TYPE  moebel = (Tisch, Stuhl, Sessel, Schrank, Sofa);
      zimmer = SET OF moebel;
VAR   x, y : zimmer;
```

→ z. B. folg. Anweisung möglich:

```
x := [ ];          {„leeres Zimmer“}
x := [Tisch..Schrank];
x := [Stuhl, Schrank, Sofa];
```

oder `IF Sofa IN y THEN x := x + [Sofa];`

Bem.: In Modula-2 anstelle [ ] nur { }.
Auch Mengenausdruck auf der rechten Seite möglich!

Speicherung:

Sei T = $\{w_1, ..., w_n\}$ der endl. vollst. geordnete Wertebereich des Grundtypes *t* .
Jede Teilmenge A ⊂ T eindeutig beschreibbar durch ihre charakteristische Funktion.

$$\chi_A : T \rightarrow \{0,1\} \; mit \; \chi_A(w_i) := \begin{cases} 1, \text{ falls } w_i \in A; \\ 0, \text{ sonst}; \end{cases}$$

→ Darstellungsmöglichkeit von Teilmengen als n-stelliges Wort über Binäralphabet bzw. Bitstring $z_A = z_1 z_2 ... z_n$ der Länge n, wobei $z_i = 1$ g. d. w. $\chi_A(w_i) = 1$ bzw. $w_i \in A$;

⇒ Geht, weil: Nur interessant, ob Element in Teilmenge; uninteressant, wie das Element aussieht;
⇒ Teilmengen ≡ binären Wörtern;
⇒ entspr. einfach die Mengenoperationen durch log. Verknüpfungen der entspr. Bitfolgen;
→ Jede Implementierung hat nur best. Anzahl El. für Basistyp; abhängig von Wortlänge. {Interessant: Solch abstrakter math. Apparat wie Mengen hat ganz einfache Implementierung.}
→ Angeg. Darstellungsmöglichkeit nur für Grundtypwertebereiche geringer Kardinalität praktikabel, da Darstellung von A mit $|A| << |T|$ Speicherplatzverschwendung.
⇒ später (2.Sem): Darstellung über Bäume (dynamische DS).

4.5.4 Der FILE-Typ

* bisher:

(i) betr. DS – Feld, Satz, Menge ⇒ „statische" Struktur von jeweils endlicher Kardinalität;
in Praxis oft DS benötigt, deren Komponentenanzahl sich während der Laufzeit dyn. ändern kann ⇒ „dynamische" Struktur

(ii) alle Daten im Arbeitsspeicher ⇒ verloren nach Abschalten des Rechners; die meisten Anwendungen fordern permanentes Speichern;

⇒ **1** geeignetes Hilfsmittel: Datei (file);

Datei : = Menge von Objekten (Daten), die nach einem log. Kriterium zusammengehören und als identifizierbare Einheit gespeichert werden.

Datei – zus.-gesetzte DS, die aus vielen gleichartigen Basiselementen (Datensätzen) besteht;

A) Allgemeine Grundlagen

Geg.: ∀ Basismenge B.

Betrachten nun B^* – Menge aller Wörter über B, d.h. $B^* = B^0 \cup B^1 \cup B^2 \cup \ldots$, wobei $B^0 := \varnothing$.

Sei $x \in B^*$ ein belieb. Wort.
Dann ex. $k \in \mathbf{N}$: $x = (b_1 b_2 \ldots b_k)$, $b_i \in B$, $i = 1(1)k$. Derartige El. aus B^* dienen als Werte für eine Variable vom File-Typ.

Welches sind die praktisch relevanten Aufgaben?

1. Problem: $x \leftarrow (b_1 b_2 \ldots b_k)$

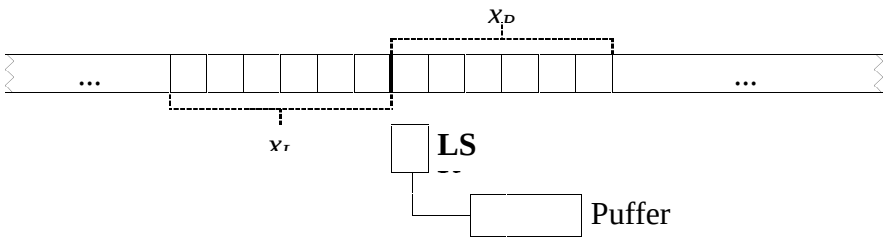
weil – bei Wertzuweisung an solche Variable ist das k zum Zeitpunkt der Def. dieser Var. unbekannt!

$\Rightarrow$ Speicherplatz für x nicht apriori definiert.

$\Rightarrow$ Frage: Wie vernünftig damit umgehen?! Ein Antwort liefert das folgende

- **Speichermodell:**

unendliches Band mit Feldern, die exakt ein El. aus der Basismenge B aufnehmen können:



weiter ∃ ein LS-Kopf (Lese-/Schreibkopf) der über genau 1 Feld stehen kann;
daran ein Puffer zur Aufnahme der Information angeschlossen;

$\Rightarrow$ $x = < x_L > < x_R >$ bzw. $x = x_L \circ x_R$

x_L – linke Teil des Wortes x, links vom LSK;
x_R – rechte Teil des Wortes x, Zelle unter LSK und rechts vom LSK;

Welche
* Operationen ?

LESEN: durch folg. Elementaroperationen beschreibbar $\Rightarrow$

(L.1) Rücken des LSK um 1 Feld nach rechts.
(L.2) Inhalt des LSK wird in Puffer geschrieben.
{Vor.: Direkter Zugriff zum Feldinhalt möglich.}

Frage in dem Zus.-hang: Wie zum 1. El. des Wortes x ?
$\rightarrow$ Brauchen/schaffen eine Operation, die den „Ur"-Zustand herstellt.
$\Rightarrow$ Operation

RÜCKSETZEN:

(RS.1) LSK auf 1. El. (Feld) von x gesetzt. (es gilt: $x_L = <>$ und $x_R = x$)
(RS.2) Inhalt des LSK wird in Puffer geschrieben.

SCREIBEN:

(S.1) Positionieren LSK so, daß $x_R = <\ >$.
(S.2) Inhalt des Puffers in Feld unter LSK.
(S.3) LSK um 1 Feld nach rechts. {damit $x_R = <\ >$ gesichert}

⇒ nur durch Anhängen von Zeichen wird „geschrieben“;
⇒ kann grundsätzlich nur auf ein leeres Wort schreiben;

Def. 4.4:

Ein Objekt mit den eben so def. Operationen LESEN, RÜCKSETZEN und SCHREIBEN heißt

sequentielles File.

Bem.: Das math. Strukturprinzip beim sequ. File ist die Bildung von Folgen (Sequenzen).

*** Abb. auf PS-n:** ∃ versch. Abb.-mechanismen dieses Modells auf Programmiersprachen.

In Pascal z. B. wie folgt:

1° Typvereinbarung: TYPE *filetyp* = FILE OF *t* ;

t beschreibt Basistyp, wodurch die Länge eines Feldes (auf Band) festgelegt wird; *filetyp* legt Wertemenge und Operationen fest;

2° Arbeit mit einem Objekt des Typs *filetyp* ⇒
Var.-vereinbarung: VAR *f* : *filetyp*;

Generell bzw. in der Regel:

folg. Standard-Operatoren für Files;

Wichtig: Alle File – Operatoren verwenden implizit eine Hilfsvariable, die Puffervariable:

- bei VAR *f* : *filetyp* wird gleichzeitig
  f↑ : *filetyp* eingerichtet;
- mit *f*↑ ist die aktuelle File-Komponente (auf welcher der LSK steht) assoziiert, d. h. beim Lesen wird der Wert der akt. File-Komponente der Puffervariablen *f*↑ zugewiesen; beim Schreiben wird der Wert der Puffervariablen *f*↑ der akt. File-Komponenten zugewiesen;

(1) REWRITE(*f*) : Es wird das leere File *f* neu angelegt; der Wert von $f\uparrow$ bleibt unverändert.
(2) PUT(*f*) : Anhängen (Schreiben) des Wertes von $f\uparrow$ an die Datei *f* (der Wert von $f\uparrow$ muß vorhanden sein); LSK eine Stelle nach rechts;
(3) RESET(*f*) : $f\uparrow$ wird der Wert der 1. Komp. von File *f* zugewiesen (LSK auf 1. Komp. gesetzt).
(4) GET(*f*) : Lesen eines Wertes in die Puffervariablen $f\uparrow$; LSK eine Stelle nach rechts;
(5) EOF(*f*) : Abfrage auf „End of File". {→TRUE, wenn Fileende überschritten; FALSE anderenfalls}

!!! Modell der Speicherung ⇔ entspr. Operationen !!!

In der Praxis oft sinnvoll, die Operation des Weitersetzens der LSK-Position mit dem Zugriff auf die Puffervar. zu verbinden.
Dazu 2 Prozeduren:

sei VAR *f* : *filetyp*;
VAR *v* : *t*; {TYPE *filetyp* = FILE OF *t*}

READ(*f, v*) ist def. durch *v* := $f\uparrow$; GET(*f*);
WRITE(*f, v*) ist def. durch $f\uparrow$:= *v*; PUT(*f*);

liest aus angeg. Datei den Wert der Puffervar. in entspr. Var.;
schreibt Wert einer Var. in angeg. Datei;

Bem.: Vor Ausführung von READ muß NOT EOF(*f*) sein!

Nächste Frage: Wie das betr. Speichermodell realisieren?

→ Frage: Wie bzw. wo kann das Band abgelegt werden?
Hauptspeicher ⇒ dann aber Feld-Typ nutzbar.
Also nicht HS, sondern folg. 2-stuf. Hierarchie →

HS: Puffer;
Zusatzsp.: Band;
↓
Welche Form?

a) durch Def. eines folgenden Objektes:

VAR *input* : FILE OF CHAR;
→ Tastatur, über die eine ∀ lange Folge von Zeichen eingebbar; (Band ≡ Tastatur)
⇒ gar keine andere Variante als das sequ. Verarbeiten;
input := Standart-Eingabe-File; auf Tastatur „gelegt";

VAR *output* : FILE OF CHAR;
:= Standart-Ausgabe-File;
→ wird Drucker/Bildschirm zugeordnet; (Band ≡ BS)
⇒ auch hier sequ. Verarbeitung;

!!! Das logische Band wird physisch anders dargestellt !!!

b) Direktzugriffsspeicher: Dateien; (Dateisystem des Betriebssystems)
VAR *datei* : FILE OF CHAR;
Synonym „Datei“ und „File“ hierdurch!

Sollen Files aus Umgebung des Pascal-Pr. benutzt werden, so muß ich sie in der Parameterliste des Kopfes angeben:

PROGRAM *name* (< liste von files >);

2 Regeln dabei:

1. Fileobjekte innerhalb eines Pr. leben nur so lange, wie das Pr. lebt (temporäre Datei).
2. Permanent vorhandene in Umgebung → wie oben angegeben.

Bem.:
Warum bei Arbeit mit *input*, *output* nicht (1)–(4) von Seite 24 (Standardoperationen) benutzt ? Dabei 2 Dinge zu beachten:

a) *input*, *output* sind vordef. Textdateien;
READ und WRITE sind nun Operationen, die mittels PUT und GET implementiert;
b) RESET und REWRITE erfolgt bei vordef. Dateien automatisch;

Zusf. Datei:

(1) als Erweiterung des Var.-konzepts;
! E/A von Daten; ! Permanente Speicherung von Daten;

(2) als Erweiterung des Feld-konzepts (dyn. DO bzw. DS);
! Vorteile aus Dynamik mit Nachteilen bzgl. Handhabung erkauft;

B) Turbo-Pascal

* Dateivariablen sind vom Typ FILE;

TP unterscheidet zwischen 3 Dateiarten:

(I) Textdateien: Textdateivariable mit vordef. Datentyp TEXT vereinbart →

VAR *f* : TEXT;

Textdatei := Aneinanderreihung von Zeichen;
in Zeilen eingeteilt;
jede Zeile endet mit Zeilenendezeichen;

(II) typisierte Dateien: lin. Sequenz von Datensätzen gleichen Typs;

VAR *f* : FILE OF < *komponententyp* > ;
{integer; real; ...}

! Als Komponententyp alles außer FILE zugelassen !

(III) untypisierte Dateien: Folge von Bytes ohne best. Struktur;
(typenlose);

VAR *f* : FILE;

hier keine Operationen mit den einzelnen Komponenten;
nur sog. Blockoperationen;

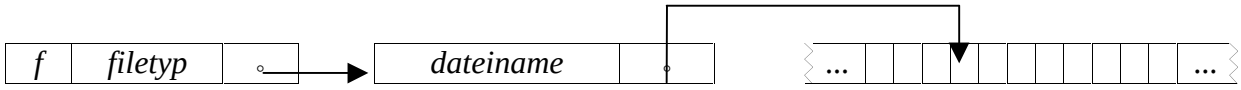
* ∃ Arbeitsschritte, die für alle Dateiarten gleich:

1. Schritt

- vor Benutzung einer Dateivariablen muß sie mit externen Datei verbunden werden, d. h. die Dateivariable wird mit dem Dateinamen der externen Datei verknüpft;

durch Prozedur:
ASSIGN (*f, dateiname*);
f – Dateivar. bel. Art;
dateiname – String-Typ; entspr. Pfadangabe, wenn nicht auf akt. Laufwerk.

⇒ Das Ansprechen der Dateivar. im Pr. bewirkt, daß sich entspr. Operationen auf die externe Datei beziehen.
Eine Art 2-stuf. Bezuges:



hat nun Beziehung zum
Ort des „Bandes“

2. Schritt

bis jetzt noch keine Dateiarbeit möglich;
fraglich, ob Datei mit angegebenem Namen existiert;
– ∃ Datei ⇒ öffnen: RESET(*f*);
– ∄ Datei ⇒ neu anlegen: REWRITE(*f*); *{erzeugt und öffnet}*

Weitere Schritte entspr. des Dateityps jeweilige Rountinen;

Letzter Schritt

Nach Arbeitsende (spätestens Pr.-ende) muß die Datei geschlossen werden mit

CLOSE(*f*);

schließt Datei, die mit Dateivar. *f* verbunden wurde;
speichert sie permanent auf der Disk/Platte unter angeg. Namen;
dabei auch Verb. zw. Dateivar. und Datei aufgehoben;

* Textdateien:

Files mit Komponenten vom Typ CHAR – wichtige Rolle;
oft Bindeglied zw. Rechner und Nutzer. Darum ein Standardname zugeordnet:

```
TYPE    TEXT = FILE OF CHAR;
```

⇒ Stand.-Eingabe- und Stand.-Ausgabe-File durch 2 Variablen:

```
VAR     input, output : TEXT;
```

→ folg. Einschränkungen

a) Auf Textfile **input** nur GET bzw. READ (nur lesen).
b) Auf Textfile **output** nur PUT bzw. WRITE (nur schreiben).

Bem.:

1. RESET bzw. REWRITE automatisch gemacht bei vordef. Files.

2. Bei beiden Textfiles:
   werden bei den Prozeduren READ und WRITE als deren erster Parameter vorausgesetzt, es sei denn der 1. Parameter ist eine andere File-Variable.

3. READ, WRITE können beliebige Anzahl Parameter haben:

(i) READ(*v1, v2, ..., vn*) ist def. durch
READ(input, *v1, v2, ..., vn*) :
liest aus input Werte in die entspr. Anzahl von Variablen;

READ(*f, v1, v2, ..., vn*) ist def. durch

```
BEGIN
        READ( f, v1);
        READ( f, v2);
              ...
        READ( f, vn);
END;
```

(ii) analog für WRITE(*v1, v2, ..., vn*) bzw. Write(*f, v1, v2, ..., vn*);