

Christian Maurer

das **M** <sup>U</sup><sub>od</sub> <sup>la</sup>**r** <sup>U</sup><sub>ni</sub> <sup>ver</sup>**S**<sub>um</sub>

v. 12. November 2011

Dr. Christian Maurer  
Keithstr. 16  
10787 Berlin

<http://murus.org/>

Die Quelltexte des  $\text{Modula-2}$  und  $\text{UJ-Sums}$  sind mit größter Sorgfalt entwickelt und werden laufend gepflegt. Kein Programmsystem dürfte jedoch jemals frei von Fehlern sein; deshalb ist auch nicht damit zu rechnen, dass *dieses* System fehlerfrei ist: Es darf nur benutzt werden „wie es ist“.

Die Modula-2- und die Java-Quelltexte von Murus sind freie Software. Sie können sie unter den Bedingungen der GNU General Public License, wie von der Free Software Foundation veröffentlicht, weitergeben und/oder modifizieren, entweder gemäß Version 3 der Lizenz oder (nach Ihrer Wahl) jeder späteren Version. Ihre Veröffentlichung erfolgt in der Hoffnung, dass sie Ihnen von Nutzen sein könnten – aber *ohne irgendeine Garantie*, auch ohne die implizite Garantie der *Marktreife* oder der *Verwendbarkeit für einen bestimmten Zweck*.

Der Originaltext der GPL ist im weltweiten Netz unter der Adresse [www.gnu.org/licenses/gpl.html](http://www.gnu.org/licenses/gpl.html) zu finden; eine deutsche Übersetzung unter [www.gnu.de/documents/gpl-3.0-de.html](http://www.gnu.de/documents/gpl-3.0-de.html).

Die Quelltexte von Murus sind nur zu Lehrzwecken konstruiert und haben rein akademischen Wert. Ihre Verwendung in Programmen könnte zu *Schäden* führen, z. B. zur Inbrandsetzung von Rechnern, zur Entgleisung von Eisenbahnzügen, zum GAU in Atomkraftwerken oder zum Absturz des Mondes ...

Object-oriented design is, in its simplest form,  
based on a seemingly elementary idea.

Computing systems perform certain actions on certain objects;  
to obtain flexible and reusable systems, it is better  
to base the structure of software on the objects  
than on the actions.

*Bertrand Meyer*

Object-oriented Software Construction, Prentice-Hall (1988), xiv



## Vorwort

In diesem Heft wird gezeigt, wie sich die Prinzipien einer Objekt-Entwicklungs-  
mit den Programmiersprachen Modula-2 und Go realisieren lassen.  
(Das Heft zu diesen Grundsätzen ist unter

<http://murus.org/doc/ObE.pdf>

im weltweiten Netz verfügbar.)

Nach einem Kapitel über die *Installation* des *Modula-2* werden in  
der *Einführung* einige Aspekte des Variablen- und Typkonzepts im im-  
perativen Paradigma zusammengefasst, deren Verständnis notwendige  
Voraussetzung für jeden objektorientierten Programmieransatz ist.

Dann wird die umfangreiche Bibliothek abstrakter Datentypen  
und -objekte erläutert, die im Laufe der letzten fünfundzwanzig Jahre  
bei meiner Lehrtätigkeit über Algorithmen und Datenstrukturen,  
Softwaretechnik und Projekte, Nichtsequentielle Programmierung und  
Mathematik entstanden ist und die die Grundlage für viele Beispiel-  
programme, Lehrprojekte und Anwendungen ist (die Modula-2-Quell-  
texte sind unter die GPL gestellt).

Im dritten Kapitel werden Mechanismen zur Konstruktion ein-  
facher *Benutzeroberflächen* vorgestellt.

Das Heft wird auf längere Sicht ein „*moving target*“ bleiben:

Der Einbezug der Programmiersprache *Go* (<http://golang.org>)  
ist ein Vorhaben, das einige Zeit benötigen wird (insofern stellt der  
erste Satz oben vorerst nur eine Absichtserklärung dar).

Auch sind noch weitere Kapitel über *Konglomerate von Objek-  
ten* und über *diverse nützliche Datentypen*, insbesondere auch für die  
*Nichtsequentielle Programmierung* geplant.

Deshalb ist es sinnvoll, gelegentlich im weltweiten Netz unter

<http://murus.org/doc/Murus.pdf>

nachzusehen, ob es eine erweiterte Version gibt.

Mitteilungen über entdeckte Fehler in den Modulen des *Modula-2* und  
über Ungereimtheiten in diesem Heft werden unter der E-Mail-Adresse

[maurer@murus.org](mailto:maurer@murus.org)

dankbar entgegengenommen.

*Christian Maurer*



## *Inhaltsverzeichnis*

VORAUSSETZUNGEN .....	1
Installation .....	1
Aktualisierung .....	3
Betrieb .....	4
EINFÜHRUNG .....	6
Modula-2 .....	6
Go .....	9
Variable konkreter Datentypen .....	11
Verweise, Wert- und Variablenparameter .....	15
Variable abstrakter Datentypen = Objekte .....	17
Wert- versus Referenzsemantik .....	18
Die Leistungen von Murus im Überblick .....	21
Standardprozeduren in den Moduln von Murus .....	21
initialisieren .....	21
terminieren, leer, leeren .....	23
kopieren .....	24
gleich, kleiner, kleinergleich .....	25
ausgeben, editieren, Codelaenge .....	26
codieren .....	27
decodieren .....	28
Die Komponenten von Murus im Überblick .....	29
BENUTZEROBERFLÄCHE .....	31
Grundsätzliches zu Benutzeroberflächen .....	31
Trennung von Ein- und Ausgabe .....	32
Tastatur und Maus .....	33
Bildschirm .....	36
Felder .....	40
Meldungen, Auswahlen, Aktionen .....	44
KONGLOMERATE VON OBJEKTEN .....	45
Folgen .....	45
Generizität durch Serialisierung .....	48
Folgen in Murus .....	48
Folgen von Objekten unterschiedlicher Größe .....	56

Kellerspeicher .....	57
Warteschlangen .....	58
Beschränkte Folgen .....	59
Prioritätswarteschlangen .....	60
Persistente Folgen .....	63
(Geordnete) Mengen .....	65
Fibonacci-Bäume und AVL-Bäume .....	65
Darstellung der Leonardo-Zahlen .....	67
Formale Potenzreihen .....	70
Persistente Mengen .....	72
ANHANG .....	73
Stichwortverzeichnis .....	73

## VORAUSSETZUNGEN

Vorausgesetzt wird ein „PC-kompatibler“ Rechner mit **Linux** als Betriebssystem. Die folgenden Hinweise beziehen sich auf **openSUSE**; sie müssen für andere Distributionen eventuell modifiziert werden.

das dasModellUllaSum basiert auf der Version 0608m des **Mocka**, des Modula-2-Compilers der GMD – der Forschungsstelle für Programmstrukturen an der Universität Karlsruhe. Quellen, Lizenzbedingungen usw. fanden sich im weltweiten Netz unter der Adresse

```
www.info.uni-karlsruhe.de/~modula/index.php
```

Sollte **Mocka** noch nicht installiert sein, muss das zuerst geschehen. Hinweise dazu sind unter der Adresse

```
http://lwb.mi.fu-berlin.de/inf/mocka/
```

zu finden.

### *Installation*

das dasModellUllaSum sollte im Verzeichnis `/usr/local` installiert werden, wovon im folgenden ausgegangen wird. Um bei einer Aktualisierung der Betriebssystemversion oder einem Wechsel der Distribution nicht alles neu installieren zu müssen, sollte für dieses Verzeichnis eine eigene Partition angelegt werden.

`root` wechselt in das Installationsverzeichnis

```
cd /usr/local
```

und sorgt dafür, dass dort die aktuelle Version der Datei `Murus.tgz` liegt; entweder durch Herunterladen in einem *Browser* von der Seite `http://murus.org/murus/` im weltweiten Netz durch einen Klick mit dem *rechten Mausknopf* oder mit dem Kommando

```
wget -N http://murus.org/murus/Murus.tgz
```

Diese Datei wird mit dem Kommando

```
tar xfv Murus.tgz
```

entpackt; dabei wird – sofern nicht bereits vorhanden – das Unterverzeichnis **Murus** erzeugt, in dem alle Quelltexte abgelegt werden.

Zur Erzeugung der Bibliotheken und Standardprogramme desModellenUlaUiverSums wechselt `root` in dieses Unterverzeichnis

```
cd Murus
```

und ruft das dort abgelegte Skript `makeMurus` auf:

```
./makeMurus
```

Dieses Skript übersetzt alle Quelltexte und bindet alle Standardprogramme desModellenUlaUiverSums, setzt alle erforderlichen Umgebungsvariablen und erweitert den Pfad, indem es im Verzeichnis `/etc/profile.d` die Datei `murus.sh` mit folgendem Inhalt ablegt:

```
export MURUS=/usr/local/Murus
export MOCKAM2="-d $MURUS/m2bin -CcallsMocka"
export MOCKALINK="-lpthread -lX11 -lGL"
export PATH=$MURUS/bin:$PATH
```

(Wenn dabei Fehlermeldungen vom Typ

```
.../Murus/XKern.mod:....: undefined reference to 'X...'
```

auftauchen, fehlt die X11-Bibliothek unter `/usr/lib/`.)

Diese Aktionen lassen sich in einem Skript zusammenfassen:

```
cd $MURUS/..
wget -N http://murus.org/Murus.tgz
tar xfzv Murus.tgz
cd Murus
./makeMurus
```

das unter dem Namen `getmurus` in `/usr/local/bin` abgelegt wird (`chmod 755 getmurus!`).

Benutzer/innen können dasModellenUlaUiverSum natürlich auch in eigener Regie installieren. Zur Installation im Benutzerbereich ist – mit geringen Änderungen – wie bei der Installation durch `root` zu verfahren:

Wenn z. B. das eigene Heimatverzeichnis als Installationsverzeichnis gewählt wird, ist `/usr/local` durch `$HOME` zu ersetzen und die Umgebungsvariable `MURUS` muss den Wert `$HOME/Murus` haben; und die Umgebungsvariablen werden nicht in `/etc/profile.d/murus.sh`, sondern in `$HOME/.bashrc` eingetragen.

### *Aktualisierung*

Die jeweils aktuelle Version der Quelltexte von  $\text{dasM}_{\text{od}}^{\text{Ula}}\text{U}^{\text{Sum}}$  und dieser Dokumentation sind im weltweiten Netz unter der Adresse

<http://murus.org/murus/Murus.shtml>

zu finden. Es ist sinnvoll, dort gelegentlich nachzuschauen, ob es eine neue Version gibt, in der aufgefundene Fehler beseitigt sind oder die leistungsfähiger ist als die vorherigen Versionen.

Mitteilungen über entdeckte Fehler im Programm oder Ungereimtheiten in dieser Dokumentation werden unter der E-Post-Adresse

[maurer@murus.org](mailto:maurer@murus.org)

dankbar entgegengenommen.

## BETRIEB

Da das `Modemultiver` Leistungen verfügbar macht, die über die üblichen Standards des tty-Konsolenbetriebs *weit hinausgehen*, nämlich

- *hochauflösende graphische Ausgaben in beliebigen Farben*
- *und den Einsatz einer Maus*

ist die Ausführung selbst anspruchsvoller ereignisgesteuerter Graphikprogramme in Konsolen möglich.

Der Betrieb unter X, d. h. in Fenstern auf graphischen Oberflächen wie z. B. KDE, Gnome oder IceWM, setzt voraus, dass die `Terminus`-Fonts, die `misc-fixed`-Fonts und die Mesa-Bibliothek (einschl. `mesa-devel`) installiert sind.

### *Konsolenbetrieb*

Eine der Konsolen 2 bis 6 wird mit `Strg+Alt+F2 ... F6` erreicht (ein einwandfreies Funktionieren in der `openSUSE-„Splash“-Konsole` (`Strg+Alt+F1`) ist nicht gesichert).

Für diesen Fall müssen

- der *Framebuffer* (Bildschirmspeicher) mit der Option `vga=n` in `/boot/grub/menu.lst`, wobei `n` eine der Zahlen 785/786 (VGA), 788/789 (SVGA), 791/792 (XGA), 794/795 (SXGA), 840/841 (SXGA+), 869 (WXGA+), 873 (WSXGA+), 838/842 (UXGA) oder 893 (WUXGA) – je nach Farbtiefe 16/24 bit – ist, eingerichtet werden,
- entweder die Benutzer in die Gruppe `video` eingetragen sein oder die Datei `/dev/fb0` für „die Welt“ les- und schreibbar sein, d. h. die Rechte `rw-rw-rw-` besitzen (`chmod 666 /dev/fb0`),
- die Datei `/dev/input/mice` für die Welt lesbar sein, d. h. die Rechte `rw-r--r--` besitzen (`chmod 644 /dev/input/mice`).

Die Änderungen per `chmod` sollte `root` durch Eintrag in die Datei `/etc/init.d/boot.local` vornehmen.

Programme mit *graphischen Ausgaben* oder *Verwendung der Maus* können in Konsolen natürlich *nicht per Login auf fernen Rechnern* ausgeführt werden, sondern *nur auf dem lokalen Rechner*, an dem Maus und Bildschirm hängen, da dabei auf eben diese die *lokalen Ressourcen* zugegriffen wird.

## *Betrieb unter X*

Unter X können alle Programme *auch auf einem fernen Rechner* ausgeführt werden, wenn dessen Ausgaben auf den lokalen Rechner *umgeleitet* werden. Am einfachsten (und sichersten) ist es, das mit der `secure shell` zu erledigen, indem man sich in einem Fenster mit dem Kommando `ssh -X host` (`host` = Name des fernen Rechners) auf ihm anmeldet und dann das Programm dort startet. Voraussetzung dazu ist, dass die `ssh`-Dienste auf den beteiligten Rechnern installiert sind, der Dämon `sshd` aktiviert und in seiner Konfiguration `/etc/ssh/sshd_config` die Zeile `X11Forwarding yes` enthalten ist und es einem erlaubt ist, sich per `ssh` auf ihm anzumelden.

Alternativ kann er in die Zugangskontroll-Liste für den X-Server aufgenommen und seine `DISPLAY`-Variable auf den lokalen Rechner gesetzt werden: dazu müssen in `/etc/sysconfig/displaymanager` die Werte der Umgebungsvariablen `DISPLAYMANAGER_REMOTE_ACCESS` und `DISPLAYMANAGER_XSERVER_TCP_PORT_6000_OPEN` auf `yes` gesetzt, der IP-Name des lokalen Rechners auf dem fernen in `/etc/X0.hosts` eingetragen oder mit dem Kommando `xhost` (Details: `man xhost`) gesetzt sowie dort die Umgebungsvariable `DISPLAY=host:0.0` (`host` = Name des lokalen Rechners) gültig machen.

## *Druck*

Drucken aus einem Programm heraus ist nur möglich, wenn `TEX` installiert und ein PostScript<sup>®</sup>-fähiger Drucker vorhanden ist.

## *Bildbearbeitung*

Zur Umwandlung von Graphiken von dem von das  $\text{M}_{\text{ode}}^{\text{U}}\text{I}^{\text{S}}\text{um}$  verwendeten `ppm`(„*portable pixmap*“)-Format in andere Formate (z. B. `gif` oder `jpg`) und umgekehrt ist die Installation von `netpbm` erforderlich.

## EINFÜHRUNG

### *Modula-2*

Die Programmiersprache *Modula-2* – NIKLAUS WIRTHS Weiterentwicklung der von ihm geschaffenen Sprache *Pascal* – erlaubt die Programmierung auf unterschiedlichsten Abstraktionsebenen

- von niedrigstmöglichen durch die Einbindung von
  - Assemblerprogrammen,
  - C-Programmen und
  - Systembibliotheken,
- über einfachste zur Entwicklung kleiner Programme
- bis zu ganz hohen durch die Zusammenfassung
  - mehrerer – auch sehr komplexer – Komponenten zu noch komplexeren,
  - ganzer Gruppen von Problemen zu Komponenten, durch die abstrakte Entwurfsmuster realisiert werden.

Warum sich *Modula-2* für die Zwecke einer streng objektbasierten Programmentwicklung hervorragend eignet, ist eins ihrer wesentlichen Merkmale: Sie bietet einen adäquaten Komponentenbegriff, den des

### *Moduls*

und dafür die

### *Zerlegung in Definitions- und Implementierungsteil*

mit folgenden fundamentalen Eigenschaften:

- der Übersetzbarkeit der Implementierung eines Moduls bereits nach Übersetzung seines eigenen und aller von ihm importierten *Definitionsmoduln* (ohne die Notwendigkeit der vorherigen Übersetzung der Implementierungen der benutzten Moduln) und
- der Möglichkeit der Konstruktion abstrakter Datentypen durch den Export *opaker Typen* in einem äußerst strengen Sinn, nämlich der *vollständigen textuellen Trennung* ihrer Realisierung von der Spezifikation in einer separaten Textdatei, dem zugehörigen *Implementierungsmodul*.

Im *Definitionsmodul* wird nur der Name des exportierten Datentyps angegeben, der damit als Träger für Parameter in den Operationen zum Zugriff auf Variable dieses Typs vorhanden ist, ohne dass seine Implementierung bekannt ist. (Wir kommen hierauf im Abschnitt über *Abstrakte Variable* ausführlich zurück.)

Einzigste Ausnahme von dieser Trennung bilden die *Hauptmoduln*, die eigentlichen *Programme*, zu denen es keinen Definitionsteil gibt.

Damit ermöglicht Modula-2 eine rigide Umsetzung des Prinzips des *information-hiding* – viel konsequenter als in Ada, C#, Java usw., bei denen alle Implementierungen den Klienten mindestens textuell bekanntgegeben werden, wenn auch wegen eines Attributs wie z. B. `private` nicht auf sie zugegriffen werden kann – und erlaubt deshalb,

*sämtliche Grundsätze einer  
objektbasierten Entwicklung  
konsequent einhalten zu können,*

wie sie z. B. im Kapitel *Systemarchitektur* des Heftes über Objektorientierte Entwicklung oder in jedem Lehrbuch über Softwaretechnik erläutert sind.

Diese Vorteile zeichnen Modula-2 vor älteren imperativen Sprachen, wie z. B. Fortran, Cobol, Algol, C oder Pascal, deutlich aus, deren Ausdrucksmöglichkeiten mit den hier postulierten Grundlagen beherrschbarer Systemarchitekturen – selbst bei sehr bescheidenen Ansprüchen an ihre Umsetzung – im Grunde völlig überfordert sind.

Eine oberflächliche Begründung für diese strikte Trennung ist, dass

- ein größeres System entwickelbar ist, indem seine Komponenten weitestgehend unabhängig voneinander entwickelt werden,
- und dabei die *Klienten*, d. h. die Personen, die zur Entwicklung eigener Komponenten vorhandene Komponenten verwenden, nicht *dadurch* völlig überfordert werden, dass ihre Arbeit die Kenntnis der – möglicherweise äußerst komplizierten – Implementierungen der benutzten Komponenten voraussetzt.

Etwas genauer beleuchtet heißt das gerade, dass die *Klienten* einer Komponente

- *nur ihre Spezifikation*, d. h. die abstrakte Beschreibung ihrer Leistungen, *kennen dürfen*,
- *nicht dagegen die Implementierung*,

um ihre eigenen Konstruktionen nicht durch implizites Wissen über Details der von ihnen benutzten Leistungen abhängig zu machen und nicht unbefugt – d. h. an der „Schnittstelle“ (= der Spezifikation) vorbei – auf die internen Daten des benutzten Moduls zuzugreifen.

(Wo kämen denn die Entwickler eines Kraftfahrzeugs hin, wenn die Konstruktion der Karosserie von technischen Details der Zylinderkopfhabe, des Antiblockiersystems oder Hinterachsdifferentials abhinge oder sogar diese Details zu beeinflussen suchte?)

Der in Modula-2 verwendete Begriff „Modul“ hat sich als Sammelbegriff für die kleineren Komponenten eines Systems eingebürgert und man spricht von *modulorientierten* Programmiersprachen.

Der *Klassenbegriff* in den neueren *objektorientierten* Sprachen ist in vieler Hinsicht damit vergleichbar; teils etwas restriktiver, aber in diversen Aspekten auch umfassender, weil weitere Eigenschaften wie *Vererbung* und neuerdings *Generizität* (*parametrische Polymorphie*) hinzukommen (was man z. B. bei funktionalen Sprachen seit etwa drei Jahrzehnten kennt, ist nach eineinhalb Jahrzehnten Entwicklung auch endlich in Java angekommen).

## *Go*

Im Herbst 2007 begann bei GOOGLE die Arbeit am Entwurf der Programmiersprache *Go*; veröffentlicht wurde Go im November 2009; die erste stabile Version erschien im Mai 2011.

Für Go gilt im Hinblick auf die Möglichkeiten der Programmierung auf verschiedenen Abstraktionsebenen und der Eignung zur objektbasierten Programmentwicklung zunächst erst einmal das gleiche wie für Modula-2.

Wir werden später auf eine ganze Reihe von Punkten zu sprechen kommen, die Go darüberhinaus auszeichnen.

Komponenten in Go sind

*Pakete („packages“).*

Insoweit ein *package* einen abstrakten Datentyp realisiert, ist auch in Go die

*Zerlegung in Spezifikation („interface“) und Implementierung*

möglich. Bei abstrakten Datentypen geht das (voreilig gesagt: leider) nicht, aber das hat zwingende systemimmanente Gründe.

In einem *Interface* werden der Name des exportierten Datentyps sowie die Signaturen von Operationen auf ihm angegeben.

Das hat zunächst die gleiche Funktion wie in Modula-2: Er ist als abstrakte Beschreibung möglicher Implementierungen vorgegeben und kann in anderen Paketen als Träger für Parameter in den Operationen zum Zugriff auf Variable dieses Typs benutzt werden, ohne dass seine Implementierung bekannt ist – mit allen im vorigen Abschnitt erwähnten Vorteilen, die das für die Architektur größerer Systeme mit sich bringt.

Hier sind wir aber an einem ersten entscheidenden Punkt, in dem der Entwurf von Go deutlich über das Konzept der *Objektbasierung* hinausgeht:

Spezifikationen können in der Weise „geschachtelt“ werden, dass *Interfaces* auf andere *Interfaces* „vererbt“ werden können.

Dabei handelt es sich um einen am Ende des vorigen Abschnitts angedeuteten Begriffe aus dem *objektorientierten* Paradigma, das z. B. auch in Java realisierbar ist.

Dazu eine persönliche Bemerkung: Ich bin der Meinung, dass die Vererbung auf der Ebene von Spezifikationen wahrscheinlich viel interessanter und weitreichender ist, als die auf der Ebene von Klassen. Das wird in dem geplanten Abschnitt zum Paket `object` von `murus` erläutert, das das Go-Analogon zu dem Abschnitt über die Standardprozeduren in Modula-2 darstellt.

Damit unterstützt Go in hohem Maße *information-hiding* mit all ihren Vorzügen, die im vorigen Abschnitt über Modula-2 genannt worden sind. Die einzige Einschränkung ist, dass es nicht möglich ist, Spezifikationen abstrakter Datenobjekte syntaktisch zu formulieren. Das kann allerdings dadurch kompensiert werden, dass statt der Konstruktion eines abstrakten Datenobjekts eine Klasse konstruiert wird, von der dann eben nur ein einziges Exemplar benutzt wird.

Lediglich bei Datenobjekten, die den Zugriff auf die Peripheriekapseln (Tastatur, Bildschirm, Drucker usw.) ist das etwas mühsam.

Auf der anderen Seite machen die Vorzüge des Paket-Konzepts von Go diesen „Nachteil“ mehr als wett:

Zu einer Spezifikation kann es in einem Paket z. B. mehrere Implementierungen geben, was für bestimmte Zwecke sehr hilfreich sein kann: ferner können klar abtrennbare Teile der Implementierung ebenfalls in das entsprechenden Paket „verpackt“ werden, und müssen nicht in andere Moduln ausgelagert werden. Softwaretechnisch ist das ein ganz erheblicher Vorteil, weil damit spezielle Dienstleistungen unterer Schichten gezielt für die Implementierung bestimmter Pakete zur Verfügung gestellt werden können, die weiter außen nicht sichtbar sind. Das Paket-Konzept ist insofern viel leistungsfähiger als das Modul-Konzept.

Wir werden an geeigneten Stellen dafür detaillierte Beispiele für diese Thesen geben.

## *Variable konkreter Datentypen*

Um grundlegende Aspekte der objektbasierten Programmierung darstellen zu können, seien zunächst die Prinzipien des imperativen Paradigmas zusammengefaßt, die sich auf das Variablen- und Typkonzept beziehen.

Unter *konkreten Datentypen* verstehen wir die bekannten rekursiv aus atomaren per Feld-, Verbund-, Mengen- und Verweiskonstruktoren zusammengesetzten Datentypen und die Prozedurtypen.

Wir werden im folgenden unter einer

### *konkreten Variablen*

immer eine *Variable eines konkreten Datentyps* verstehen.

Der Bedarf an Speicherplatz für eine konkrete Variable ist über die *Typgröße* ihres Datentyps kalkulierbar. Atomare Datentypen haben definierte Typgrößen; die Typgrößen zusammengesetzter Datentypen, z. B. Feldern oder Verbunden lassen sich daraus errechnen.

## *Modula-2*

In Modula-2 hat man folgende konkrete Datentypen:

- Die *atomaren Datentypen*
  - BOOLEAN für Wahrheitswerte,
  - CHAR für Zeichen,
  - SHORTCARD, CARDINAL, SHORTINT, INTEGER, REAL und LONG-  
REAL für Zahlen und
  - BYTE, WORD, ADDRESS und BITSET für maschinennahe Variable,
- für jeden diskreten atomaren Datentyp und je zwei Konstanten *a* und *b* dieses Typs mit  $a \leq b$  den
  - *Teilbereichstyp* [*a*..*b*],
- für Bezeichner *a*, *b*, ... den
  - *Aufzähltyp* (*a*, *b*, ... ),
- für jeden endlichen Aufzähl- oder Teilbereichstyp („*Indextyp*“) *I* und jeden konkreten Datentyp *X* das
  - *Feld* ARRAY *I* OF *X*,
- für jede endliche Folge *X*, *Y*, ... konkreter Datentypen den
  - *Verbund* RECORD *x*: *X*; *y*: *Y*; ... END
 mit *Komponenten* *x* vom Typ *X*, *y* vom Typ *Y*, ... ,

- für jeden endlichen Aufzähl- oder Teilbereichstyp  $X$  mit einer Anzahl  $\leq$  Adreßbreite des Rechners in bit die
  - Menge SET OF  $X$ ,
- für jede endliche Folge  $X, Y, \dots$  konkreter Datentypen den
  - Prozedurtyp PROCEDURE ([VAR]  $X, [VAR] Y, \dots$ )[:  $W$ ].
- für jeden konkreten Datentyp  $X$  den
  - Verweistyp (= Zeigertyp, Referenztyp) POINTER TO  $X$
 mit dem Dereferenzierungsoperator  $\hat{\cdot}$ , der einem Zeiger  $p$  vom Typ POINTER TO  $X$  diejenige Variable  $p\hat{\cdot}$  vom Typ  $X$  zuweist, „auf die  $p$  zeigt“ (mehr dazu im nächsten Abschnitt).

Mit der Deklaration einer konkreten Variablen  $x$  eines Datentyps  $X$

VAR  $x$ :  $X$ ;

ist u. a. folgendes verbunden:

- Zum Zeitpunkt der Übersetzung des Programms – d. h. durch den Übersetzer („Compiler“) – wird Speicherplatz für den Wert der Variablen  $x$  bereitgestellt, dessen Größe (d. h. „Typgröße“ seines Typs  $X$ ) per Typdeklaration festgelegt ist.
- Dieser Speicherplatz ist unter dem Namen  $x$  der Variablen innerhalb ihres Gültigkeitsbereichs „adressiert“, d. h. man kann sich den Namen der Variablen als Verweis (Zeiger, Referenz) auf die Startadresse des Speicherplatzes vorstellen,
- und er ist exklusiv für ihren Wert reserviert und steht damit für andere Zwecke nicht mehr zur Verfügung;
- seine Startadresse  $ADR(x)$  wird von der polymorphen Funktion  $ADR$  aus dem Pseudomodul SYSTEM geliefert.

Atomare Datentypen haben bestimmte Typgrößen (die von der Busbreite des verwendeten Prozessors abhängen); z. B.

- BOOLEAN, CHAR und BYTE: 1 Byte,
- SHORTCARD und SHORTINT: 2 Byte,
- CARDINAL, INTEGER, REAL, WORD, ADDRESS und BITSET: 4 Byte und
- LONGREAL: 8 Byte.

Die Typgrößen zusammengesetzter Datentypen lassen sich daraus errechnen; im Prinzip (ohne Berücksichtigung von *Alignment*) bei

- Feldern als Produkt aus der Anzahl der Werte des Indextyps und der Typgröße des Basistyps,

- Verbunden als Summe der Typgrößen ihrer Komponenten,
- Mengen als Typgröße von `BITSET`.

In Modula-2 wird die Typgröße `TSIZE(X)` eines konkreten Datentyps `X` von der polymorphen Funktion `TSIZE` aus dem Pseudomodul `SYSTEM` geliefert.

Für konkrete Variable bzw. Ausdrücke konkreter Datentypen sind in Modula-2 die üblichen *Standardoperationen* vorgesehen (wobei die einschlägigen Regeln der Typverträglichkeit zu beachten sind):

- die Wertzuweisung `:=`
  - zum Kopieren des Wertes eines Ausdrucks in eine Variable (genauer: des den Wert repräsentierenden Bitmusters in den für die Variable reservierten Speicherplatz),
- das Gleichheitsprädikat `=` und seine Negation `#`, `<>`
  - zur Überprüfung auf Übereinstimmung bzw. Ungleichheit der Werte zweier Ausdrücke (genauer: auf bitweise Übereinstimmung der Inhalte der für sie reservierten Speicherplätze),
- die Prädikate der Ordnung `<`, `<=`, `>` und `>=`
  - zum Größenvergleich der Werte von Ausdrücken,
- Zuweisung ausgezeichneter Werte wie z. B. Leerzeichen(-ketten) oder bestimmter – je nach Kontext – sinnvoller Konstanten
  - zum Leeren (Löschen) von Werten
- sowie die Prüfung auf „Leersein“ im eben genannten Sinne; ferner Operationen des „Quasistandards“, den von WIRTH im Anhang seiner Sprachbeschreibung spezifizierten „standard utility modules“, die so (oder ähnlich) alle Implementierungen der Sprache bereitstellen:
  - `Write`, `WriteString`, `WriteCard`, `WriteInt` und `WriteReal` aus dem Modul `InOut`
    - zur Ausgabe auf dem Bildschirm,
  - und `Read`, `ReadString`, `ReadCard`, `ReadInt` und `ReadReal`
    - zur Eingabe mittels Tastatur,
  - sowie gewisse Routinen aus speziellen Bibliotheken für die Abfrage der Maus
    - zur Ereignissteuerung mit ihr,
- `ADR` und `TSIZE` aus dem Pseudomodul `SYSTEM`
  - zum Zugriff auf die Repräsentation der Werte von Variablen als Bytefolgen im Arbeitsspeicher über die Startadresse und

Größe des für sie reservierten Speicherplatzes,

- `ReadNBytes` und `WriteNBytes` aus dem Modul `FileSystem`
  - zum Zugriff auf Bytefolgen im Dateisystem.

## *Go*

In Go gibt es unter anderem die folgenden konkreten Datentypen:

- Die *atomaren Datentypen*
  - `bool` für Wahrheitswerte,
  - `int8`, `int16`, `int32`, `int` und `int64` für ganze Zahlen,
  - `uint8`, `uint16`, `uint32`, `uint`, `uint64` für natürliche Zahlen mit dem Synonym `byte` für `uint8`,
  - `float32` und `float64` für reelle Zahlen,
  - `complex64` und `complex128` für komplexe Zahlen,
  - `string` für Zeichenketten,
- für jeden konkreten Datentyp `X` und jede Konstante `n` mit dem Wert einer natürlichen Zahl das
  - *Feld* `[n]X`,
- für jeden konkreten Datentyp `X` den
  - *slice* `[]X`,
- für jede endliche Folge `X, Y, ...` konkreter Datentypen den
  - *Verbund* `struct { x X; y Y; ... }`  
mit *Komponenten* `x` vom Typ `X`, `y` vom Typ `Y`, ... ,
- für jede endliche Folge `X, Y, ...` konkreter Datentypen den
  - *Funktions*typ `func ([*]X, [*]Y, ... )[: W]`.
- für jeden konkreten Datentyp `X` den
  - *Verweistyp* (= *Zeigertyp*, *Referenztyp*) `*X`  
mit dem *Dereferenzierungsoperator* `*`, der einem Zeiger `p` vom Typ `*X` diejenige Variable `*p` vom Typ `X` zuweist, „auf die `p` zeigt“ (mehr dazu im nächsten Abschnitt).
- für jeden konkreten Datentyp `X` den
  - *Kanal*typ `chan X`.

*Weiteres folgt irgendwann.*

### *Verweise, Wert- und Variablenparameter*

In diesem Abschnitt werden einige Aspekte des *Zeigerkonzepts* beleuchtet, deren Verständnis unabdingbare Voraussetzung für alles weitere, insbesondere für die Realisierung des grundlegenden Begriffs „Objekt“ in der *objektorientierten* Programmierung ist. Dazu zeigen wir an einem Beispiel, wie *Variablen-(Verweis-, Referenz-)parameter* durch *Wertparameter* ersetzt werden können.

Die in Modula-2 als Sprachbestandteil enthaltene (polymorphe) Prozedur INC, in der eine Variable um einen Wert (z. B. vom Typ CARDINAL) erhöht wird, würde man wie folgt implementieren:

```
PROCEDURE INC (VAR n: CARDINAL; w: CARDINAL);
BEGIN
  n := n + w
END INC;
```

Die Gründe, warum der erste Parameter von INC ein Variablen- und der zweite ein Wertparameter ist, liegen auf der Hand.

Es ist in der Tat möglich, den Effekt dieser Prozedur zu erzielen, ohne einen Variablenparameter zu verwenden, indem ein *Verweis* (ein Zeiger auf eine Adresse im Arbeitsspeicher) eingesetzt wird:

```
PROCEDURE inc (a: ADDRESS; w: CARDINAL);
VAR p: POINTER TO CARDINAL;
BEGIN
  p := a;
  p^ := p^ + w
END inc;
```

Der Funktionsweise ist dadurch erklärt, daß der *Dereferenzierungsoperator*  $\hat{\ }$  als Umkehroperator des *Adreßoperators* ADR definiert ist:

Für Variable p vom Typ POINTER TO X und x vom Typ X gilt

$$p = \text{ADR}(x)$$

d. h. daß p die Startadresse des für x reservierten Speicherplatzes als Wert enthält, *genau dann, wenn*

$$p^{\hat{}} = x,$$

d. h. wenn p ein Verweis auf x ist („auf x zeigt“).

Insbesondere gilt (man setze  $\hat{p}$  für  $x$  ein)

$$p = \text{ADR}(\hat{p}),$$

d. h. der Wert von  $p$  ist gerade die Startadresse des für  $\hat{p}$  reservierten Speicherplatzes; kurz:  $\hat{p}$  ist genau *die* Variable, auf die  $p$  zeigt.

Aus diesem Grunde hat die erste Anweisung  $p := a$  von `inc` den Effekt  $a = \text{ADR}(\hat{p})$ . Folglich wird über den lokalen Zeiger  $n$  auf die übergebene Adresse  $a$  zugegriffen, ab der `TSIZE(CARDINAL)` Bytes als Wert einer Variablen vom Typ `CARDINAL` interpretiert und so verändert werden, daß dieser Wert nach dem Aufruf der Prozedur um den des übergebenen Ausdrucks erhöht ist. Das liefert nun aber *genau dann* den gewünschten Effekt, wenn beim Aufruf

$$a = \text{ADR}(x)$$

für eine Variable  $x$  vom Typ `CARDINAL` gilt, d. h. wenn `inc` nicht eine Variable  $x$  vom Typ `CARDINAL` übergeben wird, sondern der Wert der Startadresse  $\text{ADR}(x)$  des für sie reservierten Speicherplatzes (was als Voraussetzung natürlich zur Spezifikation von `inc` gehört).

Dabei geht allerdings die durch den Übersetzungsvorgang garantierte Typsicherheit verloren, denn mit der Implementierung von `inc` kann nicht überprüft werden, ob ein Klient die Prozedur typgerecht, d. h. unter ihrer Voraussetzung, aufruft; eine fehlerhafte Verwendung mit unvorhersehbaren Effekten zur Laufzeit ist grundsätzlich möglich (z. B. hätte für `VAR s: ARRAY[0..3] OF CHAR` mit  $s = \text{"Affe"}$  der Aufruf `inc(ADR(s), 117378308)` den Effekt  $s = \text{"Esel"}$  zur Folge).

Das Beispiel zeigt, daß das Konzept „Variablenparameter“ nicht unbedingt Bestandteil einer Programmiersprache sein muß, sondern unter Einsatz des Adreß- und Zeigerkonzepts simuliert werden kann (genau so wird z. B. beim Programmieren in der Sprache C verfahren); es zeigt aber auch sehr deutlich das Problem der fehlenden *statischen*, d. h. zum Übersetzungszeitpunkt überprüfbaren, Typsicherheit auf.

Ferner lehrt es uns, daß – wenn wir derart auf Variable zugreifen – der „naive“ Ansatz, daß Wertparameter vor einer Veränderung der übergebenen Variablen schützen, *keineswegs* mehr gilt.

Das ist nun aber kein Widerspruch, weil beim Aufruf ja *nicht die konkrete Variable*, sondern vielmehr *ein Verweis auf sie* übergeben wird, der nach dem Aufruf natürlich *nicht* verändert ist.

### *Variable abstrakter Datentypen = Objekte*

Unter *abstrakten Datentypen* verstehen wir diejenigen Datentypen, deren Existenz zwar durch die Angabe ihres Bezeichners (und natürlich ihrer Zugriffsoperationen) in einer Spezifikation gesichert, deren Implementierung jedoch den Klienten – den Nutzern der in der Spezifikation definierten Leistungen – nicht bekannt sind.

In Modula-2 werden sie in einem *Definitionsmodul* in der Form

TYPE X;

– *also ohne Angabe ihrer Repräsentation* – definiert und deshalb auch als *opake* Datentypen bezeichnet.

Im zugehörigen *Implementierungsmodul* sind sie im einfachsten Fall als *Verweis* auf einen konkreten Datentyp realisiert, in komplexeren Situationen als Verweis auf einen Verbund, dessen Komponenten möglicherweise *ihrerseits* abstrakte Datentypen sind.

Analog zum konkreten Fall werden wir im folgenden unter einer  
*abstrakten Variablen*

immer eine Variable eines abstrakten Datentyps verstehen.

Abstrakte Variable werden zwar genau so wie konkrete vereinbart,

VAR x: X;

aber es gibt hier doch einen ganz erheblichen Unterschied:

Der Wert einer solchen Variablen ist nach dem oben gesagten ein *Verweis*, d. h. eine Variable des konkreten Datentyps ADDRESS: eine Adresse im Arbeitsspeicher, deren Wert *diejenige* Adresse im Arbeitsspeicher ist, ab der Werte von Variablen des Typs abgelegt werden, auf den verwiesen wird. Ihre Typgröße ist die – durch die Adreßbreite des zugrundeliegenden Prozessors gegebene – Typgröße des konkreten Datentyps ADDRESS.

Ihr Wert ist also von ihrem „*eigentlichen Wert*“, d. h. dem *der Variablen, auf den sie verweist*, zu unterscheiden, und ihre Typgröße hat *nichts* mit der Typgröße des *eigentlichen Werts* zu tun.

Aus diesem Grund *muß der Deklaration einer solchen Variablen* – im Unterschied zu der vom Übersetzer und Laufzeitsystem bei der Initialisierung konkreter Variablen durchgeführten Verfahrensweise – ausdrücklich *die Bereitstellung von Speicherplatz für den eigentlichen Wert folgen*.

*Das ist aber eine für den Übersetzer prinzipiell unlösbare Aufgabe:*

Das Konzept der separaten Übersetzbarkeit von Spezifikation und Implementierung hat zur Folge, daß bei der Übersetzung der „Blick“ auf den *eigentlichen Datentyp* hinter den Kulissen unmöglich, folglich die *eigentliche* Typgröße nicht bekannt ist – ganz einfach aus dem Grunde, daß die Existenz der Implementierung zu diesem Zeitpunkt nicht vorausgesetzt werden kann (was ja gerade ein wichtiger Zweck dieser unabhängigen Übersetzbarkeit ist).

Da somit *der Übersetzer* die Reservierung des eigentlich benötigten Speicherplatzes *nicht* veranlassen kann, muß diese Aufgabe von einer *übergeordneten Instanz* übernommen werden:

Die Person, die einen Quelltext entwickelt, muß – als Klient eines abstrakten Datentyps – die Deklaration jeder Variablen dieses Typs durch Einbau einer – in der Spezifikation des Datentyps enthaltenen – Anweisung ergänzen, deren Implementierung dafür zu sorgen hat, daß Speicherplatz für den *eigentlichen Wert* angelegt wird.

Dabei handelt es sich um ein charakteristisches Kennzeichen der *objektorientierten Programmierung*, weshalb wir von jetzt an

*abstrakten Variablen*

als von

*Objekten*

sprechen werden. Fazit:

*Objekte müssen explizit erzeugt werden,  
bevor sie bearbeitet werden können.*

### *Wert- versus Referenzsemantik*

Nach den Überlegungen aus dem vorigen Abschnitt stellt sich nun die Frage, welche Konsequenzen sich daraus ergeben, wenn Objekte, d. h. abstrakte Variable, „hinter den Kulissen“, also in der Implementierung, nichts anderes als Verweise sind.

Eine *Wertzuweisung*  $x := y$  hat bei *konkreten Variablen* zur Folge, daß der Wert von  $y$  in die Variable  $x$  kopiert wird. Konsequenz ist, daß es nach der Zuweisung zwei verschiedene konkrete Variable mit identischem Wert gibt, weil für die beiden Variablen unterschiedliche Speicherplätze reserviert sind. Wenn folglich danach der Wert z. B. der Variablen  $y$  verändert wird, ist die Variable  $x$  davon nicht betroffen; ihr Wert ist *nicht* verändert.

Für *Objekte*  $x$  und  $y$ , d. h. *abstrakte Variablen* eines abstrakten Datentyps, gilt *das* nun gerade *nicht*: Mit der Anweisung  $x := y$  wird vielmehr der Verweis  $y$  auf ein Objekt, d. h. lediglich die Adresse, ab der der „Wert“ von  $y$  zu finden ist, in den Verweis  $x$  kopiert.

Das hat eine völlig andere Konsequenz: Der Zeiger  $x$  verweist jetzt auf das gleiche Objekt wie der Zeiger  $y$ , d. h. die Variable  $x$  bezieht sich jetzt auf das gleiche Objekt wie  $y$ . Wenn danach das Objekt  $y$  verändert wird, ist demzufolge auch der Wert des Objekts  $x$  (genauso) verändert.

Der erstgenannte Fall ist ein Beispiel für *Wertsemantik*, der zweite für *Referenzsemantik*.

Diese Unterscheidung ist auch in anderen Fällen vorzunehmen:

Der Boolesche Ausdruck  $x = y$  liefert für konkrete Variable  $x$  und  $y$  eine Aussage darüber, ob die Werte der beiden Variablen gleich sind (*Wertsemantik*), für Objekte dagegen nur, ob die Zeiger  $x$  und  $y$  auf das gleiche Objekt verweisen (*Referenzsemantik*).

Ganz ähnlich stellt sich die Situation beim Größenvergleich dar: Der Boolesche Ausdruck  $x < y$ , der bei konkreten Variablen, für deren Typ die Relation  $<$  definiert ist, eine Aussage darüber liefert, ob der Wert von  $x$  kleiner ist als der von  $y$ , ergibt bei Objekten  $x$  und  $y$  lediglich die (völlig uninteressante) Aussage, ob der Verweis auf den Wert von  $x$  kleiner ist als der Verweis auf den Wert von  $y$ , d. h. ob der Speicherplatz für das Objekt  $x$  im Arbeitsspeicher vor dem des Objektes  $y$  liegt.

Fazit:

*Beim Umgang mit Objekten ist sorgfältig  
zwischen Wert- und Referenzsemantik zu unterscheiden.*

Auf *konkrete Variable* wird über ihre Namen zugegriffen; der Speicherplatz für ihren Wert ist per Übersetzung durch ihre Deklaration bereitgestellt, seine Größe ist durch die Angabe ihres Typs definiert; sie werden per Wertsemantik bearbeitet.

Bei *Objekten* liegen die Dinge aber ganz anders:

Auf sie wird über Verweise zugegriffen, es muß explizit veranlaßt werden, daß Speicherplatz für sie angelegt wird; und da man an die Werte der „Variablen“ über ihre „Namen“ nicht direkt herankommt, ist Referenzsemantik in der Regel (mit wenigen Ausnahmen unter engen Voraussetzungen) ein untaugliches Mittel zu ihrer Bearbeitung.

Um Objekten mit Wertsemantik beizukommen, d. h. die Effekte zu erzielen, die bei konkreten Variablen qua Wertsemantik gegeben sind, werden – in Anlehnung an die auf S. 15 genannten Standardoperationen – explizite Operationen zu folgenden Zwecken benötigt:

- Erzeugung von Objekten (mit der Reservierung von Speicherplatz für sie)
- und ihrer Vernichtung (durch die Freigabe des für sie reservierten Speicherplatzes);
- Leerung von Objekten, d. h. der Löschung ihrer Werte,
- und Überprüfung von Objekten darauf, ob sie leer sind;
- Herstellung von Kopien von Objekten
- und Überprüfung auf Übereinstimmung;
- Vergleich von Objekten bezüglich einer Ordnungsrelation;
- Darstellung von Objekten auf dem Bildschirm
- und ihrer interaktiven (per Tastatur, Maus o. ä.) Veränderbarkeit;
- Verwandlung von Objekten in serielle Bytefolgen und umgekehrt (ggf. unter Einfügung von Redundanz zur Fehlererkennung oder -korrektur), um sie z. B. in polymorphen Objektmengen temporär im Arbeitsspeicher oder persistent auf peripheren Datenspeichern abzulegen oder an Prozesse auf anderen Rechnern zu senden.

## *Die Leistungen des Modularen Universums im Überblick*

das  $\text{ModUlU}^{\text{Sum}}$  bietet eine große Vielfalt abstrakter Datentypen, die mit Realisierungen aller im vorigen Abschnitt postulierten Standardoperationen ausgestattet sind.

Die Komponenten liegen in Form von – in Modula-2 geschriebenen – Moduln vor, die *grundsätzlich* in Spezifikation und Implementierung getrennt sind.

Einigen typischen Moduln von  $\text{ModUlU}^{\text{Sum}}$  sind eigene Abschnitte gewidmet, in denen sie erläutert werden.

Zur Realisierung der o.g. Standardoperationen werden in allen Komponenten von  $\text{ModUlU}^{\text{Sum}}$ , die abstrakte *Datentypen* darstellen, die erforderlichen Prozeduren unter den Namen

- initialisieren und terminieren,
- leer,
- leeren,
- kopieren,
- gleich,
- kleiner,
- ausgeben und editieren,
- Codelaenge, codieren und decodieren

zur Verfügung gestellt.

### *Standardprozeduren in den Moduln von Murus*

#### **initialisieren**

*Genau das* ist der Zweck der Prozeduren *initialisieren*, deren Syntax für einen abstrakten Datentyp TYPE Objekte; im Prinzip wie folgt aussieht:

```
PROCEDURE initialisieren (VAR x: Objekte);
```

Im zugehörigen Implementierungsmodul beginnt die Konstruktion der Prozedur – auf der Grundlage von Typdeklarationen der Art

```
TYPE X = ... ; Objekte = POINTER TO X;
```

– immer mit der Anweisung

NEW(x),

synonym zu

ALLOCATE(x, TSIZE(X))

mit ALLOCATE aus dem Modul `Storage`, was die Bereitstellung des Speicherplatzes für die zu `x` assoziierte *dereferenzierte Variable*  $\hat{x}$  vom Typ `X` und den Eintrag der Adresse dieses Speicherplatzes in den Wert von `x` (vom Typ `POINTER TO X`) bewirkt.

In der Implementierung von `initialisieren` wird anschließend die dereferenzierte Variable  $\hat{x}$  auf einen bestimmten initialen Wert gesetzt, der als *leer* interpretiert wird (s. Abschnitt `leer`).

Der *Variablen-Parameter* beim `initialisieren` ist essentiell:

Die beim Aufruf übergebene Variable wird in der Implementierung durch die Anweisung `NEW` als Zeiger auf den gelieferten Speicherplatz zurückgegeben, also *verändert*.

Im Unterschied dazu wird dieser Verweis *allen anderen* Prozeduren (mit Ausnahme von `terminieren`) nur als *Wertparameter* übergeben (Änderungen an einem Objekt betreffen die dereferenzierte Variable, nicht den Verweis!).

Hier sieht man, warum es z. B. in Java keine Variablenparameter gibt: Würde man die *Aktionsprozedur* `initialisieren` durch eine *Funktionsprozedur* `PROCEDURE neuesObjekt(): Objekte;` ersetzen, könnte man auch in Modula-2 in diesem Kontext auf sie verzichten (eine Operation wie `terminieren` ist in Java wegen des eingebauten Speicherbereinigungsmechanismus nicht erforderlich).

Der Aufruf von Prozeduren, die Objekte als formale Parameter enthalten, *setzt grundsätzlich voraus, daß alle Objekte*, die ihnen als aktuelle Parameter übergeben werden, *initialisiert sind* (durch den vorherigen Aufruf der Prozedur `initialisieren`).

Diese Voraussetzung ist *unabdingbar*:

Der für ein Objekt qua Deklaration `VAR x: Objekte;` reservierte Speicherplatz hat anfangs – genau wie im konkreten Fall – keinen definierten Wert, enthält also eine *zufällige Adresse*.

Ein Versuch, schreibend auf die Variable zuzugreifen – hinter den Kulissen z. B. mit einer Zuweisung  $\hat{x} := \dots$  – bedeutet somit fast immer einen Schreibzugriff auf einen Adreßbereich im Arbeitsspeicher, die dem aufrufenden Prozeß vom Betriebssystem nicht „zugewiesen“

wurde und deshalb beim Aufruf mit einem Programmabbruch unter Meldung eines *Speicherzugriffsfehlers* („*segmentation fault*“) quittiert wird.

### **terminieren**

Bei dieser Prozedur handelt es sich – grob gesprochen – um das Gegenstück zu **initialisieren**.

Da Modula-2 keine automatische Speicherbereinigung („*garbage collection*“) vorsieht, sind die Entwickler/innen von Programmen oder Komponenten dafür verantwortlich, daß der Arbeitsspeicher im Laufe der Programmausführung nicht – durch ständiges Verbrauchen von Speicherplatz qua Aufruf von **initialisieren** – „zuläuft“.

Es ist daher mindestens eine Frage guten Programmierstils, den bei der Initialisierung von Objekten belegten Speicherbereich wieder freizugeben, wenn sie nicht mehr gebraucht werden.

Das geschieht durch Aufruf einer Prozedur **terminieren** mit der Syntax

```
PROCEDURE terminieren (VAR x: Objekte);
```

deren Implementierung immer mit einer Speicherbereinigung endet:

```
DISPOSE(x),
```

synonym zu

```
DEALLOCATE(x, TSIZE(X))
```

mit **DEALLOCATE** aus dem Modul **Storage**,

### **leer, leeren**

Es ist grundsätzlich sinnvoll, die „Wertemenge“ von Variablen aus abstrakten Datentypen um einen Wert „leer“ zu „augmentieren“, d. h. *leere Objekte* zuzulassen. Sie können als „undefiniert“, „unbekannt“ o. ä. – je nach Semantik der betreffenden Komponente – interpretiert werden; in der Ausgabe wird ein solcher Wert in der Regel durch eine leere Zeichenkette repräsentiert.

Damit wird auch die *Eingabe neuer Objekte* unter dem Konzept des „*Editierens*“ (Veränderns) von Werten subsumiert, indem z. B. die leere Zeichenkette überschrieben wird.

Der Überprüfung, ob eine Variable  $x$  eines abstrakten Datentyps *Objekte* einen *leeren Wert* hat, dient die Prozedur mit der Syntax

```
PROCEDURE leer (x: Objekte): BOOLEAN;
```

dem Überschreiben des Wertes einer Variablen mit dem *leeren Wert* die Prozedur mit der Syntax

```
PROCEDURE leeren (x: Objekte);
```

Beim *initialisieren* sollte jede Variable grundsätzlich „geleert“ werden und damit das Prädikat *leer* erfüllen.

### **kopieren**

Ein „naives“ Kopieren eines Objektes  $y$  auf ein Objekt  $x$  mit der Anweisung

```
x := y;
```

ist in aller Regel (von der es unter ganz bestimmten engen Voraussetzungen in Einzelfällen begründete Ausnahmen geben mag) *überhaupt nicht* sinnvoll; sie bewirkt nämlich, wie im Abschnitt über *Wert- versus Referenzsemantik* erläutert, daß jede Veränderung des *eigentlichen* Wertes von  $y$  unmittelbar auf  $x$  „durchschlägt“ und umgekehrt.

Der *eigentliche* Wert von  $y$  muß dagegen im einfachsten Fall mit

```
x^ := y^;
```

kopiert werden.

Diese Anweisung stellt jedoch keinen für einen Klienten zulässigen Quelltext dar, weil in ihr auf die interne Repräsentation des Typs der Variablen Bezug genommen wird – nur in der Implementierung ist bekannt, daß  $x$  und  $y$  Variablen vom Typ `POINTER TO ...` sind, sodaß die Variablen  $x^$  und  $y^$  definiert sind.

In der Spezifikation ist dieses Implementierungsdetail nicht sichtbar, folglich im Klientenmodul nicht verwendbar; deshalb würde sie vom Übersetzer auch mit einer Fehlermeldung quittiert.

Konsequenz ist die Bereitstellung des Konzepts des „tiefen Kopierens“ von Objekten mit Hilfe von Prozeduren mit der Syntax

```
PROCEDURE kopieren (x, y: Objekte);
```

deren Implementierung dann im einfachsten Fall wie oben angegeben aussieht (in verschachtelten Fällen anspruchsvoller).

**gleich**

Ein naiver Versuch, die „eentlichen“ Werte zweier Objekte  $x$  und  $y$  mit dem Booleschen Ausdruck

$$x = y$$

auf Gleichheit zu überprüfen, ist in der Regel genauso unsinnig wie die Zuweisung  $x := y$  beim Kopieren:

Sie sagt lediglich etwas darüber aus, ob die Zeiger  $x$  und  $y$  auf die gleiche Adresse im Arbeitsspeicher verweisen, nicht jedoch, ob die „eentlichen“ Werte der entsprechenden Variablen übereinstimmen.

Die Prüfung auf Gleichheit müßte im einfachsten Fall mit dem Booleschen Ausdruck

$$x^{\wedge} = y^{\wedge}$$

erfolgen, der jedoch – aus den beim kopieren genannten Gründen – nicht zulässig ist und deshalb in einer Prozedur mit der Syntax

```
PROCEDURE gleich (x, y: Objekte): BOOLEAN;
```

gekapselt werden muß.

**kleiner, kleinergleich**

Auch die Idee, die „eentlichen“ Werte zweier Variablen  $x$  und  $y$  eines abstrakten Datentyps, auf dem eine strikte Ordnung  $<$  definiert ist, mit dem Booleschen Ausdruck

$$x < y$$

daraufhin zu überprüfen, ob der Wert der ersten kleiner als der zweite ist, geht *völlig daneben*:

Er liefert das nichtssagende Ergebnis, ob die erste Variable im Arbeitsspeicher an einer kleineren Adresse liegt, als die zweite.

In vollständiger Analogie zu dem bei **gleich** gesagten muß die Überprüfung in einer Prozedur mit der Syntax

```
PROCEDURE kleiner (x, y: Objekte): BOOLEAN;
```

gekapselt werden, in deren Implementierung im einfachsten Fall die Variablen  $x^{\wedge}$  und  $y^{\wedge}$  verglichen werden.

Entsprechendes gilt für eine nicht-strikte Ordnung  $\leq$ , gekapselt in einer Prozedur

```
PROCEDURE kleinergleich (x, y: Objekte): BOOLEAN;
```

**ausgeben, editieren**

Der Ausgabe von Objekten und ihrer interaktiven Veränderung, was bei Variablen konkreter Datentypen mit den `Write/-Read...`-Prozeduren aus dem Modul `InOut` geschieht, dienen die Prozeduren mit der Syntax

```
PROCEDURE ausgeben (x: Objekte; ...);
PROCEDURE editieren (x: Objekte; ...);
```

wobei in der Regel in den Parametern ... die Positionierung auf dem Bildschirm bzw. innerhalb eines Fensters festgelegt wird.

Die Details werden in den jeweiligen Moduln festgelegt, da sie hochgradig von der Semantik der betreffenden Komponente abhängen.

**Codelaenge**

Die Typgröße von Variablen konkreter Datentypen ist bekannt, wie im entsprechenden Abschnitt dargelegt. Bei abstrakten Datentypen liegen die Dinge anders:

Ihre *eigentliche Typgröße* ist nur in der *Implementierung* durch die Festlegung der Repräsentation des Typs bekannt und muß Klienten deshalb vermöge einer Prozedur mit der Syntax

```
PROCEDURE Codelaenge (x: Objekte): CARDINAL;
```

bekanntgegeben werden.

Im einfachsten Fall von Verweisen auf konkrete Datentypen, d. h. falls `Objekte = POINTER TO X` für einen konkreten Datentyp `X` ist, ist ihre Implementierung durch `TSIZE(X)` gegeben; allgemein durch die Anzahl der Bytes, die zu einer eindeutig umkehrbaren Darstellung von `x` als *zusammenhängende Folge von Bytes* erforderlich ist.

Dieses Konzept ist so fundamental, daß wir dafür (den schon von WIRTH benutzten) Begriff

*Ströme*

dafür gebrauchen. Ströme im Arbeitsspeicher sind entweder durch ihre *Start- und Endadresse* oder durch ihre *Startadresse und Länge* gegeben.

**codieren**

Die Werte von Objekten werden z. B. auf peripheren Datenträgern als *Ströme* abgelegt oder beim Botschaftenaustausch oder bei Fernaufrufen über Netzverbindungen als *Ströme* an Prozesse auf anderen Rechnern gesendet: Es handelt sich bei ihnen in diesen Fällen um nichts anderes als *serielle Folgen von Bytes*, deren Interpretation als Werte irgendwelcher strukturierter Variablen auf der Maschinenebene nicht möglich ist.

Zu diesem Zweck muß der *eigentliche Wert* einer Variablen eines abstrakten Datentyps – unabhängig davon, wie komplex ihre Struktur ist – „serialisierbar“, d. h. in einen *Strom* zu verwandeln sein, und „deserialisierbar“, d. h. umkehrbar eindeutig aus einem Strom, in den er serialisiert wurde, wieder herstellbar sein.

Für eine Variable *x* eines konkreten Datentyps *X* ist das Problem dadurch gelöst, daß sie im Arbeitsspeicher als Strom der Länge  $TSIZE(X)$  ab der Adresse  $ADR(x)$  abgelegt ist. Die Variable gesondert zu serialisieren, bedeutet also nichts weiter, als diesen Strom byteweise zu kopieren, was sich mit elementaren Methoden erledigen läßt.

Mit Objekten, d. h. Variablen eines abstrakten Datentyps, geht das nun allerdings nicht so einfach:

Ein Klient weiß weder, wo die Startadresse des *eigentlichen Wertes* der Variablen liegt (was ihm auch nichts nützen würde, da dieser Wert auch *fragmentiert* im Arbeitsspeicher liegen kann, also keinen Strom darstellen muß), noch wieviel Platz für ihn gebraucht wird. Folglich muß für die Serialisierung eine Prozedur mit der Syntax

```
PROCEDURE codieren (x: Objekte; A: ADDRESS);
```

bereitgestellt werden, deren Implementierung aus den notwendigen Kopieroperationen in einen Pufferbereich ab der Adresse *A* besteht.

Entsprechend dem Hinweis zur Notwendigkeit der Initialisierung (s. *initialisieren*) setzt das natürlich voraus, daß dieser Pufferbereich vom Klienten per  $ALLOCATE(A, CodeLaenge(x))$  bereitgestellt wurde. Falls das nicht geschehen ist, fängt man sich zur Laufzeit wegen eines unzulässigen Schreibzugriffs in einen nicht freigegebenen Speicherbereich einen Speicherzugriffsfehler („*segmentation fault*“) ein.

**decodieren**

Der umgekehrte Vorgang – die Rückverwandlung eines Stroms in ein Objekt – wird – ganz analog – durch eine Prozedur mit der Syntax

```
PROCEDURE decodieren (x: Objekte; A: ADDRESS);
```

erledigt, was natürlich voraussetzt, daß der Strom ab Adresse A der Codelänge von x ein codiertes Objekt vom Typ `Objekte` darstellt.

Eine Schwäche dieses Konzeptes ist, daß eine Typüberprüfung bei der Übersetzung nicht möglich ist, sondern nur zur Laufzeit.

(Bei genügend „raffinierter“ Implementierung wäre auch *das* möglich – aber  $\text{dasModell}_{\text{Univer}}^{\text{Ula}} \text{Sum}$  ist (*noch*) nicht soweit.)

Gegen diesen Einwand hilft nur die ganz pragmatische Erwägung, daß ein Klient, der bei der Entwicklung einer Komponente gewisse Objekte in von ihm extra bereitgestellten Pufferbereichen serialisiert abgelegt hat, wohl kaum auf die (Schnaps-)Idee kommen dürfte, in der gleichen Komponente aus den Pufferbereichen plötzlich Objekte eines anderen Typs holen zu wollen . . .

## *Die Komponenten von Murus im Überblick*

Wir geben jetzt einige Beispiele für Komponenten in Form abstrakter Datentypen, mit denen  $\text{dasModul}^{\text{Ulla}}_{\text{objekt}}^{\text{Sum}}$  die verwendete Programmiersprache Modula-2 erweitert:

- Peripheriegeräte (nur als Datenobjekte) wie
  - Tastatur,
  - Bildschirm
  - und Maus (Mausknöpfe sind im Grunde Bestandteile der Tastatur, die Position der Maus dagegen des Bildschirms),
  - Drucker,
  - Dateisystem,
 (entgegen der einheitlichen Auffassung aus dem Ansatz in UNIX, sämtliche Geräte als Dateien zu betrachten),
- Felder mit Attributen (Farben, Fonts) und Editieroperationen zur Ausgabe auf bzw. Eingabe mit Peripheriegeräten,
- komfortable Versionen von
  - Wahrheitswerten, Zeichen, Zeichenketten usw. mit zugehörigen Ein-/Ausgaberoutinen,
- Zahlen höherer Genauigkeit, als üblicherweise in Programmiersprachen vorgesehen, mindestens
  - natürliche oder ganze Zahlen
 sowie Zahlen, die in der Regel nicht Bestandteile von Programmiersprachen sind, wie
  - rationale und komplexe Zahlen,
  - Zufallszahlen
- beschränkte und unbeschränkte Folgen von Objekten
  - gleicher oder unterschiedlicher Größe (mit und ohne Positionsverwaltung)
- und deren Standardanwendungen wie z. B.
  - Stapel und
  - Warteschlangen,
- sowohl
  - linear als auch
  - partiell geordnete
 Mengen von Objekten, teils mit speziellen Eigenschaften,

- Graphen,
- persistente
  - Folgen (sequentielle Dateien) und
  - geordnete Mengen ohne (B-Bäume) und
  - mit Index (indexsequentielle Dateien, B\*-Bäume),
- häufig gebrauchte „Allerwelts“-Objekte wie z. B.
  - Uhrzeiten, Kalenderdaten und Geldbeträge,
- Bestandteile der „Stammdaten“ von Personen, z. B.
  - Namen, m/w-Angaben, Anreden,
- Angaben für Anschriften usw.
  - Straßen, Postleitzahlen, Orte, Telefonnummern,
- Objekte zur Erfassung der Stammdaten von Schülern
  - Staaten, Religionen, Sprachenfolgen
- und sonstige im Schulkontext verwendbare Objekte, wie z. B.
  - Noten, Klassenstufen, Schulhalbjahre, Schulfächer, Kursnummern,
- komplexere Objekte höherer Abstraktionsstufe, wie z. B.
  - allgemeine Aufzähltypen,
  - elementargeometrische Figuren,
- Durchgriffe auf tiefliegende Leistungen der Basismaschine,
- für die nichtsequentielle Programmierung benötigte Objekte wie
  - Prozesse und Semaphore
- sowie Entwurfsmuster wie
  - kritische Abschnitte, Barrieren, Monitore,
- und Konstrukte
  - zum Botschaftenaustausch oder für Fernaufrufe,
- mehrprozeßfähige Versionen vieler der weiter oben genannten Objekte.

## BENUTZEROBERFLÄCHE

### *Grundsätzliches zu Benutzeroberflächen*

Die Aus- und Eingaberoutinen `Write...` und `Read...` aus dem Modul `InOut`, der zwar nicht Sprachbestandteil von Modula-2, aber durch den Vorschlag von N. WIRTH „Quasistandard“ ist, erlauben – ähnlich wie die entsprechenden Routinen anderer Programmiersprachen – lediglich die interaktive Steuerung anspruchsloser Testprogramme, bei denen die Gestaltung des Bildschirms keine Rolle spielt: Mit ihnen lassen sich Daten zeilenweise – ohne jeden Eingabekomfort – eingeben und die Ausgaben zeilenweise über den Bildschirm „rollen“.

Alles, was über diese primitive Form der Aus- und Eingabe hinausgeht, wie etwa

- Ereignissteuerung,
- Konstruktion von Bildschirmmasken,
- Menüs,
- graphische Ausgaben

ist mit beträchtlichem Entwicklungsaufwand verbunden, der für Lehrzwecke kaum vertretbar ist, weil dabei – mit wenig informatischem Gehalt – anspruchsvollen Lehraufgaben wertvolle Zeit geraubt wird.

Insbesondere stellt jede Form ernsthafter Arbeit unter graphischen Oberflächen erfahrungsgemäß einen Aufwand dar, der von den eigentlichen Aufgaben ablenkt und in der Regel alle Beteiligten überfordert.

Die „Programmerstellung“ in einer Lehrsituation landet deshalb früher oder später bei einem recht widersprüchlichen Problem:

Im fortgeschrittenen Stadium der Lehre – etwa bei der Arbeit an einem „Softwareprojekt“ (s. Abschnitt *Anforderungsdefinition* im Heft über  $\text{Objekt-Entwicklung}$ ) – ist eine zeilenorientierte Aus- und Eingabe von Texten völlig unbrauchbar, andererseits die Entwicklung einer graphischen Benutzeroberfläche nur sehr eingeschränkt möglich – wenn überhaupt, dann nur unter Einsatz entsprechender geeigneter „Klick“-Werkzeuge.

Für die Erstellung etwas anspruchsvollerer Programme ergibt sich daher nahezu zwangsläufig die Forderung nach einem Kompromiß in Form eines geeignet *didaktisch reduzierten* Ein-/Ausgabe-Konzepts, das zwar sowohl geschützte Eingabebereiche als auch die Möglichkeit

graphischer Ausgaben sowie den Einsatz einer Maus umfaßt – sich dabei jedoch nicht auf Berge teils extrem komplexer, mitunter sehr unübersichtlicher – Schnittstellen im industriellen Maßstab stützt (die Dokumentation der Bibliotheken, die z. B. zu `/usr/lib/libX11` oder `javax.swing` gehören, füllt mehrere Bücher), sondern diese äußerst komplizierte Materie hinter wenigen klar verständlichen und einfach benutzbaren Schnittstellen mit genügend (evtl. hoch-)leistungsfähigen Implementierungen „versteckt“.

*Genau das* leistet das  $\text{Modell}_{\text{UJ}}^{\text{Ula-Sum}}$  unter anderem, wie in den folgenden Abschnitten gezeigt wird.

### ***Trennung von Eingabe und Ausgabe***

Aus- und Eingabe werden konzeptionell grundsätzlich getrennt (was im Heft über die  $\text{Objekt-Entwicklung}_{\text{basierte}}^{\text{Ent-}}^{\text{lung}}$  ausführlich begründet ist).

Sie erfordern eigene Moduln, deren Bestandteile in Komponenten, bei denen es um die Realisierung algorithmischer Konzepte geht, den sogenannten „Fachkonzepten“, nichts zu suchen haben. (Seit Urzeiten ist das für die ingenieurmäßige Softwareentwicklung selbstverständlich – heute werden die ollen Kamellen unter „coolen“ Namen als „neu“ verkauft: „MVC“-Paradigma – „Model“, „View“, „Controller“.)

Insbesondere sind *Tastatur* und *Bildschirm* auseinanderzuhalten. (Das Prinzip von Unix, sämtliche Aus- und Eingaben in dem gemeinsamen Konzept der Dateien zu bündeln, hat damit nichts zu tun.)

das  $\text{Modell}_{\text{UJ}}^{\text{Ula-Sum}}$  enthält deshalb u. a. Moduln

- zur Ereignissteuerung
  - **Tastatur** und (dahinter versteckt) **Maus** („Controller“),
- zur komfortablen Aus- und Eingabe von Zeichenfolgen (Texten, Zahlen) und zur Erzeugung und Ausgabe graphischer Objekte (Strecken, Polygone, Kreise, Bilder usw.)
  - **Farben** und **Bildschirm** („View“),
- und zur Erzeugung von Fehlermeldungen und Hinweisen und für Menüs
  - **Meldungen, Auswahlen und Aktionen.**

Die Semantik dieser Moduln wird in den folgenden Abschnitten detailliert erläutert.

## *Tastatur und Maus*

Zur Bedienung und Steuerung eines Systems mit Tastatur und Maus müssen drei Gruppen von *Tasten* unterschieden werden:

- zur Eingabe von Zeichenketten die *Zeichentasten*,
  - die ein Echo in Form eines (fast immer) sichtbaren Zeichens auf dem Bildschirm produzieren (wobei stillschweigend unterstellt wird, daß alle Zeichen, die auf der Tastatur abgebildet sind, auch *so* auf dem Bildschirm dargestellt werden),
- zur Auslösung beabsichtigter Systemreaktionen die *Kommandotasten*
  - zur Korrektur von Zeichenketten während der Eingabe,
  - zum Abschluß von Eingaben und damit zur Systemsteuerung,
  - zur Bestätigung oder Ablehnung oder zur Auswahl von Vorschlägen des Systems,
  - zum Durchblättern von Informationsmengen und zur Auswahl von Informationen aus ihnen,
  - zur Auslösung sonstiger Systemfunktionen und
- zum Aufsuchen bestimmter Bildschirmstellen die *Maustasten*
  - zum Markieren von Daten oder Objekten, die z. B. kopiert oder entfernt werden sollen,
  - zum Verschieben von Objekten,
  - für „graphische Aktionen“ wie z. B. Setzen von Punkten oder Strecken oder zur Konstruktion komplexerer Objekte.

das Ulla Sum Modelliver Modelliver kapselt den Zugriff auf Tastatur und Maus in einem *abstrakten Datenobjekt*, dem Modul *Tastatur*, dessen Konstruktion den Modul *Maus* benutzt.

Für die *Zeichentasten* steht der mittlere Tastenblock – die alphanumerische Tastatur – des Rechners zur Verfügung; Zeichen, für die keine einzelne Taste vorgesehen ist, werden durch Kombination mit einer *Umschalttaste* ↑ (für Großbuchstaben), einer *Steuerungstaste* Strg oder einer *Metataste* Alt oder Alt Gr erreicht.

Darüberhinaus gibt es keinen einheitlichen Standard. Um von den Gegebenheiten konkreter Tastaturen oder Mäuse zu abstrahieren, werden die folgenden *Kommandos* vorgesehen:

- für die Bestätigung oder Ablehnung von Aktionen, für die Ansteuerung von bestimmten Bildschirmstellen und die Bewegungen im System in verschiedenen Richtungen die Kommandos
  - [weiter], [schluss], [zurück],
  - [nach links], [nach rechts], [nach oben], [nach unten],
  - [zum Anfang], [zum Ende], [schalte],
- zum Löschen, Ändern und Einfügen von Daten die Kommandos
  - [entferne], [füge ein],
  - [markiere], [entferne Markierung],
- zur Ansteuerung von Stellen auf dem Bildschirm mit einem Zeigeelement (Maus oder - bei Tastbildschirmen - Finger) die Kommandos
  - [hier], [hierhin], [dort], [dorthin],
  - [gehe], [ziehe], [schiebe] und
- für sonstige Systemfunktionen weitere Kommandos wie z. B.
  - [hilf], [wähle aus], [drucke] usw.

Zur Realisierung der Kommandos auf einer Tastatur oder Maus verwendet das Modell <sup>Ula</sup> <sub>Modell</sub> <sup>Sum</sup> <sub>Univer</sub> folgende Tasten, deren Semantik weitestgehend systemunabhängig ist:

- die *Eingabetaste*  $\leftarrow$  für [weiter],
- die *Schlusstaste* Esc für [schluss],
- die *Rücktaste*  $\leftarrow$  für [zurück],
- die *Pfeiltasten*  $\leftarrow$ ,  $\rightarrow$ ,  $\uparrow$  und  $\downarrow$  für [nach links], [nach rechts], [nach oben] und [nach unten];
- die *Positionierungstasten* Pos1 bzw. Ende für [zum Anfang] bzw. [zum Ende] und die *Tabulatortaste*  $\Leftrightarrow$  für [schalte],
- das *Drücken* der linken bzw. rechten *Maustaste* für [hier] bzw. [hierhin] und das *Loslassen* für bzw. [dort], [dorthin],
- die *Bewegung der Maus* bei *losgelassenen Maustasten* für [gehe] und bei *gedrückter* linker bzw. rechter *Maustaste* für [ziehe] bzw. [schiebe],
- die *Löschtaste* Entf für [entferne] und die *Einfügetaste* Einfg für [füge ein],
- die *Funktionstasten* F1, F2, ... und sonstige spezielle Tasten für andere Kommandos.

Diese Kommandos können in der „*Tiefe*“ ihrer Wirkung durch Kombination mit geeigneten Vorwahltasten, z. B. den *Umschalttasten*, den *Kontrolltasten* **Strg** und den *Metatasten* **Alt** und **Alt Gr** verstärkt werden; zu jedem Kommando gehört eine natürliche Zahl als Tiefe (0 als Basisversion, zunehmende Zahlen für größere Tiefen).

Damit sind prinzipiell wirkungsgleiche Kommandos unterschiedlicher Tiefe in Systemen möglich, wie z. B. die Bewegung in

- einem Text zum nächsten Zeichen, Wort, Satz, Abschnitt oder Kapitel,
- einem Kalender zum nächsten Tag, zur nächsten Woche, zum nächsten Monat, Jahr oder Jahrzehnt.

Als Realisierung der vertieften Kommandos wird bei das Modell  $\text{Modell}_{\text{Ua}}^{\text{Sum}}$  der bloßen *Kommandotaste* die Tiefe 0 zugeordnet und ihrer Kombination mit der *Umschalttaste*  $\uparrow$  oder *Kontrolltaste* **Strg** die Tiefe 1, mit der *Metataste* **Alt** die Tiefe 2 und der Kombination  $\uparrow + \text{Alt}$  die Tiefe 3.

Der Grund für die Entscheidung für die Wirkungsgleichheit der *Umschalttasten*  $\uparrow$  mit den *Steuerungstasten* **Strg** ist ein *ergonomischer Gesichtspunkt*:

Zur Eingabe eines Textes mit der alphanumerischen Tastatur sind – speziell bei routinierten Zehnfinger-„Blind“-Schreiber(inne)n – die *Umschalttasten* optimal geeignet, für die Kombination mit der Maus (Bildschirm im Blickfeld, nicht die Tastatur!) aufgrund ihrer peripheren Lage besser die *Steuerungstasten*.

In der *Implementierung* des Moduls **Tastatur** müssen drei Fälle völlig unterschiedlich behandelt werden, weil sie auf verschiedenen maschinennahen Systemfunktionen bzw. Bibliotheken aufsetzen (z. B. `ioctl`-Aufrufe, `kbd`-Durchgriffe, **X-Windows**):

Der Aufruf in einer *Textkonsole* (auch beim Login auf einem fernen Rechner), in einer *graphikfähigen* Konsole auf dem lokalen Rechner, oder *unter X*.

## ***Bildschirm***

Unter „Bildschirm“ wird hier immer der physikalische Bildschirm mit einer bestimmten Auflösung verstanden, der – in Abhängigkeit von der vorhandenen Technik, d. h. Graphikkarte und Monitor – im Betrieb in einer *tty-Konsole* ansprechbar ist, oder ein Fenster auf einer graphischen Oberfläche. das Modelluniversum kapselt den Zugriff auf ihn in dem *abstrakten Datenobjekt Bildschirm*.

Seine Implementierung unterscheidet drei Fälle:

- *Textbildschirme* (Einsatz der Terminal-Funktionen),
- *graphikfähige Bildschirme* („*Framebuffer*“-Programmierung) und
- *Fenster unter X* (Rückgriff auf die X11-Bibliothek).

## ***Farben***

Die Definitionen von Farben und eine Reihe von Routinen zu ihrer Manipulation sind in dem (*ausnahmsweise!*) konkreten Datentyp *Farben* gekapselt.

### ***Schrift- und Hintergrundfarben***

Es gibt zwei globale Variablen für die Schrift- und die Hintergrundfarbe des Bildschirms; standardmäßig auf weiß/schwarz eingestellt.

Die Ausgaben erfolgen in den (standardmäßig auf die Bildschirmfarben eingestellten) *aktuellen* Farben, die geändert und abgefragt werden können.

### ***Modi und Bildschirmgröße***

Mit Blick auf die gängigen technischen Standards werden *Modi* zum Betrieb unterschieden.

Im Rahmen der verfügbaren Maximalleistung des vorhandenen Rechners können einige von ihnen, und zwar diejenigen, die vom VESA-Modus der vorhandenen Graphikhardware unterstützt werden, über die entsprechenden Einstellungen in der *boot*-Prozedur des Rechners eingestellt werden.

Voreinstellung ist im Konsolenbetrieb diejenige, unter der Linux in *boot/grub/menu.lst* gestartet wird (s. *Konsolenbetrieb*, S. 2); im Betrieb unter X *VGA*.

Folgende Modi sind einstellbar:

Modus	vga =	Zeilen×Spalten im	
		Textmodus	Graphikmodus
qCGA		7×20	160×120
qTxt		12×40	320×200
CGA		15×40	320×240
Txt	768, normal	25×80	640×400
VGA	769, 785..6	30×80	640×480
PAL		36×96	768×576
WVGA		30×100	800×480
SVGA	771, 788..9	37×100	800×600
WPAL		36×128	1024×576
WSVGA		37×128	1024×600
XGA	773, 791..2	48×128	1024×768
WXGA	864, 865	50×160	1280×800
SXGA	775, 794..5	64×160	1280×1024
SXGAp		65×175	1400×1050
WXGAp	868, 869	56×180	1440×900
WXGApp		56×200	1600×900
WSXGA		64×200	1600×1024
UXGA	837, 842	75×200	1600×1200
WSXGAp	872, 873	65×210	1680×1050
HDMI		67×240	1920×1080
WUXGA	892, 893	75×240	1920×1200
SUXGA		90×240	1920×1440
QXGA	---, 850	96×256	2048×1536
WQXGA		100×320	2560×1600
QSXGA		128×320	2560×2048
QUXGA		150×400	3200×2400

Die Größenangaben zum Textmodus beziehen sich dabei auf die Schriftgröße 8×16 (s. u.).

Bildschirm verfügt über diverse Prozeduren zum Abfragen und Schalten des Modus und über Funktionen, die die Kenngrößen aus der obigen Tabelle liefern.

### *Manipulation von Bildschirmbereichen*

Es sind Routinen zur Löschung, Invertierung, Archivierung und Restaurierung von rechteckigen Bereichen des Bildschirms vorhanden.

### *Kursor*

Zur Eingabe von Texten gibt es drei Formen des Cursors (unsichtbar, Unterstrich- oder Blockkursor). Der Kursor kann in einer dieser Formen auf bestimmte Bildschirmpositionen gesetzt werden.

### *Ausgabe von Texten*

Der Bildschirm ist gerastert: in Textzeilen und -spalten und in Pixelspalten und -zeilen (siehe obige Tabelle).

Im Rahmen beider Rasterungen können einzelne Zeichen, Texte (innerhalb einer Bildschirmzeile) und Zahlen (vom Typ `CARDINAL`) an bestimmte Bildschirmpositionen geschrieben werden. Die *Koordinaten* sind dabei Paare der Form (`Zeile, Spalte`) mit Werten zwischen 0 und (`Zeilen/-Spaltenzahl() - 1`).

### *Schriftart*

Es sind vier Schriftgrößen (fester Fontbreite) einstellbar:  $6 \times 10$ ,  $8 \times 16$ ,  $12 \times 24$  und  $16 \times 32$ . Voreingestellt ist  $8 \times 16$ , was der Standardauflösung im Konsolenbetrieb entspricht.

Es werden die `Terminus`-Fonts (s. *Voraussetzungen*, S. 3) verwendet, die nicht proportional sind, was für die o. g. Rasterung notwendig ist.

### *Ausgaben graphischer Objekte*

Ein ganz entscheidender Beitrag zur genannten Forderung nach didaktischer Reduktion ist die *Vereinheitlichung der Ausgabe von Text und Graphik*.

`Bildschirm` bietet ein umfangreiches Repertoire an Routinen zur Ausgabe von

- *Punkten*,
- *Strecken, Rechtecken, Streckenzügen und Polygonen*,

- *Kreisen* und *Ellipsen* sowie
- *Bezierkurven* (bis zum Grad 34),  
teilweise auch
- *invers* und *gefüllt*.

### *Maus*

Bildschirm liefert Routinen zur Abfrage der Position des Mauszeigers und zu seiner Positionierung.

### *Synchronisation*

Für die Verwendung in nebenläufigen Programmen bietet Bildschirm eine Schloßvariable zur Sperrsynchronisation.

### *Serialisierung*

Bildschirm liefert Prozeduren zur Umwandlung von Graphikdateien zwischen dem ppm-Format und den zu ihrer Darstellung auf dem Bildschirm verwendeten internen Formaten (sowohl im *Framebuffer* als *unter X*).

### *Konsolenbetrieb versus Betrieb unter X*

Bildschirm erlaubt abzufragen, ob der aufrufende Prozeß in einer Konsole oder unter X, d. h. in einem Fenster einer graphischen Oberfläche, gestartet wurde.

Alle Routinen funktionieren auch auf graphischen Oberflächen, d. h. unter X, jedoch mit einer – in der Natur der Sache liegenden – Einschränkung:

Ein Fenster ist nie so groß wie der physikalisch vorhandene Bildschirm; deshalb ist es sinnvoll, Programme unter X nicht im maximal verfügbaren Modus, sondern dem „submaximalen“ laufen zu lassen. Auch sollte die Größe eines Fensters in diesem Falle nicht willkürlich – durch „Verziehen der Fensterrahmen“ mit der Maus – verändert werden.

### *Felder*

Für Ein- und Ausgaben von Zeichenketten auf dem Bildschirm werden Felder definierter Breite vorgesehen, aus denen sich die Bildschirmmasken zusammensetzen.

das Modul `Ulliver` enthält dafür den abstrakten Datentyp **Felder**.

Vorläufig liegen Felder grundsätzlich innerhalb einer Bildschirmzeile.

Jede Eingabe beginnt mit einer Ausgabe des Feldinhaltes (der auch nur aus Leerzeichen bestehen kann).

Vor der Eingabe einer Zeichenkette in einem Feld wird es mit einem definierten Inhalt (der auch nur aus Leerzeichen bestehen kann) vorbesetzt; damit muß zwischen der Neueingabe von Zeichenketten und der Änderung von Feldinhalten nicht unterschieden werden. Beim Editieren kann die im Feld ausgegebene Zeichenkette überschrieben werden, wobei ein gewisser Komfort vorgesehen ist. Nach dem Abschluß der Eingabe durch dafür vorgesehene Kommandos wird der Feldinhalt an das System übergeben, das die weitere Steuerung übernimmt.

### *Kursor*

Der Kursor ist grundsätzlich nur dann zu sehen, wenn Benutzerin die Möglichkeit zur Eingabe sichtbarer Zeichen offensteht. Er zeigt dabei die aktuelle Schreibposition an, an der die Eingabe eines Zeichens mit dessen Darstellung auf dem Bildschirm quittiert wird.

In diesem Fall wird zwischen Überschreibe- und Einfügemodus unterschieden, zwischen denen hin- und hergeschaltet werden kann. Der Kursor befindet sich zu Beginn der Eingabe am Feldanfang, bei leerem Feld im Einfüge-, sonst im Überschreibemodus.

Welcher der Modi eingeschaltet ist, ist an der Form des Cursors erkennbar: ein kleiner Kursor (Unterstrich) für den Einfügemodus, ein großer „Block“-Kursor (Rechteck in Buchstabengröße) für den Überschreibemodus.

### *Korrektur des Feldinhaltes*

Wenn die Eingabefelder vollständig *innerhalb einer Bildschirmzeile* liegen, gehören dazu folgende Grundsätze:

- Die Bildschirmspalte unmittelbar rechts hinter dem Eingabefeld kann zwar vom Cursor erreicht werden (nach Eingabe des letzten Zeichens im Feld *muß* der Cursor das Feld verlassen, um Mißverständnisse beim Editieren der letzten beiden Zeichen im Feld zu vermeiden), eine Eingabe von Zeichen ist dort aber nicht mehr möglich. (Als Folgerung ergibt sich daraus, daß kein Feld in die Spalte unmittelbar am rechten Bildschirmrand ragen darf.)
- Im Einfügemodus rücken alle Zeichen rechts vom Cursor um eins weiter nach rechts, sofern das Feld nicht schon voll ist; in diesem Fall ist weiteres Einfügen nicht mehr möglich. (Damit soll vermieden werden, daß Zeichen am rechten Feldrand vom Nutzer unbemerkt verschwinden, was leicht zu Fehleingaben führen kann.)

Für die Korrekturmöglichkeiten während der Eingabe sind folgende Kommandos vorgesehen (ggf. der Tiefe 1, d. h. Kommandotaste in Kombination mit einer Umschalttaste  $\uparrow$  oder Kontrolltaste **Strg** - zu ihrer Abbildung auf die Tastatur s. Abschnitt über die *Tastatur*, S. 32:

- [nach rechts]: Der Cursor springt ein Zeichen weiter nach rechts, wenn er nicht schon das Feld verlassen hat.
- [nach links]: Der Cursor springt ein Zeichen weiter nach links, wenn er nicht schon am Anfang des Feldes ist.
- [zum Anfang]: Der Cursor springt an den Feldanfang.
- [zum Ende]: Der Cursor springt hinter das letzte nichtleere Zeichen des Feldes (ggf. auf die Position hinter dem Feld).
- [entferne]: Das Zeichen, auf dem der Cursor steht, wird gelöscht; der Feldinhalt rechts von der Cursorposition rückt geschlossen ein Zeichen weiter nach links, wobei das letzte Zeichen des Feldes durch ein Leerzeichen aufgefüllt wird.
- [zurück]: Wenn der Cursor nicht schon am Feldanfang steht, wird das Zeichen links vom Cursor gelöscht und der Cursor springt ein Zeichen nach links und zieht den Feldinhalt rechts von sich nach, wobei das letzte Zeichen des Feldes durch ein Leerzeichen aufgefüllt wird.

- [zurück] [1]: Das ganze Feld wird gelöscht und der Cursor springt an den Feldanfang,
- [füge ein]: Der aktuelle Modus wird gewechselt (vom Einfüge- in den Überschreibemodus oder umgekehrt).
- [füge ein] [1]: Der Inhalt des Puffers wird hinter die Stelle kopiert, an der der Cursor steht (war der Puffer leer, passiert nichts). Der Cursor springt hinter die einkopierte Stelle.
- Andere Kommandotasten beenden die Eingabe und lösen ggf. weitere Systemfunktionen aus.

Felder können so konfiguriert werden, daß einige der oben genannten Aktionen auch „wortweise“ funktionieren:

- [nach rechts] [1]: Der Cursor springt an den Anfang des nächsten Wortes, sofern das möglich ist, (ein *Wort* ist dabei eine ununterbrochene Folge von nichtleeren Zeichen; genauer also: Der Cursor springt an die Stelle des ersten nichtleeren Zeichens rechts von ihm, das einem rechts von ihm stehenden Leerzeichen folgt, sofern es ein solches Zeichen gibt).
- [nach links] [1]: Der Cursor springt an den Anfang des Wortes, in dem er steht; steht er am Anfang eines Wortes, an den Anfang des vorigen Wortes, sofern es noch eins davor gibt.

### ***Abschluß der Eingabe***

Der Abschluß der Eingaben von Zeichenketten – und damit die Auslösung von Reaktionen des Systems auf die Eingabe – erfolgt durch Kommandos, die nicht der Korrektur des Feldinhaltes dienen, solange die Eingabe noch nicht abgeschlossen ist:

- [weiter], [schluss], [nach oben], [nach unten] und
- [nach links], [nach rechts], [zum Ende], [zum Anfang], [entferne], [füge ein] usw. in Verbindung mit Tiefen  $> 0$ .

Mit der Vielfalt dieser Kommandos ist es möglich, Benutzerin gezielt durch die Eingabefelder in einer Bildschirmmaske (ggf. auch durch verschiedene Bildschirmmasken) springen zu lassen und damit den Ablauf des Programms zu steuern; ferner können sie zur Bearbeitung von Feldinhalten als Ganzes (d. h. ohne in den Feldern zu editieren) eingesetzt werden.

Wenn es zu einer differenzierten Ablaufsteuerung notwendig ist, können weitere Kommandos zur Beendigung der Eingabe vorgesehen werden, die außerhalb des Feldeditors weiterverarbeitet werden.

Zur Steuerung eines Systems sollte nicht auf die Kommandos zum Editieren innerhalb eines Feldes zurückgegriffen werden, weil es für Benutzer verwirrend sein kann, wenn die gleiche Taste (oder Tastenkombination) je nach Kontext unterschiedliche Wirkungen hat. (Von dieser Empfehlung mag es besonders begründete Ausnahmen geben.)

### *Fehlermeldungen*

Nach dem Abschluß einer fehlerhaften Eingabe in einem Feld erscheint in einem dafür vorgesehenen Feld (entweder in der unmittelbaren Umgebung der Eingabe oder in der letzten Bildschirmzeile) ein verständlich formulierter Hinweistext auf den Fehler. Der Inhalt des Eingabefeldes bleibt stehen, um Benutzerin zu ermöglichen, den Fehler nachzuvollziehen.

Der Cursor ist jetzt nicht sichtbar. Wenn die Kenntnisnahme der Fehlermeldung von Benutzerin (als leere Eingabe) mit ← oder Esc quittiert wird, verschwindet der Fehlertext und der Cursor erscheint wieder am Anfang des betreffenden Eingabefeldes, wobei sich der Feldeditor im Überschreibemodus befindet, damit die fehlerhaft eingegebene Zeichenkette korrigiert werden kann.

***Meldungen***

Für Fehlermeldungen und Benutzer-Hinweise.

***Auswahlen***

Zur Auswahl aus Listen in Form von „*pull-down menus*“.

***Aktionen***

Zur Menüsteuerung eines Programms.

## KONGLOMERATE VON OBJEKTEN

Unter *Konglomeraten* verstehen wir im folgenden Komponenten, deren Objekte *Gesamtheiten* (*Behälter*, „*Container*“) von Objekten eines Typs sind, z. B. *Folgen*, *geordnete Mengen* oder *Graphen*.

In diesem Kapitel werden alle Konglomerate einzeln vorgestellt, die das  $\text{Mod}_{\text{U}}^{\text{Ula}} \text{Mod}_{\text{U}}^{\text{Sum}}$  enthält, da ihre Entwicklung und Untersuchung zur Grundausbildung in Algorithmen und Datenstrukturen gehört.

### *Folgen*

Eine *Folge* von Objekten eines Typs  $X$  ist ein Konglomerat der Form

$$x_0, x_1, x_2, \dots, x_{n-1}$$

wobei die  $x_i$  ( $0 \leq i < n$ ) Objekte des Typs  $X$  sind. Die Anzahl  $n$  der Objekte der Folge kann – im Rahmen der verfügbaren Speicherressourcen – beliebig groß sein. Der Fall  $n = 0$ , die *leere Folge*, ist in die Betrachtungen eingeschlossen. Die auf S. 19-26 aufgeführten Standardprozeduren werden auch für Folgen benötigt:

- zum *initialisieren* einer Folge vor jeder Verwendung mit dem Effekt, daß sie *leer* ist,
- und zu ihrem *terminieren*, wenn sie nicht mehr gebraucht wird;
- zum Test, ob eine Folge *leer* ist, d. h. keine Objekte enthält,
- eine Folge zu *leeren*, d. h. alle Objekte aus ihr zu entfernen,
- eine Folge zu *kopieren*, d. h. eine weitere Folge mit allen ihren Objekten in der gleichen Reihenfolge zu erzeugen,
- zum Test, ob zwei Folgen *gleich* sind, d. h. die gleichen Objekte in der gleichen Reihenfolge enthalten,
- zum Test, ob eine Folge in dem Sinne *kleiner* oder *gleich* einer anderen ist, daß sie eine (echte) Teilfolge der anderen ist,
- zur Berechnung der *Codelänge* von Folgen und
- zum *codieren* und *decodieren* ganzer Folgen in bzw. aus *Strömen*.

Ausgenommen sind die Prozeduren zum *ausgeben* und *editieren*, weil die Ausgabe im einfachen Fall per *traversieren* erledigt werden kann, im allgemeinen eine gesamte Folge nicht auf einen Bildschirm paßt und Folgen wohl kaum in ihrer Gesamtheit durch Editieren verändert werden, sondern nur einzelne Objekte in ihr.

Dazu kommen folgende Operationen:

- zum Test, ob zwei Folgen in dem Sinne *äquivalent* sind, daß sie gleich viele Objekte enthalten und jeweils ihre *i*-ten Objekte in einer bestimmten – durch eine *Relation*, eine Boolesche Funktion auf Paaren von Objekten, gegebenen – Beziehung stehen,
- die *Anzahl* der Objekte in der Folge zu ermitteln,
- die *Anzahl* derjenigen Objekte zu ermitteln, die eine bestimmte – durch ein *Prädikat*, eine Boolesche Funktion auf den Objekten, gegebene – Eigenschaft erfüllen,
- zum Test, ob ein bestimmtes Objekt in der Folge *enthalten* ist.

Zur Konstruktion von Zugriffsoperationen auf die Objekte in den Folgen ist es sinnvoll, auf den Folgen sozusagen einen „einen Cursor zu positionieren“, also entweder *genau eins* ihrer Objekte („auf dem der Cursor steht“) oder *keins* („der Cursor steht hinter dem letzten Objekt“) als *aktuelles Objekt* auszuzeichnen.

Als motivierende Leitidee dazu stelle man sich den Block- oder Unterstrichcursor beim Editieren einer Textzeile vor, oder den – z. B. farblich hervorgehobenen – aktuellen Eintrag in einer Menüleiste, aus der ausgewählt werden kann.

Zur Realisierung dieses Konzepts gehören die Operationen

- zum *verschieben der Position* des aktuellen Objektes *um einen Schritt nach links oder rechts* (soweit das geht),
- zum *positionieren* auf das *i*-te Objekt einer Folge,
- zum Test, ob ein Objekt als aktuelles ausgezeichnet, insbesondere
- ob das *erste* oder das *letzte* Objekt einer Folge das aktuelle ist,
- und zur Ermittlung der *Position* des aktuellen Objekts.

Der Einsatz dieser Positionierungsoperationen erlaubt eine sehr bequeme Bearbeitung von Folgen mit den grundlegenden Operationen

- zum *lesen* oder
- (über)schreiben des *i*-ten Objekts einer Folge,
- zum *einfügen* eines weiteren Objekts an einer bestimmten Stelle (vor dem aktuellen Objekt)
- und zum *entfernen* eines Objekts aus der Folge (und zwar des aktuellen).

Der Suche nach Objekten aus einer Folge mit gegebenen Eigenschaften dienen Operationen

- zum Test, ob ein (erstes oder letztes) Objekt in der Folge *existiert*, das ein gegebenes *Prädikat* erfüllt
- zum *lokalisieren* weiterer solcher Objekte sowie
- zum Test, ob *alle* Objekte einer Folge ein *Prädikat* erfüllen,

der *Bearbeitung aller Objekte* einer Folge die Operationen (*Iteratoren*, gewissermaßen „Zählschleifen“ über alle Objekte einer Folge)

- zum *traversieren* (durchlaufen) einer Folge mit einer *Bearbeitung* (einer Aktionsprozedur mit einem Objekt als Parameter)
- sowie Varianten davon zur Bearbeitung aller Objekte einer Folge, die gewisse *Prädikate* erfüllen, und zum vorzeitigen Abbruch der Traversierung, wenn ein bestimmter Zweck erreicht ist,

der *Zerlegung und Zusammenfügung* von Folgen die Operationen

- zum *filtrieren* („einsammeln“) aller Objekte aus einer Folge, die ein bestimmtes *Prädikat* erfüllen, in einer weiteren Folge,
- zum *spalten* einer Folge an einer Position in zwei Folgen,
- zum *separieren* einer Folge in zwei Folgen so, daß die eine Folge alle *diejenigen* Objekte enthält, die ein *Prädikat* erfüllt, die andere Folge alle *anderen* Objekte,
- zum *verketten* („aneinanderhängen“) zweier Folgen und
- zum *rotieren* einer Folge (erstes Objekt nach hinten bzw. letztes nach vorne).

Folgen können *geordnet* sein, d. h. daß sie ihre Objekte in der durch eine *Ordnung* gegebene Reihenfolge enthalten, wobei die Ordnung auf den Objekten durch eine *Ordnungsrelation* gegeben ist, eine Boolesche Funktion auf Paaren von Objekten, die *reflexiv*, *antisymmetrisch* und *transitiv* ist. Dieser Problemkreis wird behandelt mit Operationen

- zum *ordnen* einer Folge nach einer Ordnungsrelation, d. h. der entsprechenden „Umsortierung“ der Reihenfolge ihrer Objekte,
- zum Test, ob eine Folge *geordnet* ist,
- zum *einordnen* eines weiteren Objekts in eine geordnete Folge
- und zum *vereinigen* zweier bzgl. der gleichen Ordnung geordneter Folgen zu einer einzigen.

### *Generizität durch Serialisierung*

Nun könnte man schließen, für jeden Typ von Objekten wäre ein *eigener Folgentyp* erforderlich, d. h. sowohl für Definitions- als auch Implementierungsmodul der nahezu gleiche Quelltext – unterschieden nur nach den Typen der Objekte in den Parametern der Prozeduren.

Würden in größeren Systemen viele Folgen von Objekten unterschiedlicher Typen benötigt, wäre eine Vervielfachung algorithmisch identischer Quelltexte die Folge – mit allen daraus resultierenden Gefahren von Inkonsistenzen.

Das ist jedoch *mit der Forderung nach Wiederverwendbarkeit von Quelltexten absolut nicht vereinbar*.

Eine elegante Lösung des Dilemmas ist in Sprachen möglich, die über *parametrische Polymorphie (Generizität)* verfügen, wie z. B. in funktionalen Sprachen, in Eiffel, C++ oder (ab v. 1.5) auch in Java.

Wenn das nicht der Fall ist, wie z. B. in Modula-2, gibt es einen einfachen Ausweg, der dadurch vorgezeichnet ist, daß Objekte nur als Ströme (*serielle Bytefolgen*) auf Datenträgern gespeichert oder über das Netz gesendet werden können.

Dieser Ansatz ist bei *allen* Konglomeraten verwendbar:

Es werden *nicht unterschiedliche typangepaßte* Konglomerate konstruiert, sondern

- die Konglomerate werden nur *einmal entwickelt* – und zwar für *Ströme* – mit der Maßgabe, daß
- Objekte als *Ströme* codiert (*serialisiert*) und *in dieser Form* in Konglomerate eingefügt und
- umgekehrt *Ströme* aus Konglomeraten ausgelesen und daraus per *decodieren* die Objekte wieder „hergestellt“ werden.

Damit ist ein einfacher (aber sehr universell einsetzbarer!) Ersatz für die fehlende Generizität in Modula-2 geschaffen.

Zu diesem Zweck müssen natürlich alle abstrakten Datentypen, von deren Typ Objekte in Folgen verwaltet werden sollen, die zur Serialisierung erforderlichen Operationen (s. *Codelaenge*, *codieren* und *decodieren*, S. 24-26) zur Verfügung stellen.

### Folgen in Murus

das Modul  $\text{Ula}_{\text{Obj}}^{\text{Sum}}$  enthält den abstrakten Datentyp **Folgen**, *Folgen von Strömen beliebiger, aber fester Länge*  $> 0$ . In jeder Folge ist entweder genau ein Strom der aktuelle Strom oder der aktuelle Strom ist undefiniert.

Die Spezifikationen der *Prozedurtypen* **Prädikate**, **Relationen**, **Bearbeitungen** und **bedingte Bearbeitungen**, die in verschiedenen Operationen als Typen von Parametern vorkommen, sind im Modul **Prozedurtypen** konzentriert, der auch von vielen anderen Komponenten verwendet wird.

Da es sinnvoll ist, Operationen verschiedener Komponenten mit gleichartigen Bedeutungen auch mit gleichen Namen zu bezeichnen, ist es konsequent, auch die von ihnen exportierten Objekte gleich – schlicht als **Objekte** – zu bezeichnen. d. h. auch die *Objekte*, die der Modul **Folgen** exportiert, heißen **Objekte**. Die Alternative **Folgen** z. B. würde zu umständlichen Formulierungen bei den Klienten führen (*Folgenobjekte* hätten dann die Syntax **Folgen.Folgen** ...). Eleganter wäre natürlich ein Sprachkonzept, in der die Namen der Komponenten, die abstrakte Datentypen darstellen, automatisch auch als Bezeichner für die Objekte verwendet werden könnten – aber das ist zur Zeit ein Wunschtraum ...

Wir zeigen hier nur beispielhaft die Spezifikation einiger weniger Operationen. In jedem Fall gehört die Angabe der Voraussetzungen und Effekte dazu; beides in Form statischer Zustandsbeschreibungen.

In der Prozedur

```
PROCEDURE initialisieren (VAR X: Objekte; n: CARDINAL);
(* Vor.: X ist nicht initialisiert. n > 0.
   Eff.: X ist initialisiert und leer
         und hat die Stromlänge n.
         Der aktuelle Strom von X ist undefiniert. *)
```

wird einer Folge die Größe der (serialisierten) Objekte mitgegeben, die in ihr untergebracht werden sollen.

Damit erspart man sich, diese – immer gleiche Zahl – beim Test, ob ein Strom in der Folge *enthalten* ist, sowie beim *einfügen*, (über)schreiben und *einordnen* jedesmal von neuem zu übergeben, was mit

der Gefahr von Inkonsistenzen verbunden wäre.

Ein Objekt eines Datentyps T wird mit der Prozedur

```
PROCEDURE einfuegen (X: Objekte; A: ADDRESS);
```

```
(* Vor.: X ist initialisiert.
```

```
  Eff.: War der aktuelle Strom von X vorher undefiniert,
        ist der Strom ab A der Stromlänge von X
        hinter das Ende von X angefügt, sonst ist
        er vor den aktuellen Strom eingefügt. Anzahl
        und Reihenfolge der anderen Ströme in X und
        aktueller Strom von X sind nicht beeinflusst. *)
```

in eine Folge eingefügt, indem für A eine Startadresse übergeben wird:

- im einfachsten Fall  $ADR(x)$  für eine Variable  $x$  eines konkreten Datentyps T, deren Wert an die entsprechende Stelle kopiert wird,
- in komplexeren Fällen (bei abstrakten Datentypen) die eines vom Klienten angelegten Pufferbereichs, in den das Objekt per Aufruf von  $T.codieren(x, A)$  vorher serialisiert abgelegt wurde.

Das „Durchlaufen“ einer Folge durch den Aufruf der Prozedur

```
PROCEDURE traversieren (X: Objekte; B: Bearbeitungen);
```

```
(* Vor.: X ist initialisiert.
```

```
  Wenn X geordnet ist, ist B bzgl. dieser Ordnung
  streng monoton, d.h. wenn der Strom ab A bzgl.
  dieser Ordnung kleiner als der Strom ab A1 ist,
  dann ist auch B(A) kleiner B(A1).
```

```
  Eff.: Auf alle Ströme in X ist (in ihrer Reihenfolge
        in X) B angewandt. Aktueller Strom von X
        ist der, der vorher aktuell war. *)
```

setzt natürlich eine Bearbeitungsprozedur vom Typ der generischen Bearbeitungen voraus, deren Name `traversieren` übergeben wird; z. B. zur Ausgabe von Zeichen mit `WriteChar` aus `InOut`

```
PROCEDURE ausgeben (A: ADDRESS);
```

```
VAR C: POINTER TO CHAR;
```

```
BEGIN
```

```
  C:= A; WriteChar (C^)
```

```
END ausgeben;
```

(zur Rolle von `C` siehe *Verweise, Wert- und Variablenparameter*).

Die vollständige Spezifikation ist im unter

<http://murus.org/murus/Murus/Folgen.def>

im weltweiten Netz verfügbar.

Die *Implementierung* der *Folgen* beruht auf einer Repräsentation als *doppelt verkettete Liste* von *Knotenzellen*. Diese *Knoten* sind Verbunde aus der

- *Startadresse der Ströme*, die in den Folgen aufgehoben werden,
- und der Vorwärts- und Rückwärts-Verzeigerung in Form zweier *Verweise* auf solche Knoten:

TYPE

```
Listen = POINTER TO Knoten;
Knoten = RECORD
    Strom: ADDRESS;
    weiter, zurueck: Listen
END;
```

In einem „Verwaltungs-“Verbund werden für eine Folge die

- *Größe* der Objekte (ihre Codelänge), ihre *Anzahl*, die *Positionsnummer* des *aktuellen Knotens* sowie die
  - *Verweise* auf den *Ankerknoten* und den *aktuellen Knoten*
- teils redundant – gehalten:

```
Folgen = RECORD
    Stromlaenge, Anzahl, Position: CARDINAL;
    Anker, aktuell: Listen
END;
```

Der opake *Folgentyp* Objekte ist der Verweis auf diesen Verbund:

```
Objekte = POINTER TO Folgen;
```

Die eigentliche *Folge* besteht aus den Strömen, die ab Startadresse des ersten auf den *Ankerknoten* folgenden Knoten im Arbeitsspeicher in der Reihenfolge der *weiter*-Verweise abgelegt sind.

Der Ankerknoten selbst trägt *kein* Objekt aus der Folge; er dient als *Markierung* („*sentinel*“) zur Erkennung von Anfang und Ende der Liste und von ihm aus wird mit *weiter* auf den ersten und mit *zurueck* auf den letzten Knoten der Liste verwiesen. Der Einbau eines

solchen Markierungsknotens führt zu einer deutlichen Vereinfachung vieler Algorithmen, weil Fallunterscheidungen leicht zu treffen sind.

Die ganze Konstruktion wird kurz als *doppelt verkettete Ringliste mit Anker* bezeichnet.

Mit den redundanten Komponenten **Anzahl** und **Position** des Verwaltungsverbands sind zwei Invarianten verbunden: der Wert von

- **Anzahl** muß immer mit der Anzahl der Knoten mit Ausnahme des Ankerknotens) in der Repräsentation der Folge übereinstimmen,
- **Position** mit der Ordnungszahl des aktuellen Knotens in der Liste (den **weiter**-Verweisen folgend), von 0 für den ersten auf den Anker folgenden Knoten bis **Anzahl** für den Ankerknoten.

Sie haben den Zweck, gewisse Operationen effizienter zu machen; z. B. muß zur Bestimmung der Anzahl der Objekte einer Folge nicht die Liste traversiert werden, um die Knoten zu zählen, sondern es wird im Direktzugriff einfach der Wert von **Anzahl** geliefert.

Damit ist natürlich das Problem verbunden, bei der Entwicklung der Algorithmen für die Einhaltung der Invarianten Sorge zu tragen (was in dem Beispiel leicht ist: bei jeder Einfügung bzw. Entfernung eines Objekts in die bzw. aus der Folge wird **Anzahl** inkrementiert bzw. dekrementiert).

```

PROCEDURE initialisieren (VAR F: Objekte; n: CARDINAL);
BEGIN
  NEW (F);
  WITH F^ DO
    Stromlaenge:= n;
    NEW (Anker);
    WITH Anker^ DO
      Strom:= NIL;
      weiter:= Anker;
      zurueck:= Anker
    END;
    Anzahl:= 0;
    aktuell:= Anker;
    Position:= 0
  END
END initialisieren;
```

In dieser Implementierung werden nach der Bereitstellung von Speicherplatz für den Verwaltungsverbund und der Aufnahme der Stromlänge in ihm die Ankerzelle als einziges Element der Ringliste (ohne Strom als Inhalt) erzeugt, die qua Verweisung auf sich selbst zeigt, sie als aktueller Knoten markiert (*kein Objekt ist aktuell*) und es werden die Invarianten Anzahl und Position auf 0 gesetzt.

Genau *das* ist die Repräsentation einer *leeren* Folge.

Ein Objekt wird in eine Folge *eingefügt*, indem es in serialisierter Form an eine bestimmte Stelle gesetzt und **Anzahl** erhöht wird und ggf. **aktuell** und **Position** angepaßt werden.

Dazu dient die folgende Prozedur, die (für  $L \neq \text{NIL}$ ) bewirkt, daß vor den Knoten, auf den  $L$  vorher gezeigt hat, ein neuer Knoten mit dem Strom ab  $A$  der Länge  $n$  eingefügt ist, wobei der Strom in *den* Bereich des Arbeitsspeichers kopiert ist, deren Startadresse die erste Komponente der Knoten ist:

```

PROCEDURE vorsetzen (L: Listen; A: ADDRESS; n: CARDINAL);
VAR L1: Listen;
BEGIN
  NEW (L1);
  WITH L1^ DO
    ALLOCATE (Strom, n);
    Stroeme.kopieren (A, Strom, n);
    weiter:= L;
    zurueck:= L^.zurueck
  END;
  WITH L^ DO
    zurueck^.weiter:= L1;
    zurueck:= L1
  END
END vorsetzen;

```

Die *Entfernung* eines Objekts ist mit der Freigabe des beim Einfügen eines Knotens reservierten Platzes verbunden. Das wird mit der folgenden Prozedur erledigt, deren Effekt (für  $L \neq \text{NIL}$ ) ist, daß der Knoten, auf den  $L$  vorher gezeigt hat, entfernt ist und  $L$  auf den Knoten zeigt, auf den  $L \wedge \text{weiter}$  vorher gezeigt hat:

```

PROCEDURE ausklinken (VAR L: Listen; n: CARDINAL);
VAR Muell: Listen;
BEGIN
  Muell:= L;
  WITH Muell^ DO
    zurueck^.weiter:= weiter;
    weiter^.zurueck:= zurueck;
    DEALLOCATE (Strom, n);
    L:= weiter
  END;
  DISPOSE (Muell)
END ausklinken;

```

Mit Hilfe dieser Prozedur können sowohl einzelne Objekte aus einer Folge entfernt als auch eine ganze Folge geleert werden (natürlich unter Anpassung von *Anzahl*, *aktuell* und *Position*).

Diese beiden *modulinternen* (d. h. nicht exportierten) Prozeduren sind auch für die Implementierung diverser anderer Operationen des Moduls *Folgen* verwendbar.

Beim *terminieren* werden dual zum *initialisieren* nach der Leerung der Folge der Ankerknoten und anschließend der Verwaltungsverbund beseitigt:

```

PROCEDURE terminieren (VAR F: Objekte);
BEGIN
  leeren (F);
  DISPOSE (F^.Anker);
  DISPOSE (F)
END terminieren;

```

Die Implementierung der Operationen – abgesehen von einigen Direktzugriffen durch Einführung gewisser redundanter Daten – haben eine *lineare Komplexität* in Abhängigkeit von der *Anzahl der Objekte* in der bearbeiteten Folge, weil zur Suche nach bestimmten Objekten die Liste der Knoten traversiert werden muß.

Ausnahme ist natürlich das Ordnen einer Liste; auch bei günstiger Implementierung mittels eines höheren Sortierverfahrens führt an der damit im allgemeinen verbundenen linear-mal-logarithmischen Komplexität kein Weg vorbei.

Bei diesen einführenden Bemerkungen über die Spezifikation und Implementierung generischer Folgen soll es an dieser Stelle bleiben; auf die Implementierung der einzelnen anderen Operationen kann hier nicht eingegangen werden. Eine erschöpfende Beschreibung aller Komponenten  $\text{des } \overset{\text{Ula}}{\text{Modell}} \overset{\text{Sams}}{\text{Ujiver}}$  würde dieses Heft zu einem „Wälzer“ entarten lassen . . .

Wer nicht viele Details aus eigener mühevoller Arbeit im Studium kennt, in dem Konzept, die hier nur in Nebensätzen gestreift werden, ausführlich motiviert und häufig mit funktionalen Programmen aus wenigen Zeilen „unterfüttert“ und in dessen Verlauf unzählige Bilder verketteter Listen entwickelt werden, dem bleibt nichts anderes übrig, als sich bei Interesse durch die Spezifikation und Implementierung der Folgen alleine „durchzubeißen“.

Der Modul ist im weltweiten Netz unter

<http://murus.org/murus/Murus/Folgen.mod>

verfügbar.

### *Folgen von Objekten unterschiedlicher Größe*

Für bestimmte Anwendungen werden auch *Folgen von Strömen variabler Größe* benötigt (z. B. beim  $\text{eGriff}$ ), wenn Objekte eines Typs unterschiedlichen Platzbedarf im Speicher haben.

Die Spezifikation der meisten Operationen auf derartigen Folgen stimmt mit der der entsprechenden Operationen auf *Folgen* überein.

Da es in diesem Fall keinen Sinn gäbe, einer Folge die Typgröße der von ihr zu verwaltenden Objekte beim *initialisieren* mitzugeben, entfällt der entsprechende Parameter; dafür wird er als zusätzlicher Parameter beim Test auf *enthalten* sowie beim *schreiben*, *einfügen* und *einordnen* aufgenommen, da diesen Operationen neben der Startadresse eines Stroms jeweils auch seine Länge übergeben werden muß.

Außerdem gibt es noch eine Operation

- zur Ermittlung der *Länge* des *aktuellen* Stroms einer Folge, damit z. B. ein Klient darüber informiert werden kann, wie groß der von ihm bereitzustellende Pufferbereich zum *lesen* eines Objekts sein muß.

das  $\text{Modell}_{\text{obj}}^{\text{Ula}} \text{Sum}$  enthält den abstrakten Datentyp **VarFolgen**, *Folgen von Strömen variabler Länge*, der in seiner Funktionalität mit den **Folgen** im Grunde identisch ist.

Die Datenstruktur muß natürlich den genannten Änderungen angepaßt werden:

Die Komponente **Stromlaenge** des Verwaltungsverbundes wird aus ihm entfernt und wird stattdessen in die Knoten aufgenommen, da diese Zahl für jedes Objekt unterschiedlich sein kann.

Die *Implementierung* der **VarFolgen** unterscheidet sich nur in den oben genannten Operationen mit anderer Parametrisierung von der der **Folgen**.

### *Kellerspeicher*

Eine der wichtigsten Strukturen in der Informatik ist die eines *Kellerspeichers* (oder *Stapels*): einer Folge, in der Objekte abgelegt und der sie nach dem „*der letzte ist der erste*-Prinzip“ („*LIFO*“ = „*last in, first out*“) wieder entnommen werden.

Ein *Kellerspeicher* benötigt Operationen

- zur Ermittlung der *Anzahl* seiner Objekte, insbesondere
- zum Test, ob er *leer* ist,
- zum *einfügen* am „oberen Ende“ („*push*“) und
- zum *entfernen* des obersten Objekts, womit dieses Objekt geliefert („*top*“) und dem Stapel entnommen ist („*pop*“).

Beim *initialisieren* wird – wenn nur Objekte gleicher Größe verwaltet werden – aus dem gleichen Grund wie bei den *Folgen* eben diese Größe übergeben.

das  $\text{Modell}_{\text{Ullas}}^{\text{Sum}}_{\text{Objekt}}^{\text{iver}}$  enthält den abstrakten Datentyp **Stapel**, *unbeschränkte Kellerspeicher von Strömen beliebiger aber fester Länge*. (Hier ist mein Grundsatz verletzt, für die Bezeichner von Datentypen einen Plural zu wählen und für Datenobjekte einen Singular; insofern ist die Bezeichnung **Stapel** leider mißlich; aber „**Stapels**“ wäre wohl nicht ganz in Ordnung . . . )

Seine *Implementierung* stützt sich ganz einfach auf die *Folgen*, bei denen nur *vorne eingefügt*, *gelesen* und *entfernt* wird: sofern sie nicht leer sind, ist ihr *aktuelles Objekt* immer ihr *erstes*, das jeweils *oberste* Objekt auf dem Stapel.

## *Warteschlangen*

Eine gleichermaßen bedeutende Rolle spielen *Warteschlangen* in der Informatik: Folgen, in die Objekte auf „einer“ Seite eingefügt und aus denen sie auf der „anderen“ Seite entnommen werden („*FIFO*“ = „*first in, first out*“).

Die Operationen auf ihnen gleichen – bis auf diese Semantik – formal denen der *Stapel*.

das  $\text{Modell}_{\text{Ullrich}}^{\text{Sum}}$  enthält zwei abstrakte Datentypen: **Schlangen**, *unbeschränkte Warteschlangen, von Strömen beliebiger aber fester Länge*, sowie **BSchlangen**, *beschränkte Warteschlangen einer vorgegebenen Maximalkapazität von Strömen beliebiger aber fester Länge*.

Die *Implementierung* von **Schlangen** ist – wie *Stapel* – durch Folgen gegeben, bei denen grundsätzlich *hinten eingefügt* und *vorne gelesen* und *entfernt* wird.

Einer *beschränkten Schlange* wird beim *initialisieren* als dritter Parameter ihre *Kapazität* mitgegeben; dazu kommt eine Operation

- zum Test, ob sie *voll* ist.

Die *Implementierung* der **BSchlangen** basiert auf Feldern, die als *Ringpuffer* strukturiert sind.

### *Beschränkte Folgen*

Eine Folge heißt „*durch M beschränkt*“, wenn sie nicht mehr als  $M$  Objekte enthält.

Überlicherweise werden *beschränkte Folgen* von Objekten eines Typs  $X$  in einer Datenstruktur aufgehoben, die mit Hilfe des Feld-Konstruktors implementiert wird:

ARRAY[0..M-1] OF X.

Die Konstante  $M$  muß bereits bei der Übersetzung festgelegt sein und kann nicht erst zur Laufzeit bestimmt werden, was einen – für bestimmte Zwecke gravierenden – Nachteil darstellt:

Mitunter ist es kaum möglich, sachgerecht zwischen den widersprüchlichen Forderungen nach genügend großer Dimensionierung des Feldes und dem sparsamem Umgang mit Speicherplatz abzuwägen.

Dem „dynamischen“ Ersatz dieser statischen Konstruktion dient ein entsprechendes Konglomerat mit Operationen

- zum *initialisieren*, wobei – wie bei Folgen – die Länge der Ströme übergeben wird, darüberhinaus das Maximum  $M$  sowie ein Strom, der ein serialisiertes *leeres Objekt* darstellt, mit dem Effekt, daß nach der Initialisierung die Ströme an allen Stellen  $i$  ( $0 \leq i < M$ ) *leer* sind,
- zum *schreiben* eines Stroms an die  $i$ -te Stelle der beschränkten Folge und
- zum *lesen* eines Stroms aus der  $i$ -ten Stelle aus ihr ( $0 \leq i < M$ ).

das  $\text{Modell}_{\text{univers}}^{\text{UlaSum}}$  enthält den abstrakten Datentyp **Folgen**, *beschränkte Folgen fester Länge von Strömen beliebiger, aber fester Länge  $> 0$ .*

Seine *Implementierung* beruht auf der Bereitstellung eines Puffers auf der Halde, der in passender Größe zur Laufzeit bereitgestellt wird.

### *Prioritätsschlangen*

Prioritätsschlangen sind Warteschlangen, in die Objekte unterschiedlicher Priorität eingereiht und denen sie nach fallender Priorität entnommen werden. Die *Priorität* der Objekte ist durch eine *Ordnung* auf ihnen definiert: kleinere Objekte haben höhere Priorität.

Neben den Standardoperationen *initialisieren*, *terminieren*, Test auf *leer*, und *leeren* sind Operationen

- zur Angabe der *Anzahl* der Objekte einer Schlange,
- zum *einordnen* eines weiteren Objekts in eine Schlange (effektfrei im Falle beschränkter Schlangen, falls die Schlange voll ist; mit zufälliger Reihenfolge bei gleichpriorisierten Objekten),
- zum *entfernen* des (bzw. eines) Objekts höchster Priorität mit dessen Bereitstellung (effektfrei, falls die Schlange leer ist), und
- zum *traversieren* aller Objekte einer Schlange mit einer *Bearbeitung*.

*Beschränkte* Prioritätsschlangen enthalten zusätzlich Operationen

- zur Angabe ihrer *Maximalkapazität* und
- zum Test, ob eine Schlange *voll* ist.

das Modul `UllSum` enthält die abstrakten Datentypen `PrioSchlangen` und `BPrioSchlangen`, unbeschränkte und beschränkte geordnete Folgen von Strömen beliebiger, aber fester Länge  $> 0$ .

Die Schlangen werden als *fast perfekt ausgeglichene binäre Bäume* repräsentiert, deren *unterste Blattschicht immer von links gefüllt* ist und die die *Haldeninvariante* erfüllen: jeder Knoten hat die Eigenschaft, daß das Objekt in ihm kleiner oder gleich den Objekten in den Knoten seiner Kinder ist. Die Wurzel des Baumes enthält ein kleinstes, d. h. höchstpriorisiertes, Objekt; der Zugriff darauf hat die Komplexität  $O(1)$ .

Die *Algorithmen* basieren auf der *Halden-*(„*Heap*“-)*Struktur*: *ein-geordnet* wird durch Einfügung an der ersten freien Stelle und *entfernt* nach Wegnahme der Wurzel mit deren Ersatz durch das letzte Objekt, jeweils in der untersten Blattschicht, jeweils mit anschließendem „*ab-*“ bzw. „*aufsteigen*“ durch fortlaufenden Austausch mit dem kleineren der Kinder bzw. dem Elternknoten *solange*, bis die Haldeninvariante wiederhergestellt ist.

Sie haben wegen der garantierten fast perfekten Ausgeglichenheit der Binärbäume die Komplexität  $O(\log_2 n)$  für  $n = \text{Anzahl der Objekte}$  in der Schlange.

In der beschränkten Variante werden die Binärbäume in einem Feld realisiert, wobei die Positionen der Kind- bzw. Elternknoten durch eine einfache Indexberechnung gefunden werden:

Die Wurzel hat den Index 1, der linke bzw. rechte Kindknoten eines Elternknotens mit dem Index  $i$  hat den Index  $2i$  bzw.  $2i + 1$ ; der Elternknoten eines Knotens mit dem Index  $i$  hat den Index  $i \text{ div } 2$ .

Im unbeschränkten Fall wird mit einem Geflecht in Form eines Binärbaums gearbeitet; jeder Knoten besteht aus einem Objekt und den Verweisen auf den linken und rechten Teilbaum:

TYPE

```
Baeume = POINTER TO Knoten;
Knoten = RECORD
    Wurzel: ADDRESS;
    links, rechts: Baeume
END;
```

In einem Verwaltungsverbund werden – wie bei den Folgen – zusätzliche Daten aufgehoben:

```
PrioSchlangen = RECORD
    Anker: Baeume;
    Laenge, Anzahl: CARDINAL;
    kg: Relationen
END;
Objekte = POINTER TO PrioSchlangen;
```

In der Implementierung der Prozedur

```
PROCEDURE initialisieren (VAR X: Objekte; n: CARDINAL;
    R: Relationen);
```

zur Erzeugung einer Warteschlange werden nach der Bereitstellung von Speicherplatz für den Verwaltungsverbund `Anker := NIL` gesetzt (ein leerer Binärbaum repräsentiert eine leere Warteschlange), die beiden Parameter in die entsprechenden Komponenten übernommen und `Anzahl := 0` gesetzt.

Die Implementierungen zum *einordnen* und *entfernen* sind jedoch aufwendiger als im beschränkten Fall, weil die Verwaltung mittels der Indizes im Feld durch aufwendigere Berechnungen ersetzt werden muß:

Der letzte Knoten in einem fast perfekt ausgeglichenen Baum mit  $n > 1$  Knoten wird mit der Funktion  $f: \mathbb{N} \rightarrow 2 \times \mathbb{N}$  gefunden, die durch

$$f(n) = \begin{cases} (0, n - g(n)), & \text{falls } n - g(n) \leq \frac{g(n)}{2} \\ (1, n - \frac{g(n)}{2}), & \text{sonst} \end{cases}$$

definiert ist, wobei  $g(n)$  die größte Zweierpotenz  $\leq n$  ist, definiert durch

$$g(n) = \begin{cases} n, & \text{falls } n \leq 2 \\ 2 \cdot g(n \operatorname{div} 2), & \text{sonst.} \end{cases}$$

Das ermöglicht den rekursiven Abstieg von der Wurzel zum letzten Knoten: Er befindet sich *genau dann* im *linken* Teilbaum, wenn  $\pi_0(f(n)) = 0$ , und  $\pi_1(f(n))$  ist die Anzahl der Knoten des Teilbaums, in dem er sich befindet.

## *Persistente Folgen*

Unter *persistenten Folgen* werden solche Folgen verstanden, deren Objekte *dauerhaft* auf einem Datenträger gespeichert werden, d. h. bei einem Aufruf eines Programms, das sie verwendet, *diejenigen* Werte haben, die sie am Ende des letzten Programmlaufs hatten.

Ihre Spezifikation unterscheidet sich von der der *Folgen* lediglich *darin*, daß sie einen „Henkel“ haben, mit dem sie wieder *auffindbar* sind: Einen *Namen im Dateisystem*.

Dazu gibt es Operationen zum

- *benennen* und
- *umbenennen* sowie
- zur Bestimmung der *Anzahl der Bytes* (der „Dateilänge“)

der entsprechenden zugrundeliegenden *Dateien*.

das Modell  $\text{Ull}_{\text{Datei}}^{\text{Sum}}$  enthält den abstrakten Datentyp **PFolgen**, *persistente Folgen von Strömen beliebiger, aber fester Länge*  $> 0$ , die anhand ihres Namens im Dateisystem eindeutig identifizierbar sind.

**Konzept:** Objekte werden als Ströme (Bytefolgen) codiert; der Zugriff auf ein Objekt erfolgt über die direkte Positionierung auf das entsprechende Byte in der jeweiligen Datei nach der Formel  $\text{Dateilänge} = \text{Satzlänge} \cdot \text{Anzahl der Datensätze}$ .

Ihre *Implementierung* erfolgt in Form sequentieller Dateien von Datensätzen beliebiger, aber fester Länge; im tieferliegenden Modul **PFolgenkern** sind die Zugriffe auf das Dateisystem unter Benutzung von Aufrufen aus der C-Standardbibliothek konzentriert.

Die Repräsentation eines Datenobjekts vom Typ **PFolgen** besteht aus einem Verweis auf einen Verbund aus

- dem *Dateinamen*, einer Zeichenkette, die einen im Betriebssystem gültigen Dateinamen bezeichnet, sowie einem *temporären* Namen des gleichen Typs zu Zwecken der temporären Umbenennung;
- einem abstrakten Datenobjekt *Kern*, in dem alle betriebssystemabhängigen Teile der Dateizugriffe konzentriert sind, um eine Portierung der **PFolgen** auf andere Betriebssysteme unter Beibehalt der Quelltexte aller Algorithmen zu ermöglichen;
- der *Satzlänge* der Ströme, die die einzelnen Datensätze der zugrundeliegenden Datei darstellen; = als Bytefolgen codierte Objekte

- *Satzposition* zur Fixierung der jeweiligen Position des ausgezeichneten Objekts (entspricht der Rolle des Attributs `Position` bei den Folgen);
- *verarbeitet*, die Anzahl der bei einem Dateizugriff erfolgreich gelesenen bzw. geschriebenen Bytes;
- zwei *Pufferbereiche* im Arbeitsspeicher zur temporären Zwischenspeicherung von Strömen.

TYPE

```
Dateien = RECORD
    Dateiname,
    tempName: Dateibaum.Dateinamen;
    Kern: Pfolgenkern.objekte;
    Satzlaenge,
    Satzposition,
    verarbeitet: CARDINAL;
    Puffer1,
    Puffer1: ADDRESS;
END;
Objekte = POINTER TO Dateien;
```

Idealistische Abstützung das klassische Konzept der Zugriffe auf sequentielle Dateien, wie von WIRTH in Pascal realisiert (mit dem Blick auf die historische Situation: Bänder als periphere Datenträger)

- `reset`, `rewrite`, `read`, `write`, `eof` und `seek`

sowie der Anbindung an das *Betriebssystem* zur Realisierung dieser Operationen durch Aufrufe aus der entsprechenden *Systembibliothek* folgen

*demnächst.*

### *(Geordnete) Mengen*

Unter *Mengen* werden im folgenden *grundsätzlich geordnete Mengen* verstanden.

Es gehört natürlich – anders als bei *Folgen*) – zum Konzept, daß die Objekte in einer Menge paarweise voneinander verschieden sind (für ein Element  $x$  und eine Menge  $M$  gilt entweder  $x \in M$  oder aber  $x \notin M$ , d. h. insbesondere z. B.  $\{x, x\} = \{x\}$ ).

Es gibt in der Informatik auch den Begriff der *Multimengen*, einer Abart des mathematischen Mengenbegriffs, in denen Elemente mehrfach vorkommen dürfen, in dem folglich die Multimenge „ $\{x, x, x\}$ “ nicht mit „ $\{x\}$ “ übereinstimmt (Mathematiker/innen mögen mir den Gebrauch der geschweiften Klammern an dieser Stelle nachsehen).

*Detailliertere Ausführungen folgen eventuell irgendwann.*

das  $\text{Modell}_{\text{Ull}}^{\text{Sum}}$  enthält den abstrakten Datentyp **Mengen**, *Geordnete Mengen von Strömen beliebiger, aber fester Länge*.

Ihre *Implementierung*: Binäre AVL-Bäume mit Positionsverwaltung.

Hierzu nun in den folgenden Abschnitten etwas Theorie.

### *Fibonacci-Bäume und AVL-Bäume*

Fibonacci-Bäume sind rekursiv wie folgt definiert:

$$F(n) = \begin{cases} \text{leerer Baum} & \text{für } n < 0 \\ \text{Baum mit Wurzel, } F(n-2) \text{ als linkem} \\ \text{und } F(n-1) \text{ als rechtem Teilbaum} & \text{für } n \geq 0 \end{cases}$$

Die Rekurrenzgleichung für die Anzahl der Knoten dieser Bäume in Abhängigkeit von der Höhe  $n$  lautet folglich

$$(1) \quad l(n) = \begin{cases} 1 & \text{für } n = 0 \\ 2 & \text{für } n = 1 \\ 1 + l(n-2) + l(n-1) & \text{für } n \geq 2. \end{cases}$$

Diese Zahlen heißen LEONARDO-ZAHLEN; sie erinnern in ihrem Aufbau stark an die Fibonacci-Zahlen.

Wenn wir die dritte Gleichung von (1) durch  $l(n-1)$  dividieren und

$$(2) \quad a_n = \frac{l(n)}{l(n-1)} \quad \text{für } n \geq 1$$

setzen, erhalten wir

$$a_n = 1 + \frac{1}{a_{n-1}} + \varepsilon_n \quad \text{mit } \varepsilon_n = \frac{1}{l(n-1)}.$$

Grenzwertbildung liefert daraus

$$(3) \quad \lim_{n \rightarrow \infty} a_n = a,$$

wobei  $a = \frac{1}{2} + \frac{\sqrt{5}}{2} \approx 1,61803$  die positive Lösung der quadratischen Gleichung  $x^2 = 1 + x$  des goldenen Schnitts ist. (2) und (3) liefern asymptotisch, d. h. für größere  $n$ ,

$$l(n+1) \approx a \cdot l(n),$$

deshalb wegen  $l(0) = 1$  für die Knotenzahl  $l(n)$  in Abhängigkeit von der Höhe  $n$  asymptotisch

$$(4) \quad l(n) \approx a^n.$$

Für die Umkehrung, d. h. die Berechnung der Höhe  $n$  zu gegebener Knotenzahl  $k$ , erhalten wir durch Logarithmieren daraus

$$\log_2 k \approx \log_2(a^n) = n \cdot \log_2 a,$$

folglich

$$n \approx \frac{\log_2 k}{\log_2 a} = 1,44 \cdot \log_2 k$$

$$\text{mit } 1,44 \approx \frac{1}{\log_2 1,61803}.$$

AVL-Bäume (VON ADELSON-VELSKII UND LANDIS, 1962) sind wie folgt definiert:

- Der leere Baum ist ein AVL-Baum.
- Wenn L und R AVL-Bäume sind, die sich um höchstens 1 in der Höhe unterscheiden, ist ein Baum mit einer Wurzel und mit L als linkem und R als rechtem Teilbaum ein AVL-Baum.

Per Induktion ist leicht nachzuweisen, daß Fibonacci-Bäume

- AVL-Bäume sind (weil  $F(n)$  die Höhe  $n$  hat) und daß sie
- bei gegebener Höhe eine minimale Knotenzahl haben (wenn ein Knoten im linken oder rechten Teilbaum entfernt wird, verringert sich die Höhe des Baums um 1 – im ersten Fall nach einer einfachen Linksrotation).

Sie sind damit in *dem Sinne* die „schlechtestmöglichen“ AVL-Bäume, daß jeder AVL-Baum bei gegebener Höhe mindestens so viele Knoten wie der Fibonacci-Baum gleicher Höhe hat – mit anderen Worten, daß ein AVL-Baum nicht höher sein kann als ein Fibonacci-Baum mit gleich vielen Knoten.

Nach den Überlegungen aus dem vorigen Abschnitt ist *die Höhe eines AVL-Baums im schlimmsten Fall nur etwa 44% größer als die eines bestmöglich ausgeglichenen Baumes mit gleich vielen Knoten*, d. h. insbesondere ist logarithmische Suchzeit bei ihnen garantiert.

### ***Explizite Darstellung der Leonardo-Zahlen***

Die Leonardo-Zahlen (*Anzahlen der Knoten in Fibonacci-Bäumen*) sind rekursiv durch

$$(1) \quad l(n) = \begin{cases} 0 & \text{für } n \leq 0 \\ 1 + l(n-2) + l(n-1) & \text{für } n > 0 \end{cases}$$

definiert. Für  $n \geq 0$  ergibt sich die Folge

$$0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, 376, \dots$$

Durch Einsatz zweier Akkumulatoren läßt sich das auch endrekursiv

formulieren:

$$\begin{aligned}
 l(0) &= 0 \\
 l(n) &= l'(n-1, 1, 0) \quad \text{für } n > 0 \\
 \text{mit } l'(n, a, b) &= \begin{cases} a & \text{für } n = 0 \\ l'(n-1, 1+a+b, a) & \text{für } n > 0. \end{cases}
 \end{aligned}$$

Mit den formalen Potenzreihen

$$(2) \quad f(X) = \sum_{m=0}^{\infty} l(m)X^m = X + 2X^2 + 4X^3 + 7X^4 + 12X^5 + \dots$$

$$\text{und } q(X) = \frac{1}{1-X} = \sum_{m=0}^{\infty} X^m = 1 + X + X^2 + X^3 + X^4 + \dots$$

erhalten wir durch Einsetzen von (1)

$$\begin{aligned}
 Xf(X) + X^2f(X) + q(X) &= \sum_{m=0}^{\infty} l(m)X^{m+1} + \sum_{m=0}^{\infty} l(m)X^{m+2} + \sum_{m=0}^{\infty} X^m \\
 &= \sum_{n=0}^{\infty} (l(n-1) + l(n-2) + 1)X^n \\
 &= 1 + \sum_{m=0}^{\infty} l(m)X^m = 1 + f(X),
 \end{aligned}$$

nach  $f(X)$  aufgelöst:

$$(3) \quad f(X) = \frac{X}{(1-X-X^2)(1-X)}.$$

Die Faktorisierung von  $1 - X - X^2 = (1 - aX)(1 - bX)$  liefert das Gleichungssystem

$$(4) \quad \begin{aligned} a + b &= 1 \\ ab &= -1 \end{aligned}$$

mit den Lösungen

$$(5) \quad a = \frac{1}{2} + \frac{\sqrt{5}}{2}, \quad b = \frac{1}{2} - \frac{\sqrt{5}}{2}.$$

Der Ansatz zur Partialbruchzerlegung von (3)

$$(6) \quad \frac{1}{(1-X-X^2)(1-X)} = \frac{A}{1-aX} + \frac{B}{1-bX} + \frac{C}{1-X}$$

liefert die Gleichung

$$A(1-bX)(1-X) + B(1-aX)(1-X) + C(1-aX)(1-bX) = 1,$$

und daraus mit (4) nach Koeffizientenvergleich das lineare Gleichungssystem

$$(7) \quad \begin{aligned} A + B + C &= 1 \\ (1+b)A + (1+a)B + C &= 0 \\ bA + aB - C &= 0 \end{aligned}$$

mit den Lösungen

$$(8) \quad A = 1 + \frac{2}{5}\sqrt{5}, \quad B = 1 - \frac{2}{5}\sqrt{5}, \quad C = -1.$$

Einsetzen von (6) in (3) in Verbindung mit der allgemeinen geometrischen Potenzreihe

$$\frac{1}{1-cX} = \sum_{m=0}^{\infty} c^m X^m$$

ergibt

$$\begin{aligned} f(X) &= \frac{A}{1-aX}X + \frac{B}{1-bX}X + \frac{C}{1-X}X \\ &= A \sum_{m=0}^{\infty} a^m X^{m+1} + B \sum_{m=0}^{\infty} b^m X^{m+1} + C \sum_{m=0}^{\infty} X^{m+1} \\ &= \sum_{m=0}^{\infty} (Aa^m + Bb^m + C)X^{m+1} \\ &= \sum_{n=1}^{\infty} (Aa^{n-1} + Bb^{n-1} + C)X^n. \end{aligned}$$

Nach (2) erhalten wir durch Koeffizientenvergleich für die Leonardo-Zahlen

$$l(n) = Aa^{n-1} + Bb^{n-1} - 1 \quad \text{für } n > 0.$$

Die dritte Gleichung des Systems (7) liefert  $\frac{A}{a} + \frac{B}{b} - 1 = 0$ , deshalb ist dieses Ergebnis auch für  $n = 0$  korrekt.

Durch Einsetzen von (5) und (8) erhalten wir die explizite Darstellung der Leonardo-Zahlen

$$l(n) = \left(1 + \frac{2}{5}\sqrt{5}\right) \left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right)^{n-1} + \left(1 - \frac{2}{5}\sqrt{5}\right) \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)^{n-1} - 1$$

für alle natürlichen Zahlen  $n$ .

*Die Leserinnen und Leser sollten sich durch Nachrechnen von der Korrektheit der Lösungen (5) und (8) überzeugen.*

### **Formale Potenzreihen**

Für Mengen  $A$  und  $B$  bezeichne  $A^B$  die Menge der Abbildungen von  $B$  nach  $A$ . Sei  $X$  eine *Unbestimmte* (ein Symbol).

Wir betrachten die Menge  $H = \mathbb{N}^{\{X\}}$ . Es gilt  $H \cong \mathbb{N}$ , denn die Abbildung

$$f: \mathbb{N} \rightarrow H, \quad \text{definiert durch } f(n)(X) = n,$$

ist bijektiv; mit der Bezeichnung  $X^n = f(n) \in H$  ist

$$H = \{X^n \mid n \in \mathbb{N}\}.$$

$H$  bildet bezüglich der durch

$$X^n \cdot X^k = X^{n+k}$$

definierten Multiplikation mit  $1 = X^0$  als neutralem Element eine kommutative Halbgruppe, wovon man sich durch Nachrechnen leicht

überzeugen kann;  $f$  ist darüberhinaus wegen  $f(0) = X^0 = 1$  und  $f(n+k) = X^{n+k} = X^n \cdot X^k = f(n) \cdot f(k)$  ein Isomorphismus.

Für einen kommutativen Ring  $A$  sei

$$A[[X]] = A^H.$$

Jedes  $p \in A[[X]]$  läßt sich dann eindeutig in der Form

$$p = \sum_{n \in \mathbb{N}} a_n X^n = a_0 + a_1 X + a_2 X^2 + \dots \quad \text{mit} \quad a_n = p(X^n)$$

schreiben, denn aus dem Term  $\sum_{n \in \mathbb{N}} a_n X^n$  läßt sich  $p$  durch die Definition  $p(X^n) = a_n$  zurückgewinnen. Durch

$$\begin{aligned} \sum_{n \in \mathbb{N}} a_n X^n + \sum_{n \in \mathbb{N}} b_n X^n &= \sum_{n \in \mathbb{N}} c_n X^n \quad \text{mit} \quad c_n = a_n + b_n \\ \text{und} \quad \sum_{n \in \mathbb{N}} a_n X^n \cdot \sum_{n \in \mathbb{N}} b_n X^n &= \sum_{n \in \mathbb{N}} c_n X^n \quad \text{mit} \quad c_n = \sum_{\substack{i, k \in \mathbb{N} \\ i+k=n}} a_i b_k \end{aligned}$$

sind eine Addition und eine Multiplikation auf  $A[[X]]$  definiert, bezüglich derer  $A[[X]]$  einen kommutativen Ring bildet (Beweis durch Nachrechnen).

$A[[X]]$  heißt der *Ring der formalen Potenzreihen über  $A$* .

Per Induktion läßt sich zeigen, daß  $A[[x]]$  nullteilerfrei ist, wenn  $A$  es ist; für einen Körper  $A$  ist folglich auch der *Quotientenring der formalen Potenzreihen* (der Ring der Brüche von formalen Potenzreihen) ein *Körper*, d. h. man kann mit diesen Brüchen z. B. über  $\mathbb{Q}$  oder  $\mathbb{R}$  ganz gewöhnlich – wie mit Zahlen – rechnen.

Abschließend sei bemerkt, daß diese Konstruktion die eines Polynomrings  $A[X]$  in einer Unbestimmten  $X$  über einem Ring  $A$  umfaßt:

Man ersetze  $H = \mathbb{N}^{\{X\}}$  durch  $\mathbb{N}(\{X\})$  (wobei  $A^{(B)}$  für einen Ring  $A$  die Menge der Abbildungen von  $B$  nach  $A$  bezeichnet, die fast überall 0 sind, d. h. für die  $f(b) \neq 0$  nur für endlich viele  $b \in B$  gilt).

Die gebrochen rationalen Funktionen über  $\mathbb{R}$  bilden damit einen Unterkörper des Quotientenkörpers der formalen Potenzreihen  $\mathbb{R}[[X]]$ .

### *Persistente Mengen*

Unter *persistenten Mengen* werden *solche* geordneten Mengen verstanden, deren Objekte vermöge Speicherung auf einem Datenträger die Laufzeit eines Programms überdauern (s. *Persistente Folgen*).

Der Unterschied zwischen *ihrer* Spezifikation und der der *Mengen* ist analog zu dem zwischen der Implementierung der *persistenten Folgen* und der der *Folgen*.

das  $\text{Modell}_{\text{U}}^{\text{Ula}} \text{U}^{\text{Sum}}$  enthält den abstrakten Datentyp **PMengen**, *persistente geordnete Mengen von Strömen beliebiger, aber fester Länge*.

Ihre (sehr aufwändige) *Implementierung*: B-Bäume von Strömen gleicher Länge mit Positionsverwaltung.

### *Persistente Mengen mit Indexen*

das  $\text{Modell}_{\text{U}}^{\text{Ula}} \text{U}^{\text{Sum}}$  enthält den abstrakten Datentyp **PIMengen**, *persistente geordnete Mengen von Strömen beliebiger, aber fester Länge*.

Ihre *Implementierung*: ISAM-Dateien (sequentielle Dateien mit mehreren Indexbäumen) von Datensätzen gleicher Größe mit Positionsverwaltung.