Inhaltsverzeichnis

1 Einführung in UNIX **5**
1.1 Etwas über Betriebssysteme . . . 5
1.1.1 UNIX, LINUX, KDE, CDE . . . 6
1.1.2 Die Etikette im Rechnerpool des Fachbereichsnetzes . . . 7
1.2 An- und Abmelden beim System . . . 7
1.3 Aufbau von UNIX-Kommandos . . . 9
1.4 Editoren . . . 9
1.5 Drucken . . . 10
1.6 Verzeichnisse und Dateien . . . 11
1.6.1 Dateien und Pfade . . . 11
1.6.2 Grundgerüst des Verzeichnisbaumes . . . 13
1.6.3 Navigation im Verzeichnisbaum . . . 13
1.6.4 Ausgabe von Inhaltsverzeichnissen . . . 15
1.6.5 Zugriffsrechte für Dateien und Verzeichnisse . . . 16
1.6.6 Absolute und symbolische Modi . . . 17
1.7 UNIX-Shells . . . 18
1.7.1 Kommandos . . . 18
1.7.2 Platzhalter in Dateinamen . . . 19
1.7.3 Umleitung . . . 19
1.7.4 Vorverarbeitung der Kommandos . . . 20
1.8 Prozesse . . . 21
1.9 Arbeiten auf entfernten Rechnern . . . 22
1.10 Weitere nützliche Befehle . . . 23

2 Algorithmen **25**
2.1 Sequenzen . . . 26
2.1.1 Beispiel:
Algorithmus zur Zubereitung einer Tasse Pulverkaffee . . . 26
2.1.2 Verfeinerung . . . 26
2.2 Auswahl (Selektion) . . . 27

2.2.1 Beispiel: Kaffeekochen
Verfeinerung von Schritt 2.1 durch Auswahl 27
2.3 Schleifen (Wiederholung, Iteration) 28
2.3.1 Beispiel: Kaffeekochen
Schritt 2.1 mit Wiederholungsschleife 28
2.3.2 Beispiel: Maximumsbestimmung 29
2.3.3 Beispiel: EUKLIDischer Algorithmus zur Berechnung des größten gemeinsamen Teilers 30
2.3.4 Beispiel: Berechnung von Potenzen x^n 30
2.4 Funktionen 31
2.4.1 Algorithmus: Zeichne zwei konzentrische Quadrate 31
2.5 Vom Algorithmus zum Programm 32
2.6 Einige Grundsätze zur Programmierung 34

3 Einführung in die `Modula-3`-Programmierung 36
3.1 Überblick 36
3.1.1 Was ist `Modula-3`? 36
3.1.2 Literatur 37
3.1.3 Das nullte und kürzeste `Modula-3`-Programm 37
3.1.4 Ein erstes Programm: Ausgabe mit `IO.Put` 38
3.1.5 Steuerzeichen für die Ausgabe 39
3.1.6 Exkurs: Modulares Programmieren 40
3.1.7 Funktionen 41
3.1.8 Einige Regeln für Funktionen 42
3.2 Variablen und ordinale Typen 42
3.2.1 Variable und ihr Datentyp, ganze Zahlen 42
3.2.2 Weitere Beispiele für formatierte Ausgabe 45
3.2.3 Ausdrücke 47
3.2.4 Aufzählungen 49
3.2.5 Unterbereiche 50
3.2.6 Mengen 51
3.2.7 Typumwandlungen 52
3.2.8 Inkrement- und Dekrementfunktionen 53
3.2.9 Eingabe mit `Lex` 53
3.3 Anweisungen zur Auswahl 54
3.3.1 Logische und Vergleichsoperatoren 54
3.3.2 `IF`-Anweisungen 57
3.3.3 `CASE`-Anweisung 59
3.4 Funktionen und Prozeduren 61

3.4.1 Definition . . . 61
3.4.2 Aufruf . . . 61
3.4.3 Beispiel: Call by value . . . 62
3.4.4 Beispiel: Funktion mit Rückgabewert . . . 63
3.4.5 Beispiel: Funktion mit mehreren Eingaben . . . 64
3.4.6 Bezugsrahmen von Variablen . . . 64
3.4.7 Programm mit lokalen Variablen . . . 65
3.4.8 Programm mit globalen und lokalen Variablen . . . 66
3.4.9 Wiederholung: Funktionsparameter sind lokale Variablen . . . 68
3.4.10 Call by reference . . . 68
3.4.11 Rekursive Funktionen . . . 71
3.5 Schleifen . . . 72
3.5.1 `WHILE`- und `REPEAT`-Schleife . . . 72
3.5.2 `LOOP`-Schleife . . . 73
3.5.3 `FOR`-Schleife . . . 73
3.6 Fließkommavariablen . . . 75
3.6.1 Formatierung . . . 78
3.6.2 Beispiel: Berechnung von π nach Leibniz . . . 78
3.6.3 Mathematische Funktionen . . . 80
3.7 Felder (Arrays) . . . 84
3.7.1 Eindimensionale Felder . . . 84
3.7.2 Mehrdimensionale Felder . . . 87
3.7.3 Felder variabler Größe . . . 87
3.7.4 Felder und Mengen . . . 88
3.8 Zeichenketten (Texte) . . . 89
3.8.1 Der Datentyp Zeichen (`CHAR`) . . . 89
3.8.2 Der Datentyp `TEXT` . . . 91
3.8.3 Ein- und Ausgabe von Texten . . . 91
3.8.4 Ein Beispiel für Text-Verarbeitung: Test auf Bereichsüberschreitung . . . 93
3.9 Datenverbünde . . . 95
3.10 Zeiger (Pointer) . . . 97
3.10.1 Operationen auf Zeigern . . . 97
3.10.2 Zeiger auf Datenverbünde . . . 100
3.10.3 Felder von Feldern . . . 105
3.10.4 Funktionenvariablen . . . 107
3.11 Umgang mit Dateien . . . 108
3.11.1 Beispiel zu Textdateien . . . 110
3.12 Ausnahmebehandlung . . . 111

3.12.1 TRY-EXCEPT-Konstrukt 112
3.12.2 TRY-FINALLY-Konstrukt 115
3.12.3 Ausnahmezustände selbst ausrufen 116
3.13 Module 118
3.13.1 Einbinden von Modulen 119
3.13.2 Module selbst erstellen 119
3.13.3 Pakete verwalten 120
3.14 Objekte 121
3.14.1 Klassen 122
3.14.2 Arbeiten mit Objekten 122
3.14.3 Deklaration von Klassen 124

A Übersichten **128**
A.1 Vorrangregeln 128

B Wichtige Module der Standardbibliotheken m3core und libm3 **130**
B.1 libm3: Fmt 130
B.2 libm3: Lex 136
B.3 libm3: Scan 139
B.4 m3core: Text 140
B.5 libm3: ASCII 142
B.6 libm3: Stdio 144
B.7 libm3: IO 145
B.8 libm3: Rd 147
B.9 libm3: Wr 154
B.10 libm3: FileRd 158
B.11 libm3: FileWr 159
B.12 libm3: File 161
B.13 m3core: Float 164
B.14 libm3: Math 168
B.15 libm3: Random 172

Index **175**

Kapitel 1

Einführung in UNIX

1.1 Etwas über Betriebssysteme

Aufbau eines Computer-Systems (ganz grob):

- CPU (central processing unit)
- interne Speicher: RAM, ...
- externe Speicher: Festplatte, CD-Rom, Diskette, ...
- Eingabegeräte: Tastatur, Maus, Scanner, ...
- Ausgabegeräte: Bildschirm, Drucker, ...
- Stromversorgung u.v.m.

Zu diesen diversen Hardware-Komponenten kommt Software: Anwendungsprogramme, Systemsoftware, Compiler usw.
All dies ist für uns nur sinnvoll nutzbar, weil es ein *Betriebssystem* gibt, das uns viel Arbeit abnimmt, z.B.

- Tastatureingabe erkennen und weiterleiten
- Programm starten, den ordnungsgemäßen Ablauf kontrollieren und das Programm beenden
- Dateien speichern und verwalten

Allgemein: Betriebssystem ist die zusammenfassende Bezeichnung für alle Systemprogramme, die die Ausführung von Benutzeranweisungen, die Verteilung von Betriebsmitteln auf die einzelnen Anweisungen und die Aufrechterhaltung des Rechnerbetriebs steuern und überwachen.
Beispiele:

- MS-DOS, Windows95, WindowsNT, ...
- VMS, ...
- UNIX und seine Derivate (LINUX, Solaris, ...)

1.1.1 UNIX, LINUX, KDE, CDE

UNIX:

- multi-user-system (Netzwerkbetrieb)
- multi-tasking
- time sharing
- Rechnerunabhängigkeit, Portabilität
- Kommandozeilenorientierung
- leistungsfähige Benutzerschnittstellen (Shells)
- programmiert größtenteils in `C`

Etwas zur Geschichte

∼1970	erste UNIX-Version, BellLabs (AT&T)
∼1975	Weiterentwicklungen, u.a. University of Berkeley und verschiedene Hardware-Produzenten
1983	UNIX V5
seit 1991	Entwicklung von LINUX als UNIX-Derivat für Intel386-Prozessor Linus Torvalds u.v.a.

Heute ist ein UNIX-Betriebssystem mit einer graphischen Oberfläche verbunden: KDE, Gnome, CDE, FVWM, ...

- Desktops inkl. Titelleiste
- verschiedene Fenster (für Editor, Email, WWW, Terminal, ...)
- Fenster verschieben, vergrößern, schließen, ...
- pulldown-Menüs
- Navigation mit der Maus

1.1.2 Die Etikette im Rechnerpool des Fachbereichsnetzes

- Induktionskarte beim Betreten und Verlassen der Ebene 0 benutzen.
- Niemals einen Computer selbst ausschalten oder neustarten.
- Fehler oder Probleme an das Technik-Team melden: dringende Fehler sofort, alles andere per Email an `service@informatik.uni-bremen.de`.
- Ein Blick in die Dokumentation hilft meist mehr als fragen.
- Sorgsam mit Hardware, Software und Netzlast umgehen.
- Keine personenbezogenen Daten verarbeiten oder ausspionieren.
- Pfiffige Passwörter wählen und geheimhalten.
- Den Bildschirm nicht länger als ca. 10 Minuten blockieren.
- Auf Druckaufträge achten und lange Druckaufträge außerhalb der Stoßzeiten starten.
- Nur einen Arbeitsplatzrechner belegen.
- Web-Surfen mit möglichst geringer Netzlast.
- Ruhe bewahren, die Nachbarn möchten sich auch auf ihre eigenen Aufgaben konzentrieren.
- Das Ausloggen nicht vergessen.

Weitere Informationen:
`http://www.informatik.uni-bremen.de/t/info/text/miteinander.html`

1.2 An- und Abmelden beim System

Zu Beginn jeder Sitzung muss man sich mit seiner Benutzerkennung (account, login name) und seinem Passwort anmelden.

Dann muss die graphische Oberfläche gestartet werden. Auf dem Bildschirm erscheint:

Commands for starting the desktops:

startx	(pure X11)
X.fvwm	(same as startx
X.cde	(Sun Desktop)
X.kde	(K Desktop Environment)
X.gnome	(Gnome Desktop)

Eingabe dann z.B.

```
-> X.kde
```

Zum Abmelden beendet man zunächst KDE (Button unten rechts oder Menü), anschließend die UNIX-Sitzung:

```
-> exit
```

Wenn man den Rechner nur kurzzeitig (!) verlässt, sollte man ihn zur Sicherheit sperren: Mausklick links, Menüpunkt „Lock Screen“, oder Button rechts unten oder Menü.
Passwörter dienen der eigenen Sicherheit und der des Rechnernetzes, deshalb
- nicht vergessen,
- nicht aufschreiben,
- nicht weitergeben (auch nicht per Email).

Die Sicherheit wird regelmäßig überprüft und Passwörter ggfs. gesperrt (zum Entsperren an Technischen Mitarbeiter wenden).

Zum Ändern seines Passworts gibt es einen UNIX-Befehl:

```
-> yppasswd
```

Zunächst wird das alte Passwort eingegeben, dann zweimal das neue.
Ein paar Tipps zur Wahl des Passworts:

- Mindestens 8 Zeichen, davon mindestens 2 Buchstaben und 2 Nichtbuchstaben
- Keine Namen, Geburtsdaten, Telefonnummern u.ä.
- Z.B. Anfangsbuchstaben eines Gedichts/Liedes plus ein Sonderzeichen

1.3 Aufbau von UNIX-Kommandos

Unter UNIX kommuniziert der Benutzer mit dem Computer über Kommandos, die in einem Terminal-Fenster (genauer in einer Shell, siehe Abschnitt 1.7) eingegeben werden. Ein Terminal öffnet man über das KDE-Menü, man schließt es mit
`-> exit`

Kommandos besitzen unterschiedliche Argumente und diverse Optionen. In der Regel folgen Kommandoaufrufe diesem Schema:

Kommando [-Optionen] [Argumente]

Wirklich einheitlich sind die Schemata aber leider nicht.
Leerzeichen und Groß- und Kleinschreibung müssen beachtet werden. Wie ein Kommando aufgerufen wird, ist in den *Manual Pages* ausführlich dokumentiert:
`-> man Kommandoname`
`-> man -k vermuteterKommandoname`

Zum Blättern in der Dokumentation benutzt man die ⟨RETURN⟩-Taste, die ⟨SPACE⟩-Taste und die ⟨b⟩-Taste; mit der ⟨q⟩-Taste verlässt man sie.

Weitere Befehle:

`-> more dateiname` (Anzeigen von Textdateien)

`-> less dateiname` (ebenso, mit mehr Komfort)

`-> who` (wer arbeitet am gleichen Rechner)

`-> finger name` (Suche nach Login- und Benutzernamen)

⟨CTRL⟩⟨c⟩ beendet die laufende Kommandoausführung

1.4 Editoren

Um eine Text-Datei (z.B. den Quelltext für ein Programm) zu erzeugen, zu bearbeiten und zu speichern, benötigt man einen Editor. Dabei geht es um einfache Dateien, keine Texte mit unterschiedlichen Schriftgrößen und -arten, Seitennummerierung, Kopf/Fuß-Zeile, eingebundenen Bildern u.ä. – dafür benutzt man LaTeX, Word oder andere Programme.

Unter UNIX stehen verschiedene Editoren zur Verfügung, z.B.

`vi`	für UNIX-Freaks
`nedit`	für „normale“ Nutzer
`emacs`	auch für „normale“ Nutzer
	viel mehr als ein einfacher Texteditor, aber dafür auch gut geeignet

Zum Bedienen von `nedit` benutzt man seine Menüleiste oder die Tastatur:

Start	Menü von KDE unter „Editoren“ (oder `-> nedit &`)
Ende	Menü „File/Exit“ (oder ⟨CTRL⟩⟨q⟩)
Datei öffnen	Menü „File/Open...“ (oder ⟨CTRL⟩⟨o⟩) Datei in der Liste auswählen
Speichern	Menü „File/Save“ (oder ⟨CTRL⟩⟨s⟩)

Außerdem verfügt `nedit` über viele komfortable Funktionen: Suchen und Ersetzen, Ausschneiden und Einfügen, Undo, Rechtschreibprüfung u.v.m.

1.5 Drucken

Zum Ausdrucken von Dateien steht unter UNIX der Befehl `lpr` zur Verfügung. Auf den Rechnern der 0.ten Ebene benutzt man jedoch den Befehl `lprx`, z.B.

```
-> lprx dateiname
```

Pro Semester hat jeder Student nur eine bestimmte Anzahl (z.Zt. 300) von freien Druckseiten. Um sein Druckkontingent (*Quota*) nicht zu stark zu belasten, kann man mehrere Seiten auf ein Blatt drucken. Dazu gibt es für `lprx` die Druckoption `pages`, die die Werte 2,3,4,8,10 annehmen kann:

```
-> lprx -Pdruckername -pages -x dateiname
```

In Ebene 0 stehen die Drucker lw0, lw2, lw3 (farbig) zur Verfügung; standardmäßig wird lw0 benutzt, einen anderen Drucker kann man mit der `-P`-Option auswählen. Mit der Option `-x` werden Vorder- und Rückseite bedruckt.

Das Kommando `lpq` gibt eine Liste aller Druckaufträge aus; z.B. in folgender Form:

```
Rank     owner (class)   Job   Files      Total size   (Time)
```

```
active   benner      79   titel.ps   132 438      bytes
Ist      jschmidt    83   daten          488      bytes
2nd      benner      84   zv.text      6 212      bytes
```

Die Zahl in der Spalte `Job` gibt eine Identifikationsnummer (ID) an, mit der man den Druckauftrag löschen kann. Dies geschieht mit
`-> lprm jobid`

Sein Druck-Quota kann man sich anschauen mit
`-> pacc`

1.6 Verzeichnisse und Dateien

Alle Daten, Texte, Programme usw. sind unter UNIX in einer baumartigen Struktur angeordnet:

- Eine *Datei* (file) ist ein „Behälter“, der Daten, Texte, Programme enthält (UNIX-Kommandos sind auch Dateien).
- Ein *Verzeichnis* (auch Ordner, directory) ist eine Zusammenfassung mehrerer Dateien und Ordner. Durch Verzeichnisse kann man zusammengehörige Dateien bündeln und Daten strukturieren. Verzeichnisse, die in Verzeichnissen enthalten sind, nennt man auch *Unterverzeichnisse*.
- Sämtliche Dateien und Verzeichnisse in einem UNIX-Dateisystem befinden sich in Unterverzeichnissen eines großen Hauptverzeichnisses, der sogenannten *Wurzel*. Diese wird mit `/` bezeichnet, ansonsten dient der Schrägstrich zum Trennen von Verzeichnisnamen in einem Pfad.
- Ein *Verweis* (link) verweist auf eine Datei oder ein Verzeichnis und verhält sich dann wie eine Datei bzw. ein Verzeichnis, nur dass der Speicherplatz nur einmal benötigt wird. Wir werden davon in dieser Einführung keinen Gebrauch machen.

1.6.1 Dateien und Pfade

Oft muss man einem Kommando den Namen einer Datei oder eines Verzeichnisses übergeben, mit dem das Kommando irgendetwas anstellen soll.

- Die einfachste Möglichkeit ist, allein den Dateinamen anzugeben. Das funktioniert, wenn sich die Datei im *aktuellen Verzeichnis* befindet (Abschnitt 1.6.3).

```
more editor.txt
```

- Angenommen die Datei befindet sich in einem Unterverzeichnis, dann kann man dem Dateinamen mit Schrägstrich getrennt die übergeordneten Verzeichnisnamen voranstellen. Diese Liste von Verzeichnissen nennt man *Pfad*, denn sie beschreibt gewissermassen den Weg zu einer Datei.

  ```
  more vorlesungen/rechprak/texte/editor.txt
  ```

- Angenommen die Datei befindet sich im direkt übergeordneten Verzeichnis, darauf kann man mit dem Verweis `..` zugreifen:

  ```
  more ../editor.txt
  ```

  Das lässt sich auch kombinieren:

  ```
  more ../../texte/editor.txt
  ```

  zeigt die Datei `editor.txt` an, welche sich im Unterverzeichnis `texte` des Überüberverzeichnisses des aktuellen Verzeichnisses befinden muss.

- Der Punkt `.` bezeichnet ein Verzeichnis selbst,

  ```
  more ./editor.txt
  ```

  bedeutet das gleiche wie

  ```
  more editor.txt
  ```

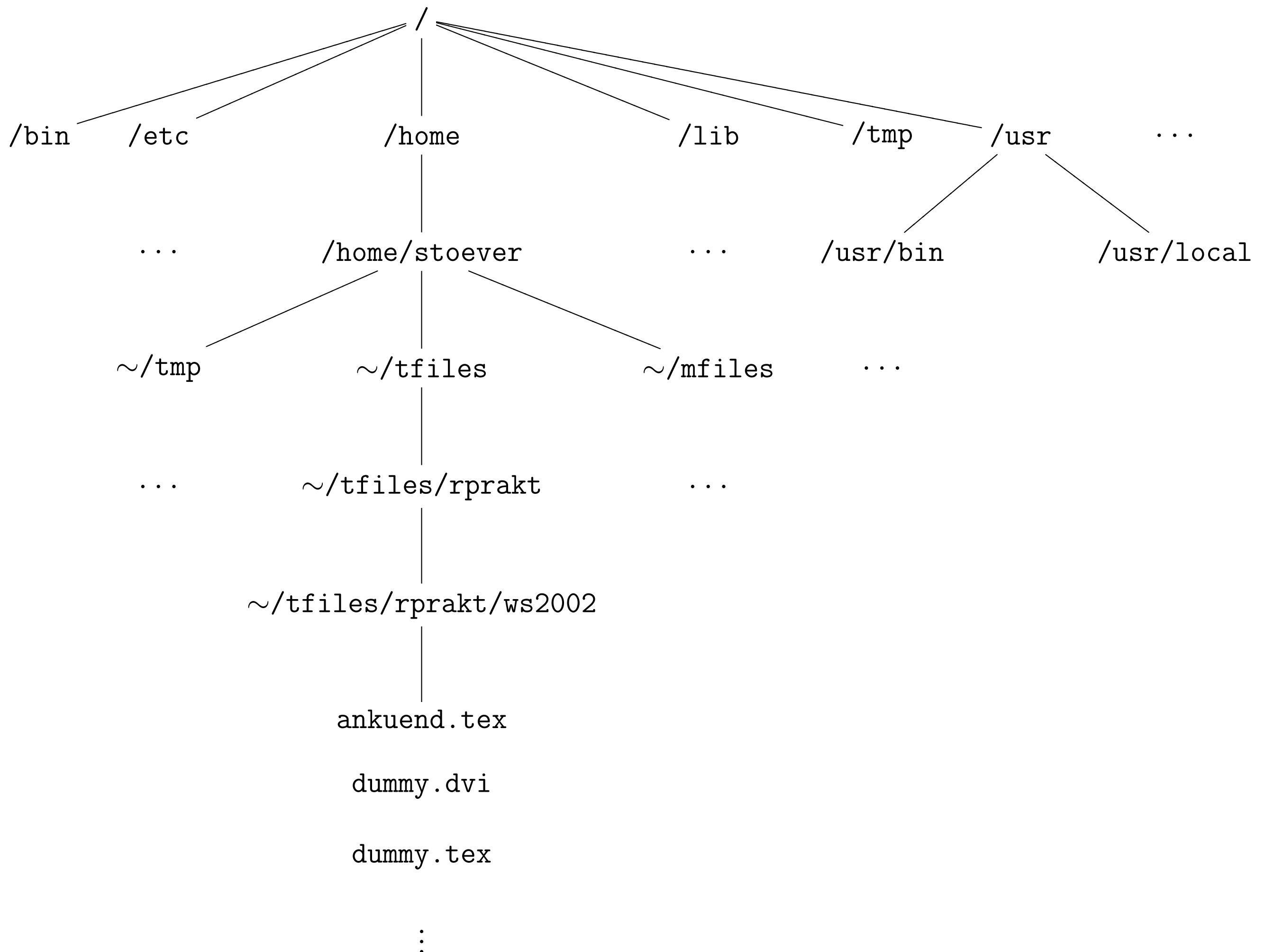
  trotzdem ist der Punkt nicht überflüssig, denn wenn ein Kommando ein Verzeichnis als Argument erwartet, dann kann man mit einem einfachen Punkt das aktuelle Verzeichnis übergeben.

- Die bisher aufgezählten Pfade waren *relative Pfade* denn sie gingen immer vom aktuellen Verzeichnis aus. Im Gegensatz dazu gibt es *absolute Pfade*. Diese beginnen mit einem Schrägstrich `/`, dem Wurzelverzeichnis:

  ```
  more /home/meyer/editor.txt
  ```

1.6.2 Grundgerüst des Verzeichnisbaumes

Der Verzeichnisbaum unter UNIX hat in etwa folgende Struktur:



Benutzer legen ihre privaten Daten in ihrem *Heimverzeichnis* ab. Die Heimverzeichnisse aller Nutzer befinden sich im Verzeichnis `/home`, das Heimverzeichnis des Benutzers mit dem Nutzerkürzel `meyer` also in `/home/meyer`. Dabei ist ~ eine Abkürzung für das Heimverzeichnis des Nutzers und `$HOME` die entsprechende Shell-Variable (Abschnitt 1.7) mit dem Pfad des Heimverzeichnisses als Wert.
Nach dem Einloggen befindet man sich im eigenen *Heimverzeichnis*.

1.6.3 Navigation im Verzeichnisbaum

Um sich im Dateibaum zu bewegen, zum Erzeugen oder Löschen von Dateien und Verzeichnissen gibt es eine Reihe von Befehlen:

`-> pwd`	print working directory zeigt das aktuelle Verzeichnis an
`-> cd ..`	change directory Wechsel ins übergeordnete Verzeichnis
`-> cd verzeichnis`	Wechsel in Unterverzeichnis (relativer Pfad)
`-> cd /home/meyer`	Wechsel ins home-Verzeichnis von meyer (absoluter Pfad)
`-> cp datei1 datei2`	copy Kopieren von Dateien – Vorsicht
`-> cp datei1 verzeichnis`	Kopiere Datei ins Unterverzeichnis
`-> mv datei1 datei2`	move Verschieben (Umbenennen) von Dateien
`-> rm datei`	remove Löschen einer Datei – Vorsicht!
`-> rm -i datei`	Löschen mit Sicherheitsabfrage
`-> rm -R verzeichnis`	rekursives Löschen von Verzeichnissen mit dessen Inhalt – VORSICHT!
`-> mkdir`	make directory Anlegen von neuen Verzeichnissen
`-> rmdir`	remove directory Löschen von leeren Verzeichnissen

1.6.4 Ausgabe von Inhaltsverzeichnissen

Den Inhalt eines Verzeichnisses kann sich mit `ls` (Kurzform) oder `ll` (Langform) anschauen. Mit `ls -a` sieht man auch die „versteckten“ Dateien (Systemdateien, Voreinstellungen u.ä.).

```
total 1836
drwxr-xr-x    5 stoever  wimi         512 Oct 17 11:10 KUTTA_sicherung
-rw-r--r--    1 stoever  wimi       35328 Aug 28 15:18 Kontrakt_FB3.doc
drwx------   18 stoever  wimi        1024 Feb  5 14:21 Mail
drwx------    4 stoever  wimi         512 Aug 15 08:30 fortran
drwxr-xr-x   11 stoever  wimi        1536 Dec 10 12:52 images
-rw-r--r--    1 stoever  wimi       18874 Dec 20 13:18 kontrakt.txt
drwxrwxrwx   16 stoever  wimi         512 Jan 17 14:41 mfiles
drwxrwxr-x    4 stoever  wimi        1536 Feb  4 10:08 public_html
-rw-r--r--    1 stoever  wimi     1296336 Feb  4 08:44 teschke.tar.gz
drwx--x--x   20 stoever  wimi        2048 Feb  4 12:26 tfiles
drwxrwxrwx    3 stoever  wimi        1536 Jan 28 17:21 tmp
drwxr-xr-x   17 stoever  wimi        2560 Feb  3 15:41 word
```

Man sieht

- die Zugriffsrechte,
- einen Referenzzähler,
- den Besitzer der Datei bzw. des Verzeichnisses,
- die Gruppe des Besitzers,
- die Größe der Datei (in Bytes),
- das Datum der letzten Änderung und
- den Namen der Datei bzw. des Verzeichnisses.

Datei- und Verzeichnisnamen bestehen aus einer Folge von Buchstaben und Zahlen (bis zu 256 Zeichen) und dürfen keine Sonderzeichen enthalten. Die Groß- und Kleinschreibung wird unterschieden. Man sollte „sprechende“ Namen verwenden, damit man selbst (und vielleicht auch andere) vermuten kann, was die Datei enthält. Oft sind Dateinamen als `name.extension` aufgebaut, wobei die Endung über den Typ der Datei Auskunft gibt, z.B.

`name.i3`	`Modula-3`-Schnittstelle,
`name.m3`	`Modula-3`-Programm,
`name.c`	`C`-Programm,
`name.o`	Objektdatei, wird vom Computer aus .c erzeugt,
`name.tex`	TEX-Datei,
`name.ps`	Postscript-Datei,
`name.doc`	Word-Dokument,
`name.html`	Datei im HTML-Format,
`.name`	versteckte Dateien: Systemdateien, Einstellungen, etc.

1.6.5 Zugriffsrechte für Dateien und Verzeichnisse

Die Zugriffsrechte legen fest, welche Nutzer wie auf Dateien und Verzeichnisse zugreifen dürfen. Dabei werden Dateien und Verzeichnisse im Wesentlichen gleich behandelt. Ein Verzeichnis erkennt man daran, dass an der ersten Stelle ein `d` steht.

Es gibt drei Klassen von Benutzern:

user	Eigentümer
group	Gruppe
others	Andere

Und drei Arten von Zugriffsrechten:

`r`	read	Leserechte
`w`	write	Schreibrechte
`x`	eXecute	Ausführbarkeit bei Dateien, Zutritt zu Verzeichnissen

Die Zugriffsberechtigung ergibt sich aus der Kombination der Benutzerklassen und der Zugriffsarten, z.B.

`r-- r-- r--`	Alle können lesen, aber keiner darf schreiben.
`rwx --- ---`	Der Eigentümer darf alles, alle anderen nichts.
`rw- r-- r--`	Alle dürfen lesen, aber nur der Eigentümer darf schreiben.

Der Besitzer einer Datei oder eines Verzeichnisses kann die Zugriffsrechte ändern. Man kann dafür *absolute* oder *symbolische* Modi verwenden:

`chmod mode dateiname` change modus

1.6.6 Absolute und symbolische Modi

Absolute Modi:

400	read	Leseberechtigung des Eigentümers
200	write	Schreibberechtigung des Eigentümers
100	execute	Ausführungsberechtigung des Eigentümers
40	read	Leseberechtigung der Gruppe
20	write	Schreibberechtigung der Gruppe
10	execute	Ausführungsberechtigung der Gruppe
4	read	Leseberechtigung der Anderen
2	write	Schreibberechtigung der Anderen
1	execute	Ausführungsberechtigung der Anderen

z.B. `rw- r-- r--` ergibt $400 + 200 + 40 + 4 = 644$, also setzt

```
-> chmod 644 datei1
```

die Rechte für diese Datei entsprechend.

Symbolische Modi werden durch Angabe von Benutzerklasse, Operation und Zugriffsrecht definiert:

```
[who] op permission
```

Dabei bedeuten die einzelnen Ausdrücke das Folgende:

who:	`u`	User
	`g`	Gruppe
	`o`	Andere
	`a`	Alle (Voreinstellung, falls `who` weggelassen)
op:	`+`	Rechte zufügen
	`-`	Rechte wegnehmen
	`=`	Rechte explizit setzen
permission:	`r`	Leserechte
	`w`	Schreibrechte
	`x`	Ausführungsrechte

Z.B.: Ändern von `rwx rw- r--` in `rw- r-- r--` geschieht durch

```
-> chmod u-x, g-w dateiname
```

1.7 UNIX-Shells

1.7.1 Kommandos

Öffnet man ein Terminal-Fenster, dann läuft im Hintergrund eine *Shell*, die die eingegebenen Kommandos interpretiert und dann ausführt bzw. eine Fehlermeldung herausgibt. Die Shell legt sich wie eine Schale um den UNIX-Kern und vereinfacht den Dialog zwischen Benutzer und Betriebssystem.
Von den verschiedenen Shells wird hier standardmäßig die `bash` (*Bourne again Shell*) benutzt.

Ein paar Vorteile:

- Editieren in der Kommandozeile
- Zurückholen bereits eingegebener Befehle mit den Cursor-Tasten
- Vervollständigung von Dateinamen mit der `<TAB>`-Taste
  (wenn die Vervollständigung eindeutig möglich ist)
- Definition eigener Namen für häufig benutzte Kommandos mit dem `alias`-Mechanismus

Zu jedem UNIX-Kommando existiert eine entsprechende, ausführbare Datei. Diese ist in der Regel eine Binärdatei, also kein Text, den man sich anschauen kann. Die meisten dieser Dateien liegen in Verzeichnissen wie `/bin`, `/usr/bin`.
Damit die Shell den Befehl ausführen kann, muss sie die dazugehörige Datei finden. Dafür hat sie einen Suchpfad, der in der Systemvariablen `PATH` gespeichert ist. Man kann sich diesen Pfad anzeigen lassen mit

```
-> echo $PATH
```

Ein weiteres Verzeichnis kann man dem Pfad z.B. mit

```
-> export PATH=$PATH:/home/name/MeineBefehle
```

hinzufügen. Dann schaut die Shell auch in `/home/name/MeineBefehle` nach, ob dort eine ausführbare Datei des eingegebenen Namens existiert.
Erfahrene UNIX-Nutzer schreiben und benutzen sogenannte *Shell-Skripte*: Programme, die innerhalb einer Shell ausgeführt werden (ohne Kompilation).

1.7.2 Platzhalter in Dateinamen

Durch Benutzung spezieller Sonderzeichen, der *Platzhalter*, auch *Jokerzeichen* oder *Wildcards* genannt, kann man Dateinamen abkürzen oder mehrere Dateien gleichzeitig ansprechen. Die Shell analysiert das verwendete Suchmuster, ersetzt es durch alle dazu passenden Dateinamen und übergibt das Ergebnis an das angegebene Kommando.

`?` steht für genau ein Zeichen (?? für zwei, ??? für drei, etc.).
`*` steht für kein, ein oder beliebig viele Zeichen.
`[ ]` steht für eine Zeichenauswahl.

Beispiele:

`-> ls prog?.c`	listet `prog1.c`, `prog2.c`, `progx.c`, ... (soweit im Verzeichnis vorhanden)
`-> ls prog[0-9].c`	listet `prog`n`.c`, wobei $n \in \{0, 1, ..., 9\}$
`-> rm *.ps`	löscht alle Postscript-Dateien (in diesem Verzeichnis)

1.7.3 Umleitung

Die Eingabe für Kommandos erfolgte bis jetzt immer über die Tastatur und das Ergebnis wurde auf dem Bildschirm ausgegeben. Man kann aber sowohl Ein- wie Ausgabe auch umleiten, z.B.

`-> ll > datei`
Das Inhaltsverzeichnis wird in eine Datei geschrieben.

`-> cat datei1 datei2 > datei3` (`cat` wie concatenate)
Verbinde `datei1` und `datei2`, schreibe das Ergebnis in `datei3`

`-> cat datei4 >> datei3`
Hänge `datei4` noch an `datei3` an.

`-> mail nutzer < datei` (`mail` ist ein UNIX-internes Email-Programm)
Schicke die Datei per `mail` an den Nutzer.

Eine besondere Form der Umleitung ist die *Pipe* (von Pipeline). Sie schaltet mehrere Kommandos hintereinander, wobei die Ausgabe des einen Kommandos als Eingabe des nächsten dient. Beispiel:

`-> ll | sort -r -k5`
Das Inhaltsverzeichnis wird der Größe nach sortiert.

Man kann alle diese Möglichkeiten auch miteinander kombinieren, z.B.

`-> ll | sort -r -k5 | lprx`
Inhaltsverzeichnis sortieren und dann ausdrucken.
`-> ll | sort -r -k5 > datei`
Inhaltsverzeichnis sortieren und dann in eine Datei schreiben.

Weitere Sonderzeichen sind z.B.

&	Programm vom Terminal abkoppeln, siehe Abschnitt 1.8
$	Shell-Variable abrufen
“	Ausgabe von Kommando als Argument benutzen
;	Kommandos hintereinander ausführen
&&	Kommandos hintereinander ausführen bis Fehler auftritt
␣	Leerzeichen
Zeilenende	Kommando starten

Mit \ (backslash) wird die Bedeutung von Sonderzeichen aufgehoben, z.B.

```
-> rm datei1 datei2 datei3 \
> datei4 datei5
```

Setzt man einen Befehl (oder einen Teil davon) in einfache Hochkommata, wird dieser Text genauso (ohne Ersetzungen o.ä.) übernommen.

1.7.4 Vorverarbeitung der Kommandos

Noch bevor das eigentliche Kommando (Programme) gestartet wird, erledigt die Shell einige Aufgaben. Es ist wichtig zu wissen, welche Aufgaben die Shell übernimmt und welche die Kommandos. Services, die die Shell anbietet, können für alle Kommandos genutzt werden, werden aber auch manchmal ungewollt in Anspruch genommen, ohne dass das aufgerufene Programm etwas dagegen tun könnte.
Nach dem man eine Kommandozeile mit ⟨RETURN⟩ abgeschlossen ist, passiert in etwa folgendes:

- Die Dateien zur Umleitung von Ein- und Ausgabe werden geöffnet. Im Falle der Ausgabe wird die entsprechende Datei geleert.

- Shell-Variablen, denen ein $ vorangestellt wurde, werden durch ihren Inhalt ersetzt. Beispiel:

  ```
  echo $PWD
  ```

  könnte zu

```
echo /home/ramlau
```

werden.

- Texte, die Platzhalter enthalten, werden entsprechend der vorhandenen Dateien im aktuellen Verzeichnis durch eine durch Leerzeichen unterteilte Liste ersetzt. Gibt es keine Datei, die zu dem Muster passt, wird das Muster beibehalten. Beispiel:

  ```
  ls *.tex *.bla
  ```

  könnte zu

  ```
  ls skript.tex anhang.tex *.bla
  ```

  werden.

  Vorsicht: Die Shell weiß nichts über die Bedeutung der Argumente für ein Kommando, sie ersetzt stur alles was wie ein Dateinamenmuster aussieht, auch wenn es z.B. ein Suchmuster für `grep` ist.

- Das eigentliche Kommando wird mit den umgewandelten Argumenten aufgerufen. Wie das Kommando letztlich aufgerufen wird, bekommt man zu sehen, wenn die `bash` mit der Option `-x` gestartet wird.

Die Bequemlichkeiten, die eine Shell durch ihre Automatiken erlaubt, können sich auch schnell rächen:
Beispiel: Wir suchen alle Zeilen eines Textes in denen ein > vorkommt und schreiben:

```
grep > text.txt
```

Die Argumente > und `text.txt` erreichen das Programm `grep` allerdings nie, denn die Shell hat bereits die Datei `text.txt` zum Zwecke der Ausgabeumleitung gelöscht und ruft `grep` ohne jegliche Argumente auf.

1.8 Prozesse

UNIX ist nicht nur ein Mehrbenutzer- (Multi-User) sondern auch ein Multitasking-Betriebssystem, d.h. jeder kann parallel mehrere Prozesse (Kommandos, Programme, ...) laufen lassen. Tatsächlich laufen ständig, ohne dass man es merkt, diverse, vom System ausgelöste Prozesse.

Um ein Terminal nicht zu blockieren, kann man Prozesse auch so starten, dass sie im Hintergrund laufen, z.B.

```
-> nedit &
```

Es öffnet sich das `nedit`-Fenster und im Terminal kann weitergearbeitet werden.

Mittels ⟨CTRL⟩⟨z⟩ und `bg` kann man einen Prozess in den Hintergrund verlagern, mit `fg` wird er wieder in den Vordergrund geholt.

Eine Übersicht über die laufenden Prozesse liefert der `ps`-Befehl. Durch

```
-> ps -u nutzer
```

bekommt man eine Liste seiner eigenen, aktiven Prozesse. Dabei ist für jeden Prozess eine eindeutige Kennzahl `PID` angegeben, die man zum außerplanmäßigen Beenden eines Prozesses benötigt (wenn ⟨CTRL⟩⟨c⟩ nicht möglich ist):

```
-> kill PID
-> kill -9 PID
```

Das `kill`-Kommando versucht, den Prozess so ordentlich wie möglich zu beenden. Die zweite Variante sollte nur „im Notfall“ verwendet werden, weil dann der Prozess u.U. unordentlich terminiert wird.

1.9 Arbeiten auf entfernten Rechnern

Unter UNIX ist es problemlos möglich, Rechner so zu vernetzen, dass man auf einem anderen Rechner arbeiten kann, als dem, vor dem man sitzt.
Auf einen fremden Rechner begibt man sich mit

`ssh` nutzer@rechner

. Eventuell muss man noch sein Passwort eingeben. Ist man auf dem Rechner angelangt, kann man bereits einfache Kommandozeilenprogramme starten, neben `ls`, `cp`, `rm` usw. natürlich auch den Editor `vi` oder das Textsatzsystem `latex`.
Es ist auch möglich Programme mit grafischer Benutzeroberfläche auf dem fremden Rechner zu starten, aber die Ausgabe auf dem eigenen Rechner zu sehen. Da alle Grafikdaten von dem entfernten Rechner zum eigenen übertragen werden müssen,

benötigt das ganze viel Netzbandbreite und ist auch nicht sonderlich schnell. Aber es geht!
Zunächst muss man anderen Rechnern erlauben auf dem eigenen Bildschirm Fenster zu öffnen:

```
thielema@peano> xhost lie
lie being added to access control list
```

Dann muss man auf dem anderen Rechner die Ausgabe auf den eigenen Bildschirm lenken. Den Ausgabebildschirm gibt man durch Setzen der Umgebungsvariable `DISPLAY` an. Wenn man nicht weiß, wie der eigene Rechner erreicht werden kann, weil zum Beispiel die IP-Adresse dynamisch vergeben wurde, kann man das mit dem Kommando `who` herausbekommen:

```
thielema@peano> ssh lie

thielema@lie> who
ramlau   pts/0    Feb 16 09:08
...
ramlau   pts/11   Feb 17 09:53
thielema pts/12   Feb 19 12:41 (peano.math.uni-bremen.de)

thielema@lie> export DISPLAY=peano.math.uni-bremen.de:0
```

1.10 Weitere nützliche Befehle

Ohne Anspruch auf Vollständigkeit seien noch ein paar wichtige UNIX-Befehle aufgelistet:

`find . -name '*.c' -print`	Suche nach Dateinamen
`grep suchmuster datei`	Suche nach Zeichenketten in Dateien
`which kommando`	Suche nach dem Pfad von Kommandos
`quota`	Zeigt an, ob man mehr als erlaubt Plattenplatz beansprucht. Bei Überschreitung der vorgegebenen harten Grenze kann man sein Benutzerkonto praktisch nicht mehr benutzen, da immer Platz für temporäre Dateien gebraucht wird.
`du -ks`	Berechnet den Speicherplatz, der vom Verzeichnis inklusiv Unterverzeichnis belegt wird.

`gzip filename`	Komprimieren von Dateien, Ausgabe filename.gz
`gunzip filename.gz`	Dekomprimieren von Dateien
`tar -cvf archivname.tar dirname1`	Anlegen von Datei-Archiven
`tar -tf archivname.tar`	Zeigt den Inhalt der tar-Datei
`tar -xvf archivname.tar`	Zeigt den Inhalt der tar-Datei
`slogin rechner -lnutzer`	Anmelden auf einem anderen Rechner
`sftp rechner`	secure file transfer protocol – Dateien zwischen Rechnern austauschen
`a2ps -r -f 10 -P printer datei`	ASCII-Text im Querformat drucken
`grp -setup boerse`	Anlegen der Gruppe boerse
`grp -show boerse`	Anzeigen der Daten der Gruppe
`grp -invite boerse username`	Einladen von Teilnehmern
`chgrp boerse BOERSE`	Ändern der Gruppenzugehörigkeit vom Verzeichnis BOERSE
`chmod 770 BOERSE`	Nur Mitglieder der Gruppe dürfen in das Verzeichnis
`chmod g+s BOERSE`	Alle angelegten Dateien gehören der Gruppe boerse
`scp nutzer@rechner:datei .`	Dateien zwischen Nutzern oder Rechnern austauschen

Kapitel 2

Algorithmen

Versuch einer Definition:
Ein *Algorithmus* ist eine genaue Beschreibung, wie man eine gegebene Aufgabe lösen kann. Er besteht aus einer endlichen Folge von Schritten, deren korrekte Abarbeitung die gestellte Aufgabe löst.

Beispiele für Algorithmen: Kochrezepte, Montageanleitungen, Strickmuster, ...

Die Durchführung eines Algorithmus liefert einen *Prozess*. Ein *Prozessor* ist eine Einheit, die einen Algorithmus durchführt, z.B. ein Mensch oder die CPU des Computers.
Aber ein Algorithmus ist **unabhängig** von seiner Umsetzung in einen Prozess und seiner Abarbeitung durch einen Prozessor.

Die Entwicklung eines Algorithmus ist um so schwieriger, je komplexer die zu lösende Aufgabe ist. Deshalb zerlegt man die Aufgabe in Teilprobleme, zu deren Lösung man dann Algorithmen entwickelt. Oder man zerlegt die Teilaufgabe wiederum in Teilaufgaben usw.
Dieses methodische Vorgehen heißt *schrittweise Verfeinerung*.

Zu einem Algorithmus gehören (in der Regel) Eingabedaten und Ausgabedaten. Der Ausführungsteil besteht aus einer endlichen Folge von Anweisungen, die eine Reihe von typischen Bausteinen benutzen, die man auch ineinander verschachteln kann.

Wichtige, prinzipielle Fragen beim Algorithmenentwurf sind

- Korrektheit: Löst der Algorithmus die gegebene Aufgabe korrekt?
- Komplexität: Vergleich von Algorithmen nach Zeit- und Speicherbedarf

- Berechenbarkeit: Kann es einen Algorithmus geben, der das gegebene Problem löst?

2.1 Sequenzen

Ein Algorithmus besteht aus einer Folge von Schritten, so dass

- zu einem bestimmten Zeitpunkt nur ein Schritt ausgeführt wird (serieller/sequentieller Algorithmus im Unterschied zu parallelen Algorithmen),
- jeder Schritt genau einmal ausgeführt wird,
- die Reihenfolge der Ausführung der Schritte die gleiche Folge ist, in der sie niedergeschrieben sind,
- mit Beendigung des letzten Schrittes der Algorithmus endet.

2.1.1 Beispiel: Algorithmus zur Zubereitung einer Tasse Pulverkaffee

1. *Koche Wasser*
2. *Gib Kaffeepulver in die Tasse*
3. *Fülle Wasser in die Tasse*

2.1.2 Verfeinerung

1.1 *Fülle Kessel*
1.2 *Schalte Kessel an*
1.3 *Warte, bis es kocht*
1.4 *Schalte Kessel aus*

2.1 *Öffne Kaffeeglas*
2.2 *Entnehme einen Löffel Kaffee*
2.3 *Kippe Löffel in die Tasse*
2.4 *Schließe Kaffeeglas*

3. *Fülle Wasser in die Tasse*

Der Algorithmus ist unflexibel, die Ausführung starr, er enthält keine Alternativen, wenn ein Schritt nicht durchführbar ist.

Was passiert z.B., wenn das Kaffeeglas leer ist oder was ist zu tun, wenn mehrere Tassen Kaffee gekocht werden sollen?

2.2 Auswahl (Selektion)

Man benötigt ein Konstrukt, dass eine alternative Aktion vorsieht, falls ein Schritt nicht möglich ist:

FALLS	Bedingung
DANN	Folge 1 von Schritten
SONST	Folge 2 von Schritten

2.2.1 Beispiel: Kaffeekochen Verfeinerung von Schritt 2.1 durch Auswahl

2.1.1 *Nehme Kaffeeglas aus dem Fach*

2.1.2 **FALLS** *Kaffeeglas leer ist*
DANN *Nehme neues Kaffeeglas*

2.1.3 *Entferne Deckel vom Kaffeeglas*

Der **SONST**-Teil kann hier entfallen, weil keine Alternativ-Aktion nötig ist.

Auswahl-Aktionen können auch ineinander geschachtelt werden („nesting“):

2.1.1 *Nehme Kaffeeglas aus dem Fach*

2.1.2 **FALLS** *Kaffeeglas leer ist*

FALLS *Weiteres Kaffeeglas vorhanden*

DANN *Nehme neues Kaffeeglas*

SONST *Breche Kaffeekochen ab*

2.1.3 *Entferne Deckel vom Kaffeeglas*

Problem: Möglicherweise ist nicht von vornherein klar, wieviele Fälle unterschieden werden müssen.

2.3 Schleifen (Wiederholung, Iteration)

Wir benötigen ein Konstrukt, in dem eine Anweisung solange wiederholt wird, bis die Teilaufgabe gelöst ist – *Schleife*n. Es gibt drei Typen:

- **WIEDERHOLE** *Folge von Anweisungen* **BIS** *Bedingung erfüllt*
  Repeat-Until-Schleife, Schleife mit Nachtest
- **SOLANGE** *Bedingung erfüllt* **FÜHRE AUS** *Anweisungsfolge*
  While-Schleife, Schleife mit Vorabtest
- **FÜR** *Zähler* **VON** *Anfang* **BIS** *Ende* *Anweisungsfolge*
  For-Schleife, Zählschleife

Achtung: Man muss die Abbruchbedingung so formulieren, dass die Schleife in jedem Fall nach endlich vielen Durchläufen beendet wird (z.B. kann man eine maximale Anzahl an Schritten vorgeben). Ebenso ist zu vermeiden, dass die Abbruchbedingung nicht erfüllt ist, aber keine weiteren Schritte mehr möglich sind.

2.3.1 Beispiel: Kaffeekochen Schritt 2.1 mit Wiederholungsschleife

2.1.1 **WIEDERHOLE**
Nehme Kaffeeglas aus dem Fach
BIS *Kaffeeglas nicht leer oder kein Glas mehr vorhanden*

2.1.2 **FALLS** *Weiteres Kaffeeglas vorhanden*
DANN *Entferne Deckel vom Kaffeeglas*
SONST *Breche Kaffeekochen ab*

2.3.2 Beispiel: Maximumsbestimmung

Eingabe: Liste von Zahlen
Ausgabe: Größte Zahl in der Liste

Setze erste Zahl der Liste als bislang größte Zahl

WIEDERHOLE

Lese nächste Zahl in der Liste

FALLS *Zahl größer als bisher größte Zahl*

DANN *Setze diese Zahl als bislang größte Zahl*

BIS *Liste erschöpft*

Schreibe die bislang größte Zahl nieder

Problem: Wenn die Liste nur aus einer Zahl besteht, dann kann die erste Anweisung des Schleifenrumpfs nicht ausgeführt werden. Ein Prozessor, der diesen Algorithmus ausführen soll, befindet sich so in einem undefinierten Zustand.

Lösung: Schleife mit Vorabtest

Setze erste Zahl der Liste als bislang größte Zahl

SOLANGE *Liste nicht erschöpft* **FÜHRE AUS**

Lese nächste Zahl in der Liste

FALLS *Zahl größer als bisher größte Zahl*

DANN *Setze diese Zahl als bislang größte Zahl*

Schreibe die bislang größte Zahl nieder

2.3.3 Beispiel: EUKLIDischer Algorithmus zur Berechnung des größten gemeinsamen Teilers

Der größte gemeinsame Teiler von x und y mit $\{x, y\} \subset \mathbb{N}_0$ ist die natürliche Zahl q, so dass x und y Vielfache von q sind und q wiederum Vielfaches jedes anderen gemeinsamen Teilers von x, y ist.
Bezeichnung: $\text{ggT}(x, y)$
Folgerungen:

$$\begin{aligned} \text{ggT}(x, y) &= \text{ggT}(y, x) \\ \text{ggT}(x, 0) &= x \end{aligned}$$

Eingabe: Zwei natürliche Zahlen x und y
Ausgabe: $\text{ggT}(x, y)$

Lies x und lies y

SOLANGE $y \neq 0$ **FÜHRE AUS**

Berechne Rest von x/y

Ersetze x durch y

Ersetze y durch den Divisionsrest

Schreibe als Ergebnis x

2.3.4 Beispiel: Berechnung von Potenzen x^n

Eingabe: Reelle Zahl x, natürliche Zahl n
Ausgabe: x^n

Übernehme die Werte von x und n

Setze Produkt auf 1

FÜR k **VON** 1 **BIS** n

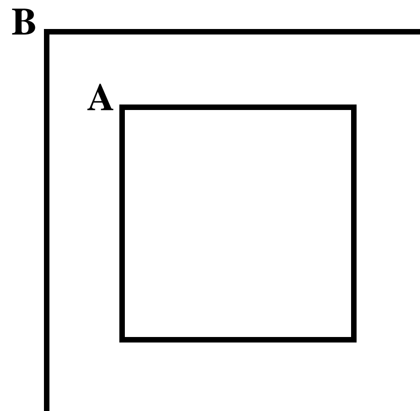
Multipliziere Produkt mit x

Schreibe Produkt nieder

2.4 Funktionen

Funktionen erleichtern die schrittweise Verfeinerung eines Algorithmus und erlauben die wiederholte Verwendung von Algorithmenteilen, ohne diese Teile erneut eingeben zu müssen.

Beispiel: Zeichengerät/Plotter zeichnet zwei konzentrische Quadrate



Verfügbare Funktionen	Funktionalität
`forward(`x`)`	Bewege Stift um x cm vorwärts
`right(`α`)`	Drehe Stift um α° nach rechts
`down(`Y`)`	Senke Stift auf den Punkt Y
`up`	Hebe Stift vom Papier ab

`Zeichne_Quadrat(`x, Y`)`	`down(`Y`)`
	WIEDERHOLE vier mal
	    `forward(`x`)`
	    `right(90)`
	`up`

2.4.1 Algorithmus: Zeichne zwei konzentrische Quadrate

`Zeichne_Quadrat(`$10, A$`)`

`Zeichne_Quadrat(`$15, B$`)`

2.5 Vom Algorithmus zum Programm

Computer haben eine eigene, vom Typ der CPU abhängige *Maschinensprache*. Unter *Assembler* versteht man die mnemotechnische Umsetzung der Maschinensprache: Die Steuerbefehle sind durch Zeichen statt durch pure Bits verschlüsselt.

Größere Projekte in Assembler zu schreiben, ist zu aufwändig und zu fehleranfällig. Deshalb wurden (und werden) höhere Programmiersprachen entwickelt, z.B.

- C, FORTRAN, Pascal, Modula-2, Basic, Algol, ... (imperative Sprachen)
- C++, Java, Modula-3, ... (imperative objektorientierte Sprachen)
- LISP, Haskell, ... (funktionale Programmiersprachen)

Daneben gibt es diverse Programmiersprachen für spezielle Anwendungen, wie SQL (Datenbanken), HPC Fortran (Parallelrechnen) oder MATLAB.

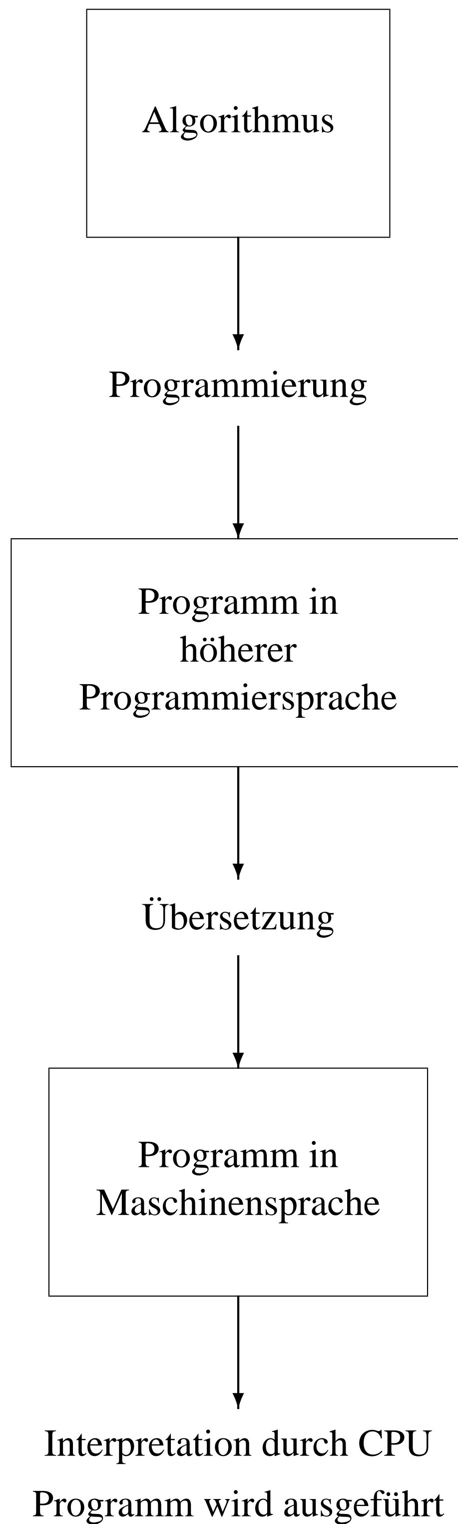
Höhere Programmiersprachen sind in der Regel rechnerunabhängig, ihre Beherrschung erfordert die Kenntnis von

- Alphabet, d.h. der zugrundeliegende Zeichenvorrat;
- Syntax, d.h. zulässige Zeichenkombinationen (insbesondere Schlüsselwörter und Befehle) und ihre Grammatik;
- Semantik, d.h. Bedeutung und Sinnhaftigkeit formal korrekter Sätze.

Damit ein Algorithmus auf einem Computer ausgeführt werden kann, muss er in einer Programmiersprache implementiert werden, dieses Programm muss in die Maschinensprache des Prozessors übersetzt (*kompiliert*) werden, so dass ein ausführbares Programm (eine Folge von Befehlen in Maschinensprache) entsteht.

- Compiler: Der gesamte Quelltext wird übersetzt und ein ablauffähiges Programm erzeugt.
- Interpreter: Es wird je ein Befehl übersetzt und abgearbeitet.

Noch einmal: Ein Algorithmus ist **unabhängig** von der Programmiersprache, diese ist lediglich ein Mittel zum Zweck!


Algorithmus
Programmierung
Programm in
höherer
Programmiersprache
Übersetzung
Programm in
Maschinensprache
Interpretation durch CPU
Programm wird ausgeführt

2.6 Einige Grundsätze zur Programmierung

Ein gutes Computerprogramm sollte ein paar Anforderungen erfüllen:

- Verständlichkeit
- Fehlerfreiheit
- Sicherheit
- Effizienz
- Systemunabhängigkeit

Um insbesondere dem ersten Punkt zu genügen, sollten folgende Grundsätze befolgt werden:

- Am Anfang des Programms und jeder darin definierten Funktion steht ein Programmkopf mit
  - Autor, Datum
  - Kurzbeschreibung / Zweck / Methode
  - Beschreibung der Ein- und Ausgabeparameter
  - evtl. Referenzen oder Versionsnummer
- Kompliziertere Anweisungen sollten durch Kommentare erläutert werden.
- Jedes `Modula-3`-Programm hat die feste Struktur:
  - Kopf
  - Import anderer Module
  - Deklarationen und Prozeduren
  - Hauptprogramm
  - Ende
- Variablennamen sollten für sich sprechen und gewisse Konventionen beachten (z.B. bezeichnen `i`, `j`, `k`, `l`, `m`, `n` typischerweise Ganzzahl-Variable).
- Konventionen der Sprache einhalten. Bei `Modula-3`: Großgeschrieben werden Modulnamen, Typen, Prozeduren. Kleingeschrieben werden Variablen, Datenverbund-Elemente, Objekt-Methoden. Ein Modul sollte den Namen des implementierten Datentyps tragen. Dieser Typ wird einfach `T` genannt. Beispiel: `Rd.T` für einen Eingabestrom.

- Bibliotheksfunktionen verwenden statt selbst schreiben.
- Eingaben müssen daraufhin überprüft werden, ob sie den gestellten Anforderungen genügen. Gleiches gilt für Eingabeparameter von Funktionen.
- Typen so wählen, dass sie die Realität möglichst genau modellieren.
- Durch zusätzliche Abfragen (`ASSERT`) Invarianten zur Laufzeit überprüfen.
- Warnungen des Übersetzers ernst nehmen. Nach Ursachen suchen, statt die Warnungen zu unterdrücken. Mit Kommentar begründen, falls man doch eine Warnung unterdrückt.

Kapitel 3

Einführung in die Modula-3-Programmierung

3.1 Überblick

3.1.1 Was ist Modula-3?

- Modula-3 ist ein Nachfolger von Modula-2 und Pascal und wurde Ende der 1980er Jahre von DEC (Digital Equipment Corporation) und Olivetti entwickelt.
- Modula-3 ist eine imperative Programmiersprache: Programme bestehen aus aneinandergereihten Anweisungen.
- Modula-3 ist eine strukturierte Programmiersprache: Es gibt kein GOTO, dafür Unterprogramme, Schleifen, Abfragen, etc.
- Modula-3 ist sehr streng, erzieht zu gutem Programmierstil, deckt viele Fehler noch vor Programmstart auf und ist auf Grund seiner Modularisierung sehr gut für große Projekte geeignet.
- Modula-3 ist eine moderne objektorientierte Programmiersprache: Unterstützt werden Objektklassen (nur einfaches Erben), Ausnahmen (Exceptions), automatische Speicherverwaltung (Garbage-Collector), Programmierung allgemeiner Datenstrukturen (Templates), nebenläufige Ausführung (Threads).
- Modula-3 ist eine Systemprogrammiersprache: Es lassen sich sehr effiziente maschinennahe Programme schreiben, z.B. Gerätetreiber oder Betriebssysteme, aber systemnahe und höhere Programmteile sind streng getrennt.
- Trotzdem ist Modula-3 relativ einfach zu erlernen: Die Sprachdefinition umfasst nur 60 Seiten!

3.1.2 Literatur

– László Böszörményi und Carsten Weich: „Programmieren mit Modula-3. Eine Einführung in stilvolle Programmierung.“, Springer, 1995, ISBN 3540579117

– Samuel P. Harbison: „Modula-3“, Prentice Hall, 1992, ISBN 0135963966

– Robert Sedgewick: „Algorithms in Modula-3“, Addison-Wesley, 1993, ISBN 0201533510

– http://www.m3.org/

– Modula-3, Einführung http://www1.cs.columbia.edu/graphics/modula3/tutorial/www/m3_toc.html

– Modula-3, Sprachdefinition http://www.research.compaq.com/SRC/m3defn/html/m3.html

3.1.3 Das nullte und kürzeste Modula-3-Programm

```
MODULE Main;
(* $Id: Empty.m3,v 1.2 2004/01/16 18:15:09 thielema Exp $ *)

BEGIN
END Main.
```

Zum Testen in Texteditor eingeben, in ein neues Verzeichnis Empty/src unter dem Namen Main.m3 speichern, dann im Verzeichnis Empty mit dem Compiler cm3 übersetzen:

```
Empty> cm3
```

Dies erzeugt das ausführbare Programm LINUXLIBC6/prog, das man ausführen kann mit

```
Empty> LINUXLIBC6/prog
```

Es passiert (natürlich) nichts.
Aber man erkennt am Code schon einige Bausteine der Modula-3-Syntax:

- Kommentare, d.h. Programmteile, die vom Compiler ignoriert werden, sind durch (* *) gekennzeichnet.
- Ein Programmteil beginnt immer nach dem Kopf des entsprechenden Konstrukts (z.B. BEGIN für einen Block) und endet mit einem END.
- Eine Datei enthält genau ein Modul, das Hauptmodul heißt Main.
- Jedes Modula-3-Modul besitzt ein Hauptprogramm.

3.1.4 Ein erstes Programm: Ausgabe mit `IO.Put`

Als Erstes soll etwas auf der Konsole ausgegeben werden. Dafür gibt es z.B. die Funktion `IO.Put`, die von `Modula-3` zusammen mit anderen Standard-Funktionen zur Ein-/Ausgabe in dem Modul `IO` zur Verfügung gestellt wird. Mit dem Befehl `IMPORT` wird das Modul in den Programmtext eingefügt.
Man beachte, dass der Aufruf von `IO.Put` mit einem Semikolon abgeschlossen wird.

```
MODULE Main;
(* $Id: ErstesProg0.m3,v 1.2 2004/01/16 18:15:09 thielema Exp $ *)

IMPORT IO;

BEGIN
  IO.Put(Mein erstes Programm in Modula-3!);
END Main.
```

```
ErstesProg0> cm3
new source -> compiling Main.m3
"../src/Main.m3", line 7: syntax error: missing ')' (erstes)
"../src/Main.m3", line 7: Illegal character (33)
2 errors encountered
compilation failed => not building program "prog"
Fatal Error: package build failed
```

Wieso entstehen Fehler? Eingabe von `IO.Put` sollte ein *Textliteral* sein, welches aber durch Anführungszeichen gekennzeichnet sein muss.

```
MODULE Main;
(* $Id: ErstesProg1.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO;

BEGIN
  IO.Put("Mein erstes Programm in Modula-3!");
END Main.
```

```
ErstesProg1> LINUXLIBC6/prog
Mein erstes Programm in Modula-3!ErstesProg1>
```

Warum erscheint das Prompt `ErstesProg1>` hinter dem ausgegebenen Text?
In dem Text, der `IO.Put` übergeben wird, muss noch ein Befehl zum Zeilenumbruch eingebaut werden. Dies geschieht mit dem Steuerbefehl `\n`.

```
MODULE Main;
(* $Id: ErstesProg2.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO;

BEGIN
  IO.Put("Mein erstes Programm in Modula-3!\n");
END Main.
```

```
ErstesProg2> LINUXLIBC6/prog
Mein erstes Programm in Modula-3!
ErstesProg2>
```

Hinweis: Zeichenfolgen wie `\n` werden bereits beim Übersetzen durch ihre ASCII-Codes ersetzt, also zum Beispiel `\n` durch die Nummer 10.

3.1.5 Steuerzeichen für die Ausgabe

Steuerzeichen	Name	Ausgabe
`\f`	**FF** (formfeed)	Seitenvorschub wird ausgelöst
`\n`	**NL** (newline)	Cursor geht zum Anfang der nächsten Zeile
`\r`	**CR** (carriage return)	Cursor geht zum Anfang der aktuellen Zeile
`\t`	**HT** (horizontal tab)	Cursor geht zur nächsten horizontalen Tabulatorposition
`\v`	**VT** (vertical tab)	Cursor geht zur nächsten vertikalen Tabulatorposition
`\"`		doppelte Anführungszeichen oben: "
`\'`		Anführungszeichen oben: '
`\?`		Fragezeichen: ?
`\\`		Backslash: \

3.1.6 Exkurs: Modulares Programmieren

Ein `Modula-3`-Programm besteht praktisch immer aus mehreren Modulen, denn bereits die Standardfunktionen sind in Modulen organisiert, welche man nach Bedarf in eigene Programme einbinden kann.
Durch Module wird ein Programm in überschaubare Happen aufgeteilt. Variablen und Funktionennamen in den einzelnen Modulen können nicht in Konflikte geraten, denn im Normalfall greift man auf solche Namen in der Form `Modul.Funktion` (sogenanntes *Qualifizieren*) zu. Die Reihenfolge der `IMPORT`-Anweisungen spielt keine Rolle für den Programmfluss. Es ist kaum möglich, dass durch das Einbinden eines weiteren (existenten :-) Moduls das Programm nicht mehr übersetzt werden kann oder aber völlig anders (insbesondere gar nicht mehr) funktioniert.
Zu `Modula-3` gehört immer auch ein System zur Verwaltung der Module. Es ermittelt Abhängigkeiten zwischen den Modulen und übersetzt nur die Module, die sich geändert haben, oder die von Modulen abhängen, welche sich geändert haben. Dieses System gibt auch die Verzeichnisstruktur für die Programmdateien vor:

`Projekt`	Hauptverzeichnis des Projektes
`  src`	Programmtexte, Steuerdateien
`  LINUXLIBC6`	vom Compiler für Linux erzeugte Dateien
`  SOLgnu`	vom Compiler für SUNs Solaris erzeugte Dateien
`  NT386`	vom Compiler für Windows erzeugte Dateien

Auch für kleine Programme wird der ganze Aufwand fällig, der eigentlich für große Projekte gedacht ist. (Siehe Abschnitt 3.13)

3.1.7 Funktionen

Funktionen erlauben die schrittweise Verfeinerung eines Programmes. Wann immer ein Programmabschnitt an mehreren Stellen eines Programmes nur leicht abgeändert auftritt, sollte man diesen Programmteil vielleicht besser in eine eigenständige Funktion umwandeln. Verbesserungen und Erweiterungen dieser Funktion kommen dann allen Aufrufen zu gute. (vgl. Abschnitt 2.4).
Die Reihenfolge von Funktionsdefinition und Funktionsaufruf im Programmtext ist egal!

```
MODULE Main;
(* $Id: AusgabeFunktion.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO;

PROCEDURE Text1 () =
  BEGIN
    IO.Put("Mein erstes C-Programm.\n");
  END Text1;

PROCEDURE Text2 () =
  BEGIN
    IO.Put("Hello world!\n");
  END Text2;

BEGIN
  Text1();
  Text2();
END Main.
```

3.1.8 Einige Regeln für Funktionen

- Funktionsaufrufe enthalten *immer* `( )`. In diesen Klammern können Argumente (Eingabeparameter) an die Funktion übergeben werden (mehr dazu in Abschnitt 3.4).
- Funktionsaufrufe sind Anweisungen. Daher werden sie mit einem Semikolon abgeschlossen.
- Funktionsnamen (gilt auch für Variablen-, Strukturnamen)
  - müssen mit einem Buchstaben anfangen und dürfen nur Buchstaben, Ziffern, Unterstriche enthalten;
  - Groß- und Kleinschreibung werden unterschieden, z.B. sind

    `main, Main, mAin, MAIN`

    unterschiedliche Namen;
  - dürfen nicht `Modula-3`-Schlüsselwörter sein, dürfen diese aber enthalten;
- Mehr zu Parameterübergabe und lokalen und globalen Variablen in Abschnitt 3.4.8.

3.2 Variablen und ordinale Typen

3.2.1 Variable und ihr Datentyp, ganze Zahlen

Im Computer werden alle Daten durch Nullen und Einsen (Bits) dargestellt, erst eine hohe Programmiersprache interpretiert sie als Zahlen, Zeichen oder ähnliches. Variablen entsprechen Speicherzellen, Typen entsprechen der Interpretation des Speicherinhalts. Der gleiche Inhalt kann abhängig vom Typ verschiedene Bedeutungen haben. Beispielsweise entspricht die Bitfolge 00100000 als ganze Zahl einer 32 und als ASCII-Zeichen einem Leerzeichen. Der Typ taucht im übersetzten Programm nicht mehr auf! Daher muss der Compiler bereits beim Übersetzen sicherstellen, dass Speicherzellen (Variablen) nur mit Bitmustern verträglicher Typen beschrieben werden. Dadurch sind die Programme sehr effizient und die Typentests decken ziemlich schnell „dumme“ Fehler im Programm auf!
Man muss unterscheiden:

- Objekt (Daten): Bitfolge bestimmter Länge, von eindeutigem Datentyp.
- Wert eines Objekts: Erhält man durch Interpretation der Bitfolge.

- Variable: Speicherbereich bestimmter Länge. Er enthält ein Objekt passenden Typs, zu verschiedenen Zeitpunkten in der Regel auch verschiedene Objekte.
- Wert einer Variablen: Wert des Objekts, den die Variable zum aktuellen Zeitpunkt enthält.
- Konstante: Objekt, dessen Wert sich nicht ändern kann.

Die vom Computer verarbeiteten Daten sind in verschiedene *Datentypen* eingeteilt. In `Modula-3` kann man aus den maschinennahen Standardtypen

- ganze Zahlen: `INTEGER`, `CARDINAL`
- rationale Zahlen: `REAL`, `LONGREAL`, `EXTENDED`
- Zeichen: `CHAR`
- Wahrheitswert: `BOOLEAN`
- Aufzählung

viele verschiedene, komplexe Datentypen aufbauen. Zu jedem Datentyp gehören unterschiedliche, sinnvolle Operationen.
Die Wahl des Typs muss genau bedacht werden. Je spezieller der Typ, desto besser kann der Compiler Flüchtigkeitsfehler erkennen. Je weniger Typumwandlungen später nötig sind, desto besser.
Zu jedem Typ gibt es sogenannte *Literale*, also Textschnippsel, die einen Wert eines bestimmten Typs repräsentieren. Zum Beispiel ist `'A'` ein Literal für ein Zeichen, besitzt den Typ `CHAR`. Hingegen ist `12` ein Literal für den Wert 12 vom Typ `INTEGER`. Dieses ist völlig verschieden von `12.0`, einem Literal für eine (einfach-genaue) Fließkommazahl mit Wert 12. Sobald man einer Zahl Nachkommastellen verpasst, betrachtet der Compiler sie als Fließkommazahl.
Variablen müssen vor ihrer Benutzung *deklariert* werden. Damit wird zum einen Speicher für die Variable reserviert und zum anderen wird ein Name vergeben und der Typ (also die Interpretation des Speicherbereiches) festgelegt. Variablen werden niemals automatisch angelegt und lassen sich auch nicht beliebig löschen. Das erlaubt eine effiziente Implementation, hat sich aber auch als hervorragendes Mittel zur Fehlervermeidung bewährt.
So werden Variablen deklariert:

```
VAR
  a: INTEGER;
  b: [-10..10] := 3;
  c            := TRUE;
```

Hiermit werden drei Variablen eingeführt:

- `a` wird als Ganzzahlvariable vereinbart. Ihr Wert ist zunächst ungewiss.
- `b` wird als Ganzzahlvariable deklariert, die höchstens 10 und mindestens -10 sein darf. Ihr Wert wird mit 3 festgelegt.
- `c` wird mit einer verkürzten Schreibweise deklariert und initialisiert. Das ist immer dann sinnvoll, wenn sich aus dem Initialisierungswert der passende Typ ergibt, hier ergibt sich aus `TRUE` der Typ `BOOLEAN` (Wahrheitswert).

Neben Variablen gibt es noch Konstanten. Deren Wert muss bereits zum Zeitpunkt der Übersetzung bekannt sein und ändert sich während des Programmlaufes nicht:

```
CONST
  AntwortAufUniversumLebenRest = 42;
  FrageFuerAntwort : TEXT =
    "What do you get if you multiply six by nine?";
```

Von Zahlen in der internen Computerdarstellung haben wir nicht viel. Wir wollen uns die Inhalte der Variablen auch irgendwie ansehen. In den Standardbibliotheken von `Modula-3` sind Aufbereitung der Daten als Text und die Ausgabe weitgehend getrennt. Die Routinen lassen sich daher sehr flexibel einsetzen, sind aber nicht so effizient. Im Modul `Fmt` gibt es zahlreiche Funktionen zum Darstellen von verschiedenen Daten als Text:

`Fmt.Int` Konvertiert eine ganze Zahl in einen Text.

`Fmt.F` Fügt bis zu 5 Textteile in eine Maske ein. Die Maske ist ein Text, der als Platzhalter die Zeichenfolge `%s` enthält. Diese Platzhalter werden von `Fmt.F` durch die folgenden Textteile ersetzt:

```
Fmt.F("Am %s.%s. kommt der %s.",
   Fmt.Int(24), Fmt.Int(12), "Weihnachtsmann")
```

ergibt `"Am 24.12. kommt der Weihnachtsmann."`

`Fmt.FN` Wie `Fmt.F`, erwartet aber ein Feld von Texten (siehe Abschnitt 3.7) und verarbeitet beliebig viele Textteile.

Beispiel:

```
MODULE Main;
(* $Id: IntAusgabe.m3,v 1.2 2004/01/16 18:15:09 thielema Exp $ *)

IMPORT IO, Fmt;
```

```
VAR
  zahl1: INTEGER;
  zahl2: INTEGER;
  zahl3: INTEGER;

BEGIN
  zahl1 := 19;
  zahl2 := -23;
  zahl3 := 299;

  IO.Put(Fmt.F("Die Zahlen lauten %s, %s und %s.\n",
               Fmt.Int(zahl1), Fmt.Int(zahl2), Fmt.Int(zahl3)));
END Main.
```

Verkürzt:

```
MODULE Main;
(* $Id: IntAusgabeKurz.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO, Fmt;

VAR
  zahl1: INTEGER := 19;
  zahl2: INTEGER := -23;
  zahl3: INTEGER := 299;

BEGIN
  IO.Put(Fmt.F("Die Zahlen lauten %s, %s und %s.\n",
               Fmt.Int(zahl1), Fmt.Int(zahl2), Fmt.Int(zahl3)));
END Main.
```

```
IntAusgabe> cm3
IntAusgabe> LINUXLIBC6/prog
Die Zahlen lauten 19, -23 und 299.
```

3.2.2 Weitere Beispiele für formatierte Ausgabe

- Rechtsbündige Ausgabe mit 6 (Leer-)Stellen

```
IO.Put(Fmt.F("Wert 1 = %6s\n", Fmt.Int(42)));
IO.Put(Fmt.F("Wert 2 = %6s\n", Fmt.Int(42424)));

Wert 1 =     42
Wert 2 =  42424
```

- Benötigt die Variable mehr Platz als im Format angegeben ist, werden trotzdem alle Ziffern angezeigt

```
IO.Put(Fmt.F("Wert 2 = %1s\n", Fmt.Int(42424)));

Wert 2 = 42424
```

- Linksbündige Ausgabe mit 6 (Leer-)Stellen

```
IO.Put(Fmt.F("Wert 1 = %-6s\n", Fmt.Int(42)));
IO.Put(Fmt.F("Wert 2 = %-6s\n", Fmt.Int(42424)));

Wert 1 = 42
Wert 2 = 42424
```

- Rechtsbündige Ausgabe mit 6 Stellen, Auffüllen mit Nullen

```
IO.Put(Fmt.F("Wert 1 = %06s\n", Fmt.Int(42)));
IO.Put(Fmt.F("Wert 2 = %06s\n", Fmt.Int(42424)));

Wert 1 = 000042
Wert 2 = 042424
```

- Ausgabe von „%“

```
IO.Put(Fmt.F("Wert 1 = %s%\n", Fmt.Int(42)));

Wert 1 = 42%
```

- Ausgabe von Zahlen im Oktal- oder Hexadezimalformat

```
IO.Put(Fmt.F("Wert dezimal = %s\n", Fmt.Int(2748,10)));
IO.Put(Fmt.F("Wert oktal = %s\n", Fmt.Int(2748,8)));
IO.Put(Fmt.F("Wert hexadezimal = %s\n", Fmt.Int(2748,16)));

Wert dezimal = 2748
Wert oktal = 5274
Wert hexadezimal = abc
```

3.2.3 Ausdrücke

Grundrechenarten

Die Symbole `+`, `-`, `*`, `DIV` werden für die Grundrechenarten der ganzen Zahlen benutzt, wobei `*`, `DIV` stärker binden als `+`, `-` (Punkt- vor Strich-Rechnung, siehe auch Abschnitt A.1). Die Operation `DIV` (Division mit Abrunden) ist eine spezielle Abwandlung für ganze Zahlen, da dort die Division im allgemeinen nicht ausführbar ist. `DIV` wird durch `MOD` ergänzt, welches den Rest bei der Division bestimmt. Es gilt also

```
a DIV b * b + a MOD b = a.
```

So verhalten sich die Operationen bei negativen Zahlen:

```
 13 DIV  4 =  3
 13 MOD  4 =  1
-13 DIV  4 = -4
-13 MOD  4 =  3
 13 DIV -4 = -4
 13 MOD -4 = -3
-13 DIV -4 =  3
-13 MOD -4 = -1
```

Das Ergebnis einer Berechnung kann man zum Beispiel wieder einer Variablen zuweisen:

- „:=“ weist der links stehenden Variablen den Wert zu, den der Ausdruck auf der rechten Seite hat.
- Falls Ausdruck auf der rechten Seite eine Berechnung erfordert, so wird dieser zunächst ausgewertet und erst dann der Variablen zugewiesen (und gegebenenfalls der Typ angepasst).

- Vor „:=“ darf kein konstanter Ausdruck stehen und kein Ausdruck, der Berechnungen erfordert.

Beispiel:

```
MODULE Main;
(* $Id: IntRechnen.m3,v 1.2 2004/01/16 18:15:09 thielema Exp $ *)

IMPORT IO, Fmt;

VAR zahl1, zahl2, zahl3, zahl4: INTEGER;

BEGIN
  zahl1 := 3 * 2;
  zahl3 := zahl1 * 2;
  zahl2 := zahl3;
  zahl3 := zahl3 * 2;
  zahl4 := zahl3 DIV 5;
  zahl1 := zahl2 * zahl4 + zahl3;
  IO.Put(Fmt.F("Das Ergebnis lautet : %s\n", Fmt.Int(zahl1)));
END Main.
```

```
IntRechnen> cm3
IntRechnen> LINUXLIBC6/prog
Das Ergebnis lautet : 72
```

Wie kommt das Ergebnis zustande?

	zahl1	zahl2	zahl3	zahl4
zahl1 := 6	6	?	?	?
zahl3 := 12	6	?	12	?
zahl2 := zahl3	6	12	12	?
zahl3 := 12 * 2	6	12	24	?
zahl4 := 24 DIV 5	6	12	24	4
zahl1 := 12 * 4 + 24	72	12	24	4

Einsatz von Ausdrücken

An nahezu jeder Programmstelle, an der eine Variable stehen kann, kann auch ein berechenbarer Ausdruck (einschließlich Literale) stehen.
Beispiele:

```
z3 := z1 + z2;
IO.Put(Fmt.F("%s plus %s gleich %s",
             Fmt.Int(z1), Fmt.Int(z2), Fmt.Int(z3)));
```

Oder kürzer (falls `z3` nicht weiter benötigt):

```
IO.Put(Fmt.F("%s plus %s gleich %s",
             Fmt.Int(z1), Fmt.Int(z2), Fmt.Int(z1 + z2)));

IO.Put(Fmt.F("%s plus %s gleich %s",
             Fmt.Int(5), Fmt.Int(7), Fmt.Int(5+7)));
```

oder noch kürzer

```
IO.Put(Fmt.F("5 plus 7 gleich %s", Fmt.Int(5 + 7)));
```

Überläufe

Was passiert bei Überschreitung des Zahlbereichs („Overflow“)?
Es wird ein Fehler ausgelöst, der das Programm abbricht, wenn er nicht behandelt wird. (Siehe Abschnitt 3.12)
Problem: Die meisten `Modula-3`-Compiler bauen auf Codeerzeugern von `C`-Compilern auf. Da `C` keine Überlaufabfrage kennt, können diese Codeerzeuger ebenfalls keinen Überlauftest ins Programm einbauen und so bleiben Überläufe über die größte mit `INTEGER` darstellbare Zahl hinaus unerkannt!

3.2.4 Aufzählungen

Neben den ganzen Zahlen sind die Aufzählungen ein wichtiger Grundtyp. Aufzählungen sind wie ganze Zahlen *ordinale Typen* (nicht ordinäre Typen! :-).
Beispiele:

```
TYPE
  Grundfarbe = {rot, gruen, blau};
  Modefarbe  = {mauve, gruen, schaumolweiss, rotbraungrau};
  Status     = {pause, maustasteLosgelassen, maustasteGedrueckt};
  BOOLEAN    = {FALSE, TRUE};
```

Der Typ `BOOLEAN` ist bereits vordefiniert und wird von Operatoren wie `AND` und `OR` verwendet. (Siehe Abschnitt 3.3.1) Den Aufzählungselementen werden intern aufsteigend ganze Zahlen bei 0 beginnend zugeordnet. Zum Beispiel entspricht `Grundfarbe.bl` dem Wert 2. Von diesem Wissen sollte man allerdings so wenig wie möglich Gebrauch machen. Im Prinzip sind Aufzählungselemente Ganzzahlkonstanten mit dem

Unterschied, dass man mit ihnen nicht rechnen kann und dass sie unverträglich mit ganzen Zahlen und anderen Aufzählungen sind.
Beispiele:

Erlaubt:

```
VAR
  st: Status;
BEGIN
  st := Status.pause;
  st := LAST(Status);
END;
```

Verboten:

```
VAR
  st:    Status;
  zahl:  INTEGER;
  farbe: Grundfarbe;
BEGIN
  st    := 1;
  st    := Status.pause + 1;
  zahl  := Grundfarbe.gruen;
  farbe := Modefarbe.gruen;
  farbe := Status.pause;
END;
```

Da Aufzählungen mit keinem anderen Typ verträglich sind, kann man den Wert von Aufzählungsvariablen nicht einmal ohne weiteres ausgeben! Folgende Lösungen bieten sich an:

1. Mit `ORD` (Abschnitt 3.2.7) wandelt man den Wert in eine ganze Zahl um und gibt diese aus.

   Probleme:

   - Gibt man die Zahl für den Benutzer aus, weiß er wahrscheinlich nichts mit dem Zahlenwert anzufangen.
   - Gibt man die Zahl in eine Datei aus, die später wieder eingelesen werden soll, muss man sicherstellen, dass sich die Reihenfolge der Aufzählungselemente später nicht mehr ändert.

2. Man kann sehr elegant einen Aufzählungswert in etwas anderes umwandeln, in dem man auf ein konstantes Feld mit dem Aufzählungswert als Index zugreift. (siehe Abschnitt 3.7)

3.2.5 Unterbereiche

Variablen vom Typ `INTEGER` können in der Regel sehr große und sehr kleine Werte annehmen, auf einem 32-Bit Rechner zum Beispiel ganze Zahlen von -2147483648 bis 2147483647. Wenn man schon weiß, dass man einer Variable nur Werte aus einem kleinen Intervall zuweisen wird, kann man dies dem Compiler durch Unterbereiche

anzeigen. Er sorgt dann während der Übersetzung und während des Programmlaufes dafür, dass die Intervallgrenzen nicht überschritten werden.
Beispiele:

```
TYPE
  Quadrant       = [1..4];
  Schulnote      = [1..6];
  Abipunkte      = [0..840];
  Maus           = [Status.maustasteLosgelassen ..
                     Status.maustasteGedrueckt];
  Grossbuchstabe = ['A'..'Z'];
  CARDINAL       = [0..LAST(INTEGER)];
```

Man sieht, dass Unterbereiche auch von anderen Typen gebildet werden können, zum Beispiel Aufzählungen und Zeichen. Der Typ `CARDINAL` der nicht-negativen ganzen Zahlen ist wie im Beispiel angegeben vordefiniert.

3.2.6 Mengen

`Modula-3` stellt einen Typ für Mengen zur Verfügung.
Beispiel:

```
TYPE
  Gemuet     = {sauer, stinkig, launisch,
                normal, gutDrauf, ausgeflippt};
  Gemuetsmix = SET OF Gemuet;

CONST
  ansprechbar  = Gemuetsmix {Gemuet.normal, Gemuet.gutDrauf};
  grosserBogen = Gemuetsmix {Gemuet.sauer, Gemuet.stinkig};

  vokale = SET OF CHAR {'a', 'e', 'i', 'o', 'u'};
```

Manche Rechensymbole, die schon für Zahlen definiert sind, haben für Mengen eine andere Bedeutung. Das sind die Operationen auf Mengen:

	Bezeichnung	Umschreibung	Logik
		Operationen	
+	Vereinigung		x OR y
-	Mengendifferenz		x AND NOT y
*	Schnitt		x AND y
/	symmetrische Differenz	A/B = (A-B)+(B-A)	x XOR y
		Relationen	
IN	Element von	a IN A - a enthalten in A	
<=	Teilmenge	A<=B = A-B={}	NOT x OR y
<	echte Teilmenge	A<B = A<=B AND A#B	
>=	Obermenge	A>=B = B-A={}	x OR NOT y
>	echte Obermenge	A>B = A>=B AND A#B	

Hinweis: ‚#‘ steht ‚ungleich‘ (Abschnitt 3.3.1), x steht für e IN A, y steht für e IN B und der Ausdruck in der Spalte „Logik“, Abschnitt „Operationen“ ist äquivalent zum Ausdruck e IN A *op* B, wobei *op* die betrachtete Operation ist.

3.2.7 Typumwandlungen

Modula-3 ist sehr streng, was Typen anbetrifft. Ganze Zahlen, Fließkommazahlen, Zeichen, Aufzählungen, Mengen können nicht in einer Operation gemischt werden. Dadurch können beispielsweise Fehler bei Zuweisungen oder fälschlich vertauschte Funktionsparameter noch vor dem ersten Programmlauf aufgedeckt werden. Obwohl für Additionen immer das Zeichen + verwendet wird, so haben doch die Additionen von ganzen Zahlen und von Fließkommazahlen etwas andere Eigenschaften. Auslöschungsfehler und Rundungsfehler können nur Fließkommazahlen auftreten. Deshalb muss man dem Compiler über die Typen der Operanden mitteilen, welche Operation man meint.
Durch Konvertierungen kann man Operanden notfalls anpassen. Häufige Konvertierungen sind ein deutliches Zeichen für ungeeignet gewählte Typen und sollten vermieden werden.
Folgende Funktionen helfen beim Konvertieren, hier am Beispiel der fiktiven Variable x erklärt:

INTEGER → CHAR	VAL(x, CHAR)
INTEGER → Aufzählung	VAL(x, Aufzählung)
INTEGER → LONGREAL	FLOAT(x, LONGREAL)
LONGREAL → INTEGER	ROUND(x), TRUNC(x), FLOOR(x), CEILING(x)
CHAR → INTEGER Aufzählung → INTEGER	ORD(x)

3.2.8 Inkrement- und Dekrementfunktionen

Eine Kurzschreibweise für das Erhöhen und Verringern eines ordinalen Wertes (also auch Aufzählungen und Zeichen) bieten die eingebauten Prozeduren `INC` bzw. `DEC`.

`INC(n);`	`n := n + 1;`
`DEC(n);`	`n := n - 1;`
`INC(n,d);`	`n := n + d;`
`DEC(n,d);`	`n := n - d;`

3.2.9 Eingabe mit `Lex`

Bisher kennen wir `IO.Put` zur Ausgabe auf dem Bildschirm und einige Routinen des Modules `Fmt` zum Aufbereiten verschiedener Daten als Text. Mit den Funktionen aus dem Modul `Lex` kann man auch vom Benutzer eingegebene Werte einlesen:

```
variable := Lex.Int(Stdio.stdin);
```

Damit der Benutzer weiß, dass und vor allem welche Eingabe von ihm erwartet wird, sollte man `Lex.Int` mit einem `IO.Put` kombinieren.

```
MODULE Main;
(* $Id: LexStdin.m3,v 1.2 2004/01/16 18:15:09 thielema Exp $ *)

IMPORT IO, Fmt, Lex, Stdio;

VAR
  z1: CARDINAL;
  z2: INTEGER;

<* FATAL ANY *>
BEGIN
  IO.Put("Zahl eingeben: ");
  z1 := Lex.Unsigned(Stdio.stdin, 10);
  IO.Put(Fmt.F("Die Zahl lautet: %s\n", Fmt.Int(z1)));
  IO.Put("Noch eine Zahl eingeben (hexadezimal): ");
  z1 := Lex.Unsigned(Stdio.stdin);
  IO.Put(Fmt.F("Die Zahl lautet: %s\n", Fmt.Int(z1)));
  IO.Put("Und noch eine Zahl eingeben: ");
  z2 := Lex.Int(Stdio.stdin);
END Main.
```

```
LexStdin> LINUXLIBC6/prog
Zahl eingeben: 10
Die Zahl lautet: 10
Noch eine Zahl eingeben (hexadezimal): 10
Die Zahl lautet: 16
Und noch eine Zahl eingeben: -42
```

Falsche Eingaben lösen einen Fehler aus, der das Programm abbricht, falls er nicht behandelt wird. Der Compiler warnt den Programmierer vor unbehandelten Fehlern. Diese Warnung wiederum haben wir mit dem sogenannten Pragma `<* FATAL ANY *>` unterdrückt. Das ist unsauber, aber es gibt zwei gute Gründe dies zu tun:

- Man ist sich sehr sicher, dass die prinzipiell möglichen Fehler in dieser konkreten Situation nicht auftreten können. Besser ist es dann immer noch, die Fehler namentlich hinter `FATAL` aufzuführen. Außerdem sollte man sich einen Kommentar abringen, der beschreibt, warum die unterdrückten Fehler nicht auftreten können.
- Man schreibt ein Beispielprogramm für Studenten, die sich das Programm eh nicht so genau anschauen.

Wie man Fehler korrekt behandelt, werden wir in Abschnitt 3.12 sehen.

3.3 Anweisungen zur Auswahl

3.3.1 Logische und Vergleichsoperatoren

Zur Formulierungen von Bedingungen in Auswahlanweisungen oder Schleifen benötigt man *Vergleichsoperatoren* und *logische Operatoren*, z.B. zur Negation oder Verknüpfung von Bedingungen.

Vergleichsoperator	Bedeutung
`a < b`	`TRUE`, wenn `a` kleiner `b`
`a <= b`	`TRUE`, wenn `a` kleiner oder gleich `b`
`a > b`	`TRUE`, wenn `a` größer `b`
`a >= b`	`TRUE`, wenn `a` größer oder gleich `b`
`a = b`	`TRUE`, wenn `a` gleich `b`
`a # b`	`TRUE`, wenn `a` nicht gleich `b`

Wichtig ist, das die Operanden immer verträglich sein müssen. Wenn wie in Abschnitt 3.2.4 die Typen `Grundfarbe` und `Modefarbe` verschieden sind, dann ist ein

Vergleich zwischen Ausdrücken dieser Typen verboten. Mit gutem Recht könnte der Übersetzer den Ausdruck `Grundfarbe.gruen = Modefarbe.gruen` als Ausdruck mit konstantem Wert `FALSE` auffassen, aber in der Regel ist ein solcher Ausdruck auf einen Programmierfehler zurückzuführen. Deswegen weist der Übersetzer ihn zurück.
In `Modula-3` gehören Wahrheitswerte zum Typ `BOOLEAN`, welcher unverträglich mit anderen Typen ist. Daher werden Ausdrücke wie `(a=b)+1` sofort als Fehler erkannt. Ausgegeben werden Wahrheitswerte so:

```
MODULE Main;
(* $Id: Wahrheitswerte.m3,v 1.4 2004/02/18 11:46:42 thielema Exp
   $ *)

IMPORT IO, Fmt;

VAR grundschueler, abiturient: BOOLEAN;

BEGIN
  (* Wer hat recht? *)
  grundschueler := 5 * 7 = 35;
  abiturient := 5 * 7 = 36;
  IO.Put(Fmt.F("Die Antwort des Grundschuelers ist %s\n"
               & "Die Antwort des Abiturienten ist %s\n",
             Fmt.Bool(grundschueler), Fmt.Bool(abiturient)));
END Main.
```

Mit dem *Negationsoperator* `NOT` wird der Wahrheitswert einer Bedingung genau umgekehrt, z.B.

```
w := 101;
NOT (w = 100)       -->  TRUE
NOT  w = 100        -->  TRUE
NOT  w > 100        -->  FALSE
```

Die Vorrangreihenfolge ist:

1. Arithmetische Operationen
2. Relationen
3. logische Operationen

(vollständige Liste in Abschnitt A.1). Daher kann die Klammer um `w = 100` entfallen.

Zur Verknüpfung von Bedingungen gibt es die logischen Operatoren `AND` und `OR`. Die Relationen = und # sind ebenfalls sinnvoll für Wahrheitswerte.

`AND`	„Und“	Verknüpfung ist genau dann wahr, wenn beide Bedingungen wahr sind.
`OR`	„Oder“	Verknüpfung ist genau dann wahr, wenn mindestens eine der beiden Bedingungen wahr ist.
=	„Genau dann wenn“	Verknüpfung ist genau dann wahr, wenn beide Bedingungen den gleichen Wahrheitswert besitzen.
#	„Entweder oder“	Verknüpfung ist genau dann wahr, wenn beide Bedingungen unterschiedliche Wahrheitswerte besitzen.

Bedingung 1	Bedingung 2	Bedingung 1 `AND` Bedingung 2
`FALSE`	`FALSE`	`FALSE`
`FALSE`	`TRUE`	`FALSE`
`TRUE`	`FALSE`	`FALSE`
`TRUE`	`TRUE`	`TRUE`

Bedingung 1	Bedingung 2	Bedingung 1 `OR` Bedingung 2
`FALSE`	`FALSE`	`FALSE`
`FALSE`	`TRUE`	`TRUE`
`TRUE`	`FALSE`	`TRUE`
`TRUE`	`TRUE`	`TRUE`

Beispiel: Ist eine Zahl in einem Intervall enthalten?
Die mathematische Notation $20 \leq x \leq 40$ ist zwar gängig und schön kompakt, aber strenggenommen falsch. Deswegen lässt sie sich auch nicht direkt in ein `Modula-3`-Programm übertragen. Schreibt man nämlich `20 < x < 40`, dann fasst der Übersetzer dies als `(20 < x) < 40` auf. Der Ausdruck `(20 < x)` ist vom Typ `BOOLEAN`. Wie aber soll man einen Wahrheitswert mit der ganzen Zahl `40` vergleichen? Folgerichtig lehnt der Übersetzer diesen Ausdruck ab.
Die mathematische Schreibweise $20 \leq x \leq 40$ ist eine Kurzschreibweise für $20 \leq x \wedge x \leq 40$ und in dieser Form muss man es auch dem Übersetzer servieren:

$x \in [20, 40]$	$\Longleftrightarrow$	`20 <= x AND x <= 40`
$x \notin [20, 40]$	$\Longleftrightarrow$	`x < 20 OR 40 < x`

Verknüpfungen mit `AND` und `OR` werden von links nach rechts abgearbeitet. Die Auswertung endet, wenn Wert des Ausdrucks feststeht (*Lazy evaluation*). Zum Beispiel wird für `x=17` im letzten Beispiel nur `x<20` ausgewertet, da dann der Gesamtausdruck wahr ist und es auch bleibt.
Diese Vorgehensweise ist mehr als eine nette Optimierung. Von ihr kann der korrekte Programmlauf abhängen. Beispiel: Wenn `y = 0` dann führt der Ausdruck `x MOD`

y zum Programmabbruch wegen einer Division durch 0. Wenn in einer konkreten Situation im Programm `y=0` als „nicht teilbar“ zu werten ist, dann ist der korrekte Ausdruck dafür `y # 0 AND x MOD y = 0`, denn der rechte Operand von `AND` wird nur ausgewertet, wenn `y # 0`. Dagegen führt der Ausdruck `x MOD y = 0 AND y # 0` nach wie vor unter Umständen zum Programmabbruch.

3.3.2 `IF`-Anweisungen

Die in Abschnitt 2 eingeführte Anweisung zur Auswahl kann, wen wundert's, direkt in `Modula-3`-Syntax übersetzt werden:

```
FALLS Bedingung        IF Bedingung
DANN                   THEN
  ...                    ...
SONST                  ELSE
  ...                    ...
                       END
```

Beispiel: Maximums-Bestimmung

```
VAR
  max: INTEGER;
BEGIN
  IF x > y THEN
    IF x > z THEN
      max := x;
    ELSE
      max := z;
    END;
  ELSE
    IF y > z THEN
      max := y;
    ELSE
      max := z;
    END;
  END;
  (* max enthaelt jetzt
     das Maximum der Zahlen x, y, z *)
END;
```

Bei `IF`-Anweisungen kann der `ELSE`-Zweig auch weggelassen werden.
Außerdem kann man mit dem `ELSIF`-Teil die Verzweigungstiefe reduzieren. Der Programmabschnitt

```
IF A THEN
  B;
ELSE
  IF C THEN
    D;
  ELSE
    E;
  END;
END;
```

reduziert sich dadurch zu

```
IF A THEN
  B;
ELSIF C THEN
  D;
ELSE
  E;
END;
```

.
Im Beispiel der Maximumsbestimmung:

```
VAR
  max: INTEGER;
BEGIN
  IF x > y THEN
    IF x > z THEN
      max := x;
    ELSE
      max := z;
    END;
  ELSIF y > z THEN
    max := y;
  ELSE
    max := z;
  END;
END;
```

Nun ist allerdings die Symmetrie der Verschachtelung hinüber und es bleibt eine Geschmacksfrage, ob dieses Konstrukt leichter zu lesen ist.

3.3.3 `CASE`-Anweisung

Mit geschachtelten `IF`-Anweisungen lassen sich grundsätzlich alle Entscheidungsfolgen realisieren, die Programme werden dadurch aber mitunter unübersichtlich. Statt

```
IF x = a THEN
  A;
ELSIF x = b THEN
  B;
ELSIF x = c THEN
  C;
ELSE
  Z;
END;
```

verwendet man besser

```
CASE x OF
| a =>
  A;
| b =>
  B;
| c =>
  C;
ELSE
  Z;
END;
```

Beispiel:

```
MODULE Main;
(* $Id: Case.m3,v 1.3 2004/01/22 17:46:35 thielema Exp $ *)

IMPORT IO, Lex, Stdio;

VAR z: INTEGER;
```

```
<* FATAL ANY *>
BEGIN
  IO.Put("Bitte geben Sie eine Zahl ein:\n");
  z := Lex.Int(Stdio.stdin);
  IO.Put("\n");

  CASE z OF
  | 0 => IO.Put("Die Zahl heißt Null.\n");
  | 1 => IO.Put("Die Zahl heißt Eins.\n");
  | 2 .. 3, 5, 7 => IO.Put("Die Zahl ist prim.\n");
  | 42 => IO.Put("Diese Zahl beantwortet alles.\n");
  | 100 .. 999 =>
      IO.Put("Die Zahl ist dreistellig (Peanuts).\n");
  ELSE
    IO.Put("Zu dieser Zahl fällt mir nichts ein!\n");
  END

END Main.
```

Der Auswahlausdruck (hier z) muss von ordinalem Typ sein. Außerdem dürfen für die Verzweigungen Listen konstanter Ausdrücke oder Unterbereiche verwendet werden. Die Reihenfolge der `| x =>`-Verzweigungen ist egal. Ein Fall darf nicht mehrfach behandelt werden. Daher ist auch

```
  CASE x OF
  | 0 => A;
  | -10..10 => B;
  END;
```

verboten.

3.4 Funktionen und Prozeduren

Funktionen dienen der strukturierten Programmierung: Wiederholt genutzte Programmteile werden aus dem Hauptprogramm ausgelagert. (Siehe Abschnitt 3.1.7)

3.4.1 Definition

*Prozedur*en und *Funktion*en sind *Unterprogramm*e, also Teile eines großen Programmes aber nicht eigenständig lauffähig.
Prozeduren können durch Parameter gesteuert werden. Die Parameter werden im Prozedurkopf (*Signatur*) vereinbart und stehen innerhalb der Prozedur als Variable zur Verfügung.

```
PROCEDURE Name (var1, var2: TypA; var3: TypB; ) =
  BEGIN
    ...
  END Name;
```

Mit dem `RETURN`-Befehl kann man die Prozedur vorzeitig verlassen.
Außerdem kann eine Prozedur einen (aber nur höchstens einen) Wert zurückgeben. Mehrere Rückgabewerte muss man in Datenverbünden zusammenfassen. Dazu mehr in Abschnitt 3.9. Der Typ des Rückgabewertes wird Doppelpunkt getrennt hinter die Parameterliste geschrieben.
Solche Prozeduren heißen dann auch Funktionsprozeduren oder kurz Funktionen. Mit dem `RETURN`-Befehl werden bei solchen Prozeduren die Werte zurückgegeben.

```
PROCEDURE Name (var1, var2: TypA; var3: TypB; ): TypRueckgabe =
  BEGIN
    ...
    RETURN wert;
  END Name;
```

Dem `RETURN`-Befehl kann auch ein berechenbarer Ausdruck übergeben werden. Im Gegensatz zur Prozedur ohne Rückgabewert *muss* eine Funktion immer durch `RETURN` verlassen werden. Es ist ein Fehler, wenn das Ende einer Funktionsprozedur erreicht wird.

3.4.2 Aufruf

Eine Prozedur ruft man durch Nennung des Namens mit anschließender geklammerter kommagetrennter Liste aus Ausdrücken auf, etwa: `Name(1,2,3)`. Auch `IO.Put`

und `Fmt.F` sind ganz normale Prozeduren, die wir bereits auf diesem Wege aufgerufen haben.
Die übergebenen Ausdrücke werden zunächst ausgewertet und dann in die „lokalen Variablen“ der Prozedur kopiert. Diese Methode heißt *Call by value* im Gegensatz zu *Call by reference*. Änderungen an den Parameter-Variablen innerhalb der Prozedur wirken sich folglich nicht auf die Variablen im Hauptprogramm aus.

3.4.3 Beispiel: Call by value

```
MODULE Main;
(* $Id: CallByValue.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO, Fmt;

PROCEDURE PutInt (n: INTEGER; ) =
  BEGIN
    IO.Put(Fmt.Int(n));
  END PutInt;

BEGIN
  IO.Put("Jetzt kann ich Zahlen direkt ausgeben:\n");
  PutInt(42);
  IO.Put("\n");
END Main.
```

```
CallByValue> LINUXLIBC6/prog
Jetzt kann ich Zahlen direkt ausgeben:
42
```

3.4.4 Beispiel: Funktion mit Rückgabewert

```
MODULE Main;
(* $Id: QuadratFunktion.m3,v 1.3 2004/03/04 10:17:37 thielema Exp
   $ *)
(* Funktion zum Quadrieren einer ganzen Zahl *)

IMPORT IO, Fmt, Lex, Stdio;

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

VAR x: INTEGER;
<* FATAL ANY *>
BEGIN
  IO.Put("Bitte geben Sie eine Zahl ein: ");
  x := Lex.Int(Stdio.stdin);

  IO.Put(Fmt.F("Das Quadrat von %s ist %s.\n", Fmt.Int(x),
               Fmt.Int(Quadrat(x))));
END Main.
```

```
QuadratFunktion> LINUXLIBC6/prog
Bitte geben Sie eine Zahl ein: 7
Das Quadrat von 7 ist 49.
```

Hinweis: Funktionen sollten entweder etwas ausgeben oder rechnen, aber nie beides tun. Niemand kann eine Rechenfunktion weiterverwenden, die ihre Ergebnisse auf den Bildschirm schreibt. Niemand rechnet zum Beispiel damit, dass `sin` etwas auf die Konsole schreibt. Lediglich vorübergehend für Tests sollte man die Konsole benutzen.

3.4.5 Beispiel: Funktion mit mehreren Eingaben

```
MODULE Main;
(* $Id: MaximumKurz.m3,v 1.1 2004/01/21 14:34:25 thielema Exp
   $ *)
(* Funktion zur Maximumbestimmung *)

IMPORT IO, Fmt, Lex, Stdio;

PROCEDURE Max (a, b: INTEGER; ): INTEGER =
  BEGIN
    IF a > b THEN RETURN a; ELSE RETURN b; END;
  END Max;

VAR z1, z2: INTEGER;
<* FATAL ANY *>
BEGIN
  IO.Put("Bitte geben Sie die 1. Zahl ein: ");
  z1 := Lex.Int(Stdio.stdin);
  IO.Put("Bitte geben Sie die 2. Zahl ein: ");
  z2 := Lex.Int(Stdio.stdin);
  IO.Put(
    Fmt.F("Die groessere Zahl ist %s.\n", Fmt.Int(Max(z1, z2))));
END Main.
```

```
Maximum> LINUXLIBC6/prog
Bitte geben Sie die 1. Zahl ein: 2
Bitte geben Sie die 2. Zahl ein: 6
Die groessere Zahl ist 6.
```

3.4.6 Bezugsrahmen von Variablen

Wir wissen, dass Änderungen an den Eingabeparametern innerhalb einer Funktion keine Wirkung außerhalb der Funktion haben, die Änderungen wirken sich nur lokal in der Funktion aus. Variablen die innerhalb einer Funktion angelegt werden, sind ebenfalls *lokale Variable*. Daneben heißen die Variablen des Hauptprogrammes *globale Variable*n.

3.4.7 Programm mit lokalen Variablen

Was passiert, wenn es lokale und globale Variablen gleichen Namens gibt?

```
MODULE Main;
(* $Id: LokaleVariable.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)
(* Call by value: Lokale Variable beeinflusst globale Variable
   nicht. *)

IMPORT IO, Fmt;

PROCEDURE Aendern () =
  VAR x: INTEGER := 10;
  BEGIN
    IO.Put(Fmt.F("x = %s\n", Fmt.Int(x)));
  END Aendern;

VAR x: INTEGER := 4;

BEGIN
  IO.Put(Fmt.F("x = %s\n", Fmt.Int(x)));
  Aendern();
  IO.Put(Fmt.F("x = %s\n", Fmt.Int(x)));
END Main.
```

```
LokaleVariable> LINUXLIBC6/prog
x = 4
x = 10
x = 4
```

Die Variable `x` in `Aendern()` ist eine andere als die Variable `x` im Hauptprogramm. Die Zuweisung an `x` in `Aendern` und die Konvertierung in einen Text mit `Fmt.Int` in `Aendern` benutzen eine andere Variable als die entsprechenden Anweisungen im Hauptprogramm. Trotzdem dürfen beide Variablen den gleichen Namen tragen.
Es wird immer die lokalste der Variablen mit dem verwendeten Namen angesprochen. Der Inhalt anderer Variablen gleichen Namens ist in der Funktion nicht zugänglich. Es ist also wichtig, den Bezugsrahmen (Sichtbarkeitsbereich, Gültigkeitsbereich, Lebensdauer) jeder Variablen zu kennen.

3.4.8 Programm mit globalen und lokalen Variablen

```
MODULE Main;
(* $Id: LokaleGlobaleVariable.m3,v 1.2 2004/01/16 18:15:09
   thielema Exp $ *)
(* Unterschied lokale-globale Variablen *)

IMPORT IO, Fmt;

PROCEDURE Aendern () =
  VAR a: INTEGER;
  BEGIN
    a := 0;
    b := 0;
    IF a = 0 THEN
      VAR a: INTEGER := 20;
      BEGIN
        b := 20;
        IO.Put(
          Fmt.F("a=%2s und b=%2s\n", Fmt.Int(a), Fmt.Int(b)));
      END;
      IO.Put(Fmt.F("a=%2s und b=%2s\n", Fmt.Int(a), Fmt.Int(b)));
    END;
  END Aendern;

VAR
  a: INTEGER := 10;
  b: INTEGER := 10;
BEGIN
  IO.Put(Fmt.F("a=%2s und b=%2s\n", Fmt.Int(a), Fmt.Int(b)));
  Aendern();
  IO.Put(Fmt.F("a=%2s und b=%2s\n", Fmt.Int(a), Fmt.Int(b)));
END Main.
```

```
LokaleGlobaleVariable> LINUXLIBC6/prog
a=10 und b=10
a=20 und b=20
a= 0 und b=20
a=10 und b=20
```

Variablen können nicht nur bezüglich einer Funktion lokal sein, sondern können überall angelegt werden, wo auch eine Anweisung stehen könnte. Variablendefinitionen leiten somit einen Block ein, und nur in diesem existieren die definierten Variablen.
Wir sehen an dem Beispiel auch, dass es möglich ist, aus einer Funktion heraus auf globale Variablen zuzugreifen. Vor dieser Möglichkeit kann aber nicht genug gewarnt werden! Durch den Zugriff auf globale Variablen entwickeln sich Abhängigkeiten im Programm, die sehr schwer nachvollziehbar sind.
Je enger eine Information begrenzt ist, d.h. je kleiner der Gültigkeitsbereich einer Variablen ist, desto weniger Fehler, z.B. durch unbeabsichtigte Änderungen an der Variablen, kann man machen.
Eine Variable, die nur innerhalb einer Funktion benutzt wird, sollte immer lokale Variable der Funktion sein (z.B. Variablen für Zwischenergebnisse).
Von der Übersicht her sind *Call by value*-Funktionen ohne Zugriff auf globale Variablen optimal: Man sieht genau, was in die Berechnung eingeht und was herauskommt. Der einzige gute Grund, Funktionen nicht zu verwenden, ist Effizienz.
Call by value-Prozeduren, die keinen Wert zurückgeben, werden fast immer auf globale Variablen zugreifen oder Prozeduren aufrufen, die dies tun, denn viel mehr Möglichkeiten gibt es gar nicht, um überhaupt Aktivität zu zeigen.
Prominentes Beispiel: `IO.Put`. Tatsächlich verwendet `IO.Put` die globale Variable `Stdio.stdout`, nämlich den Standard-Ausgabestrom.
Vermeiden Sie solches Verhalten in eigenen Prozeduren! Geben Sie dem Benutzer ihrer Prozedur die Möglichkeit den Ausgabestrom selbst festzulegen.
Statt

```
PROCEDURE PutHello () =
  BEGIN
    IO.Put ("Hello!\n");
  END PutHello;
```

zu schreiben, können Sie sich zum Beispiel eine Variable vom Typ Ausgabestrom (`Wr.T`, also *Writer*, siehe Abschnitte 3.11, B.9) übergeben lassen. Im Modul `Wr` gibt es die Routine `PutText`, die wie `IO.Put` einen Text ausgibt, allerdings in eine frei wählbare Datei statt auf die Standardausgabe.

```
PROCEDURE PutHello (wr: Wr.T; ) =
  BEGIN
    Wr.PutText (wr, "Hello!\n");
  END PutHello;
```

Kurz gesagt: Arbeiten Sie immer so lokal wie möglich und so global wie nötig.

3.4.9 Wiederholung: Funktionsparameter sind lokale Variablen

```
MODULE Main;
(* $Id: LokaleVariableQuad.m3,v 1.3 2004/02/04 15:51:28 thielema
   Exp $ *)
(* Funktionsparameter sind lokale Variable (noch einmal) *)

IMPORT IO, Fmt, Lex, Stdio;

PROCEDURE Quadrat (x: INTEGER; ) =
  VAR quad: INTEGER;
  BEGIN
    x := 7;
    quad := x * x;
    IO.Put(Fmt.F("Das Quadrat von %s ist %s.\n", Fmt.Int(x),
                 Fmt.Int(quad)));
  END Quadrat;

VAR x: INTEGER;
<* FATAL ANY *>
BEGIN
  IO.Put("Bitte geben Sie eine Zahl ein: ");
  x := Lex.Int(Stdio.stdin);
  Quadrat(x);
  IO.Put(Fmt.F("x = %s\n", Fmt.Int(x)));
END Main.
```

```
LokaleVariableQuad> LINUXLIBC6/prog
Bitte geben Sie eine Zahl ein: 34
Das Quadrat von 7 ist 49.
x := 34
```

Das Beispiel zeigt noch einmal, dass die Variable `x`, mit der die Funktion `quadrat` aufgerufen wird, innerhalb der Funktion lokalen Charakter hat: Sie kann intern verändert werden, aber das hat außerhalb keine Wirkung.

3.4.10 Call by reference

Man kann in `Modula-3` Parameter auch so übergeben, dass die Funktion die Variable des Aufrufers ändern kann. Hierbei wird der Funktion nicht eine Kopie des

Variablenwertes übergeben, sondern ein Verweis auf die Variable. Insbesondere darf man an solche Parameter auch keine Ausdrücke übergeben. Das Verfahren heißt neudeutsch *Call by reference* und wird im Funktionskopf mit einem `VAR` vor dem Parameternamen angezeigt.
Im folgenden Beispiel soll gezeigt werden, wie man mit diesem Verhalten Prozeduren in der Art von `INC` und `DEC` schreiben kann.

```
MODULE Main;
(* $Id: CallByReference.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO, Fmt;

PROCEDURE Mul (VAR x: INTEGER; y: INTEGER; ) =
  BEGIN
    x := x * y;
  END Mul;

PROCEDURE Div (VAR x: INTEGER; y: INTEGER; ) =
  BEGIN
    x := x DIV y;
  END Div;

VAR
  a: INTEGER := 10;
  b: INTEGER;

BEGIN
  Mul(a, 20);
  INC(a, 100);
  Div(a, 7);

  b := (10 * 20 + 100) DIV 7;

  IO.Put(
    Fmt.F(
      "Beide Rechnung sollten das gleiche Ergebnis liefern:\n"
        & "%s = %s\n", Fmt.Int(a), Fmt.Int(b)));
END Main.
```

Mit `VAR`-Parametern sollte sparsam umgegangen werden, denn sie lassen den Benutzer der Funktion über einiges im Unklaren:

- Wird die übergebene Variable von der Funktion ausgelesen, muss der Aufrufer sie also vor Übergabe initialisieren?
- Wird die übergebene Variable von der Funktion beschrieben, also muss sich der Aufrufer selbst Kopien des Wertes anlegen, wenn er ihn noch braucht?

Verwendet man hingegen konsequent normale Parameter (der Übergabemodus heißt `VALUE`, wird aber meistens nicht hingeschrieben) zur Dateneingabe und den Funktionswert zur Ausgabe, gibt es keine Verwirrungen.
Der häufigste Grund für die Anwendung von `VAR`-Parametern ist daher die Effizienz bei größeren Daten. Bei großen Feldern (Abschnitt 3.7) ist es wesentlich effizienter, nur einen Verweis auf das Feld zu übergeben, statt das ganze Feld zu kopieren. Wenn das Feld aber gar nicht verändert wird, sollte man statt des `VAR`-Parameters lieber den Modus `READONLY` wählen. Jetzt prüft der Compiler, dass das übergebene Datum tatsächlich nicht von der Funktion geändert wird.
Faustregel: Zur Übergabe kleiner Werte (ordinale Werte, kleine Mengen, Zeiger (insbesondere auf Objekte)) normale Parameter benutzen, zur Übergabe großer Datenmengen (Felder, Datenverbünde) `READONLY`-Parameter und für die Rückgabe immer die Rückgabewerte (`RETURN`) benutzen, gegebenenfalls in Datenverbünden oder Zeigern. (Siehe Abschnitte 3.9 und 3.10.)

3.4.11 Rekursive Funktionen

Eine mächtige Programmiertechnik ist die *Rekursion*.
Funktionen, die sich selbst aufrufen oder auch zwei Funktionen, die sich wechselseitig aufrufen, heißen *rekursiv*.
Eine Rekursion erlaubt oft eine elegante Lösung. Wenn es hingegen auch eine iterative Lösung (also mit Schleife, siehe Abschnitt 3.5) gibt, ist diese meist effizienter.
Beispiel: Berechnung der Fakultätsfunktion $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$

```
PROCEDURE FakIterativ (n: CARDINAL; ): CARDINAL =
  VAR prod: CARDINAL := 1;
  BEGIN
    FOR i := 1 TO n DO
      prod := prod * i;
    END;
    RETURN prod;
  END FakIterativ;
```

Idee für die Rekursion: $n! = n \cdot (n-1)!$

```
PROCEDURE FakRekursiv (n: CARDINAL; ): CARDINAL =
  BEGIN
    IF n = 0 THEN
      RETURN 1;
    ELSE
      RETURN n * FakRekursiv(n - 1);
    END;
  END FakRekursiv;
```

Endloses rekursives Aufrufen muss vermieden werden. Dies kann man dadurch erreichen, dass die fortschreitenden rekursiven Aufrufe „einfacher“ werden (oben `n` $\rightarrow$ `n-1` $\rightarrow$ `n-2` $\rightarrow \ldots$) und immer irgendwann ein Fall eintritt (hier `n=0`), bei dem kein rekursiver Aufruf zur Berechnung des Funktionswertes mehr nötig ist. Diese einfachsten Fälle werden *Rekursionsverankerung* genannt.

3.5 Schleifen

3.5.1 WHILE- und REPEAT-Schleife

Auch die in Teil 2 eingeführten Algorithmen-Bausteine zur Wiederholung von Anweisungen werden direkt in die Modula-3-Syntax übersetzt:

```
SOLANGE Bedingung erfüllt FÜHRE AUS      WHILE bedingung DO
  Anweisungen                              Anweisungen
                                         END;

WIEDERHOLE                               REPEAT
  Anweisungen                              Anweisungen
BIS Bedingung erfüllt                    UNTIL bedingung;
```

Beispiel: Eingabeüberprüfung

1. Mit WHILE-*Schleife*

```
IO.Put("Eingabe einer Zahl zwischen 0 und 200: ");
z := Lex.Int(Stdio.stdin);
WHILE z < 0 OR 200 < z DO
  IO.Put("Eingabe einer Zahl zwischen 0 und 200: ");
  z := Lex.Int(Stdio.stdin);
END;
```

2. Mit REPEAT-*Schleife*

```
REPEAT
  IO.Put("Eingabe einer Zahl zwischen 0 und 200: ");
  z := Lex.Int(Stdio.stdin);
UNTIL 0 <= z AND z <= 200;
```

In diesem Beispiel ist die REPEAT-Schleife besser geeignet als die WHILE-Schleife, weil mindestens einmal zur Eingabe aufgefordert werden soll.
Noch ein Beispiel: WHILE ohne Anweisungsblock

```
PROCEDURE EingabeZahl (): INTEGER =
  BEGIN
    IO.Put("Bitte geben Sie eine Zahl ein: ");
    RETURN Lex.Int(Stdio.stdin);
  END EingabeZahl;
```

```
BEGIN
  WHILE EingabeZahl() # 200 DO END;
END Main.
```

Die Abfrage wird erst beendet, wenn 200 eingegeben wurde.

3.5.2 LOOP-Schleife

Es gibt außerdem die LOOP-Schleife (so was wie eine persönliche PIN-Nummer), bei der innerhalb der Schleife über den Abbruch entschieden wird. Das Kommando zum Abbruch der Schleife heißt EXIT.

```
LOOP
  IO.Put("Eingabe einer Zahl zwischen 0 und 200: ");
  z := Lex.Int(Stdio.stdin);
  IF 0 <= z AND z <= 200 THEN EXIT END;
  IO.Put("Zwischen 0 und 200, du Nase!\n");
END;
```

3.5.3 FOR-Schleife

Kennt man die genaue Zahl der Schleifendurchläufe schon vorab, sollte man immer eine FOR-Schleife benutzen. Beispiel: Berechnung der Summe $\sum_{k=1}^{100} k$

```
s := 0;                      s := 0;
k := 1;
WHILE k <= 100 DO            FOR k:=1 TO 100 DO
  INC (s, k);                  INC (s, k);
  INC (k);
END;                         END;
```

Allgemein haben FOR-Schleifen folgende Syntax:

```
FOR var := anfang TO ende BY schritt DO
  Anweisungen
END;
```

Der Teil BY schritt kann weggelassen werden, wenn schritt 1 beträgt.
Eine solche Schleife wird folgendermaßen abgearbeitet:

1. Berechne die Ausdrücke `anfang`, `ende`, `schritt`

2. Lege neue Variable `var` mit einem Typ an, der alle Werte von `anfang` bis `ende` umfasst. Diese Variable ist nur innerhalb der Schleife verfügbar und nur lesbar.

3. Weise der Variablen `var` nacheinander die Werte `anfang`, `anfang+schritt`, `anfang+2*schritt`, ... zu (also eine arithmetische Folge) und führe jeweils alle Anweisungen in der Schleife aus.

4. Stoppe, wenn der Wert von `var` den von `ende` überschreitet, falls `schritt > 0`, oder unterschreitet, falls `schritt < 0`.

Eine `FOR`-Schleife kann auch in eine `WHILE`-Schleife umgeschrieben werden.
`FOR`her:

```
FOR n := a TO b BY c DO
  A;
END;
```

Nachher:

```
VAR
  n := a;
BEGIN
  IF c >= 0 THEN
    WHILE n <= b DO
      A;
      INC(n, c);
    END
  ELSE
    WHILE n >= b DO
      A;
      INC(n, c);
    END
  END
END
```

Die `LOOP`-Schleife ist die flexibelste aber auch unübersichtlichste Schleifenform in `Modula-3`. Bevorzugt werden sollten in dieser Reihenfolge:

1. `FOR`

2. `WHILE`

3. `REPEAT`
4. `LOOP`

.

3.6 Fließkommavariablen

Intern werden rationale Zahlen auf dem Computer in der Form

$$\pm 0.z_1 z_2 \ldots z_t \cdot B^e$$

mit der Basis $B = 2$, Ziffern $z_i \in \{0, \ldots, B-1\}$ mit $z_1 \neq 0$ und einem beschränkten Exponenten $e \in \mathbb{Z}$ dargestellt.
Wieviele Bits zur Verfügung gestellt werden, um das Vorzeichen, die Ziffern und den Exponenten abzuspeichern, ist vom Typ und auch vom Computersystem abhängig. Zahlen vom Typ `REAL` sind eigentlich immer einfach genaue IEEE-Fließkommazahlen, welche Näherungen für die Werte aus

$$[-10^{38}, 10^{38}]$$

enthalten. Obwohl der Zahlentyp in `Modula-3` hochstapelnd `REAL` heißt, kann `REAL` in Wirklichkeit nur rationale Zahlen darstellen und davon auch nur eine geringe, nämlich *endliche* Auswahl. Diese Teilmenge der im Fließkommaformat darstellbaren rationalen Zahlen ist nach oben und unten beschränkt und die Zahlen sind außerdem von endlicher Genauigkeit.
Der Typ `REAL` soll als Näherung an reelle Zahlen verstanden werden, so dass man auch mit transzendenten Zahlen wie π und transzendenten Funktionen wie $\sin$ rechnen kann. Es muss aber immer gerundet werden und man hat ständig mit Rundungsfehlern zu kämpfen. Schlussendlich gibt es eine ganze mathematische Disziplin, die numerische Mathematik, welche sich intensiv solchen Problemen widmet.
Ein kleines Beispiel (ansonsten mehr dazu im Mathematischen Praktikum):

```
MODULE Main;
(* $Id: Rundungsfehler.m3,v 1.4 2004/02/26 16:14:42 thielema Exp
   $ *)
(* Rundungsfehlereinfluss: Addiere n mal 1.0/n *)

IMPORT IO, Fmt, Lex, Stdio;

VAR
```

```
  x: REAL;
  s: REAL     := 0.0;
  n: CARDINAL;

<* FATAL ANY *>
BEGIN
  IO.Put("Wie oft soll addiert werden? ");
  n := Lex.Int(Stdio.stdin);
  x := 1.0 / FLOAT(n, REAL);

  FOR i := 1 TO n DO s := s + x; END;

  IO.Put(Fmt.F("%s mal %5s = %s \n", Fmt.Int(n), Fmt.Real(x),
               Fmt.Real(s)));
  IO.Put(Fmt.F("Fehler: %s\n", Fmt.Real(ABS(s - 1.0))));
END Main.
```

```
Rundungsfehler> LINUXLIBC6/prog
Wie oft soll addiert werden? 100
100 mal  0.01 = 0.99999934
Fehler: 6.556511e-7

Rundungsfehler> LINUXLIBC6/prog
Wie oft soll addiert werden? 1000
1000 mal 0.001 = 0.9999907
Fehler: 9.298325e-6

Rundungsfehler> LINUXLIBC6/prog
Wie oft soll addiert werden? 10000
10000 mal 0.0001 = 1.0000535
Fehler: 0.00005352497
```

Konkret gibt es in `Modula-3` drei Datentypen für Fließkommavariablen:

Typ	Speicherplatz (implementations-abhängig)	Formatierung mit	Literal
`REAL`	4 Bytes	`Fmt.Real`	1.0E0 oder 1.0
`LONGREAL`	8 Bytes	`Fmt.LongReal`	1.0D0
`EXTENDED`	8-12 Bytes	`Fmt.Extended`	1.0X0

Anhand des Dezimalpunktes unterscheidet der Compiler zwischen Fließkommazahlen und ganzen Zahlen. Ausdrücke wie `1 / 2.0D0` sind daher nicht erlaubt, stattdessen muss man `1.0D0 / 2.0D0` schreiben.
Zur Typumwandlung muss man die `FLOAT`-Funktion benutzen.
Beispiel:

```
VAR
  a, b: INTEGER;
  x, y: LONGREAL;
BEGIN
  a := 3;
  b := 2;
  x := FLOAT(a,LONGREAL) / FLOAT(b,LONGREAL);
  y := FLOAT(a,LONGREAL);
END;
```

Die Grundrechenarten werden in gewohnter Art und Weise durch die Symbole `+`, `-`, `*`, `/` repräsentiert. Die Prozeduren `INC` und `DEC` dagegen sind für Fließkommazahlen nicht definiert.

3.6.1 Formatierung

Eine Fließkommazahl kann im Festkommaformat (z.B. `x=123.456`) oder in Exponentialschreibweise (z.B. `x=0.123456e-2`) ausgegeben werden, wobei angegeben werden kann, wieviele Ziffern nach dem Dezimalpunkt verwendet werden sollen. Einige Beispiele:

`x : REAL := 2367.12864;`	
`Fmt.Real(x, Fmt.Style.Auto)`	`2367.1287`
`Fmt.Real(x, Fmt.Style.Fix)`	`2367.12870000`
`Fmt.Real(x, Fmt.Style.Sci)`	`2.36712870e+03`
`Fmt.Real(x, Fmt.Style.Auto, 2)`	`2370`
`Fmt.Real(x, Fmt.Style.Fix, 2)`	`2367.13`
`Fmt.Real(x, Fmt.Style.Sci, 2)`	`2.37e+03`
`Fmt.Real(x, Fmt.Style.Auto, 2, TRUE)`	`2370.0`
`Fmt.Real(x, Fmt.Style.Fix, 2, TRUE)`	`2367.13`
`Fmt.Real(x, Fmt.Style.Sci, 2, TRUE)`	`2.37e+03`

3.6.2 Beispiel: Berechnung von π nach Leibniz

Von Leibniz stammt die Reihenentwicklung

$$\begin{aligned} \frac{\pi}{4} &= \arctan 1 \\ &= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \mp \dots \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}, \end{aligned}$$

die wir zur approximativen Berechnung von π benutzen wollen, obwohl es dazu deutlich bessere Algorithmen gibt:

```
MODULE Main;
(* $Id: ApproxPi.m3,v 1.3 2004/02/04 15:51:28 thielema Exp $ *)
(* Approximation von Pi mit Verfahren von Leibniz *)

IMPORT IO, Fmt, Lex, Stdio;

VAR
  max   : CARDINAL;
  nenner: REAL     := 1.0;
```

```
  pi    : REAL     := 0.0;

<* FATAL ANY *>
BEGIN
  IO.Put("Wieviele Reihenglieder sollen berechnet werden? ");
  max := Lex.Int(Stdio.stdin);

  FOR n := 0 TO max - 1 DO
    IF n MOD 2 = 0 THEN
      pi := pi + 1.0 / nenner;
    ELSE
      pi := pi - 1.0 / nenner;
    END;
    nenner := nenner + 2.0;
  END;
  pi := pi * 4.0;
  IO.Put(
    Fmt.F("n = %8s,  pi = %s\n", Fmt.Int(max), Fmt.Real(pi)));
END Main.
```

```
ApproxPi> LINUXLIBC6/prog
Wieviele Reihenglieder sollen berechnet werden? 200
n :=      200,  pi := 3.136593

ApproxPi> LINUXLIBC6/prog
Wieviele Reihenglieder sollen berechnet werden? 20000
n :=    20000,  pi := 3.1415472

ApproxPi> LINUXLIBC6/prog
Wieviele Reihenglieder sollen berechnet werden? 2000000
n :=  2000000,  pi := 3.141596
```

3.6.3 Mathematische Funktionen

Für jeden Fließkommazahlentyp besitzt die `Modula-3`-Standardbibliothek ein Modul für grundlegende Funktionen zur Manipulation von Fließkommazahlen.
Die Module heißen `RealFloat`, `LongFloat`, `ExtendedFloat`.

`PROCEDURE Scalb(x: T; n: INTEGER): T;`	Return `x * 2^n`.
`PROCEDURE Logb(x: T): T;`	Return the exponent of `x`. More precisely, return the unique `n` such that the ratio `ABS(x) / Base^n` is in the range `[1..Base-1]`, unless `x` is denormalized, in which case return the minimum exponent value for T.
`PROCEDURE ILogb(x: T): INTEGER;`	Like Logb, but returns an integer, never raises an exception, and always returns the `n` such that `ABS(x) / Base^n` is in the range `[1..Base-1]`, even for denormalized numbers.
`PROCEDURE NextAfter(x, y: T): T;`	Return the next representable neighbor of `x` in the direction towards `y`. If `x = y`, return `x`.
`PROCEDURE CopySign(x, y: T): T;`	Return `x` with the sign of `y`.
`PROCEDURE Finite(x: T): BOOLEAN;`	Return `TRUE` if `x` is strictly between minus infinity and plus infinity. This always returns `TRUE` on non-IEEE machines.
`PROCEDURE IsNaN(x: T): BOOLEAN;`	Return `FALSE` if `x` represents a numerical (possibly infinite) value, and `TRUE` if `x` does not represent a numerical value. For example, on IEEE implementations, returns `TRUE` if `x` is a NaN, `FALSE` otherwise.
`PROCEDURE Sign(x: T): [0..1];`	Return the sign bit of `x`. For non-IEEE implementations, this is the same as `ORD(x >= 0)`; for IEEE implementations, `Sign(-0) = 1` and `Sign(+0) = 0`.
`PROCEDURE Differs(x, y: T): BOOLEAN;`	Return `(x < y OR y < x)`. Thus, for IEEE implementations, `Differs(NaN,x)` is always `FALSE`; for non-IEEE implementations, `Differs(x,y)` is the same as `x # y`.
`PROCEDURE Unordered(x, y: T): BOOLEAN;`	Return `NOT (x <= y OR y <= x)`.
`PROCEDURE Sqrt(x: T): T;`	Return the square root of `x`. This must be correctly rounded if `FloatMode.IEEE` is `TRUE`.

`TYPE IEEEClass =`
`{SignalingNaN, QuietNaN,`
`Infinity, Normal, Denormal, Zero};`
 	
`PROCEDURE Class(x: T): IEEEClass;`
	Return the IEEE number class containing `x`.

Von `Modula-3` werden in dem Modul `Math` Schnittstellen zu einer Reihe mathematischer Standardfunktionen für den Typ `LONGREAL` zur Verfügung gestellt.

Teilweise überschneiden sich die Funktionen mit denen aus `LongFloat`. In diesem Falle sollten die Funktionen aus `LongFloat` verwendet werden.

Funktion	**Funktionsweise**
`Pi`	π, also halber Umfang des Einheitskreises
`LogPi`	$\ln \pi$
`SqrtPi`	$\sqrt{\pi}$
`E`	EULERsches e
`Degree`	Radianten pro Grad
`acos`	*Arcus Kosinus*: Umkehrfunktion des Kosinus im Bogenmaß im Intervall $[0, \pi]$. `w=acos(a);` $\iff w = \arccos(a)$
`asin`	*Arcus Sinus*: Umkehrfunktion des Sinus im Bogenmaß im Intervall $[-\pi/2, \pi/2]$. `w=asin(a);` $\iff w = \arcsin(a)$
`atan`	*Arcus Tangens*: Umkehrfunktion des Tangens im Bogenmaß. `w=atan(a);` $\iff w = \arctan(a),\ w \in [-\pi/2, \pi/2]$
`atan2`	`w=atan2(a,b);` $\iff w = \arctan(\frac{a}{b}),\ w \in [-\pi, \pi]$
`ceil`	*Obere Gaußklammer*: Nächstgrößere Ganzzahl `w=ceil(a);` $\iff w = \lceil a \rceil$
`cos`	*Kosinus*: Berechnung des Kosinus mit Argument im Bogenmaß `w=cos(a);` $\iff w = \cos(a)$
`cosh`	*Kosinus Hyperbolicus*: Berechnung des Kosinus Hyperbolicus mit Argument im Bogenmaß `w=cosh(a);` $\iff w = \cosh(a)$
`exp`	*Exponentialfunktion*: `w=exp(a);` $\iff w = e^a$
`expm1`	*Exponentialfunktion* verringert um eins: Diese Implementation ist numerisch günstiger bei kleinen Exponenten. `w=exp(a)-1;` $\iff w = e^a - 1$
`fabs`	*Absolutwert*: `w=fabs(a);` $\iff w = \lvert a \rvert$
`floor`	*Untere Gaußklammer*: Nächstkleinere Ganzzahl `w=floor(a);` $\iff w = \lfloor a \rfloor$
`fmod`	*Modulofunktion*: Analogon zu `a MOD b` für rationale Zahlen `w=fmod(a,b);` $\iff w = a \bmod b$
`frexp`	Aufspaltung von a in $a = f \cdot 2^i$, $f \in [0.5, 1)$ wird zurückgegeben, i in b gespeichert

	`f=frexp(a,b);` $\iff$ $a = f \cdot 2^b$
`ldexp`	Umkehrfunktion zu `frexp`. `w=ldexp(a,b);` $\iff$ $w = a \cdot 2^b$
`log`	*Logarithmus Naturalis*: Logarithmus zur Basis e `w=log(a);` $\iff$ $w = \ln(a) = \log_e(a)$
`log10`	*Dekadischer Logarithmus*: Logarithmus zur Basis 10 `w=log10(a);` $\iff$ $w = \lg(a)$
`log1p`	*natürlicher Logarithmus* vom um eins erhöhten Argument: numerisch günstiger bei Argumenten nahe 1 `w=log1p(a);` $\iff$ $w = \ln(a+1)$
`modf`	Aufspaltung einer Zahl a in Ganzahlanteil i und Rest $f = a - i$, $f \in [0,1)$ wird zurückgegeben, $i \in \mathbb{Z}$ in b gespeichert. `f=modf(a,b);` $\iff$ $a = b + f$
`pow`	*Potenzfunktion*: `w=pow(a,b);` $\iff$ $w = a^b$
`sin`	*Sinus*: Berechnung des Sinus mit Argument im Bogenmaß `w=sin(a);` $\iff$ $w = \sin(a)$
`sinh`	*Sinus Hyperbolicus*: Berechnung des Sinus Hyperbolicus mit Argument im Bogenmaß `w=sinh(a);` $\iff$ $w = \sinh(a)$
`sqrt`	*Quadratwurzel*: `w=sqrt(a);` $\iff$ $w = \sqrt{a}$
`tan`	*Tangens*: Berechnung des Tangens mit Argument im Bogenmaß `w=tan(a);` $\iff$ $w = \tan(a)$
`tanh`	*Tangens Hyperbolicus*: Berechnung des Tangens Hyperbolicus mit Argument im Bogenmaß `w=tanh(a);` $\iff$ $w = \tanh(a)$

Ein einfaches Beispiel:

```
MODULE Main;
(* $Id: Mathefunktionen.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)
(* Benutzung der Mathe-Bibliothek *)

IMPORT IO, Fmt, Lex, Stdio;
IMPORT Math;

VAR x: LONGREAL;
```

```
<* FATAL ANY *>
BEGIN
  IO.Put("Zahl eingeben: ");
  x := Lex.LongReal(Stdio.stdin);
  IO.Put(Fmt.F("Wurzel von %s: %s \n", Fmt.LongReal(x),
             Fmt.LongReal(Math.sqrt(x))));
  IO.Put(Fmt.F("Exponentialfunktion an der Stelle %s: %s \n",
             Fmt.LongReal(x), Fmt.LongReal(Math.exp(x))));
END Main.
```

```
Mathefunktionen> LINUXLIBC6/prog
Zahl eingeben: 2
Wurzel von 2: 1.4142135623730951
Exponentialfunktion an der Stelle 2: 7.38905609893065
```

Weitere Funktionen kann man selber programmieren.
Beispiel: Die Winkelfunktionen erwarten Argumente im Bogenmaß, für die Umrechnung zwischen Grad- und Bogenmaß kann man folgende Funktionen benutzen:

```
MODULE Main;
(* $Id: Bogenmass.m3,v 1.3 2004/02/04 15:51:28 thielema Exp $ *)
(* Bogen-und Gradmass *)

IMPORT IO, Fmt, Lex, Stdio;

CONST Degree = 0.017453292519943295769236907684D0;

PROCEDURE RadToDeg (a: LONGREAL; ): LONGREAL =
  BEGIN
    RETURN a / Degree;
  END RadToDeg;

PROCEDURE DegToRad (a: LONGREAL; ): LONGREAL =
  BEGIN
    RETURN a * Degree;
  END DegToRad;

VAR x: LONGREAL;
<* FATAL ANY *>
BEGIN
```

```
  IO.Put("Zahl im Bogenmass eingeben: ");
  x := Lex.LongReal(Stdio.stdin);
  IO.Put(Fmt.F("%s im Bogenmass = %s im Gradmass \n",
               Fmt.LongReal(x), Fmt.LongReal(RadToDeg(x))));

  IO.Put("Zahl im Gradmass eingeben: ");
  x := Lex.LongReal(Stdio.stdin);
  IO.Put(Fmt.F("%s im Gradmass = %s im Bogenmass \n",
               Fmt.LongReal(x), Fmt.LongReal(DegToRad(x))));
END Main.
```

```
Bogenmass> LINUXLIBC6/prog
Zahl im Bogenmass eingeben: 1
1 im Bogenmass := 57.29577951308232 im Gradmass
Zahl im Gradmass eingeben: 180
180 im Gradmass := 3.141592653589793 im Bogenmass
```

3.7 Felder (Arrays)

3.7.1 Eindimensionale Felder

Ein *Feld* ist die geeignete Datenstruktur, um Variablen des gleichen Typs zusammenzufassen. In dieser Art zusammengefasste Variablen (Feldelemente) können leichter in Schleifen abgearbeitet werden, und die Anzahl der Variablen kann sogar während der Laufzeit variieren!
Beispiel: Statt

```
  VAR z0, z1, z2, z3, z4, z5, z6, z7, z8, z9: INTEGER;
  BEGIN
    IO.Put("Haste ma 10 Zahlen für mich?\n");
    z0 := Lex.Int(Stdio.stdin);
    z1 := Lex.Int(Stdio.stdin);
    z2 := Lex.Int(Stdio.stdin);
    z3 := Lex.Int(Stdio.stdin);
    z4 := Lex.Int(Stdio.stdin);
    z5 := Lex.Int(Stdio.stdin);
    z6 := Lex.Int(Stdio.stdin);
    z7 := Lex.Int(Stdio.stdin);
    z8 := Lex.Int(Stdio.stdin);
```

```
    z9 := Lex.Int(Stdio.stdin);
  END;
```

schreibt man lieber

```
  VAR z: ARRAY [0 .. 9] OF INTEGER;
  BEGIN
    IO.Put(Fmt.F("Haste ma %s Zahlen für mich?\n",
                 Fmt.Int(NUMBER(z))));
    FOR i := FIRST(z) TO LAST(z) DO
      z[i] := Lex.Int(Stdio.stdin);
    END;
  END;
```

Daran erkennt man folgendes:

- Ein Feld besitzt einen Indextyp, im Beispiel den Unterbereichstyp `[0..9]`.
- Ein Feld besitzt einen Werttyp, im Beispiel den Ganzzahltyp `INTEGER`.
- Ein Feld ist selbst wieder ein Typ.
- Geschrieben wird der Feldtyp als `ARRAY Indextyp OF Wertetyp`
- Der erste und der letzte Index einer Feldvariable oder eines Feldtyps lassen sich mit `FIRST` bzw. `LAST` ermitteln. Die Größe des Feldes erfährt man mit `NUMBER`.
- Auf ein Feldelement greift man mit eckigen Klammern zu, `a[i]` entspricht etwa dem mathematischen a_i.
- Mathematisch gesehen ist ein Feld eine Funktion, die einen Wert vom Indextyp auf einen Wert vom Wertetyp abbildet.

Noch ein Beispiel, welches auf dem Typen `Grundfarbe` aus dem Abschnitt 3.2.4 aufbaut:

```
TYPE
  Mischfarbe = ARRAY Grundfarbe OF LONGREAL;
```

Folgende Beispiele zeigen, wie man Felder mit festem Inhalt anlegt:

```
CONST
  orange = Mischfarbe{1.0D0, 0.5D0, 0.0D0};
  lila   = Mischfarbe{1.0D0, 0.0D0, 1.0D0};
```

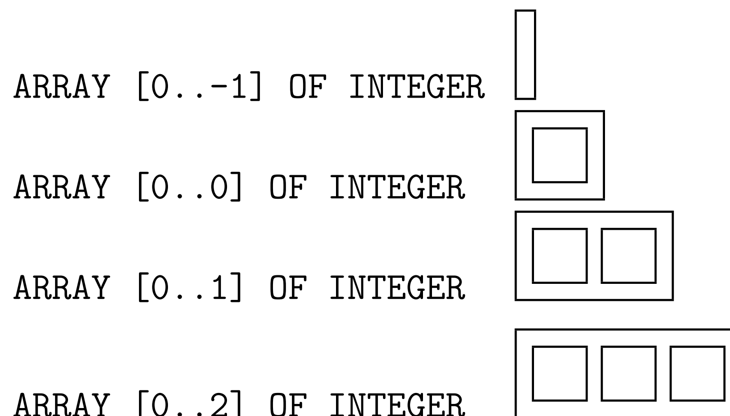
oder auch

```
CONST
  vorzeichen =
    ARRAY BOOLEAN OF TEXT{"negativ", "0 oder positiv"};
VAR
  z: INTEGER;
BEGIN
  IO.Put("Bitte geben sie eine ganze Zahl ein: ");
  z := Lex.Int(Stdio.stdin);
  IO.Put(Fmt.F("Die Zahl ist %s.\n", vorzeichen[z >= 0]));
END;
```

Das letzte Beispiel beantwortet auch die in Abschnitt 3.2.4 aufgeworfene Frage nach der Konvertierung eines Aufzählungswertes in einen Text.
Wichtig: Obwohl ein Feld mit genau einem Element genau so viel Information enthält, wie eine Variable vom Elementtyp, sind beide Dinge grundsätzlich verschieden. Beispiel:

```
VAR
  a: ARRAY [0..0] OF INTEGER;
  b: INTEGER;
```

Die Variablen `a` und `b` verwalten zwar die gleiche Menge Information, allerdings ist `a` ein Feld, man kann z.B. auf dessen Elemente zugreifen (`a[0]`) aber nicht damit rechnen (`a+1` ist verboten), dagegen ist `b` eine Zahl, man kann damit rechnen, aber keine Feldoperationen darauf anwenden. Insbesondere kann man `a` und `b` nicht einander zuweisen. Stattdessen führen die Zuweisungen `a[0] := b` und `b := a[0]` zum gewünschten Ergebnis.
Man kann sich ein Feld wie einen Rahmen vorstellen: Egal, ob ein Feld aus keinem, einem oder mehr als einem Element besteht, der Rahmen ist immer da.



3.7.2 Mehrdimensionale Felder

Auch mehrdimensionale Felder werden von Modula-3 direkt unterstützt, so lange die Dimension konstant ist. Die Ausdehnung in jeder Dimension kann dagegen variabel sein, dazu mehr im Abschnitt 3.10.1. Ein mehrdimensionales Feld ist einfach ein Feld von Feldern, wie hier am Spielfeld von *TicTacToe* demonstriert:

```
TYPE Symbol = {leer, o, x};
VAR z: ARRAY [0 .. 2] OF ARRAY [0 .. 2] OF Symbol;
BEGIN
  z[0][1] := Symbol.o;
END;
```

Hierbei könnte man `ARRAY [0 .. 2] OF Symbol` als eine Zeile von drei Symbolen auffassen, und `ARRAY [0 .. 2] OF ARRAY [0 .. 2] OF Symbol` als eine Spalte von Symbolzeilen. Umgekehrt stehen `z` für das zweidimensionale Spielfeld, `z[0]` für die oberste Zeile, und `z[0][1]` für das mittlere Symbol der obersten Zeile.
Neben dieser analytischen Variante gibt es für Schreibfaule auch Abkürzungen, sowohl beim Typ als auch beim Zugriff auf Feldelemente:

```
TYPE Symbol = {leer, o, x};
VAR z: ARRAY [0 .. 2], [0 .. 2] OF Symbol;
BEGIN
  z[0, 1] := Symbol.o;
END;
```

3.7.3 Felder variabler Größe

Der große Vorteil von Feldern gegenüber einzelnen Variablen besteht darin, dass Felder nicht nur sehr groß, sondern auch noch in der Größe variabel sein können. Das heißt nicht, dass leicht neue Elemente hinzugefügt und entfernt werden können, sondern dass man ein Feld beliebiger Größe anlegen kann, dessen Größe sich später nicht mehr ändern kann. Benötigt man mehr Flexibilität, kann man auf Module der Standardbibliothek wie `Sequence` oder `List` zurückgreifen.
Ein Feld mit variabler Größe, genaugenommen ein Feld, dessen Größe zum Zeitpunkt der Übersetzung nicht bekannt ist, heißt *offenes Feld*. Der Typ eines offenen Feldes wird zum Beispiel so geschrieben: `ARRAY OF INTEGER`. Der Indextyp wird nicht hingeschrieben und ist ein Unterbereich der ganzen Zahlen beginnend bei 0.
Es ist nicht möglich Variablen vom Typ *offenes Feld* anzulegen, dagegen können Variablen Zeiger auf offene Felder sein (einmal mehr der Verweis auf Abschnitt 3.10.1) und Funktionsparameter können selbst offene Felder sein.

Beispiel: Durchschnitt von n Zahlen

```
MODULE Main;
(* $Id: ArithMittel.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)
(* Durchschnitt von einer beliebigen Anzahl Zahlen *)

IMPORT IO, Fmt;

PROCEDURE Durchschnitt (READONLY x: ARRAY OF LONGREAL; ):
  LONGREAL =
  VAR summe: LONGREAL := 0.0D0;
  BEGIN
    FOR n := FIRST(x) TO LAST(x) DO summe := summe + x[n]; END;
    RETURN summe / FLOAT(NUMBER(x), LONGREAL);
  END Durchschnitt;

CONST zahl = ARRAY OF LONGREAL{0.0D0, 8.0D0, 15.0D0};

BEGIN
  IO.Put("Der Durchschnitt der Zahlen");
  FOR n := FIRST(zahl) TO LAST(zahl) DO
    IO.Put(" " & Fmt.LongReal(zahl[n]));
  END;
  IO.Put(" ist " & Fmt.LongReal(Durchschnitt(zahl)) & "\n");
END Main.
```

Das Beispiel zeigt außerdem eine prominente Anwendung des `READONLY`-Übergabe-Modus, nämlich die effiziente Übergabe großer Datenmengen, ohne dass der Aufrufer um seine Daten fürchten muss (siehe Abschnitt 3.4.10). Außerdem haben wir der Übersicht wegen den `&`-Operator aus Abschnitt 3.8.2 verwendet, der zwei Texte aneinanderhängt.

3.7.4 Felder und Mengen

Vom Informationsgehalt her sind `ARRAY Typ OF BOOLEAN` und `SET OF Typ` gleichwertig. Das Feld kann man so als Menge interpretieren: Das Element `i` ist in der Menge enthalten, wenn der entsprechende Feldeintrag den Wert `TRUE` enthält. Formal, wenn a ein Feld und s eine äquivalente Menge sind, dann gilt für alle `i`: `a[i] = i IN s`.

Eine Menge wird meist kompakter gespeichert und es gibt effiziente Operationen auf den Mengen als solchen, z.B. Vereinigung. Daher wird in solchen Fällen wohl der Mengentyp den Zuschlag bekommen. Wenn der Grundbereich sehr groß oder die Größe vorab gar nicht bekannt ist, können Typen aus den Standardbibliotheken weiterhelfen: Das Modul `BitVector` ist für dichtbesetzte Ganzzahlmengen und das Modul `IntSet` für dünnbesetzte Ganzzahlmengen geeignet. „Dünnbesetzt“ soll hier heißen, dass gemessen an der Mächtigkeit des Grundbereiches die Mengen nur sehr wenige Elemente enthalten.

3.8 Zeichenketten (Texte)

3.8.1 Der Datentyp Zeichen (CHAR)

Mit Zeichen haben wir bereits in Abschnitt 3.2.1 kurz zu tun gehabt. Zeichen sind Ziffern, Buchstaben, Sonderzeichen und Steuerzeichen. Literale dieses Typs werden in einfache Hochkommata gesetzt, zum Beispiel `'A'`.
Einen Überblick über alle Zeichen und deren interne Kodierung können wir uns leicht selbst verschaffen:

```
MODULE Main;
(* $Id: ASCIITabelle.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)

IMPORT IO, Fmt;

BEGIN
  FOR c := FIRST(CHAR) TO LAST(CHAR) DO
    IO.Put(Fmt.F("Das Zeichen mit der Nummer %03s ist: %s\n",
                 Fmt.Int(ORD(c)), Fmt.Char(c)))
  END;
END Main.
```

Da `CHAR` ein ordinaler Typ ist, kann man ihn überall dort verwenden, wo auch ganze Zahlen eingesetzt werden: Für Unterbereiche, als Indextyp für Felder, als Basistyp für Mengen, als Schleifenzähler, in `CASE`-Anweisungen.
Nützliches im Umgang mit Zeichen ist im Modul `ASCII.i3` definiert (Abschnitt B.5).

```
MODULE Main;
(* $Id: ASCIIFunktionen.m3,v 1.3 2004/03/05 08:52:42 thielema Exp
```

```
   $ *)
(* Beispielprogramm zu CHAR-Variablen und ASCII.i3 *)

IMPORT IO, Fmt, Stdio, ASCII;

VAR z: CHAR;
<* FATAL ANY *>
BEGIN
  REPEAT
    IO.Put("\nBitte ein Zeichen eingeben: ");
    z := IO.GetChar(Stdio.stdin);
    (* Auf "RETURN" warten.  Lex.Skip ist zu umstaendlich. *)
    WHILE IO.GetChar(Stdio.stdin) # ASCII.NL DO END;

    IO.Put(Fmt.F("Der ASCII Code des Zeichens ist %s\n",
                 Fmt.Int(ORD(z))));
    IO.Put(Fmt.F("Zeichen als Grossbuchstabe: '%s'\n",
                 Fmt.Char(ASCII.Upper[z])));
    IO.Put(Fmt.F("... und als Kleinbuchstabe: '%s'.\n",
                 Fmt.Char(ASCII.Lower[z])));
  UNTIL z IN ASCII.Controls;
  IO.Put("\n");
END Main.
```

```
Bitte ein Zeichen eingeben: A
Der ASCII Code des Zeichens ist 65
Zeichen als Grossbuchstabe: 'A'
... und als Kleinbuchstabe: 'a'.

Bitte ein Zeichen eingeben: ?
Der ASCII Code des Zeichens ist 63
Zeichen als Grossbuchstabe: '?'
... und als Kleinbuchstabe: '?'.

Bitte ein Zeichen eingeben:
Der ASCII Code des Zeichens ist 32
Zeichen als Grossbuchstabe: ' '
... und als Kleinbuchstabe: ' '.
```

```
Bitte ein Zeichen eingeben: ^G
Der ASCII Code des Zeichens ist 7
Zeichen als Grossbuchstabe: ''
... und als Kleinbuchstabe: ''.
```

3.8.2 Der Datentyp TEXT

Eigentlich hätte man mit Feldern aus Zeichen, also `ARRAY OF CHAR`, einen gar nicht so schlechten Datentyp für Texte. Trotzdem gibt es in `Modula-3` den speziellen Typ `TEXT`, dessen Implementation effizienter oder flexibler sein kann als ein Feld von Zeichen.
Objekte vom Typ `TEXT` sind unveränderbar. Es gibt keine Operation, die etwa Zeichen im Text abändert. Stattdessen erzeugen die Standard-Funktionen aus alten Texten neue.
Textliterale sind in Anführungszeichen eingeschlossene Zeichenketten wie `"Das ist ein Text."`.
Es gibt nur eine festverdrahtete Operation für Texte: Die *Verkettung*, repräsentiert durch das Symbol `&`. Textliterale dürfen sich nicht über mehrere Zeilen erstrecken, wohl aber Text-Ausdrücke mit dem Verkettungsoperator:

```
"Hier fange ich an zu schreiben " &
"und hier geht es ganz bequem weiter."
```

Dieser Ausdruck ergibt einen einzeiligen Text! Weitergehende Operationen für Texte findet man im Modul `Text` (Abschnitt B.4).
Die Relationen `=` und `#` bedeuten bei Textvariablen übrigens nicht, dass die repräsentierten Texte gleich bzw. ungleich sind, sondern dass es sich um die gleichen Objekte handelt. In Wirklichkeit sind Variablen vom Typ `TEXT` nämlich nur Verweise auf die eigentliche Textdatenstruktur (siehe Abschnitt 3.10). Das, was wir unter dem Vergleichen von Texten verstehen, erledigt die Routine `Text.Equal`. Es folgt, dass wenn `a` und `b` Texte sind und `a = b` gilt, dann ist sicher auch `Text.Equal` wahr. Die Umkehrung gilt, wie gesagt, nicht.
Auch bei Texten der gleiche Hinweis wie bei Feldern (Abschnitt 3.7.1): Die Literale `'A'` und `"A"` sind völlig verschieden. Das erste ist ein Zeichen, das zweite ein Text, der zufälligerweise nur ein Zeichen enthält.

3.8.3 Ein- und Ausgabe von Texten

Die Ausgabe von Texten bereitet uns keinerlei Schwierigkeiten, im Prinzip machen wir das schon die ganze Zeit.

Die Eingabe ist etwas kniffeliger. Bis jetzt hatten wir es meistens mit Zahleneingaben zu tun. Bei einer Zahl ist irgendwie klar, wann die Zahl zu Ende ist. Bei Texten, die prinzipiell aus beliebigen Zeichen bestehen können, zum Beispiel auch aus Zeilenvorschüben, ist das nicht so einfach.
Die Funktion `Lex.Scan` liest einen Text vom Eingabestrom ein. Man kann eine Menge der erlaubten Zeichen übergeben. Es werden so lange Zeichen aus dem Eingabestrom gelesen, wie es sich um erlaubte Zeichen handelt. Sobald ein anderes Zeichen entdeckt wird, hört `Lex.Scan` mit Einlesen auf und gibt den bis dahin eingelesenen Text zurück. Gibt man nicht ausdrücklich eine Menge erlaubter Zeichen an, sind alle Zeichen außer Steuer- oder Leerzeichen erlaubt.

```
MODULE Main;
(* $Id: EingabeText.m3,v 1.3 2004/01/16 18:15:09 thielema Exp
   $ *)
(* Erzeugt Abkürzungen aus Wörtern, in dem es die Vokale
   entfernt *)

IMPORT IO, Lex, Stdio, Text;

CONST
  vokale = SET OF
           CHAR{'a', 'e', 'i', 'o', 'u', 'ä', 'ö', 'ü', 'A',
                'E', 'I', 'O', 'U', 'Ä', 'Ö', 'Ü'};

VAR text: TEXT;
<* FATAL ANY *>
BEGIN
  IO.Put("Bitte geben Sie ein langes Wort ein: ");
  text := Lex.Scan(Stdio.stdin, SET OF CHAR{'!'.. LAST(CHAR)});
  FOR n := 0 TO Text.Length(text) - 1 DO
    VAR c := Text.GetChar(text, n);
    BEGIN
      IF NOT c IN vokale THEN IO.PutChar(c); END;
    END;
  END;
  IO.PutChar('\n');
END Main.
```

```
EingabeText> LINUXLIBC6/prog
Bitte geben Sie ein langes Wort ein: Autofensterklorollenhäkelmütz
```

`tfnstrklrllnhklmtz`

Während die Funktionen aus dem Modul `Lex` Daten aus Eingabeströmen (also Dateien, Konsole) lesen und konvertieren, stellt das Modul `Scan` (Abschnitt B.3) ähnliche Funktionen für Texte zur Verfügung.

3.8.4 Ein Beispiel für Text-Verarbeitung: Test auf Bereichsüberschreitung

```
MODULE Main;
(* $Id: TestGanzzahltext.m3,v 1.4 2004/02/25 16:35:23 thielema
   Exp $ *)
(* Manueller Test auf Bereichsueberschreitung *)
(* vgl.  A.Willms *)

IMPORT IO, Fmt, Lex, Stdio;
IMPORT Text;
IMPORT Scan;

PROCEDURE Is16BitInt (s: TEXT; ): BOOLEAN =
  (* Test auf -32768 < Scan.Int(s) < 32768 *)
  (* Eingabe Text s *)
  (* Ausgabe TRUE falls Ungleichung erfuellt *)
  CONST maxint = ARRAY OF CHAR{'3', '2', '7', '6', '8'};
  VAR i := 0;
  BEGIN
    (* Leerer Text bedeutet: Keine Zahl *)
    IF Text.Length(s) = 0 THEN RETURN FALSE; END;

    (* Vorzeichen *)
    IF Text.GetChar(s, i) IN SET OF CHAR{'+', '-'} THEN
      INC(i);
    END;

    (* Fehlt bei Willms *)
    IF Text.Length(s) - i > 5 THEN RETURN FALSE; END;
    IF Text.Length(s) - i < 5 THEN RETURN TRUE; END;

    (* Vergleiche Text lexikographisch mit Textdarstellung von
```

```
       32768, Text.Compare wuerde das auch erledigen. *)
    FOR j := 0 TO Text.Length(s) - i - 1 DO
      VAR
        c := Text.GetChar(s, i + j);
        d := maxint[j];
      BEGIN
        IF c > d THEN
          RETURN FALSE;
        ELSIF c < d THEN
          RETURN TRUE;
        END;
      END;
    END;
    RETURN TRUE;
  END Is16BitInt;

VAR
  text : TEXT;
  passt: BOOLEAN;
<* FATAL ANY *>
BEGIN
  REPEAT
    IO.Put("Bitte Wert eingeben: ");
    text :=
      Lex.Scan(Stdio.stdin, SET OF CHAR{'+', '-', '0'.. '9'});
    EVAL IO.GetChar(Stdio.stdin); (* Schlucke RETURN *)

    passt := Is16BitInt(text);
    IF NOT passt THEN
      IO.Put("Wert ausserhalb des gueltigen Bereichs!\n\n");
    ELSE
      VAR wert := Scan.Int(text);
      BEGIN
        IO.Put(Fmt.F("Der eingelesene Text \"%s\""
                       & " entspricht dem Zahlenwert %s.\n",
                      text, Fmt.Int(wert)));
      END;
    END;
  UNTIL passt;
```

```
END Main.
```

```
TestGanzzahltext> LINUXLIBC6/prog
Bitte Wert eingeben: 42023
Wert ausserhalb des gueltigen Bereichs!

Bitte Wert eingeben: -123456
Wert ausserhalb des gueltigen Bereichs!

Bitte Wert eingeben: +12321
Der eingelesene Text "+12321" entspricht dem Zahlenwert 12321.
```

Dieser aufwändige Test der Eingaben ist zum Glück nicht notwendig. Wie man elegant Eingabefehler behandeln kann, sehen wir in Abschnitt 3.12.

3.9 Datenverbünde

In Feldern kann man Elemente des *gleichen* Datentyps zusammenfassen, die dann eine Einheit bilden und effizient sequentiell verarbeitet werden können.
Häufig will man aber auch Elemente *unterschiedlicher* Datentypen zusammenfassen, beispielsweise für geometrische Objekte (Kreise, die durch Mittelpunkt und Radius beschrieben werden, usw.) oder Personendaten (Vorname, Nachname, Alter, Adresse, ...). Dafür bietet `Modula-3` die Datenverbünde (`RECORD`). Die Syntax für die Elementliste eines *Datenverbund*es ist genau die gleiche, wie die des Variablen-Deklarationsblockes:

```
TYPE
  Person = RECORD
             vorname,
             nachname: TEXT;
             masse:    REAL := 0.0;
           END;
```

Hiermit führen wir für unseren neuen Datenverbund gleich einen neuen Namen (`Person`) ein. Damit können wir bereits eine erste kleine Datenbank aufbauen, unbestritten eines der beliebtesten Beispiele zur Demonstration von Datenverbünden.

```
CONST
  jodlerin = Person {"Lieselotte", "Hoppenstedt", 60.0};
```

```
  promis = ARRAY OF Person {
             Person {"Klaus",      "Hülsensack",  75.0},
             Person {"Gabi",       "Flüssigbrot", 55.0},
             Person {"N.N.",       "Hülsensack"},
             jodlerin
           };
```

Auf ein Element eines Datenverbundes greift man mit Hilfe eines Punktes zu: `jodlerin.vor`
Ebenso kann man Datenverbünde variablen Inhalts anlegen und erst recht Felder von Datenverbünden

```
VAR
  omis: ARRAY [0 .. 3] OF Person;
```

welche man sowohl simultan, als auch einzeln, als auch elementweise beschreiben kann:

```
  omis := promis;
  omis[0] := jodlerin;
  omis[0].masse := 100.0;
```

.
Elemente können nahezu alle erdenklichen Typen haben, insbesondere auch Datenverbünde oder Felder (außer offene Felder). Damit lässt sich ein Datenverbund noch besser strukturieren und gegebenenfalls in kleinere Stücke aufteilen, die auch einzeln gebraucht werden. Hier unser verfeinertes Beispiel aus der deutschen Verwaltung:

```
TYPE
  Geschlecht = {weib, kerl};

  Datum    = RECORD
               tag:        [1..31];
               monat:      [1..12];
               jahr:       CARDINAL;
             END;

  Person   = RECORD
               vorname,
               nachname:   TEXT;
               geschlecht: Geschlecht;
               geburt:     Datum;
```

```
          END;

  Adresse = RECORD
              strasse:    TEXT;
              haus:       CARDINAL;
              plz:        [00000..99999];
              ort:        TEXT;
            END;

  Akte    = RECORD
              person:     Person;
              wohnhaft:   Adresse;
            END;
```

3.10 Zeiger (Pointer)

Mit dem Datentyp *Zeiger* kann man Daten miteinander verketten.

```
VAR
  p: REF INTEGER;
```

Dadurch wird die Variable p deklariert, die nicht selbst eine ganze Zahl enthält, sondern nur auf eine solche Variable verweist (eine Referenz, daher REF).

3.10.1 Operationen auf Zeigern

Wie bekommt man einen Verweis auf ein existierendes Objekt? Dazu gibt es die ADR-Operation, welche aber im Normalfall (sprich: in sicheren Modulen) gesperrt ist. Das ist nötig, denn mit ADR kann man sich zum Beispiel einen Verweis auf eine lokale Variable besorgen, der noch weiterbesteht, obwohl die Variable, auf die verwiesen wird, gar nicht mehr existiert:

```
PROCEDURE Illegal (): ADDRESS =
  VAR
    a: INTEGER := 42;
  BEGIN
    RETURN ADR(a);
  END Illegal;
```

Diese Prozedur würde einen Verweis auf a zurückgeben, aber in dem Moment, wo der Aufrufer diesen Wert erhält, existiert die Variable a gar nicht mehr.

Tatsächlich benötigt man die `ADR`-Funktion nicht, so lange man auf hoher Ebene programmiert. In der Regel bekommt man Verweise nur auf neu erzeugte Objekte. Dafür gibt es die `NEW`-Funktion. Diese reserviert Speicher für ein Feld, Datenverbund oder Objekt und gibt einen Verweis darauf zurück.
Das besondere an Verweisen der Marke `REF` ist, (im Gegensatz zu `UNTRACED REF`) dass sie während der Laufzeit beobachtet werden und wenn kein Verweis mehr auf ein bestimmtes Objekt existiert, wird dieses Objekt gelöscht. Das dafür zuständige Untersystem heißt *Garbage Collector*.
Als Besonderheit bei offenen Feldern erwartet `NEW` außer dem Typ auch die tatsächlichen Ausdehnungen des Feldes in jeder Dimension mit Kommata getrennt. Ein Feld für das Spiel „Raummühle“ (auch „3D-TicTacToe“) legt man zum Beispiel mit

```
NEW(REF ARRAY OF ARRAY OF ARRAY OF Symbol, 4, 4, 4)
```

an.
Mit `NEW` lässt sich endlich vernünftig mit offenen Feldern arbeiten. Zur Erinnerung: Als Variablen oder Rückgabewerte von Funktionen sind offene Felder nicht erlaubt, wohl aber Verweise auf offene Felder.

```
PROCEDURE Scale (READONLY x: ARRAY OF LONGREAL; k: LONGREAL; ):
  REF ARRAY OF LONGREAL =
  VAR z: REF ARRAY OF LONGREAL;
  BEGIN
    z := NEW(REF ARRAY OF LONGREAL, NUMBER(x));
    FOR i := FIRST(x) TO LAST(x) DO z^[i] := x[i] * k; END;
    RETURN z;
  END Scale;
```

Hier wurde bereits die dritte wichtige Operation für Zeiger verwendet: Das Dereferenzieren (Symbol: `^`). Dereferenzieren ist die Umkehrung zum Referenzieren (`ADR`). Im obigen Beispiel ist `z` der Verweis auf ein offenes Feld, `z^` ist das Feld selbst, `z^[i]` bezeichnet das `i`. Element des Feldes, auf das `z` zeigt. Weil diese Kombination so häufig auftritt, darf man auch kurz `z[i]` schreiben:

```
PROCEDURE Scale (READONLY x: ARRAY OF LONGREAL; k: LONGREAL; ):
  REF ARRAY OF LONGREAL =
  VAR z := NEW(REF ARRAY OF LONGREAL, NUMBER(x));
  BEGIN
    FOR i := FIRST(x) TO LAST(x) DO z[i] := x[i] * k; END;
    RETURN z;
  END Scale;
```

In einem etwas komplexeren Beispiel erstellen wir eine HILBERT-*Matrix* und geben diese aus:

```
MODULE Main;
(* $Id: HilbertMatrix.m3,v 1.2 2004/03/01 15:44:15 thielema Exp
   $ *)

IMPORT IO, Fmt;

TYPE Matrix = REF ARRAY OF ARRAY OF LONGREAL;

PROCEDURE HilbertMatrix (n: CARDINAL; ): Matrix =
  VAR hilb := NEW(Matrix, n, n);
  BEGIN
    FOR i := 0 TO n - 1 DO
      FOR j := 0 TO n - 1 DO
        hilb[i, j] := 1.0D0 / FLOAT(i + j + 1, LONGREAL);
      END;
    END;
    RETURN hilb;
  END HilbertMatrix;

PROCEDURE FmtMatrix (z: Matrix): TEXT =
  VAR t: TEXT := "";
  BEGIN
    FOR i := 0 TO LAST(z^) DO
      FOR j := 0 TO LAST(z[i]) DO
        t := t & Fmt.Pad(Fmt.LongReal(z[i, j]), 20);
      END;
      t := t & "\n";
    END;
    RETURN t;
  END FmtMatrix;

<* FATAL IO.Error *>
BEGIN
  IO.Put("Größe der Hilbert-Matrix: ");
  IO.Put(FmtMatrix(HilbertMatrix(IO.GetInt())));
END Main.
```

3.10.2 Zeiger auf Datenverbünde

Aus Zeigern, Datenverbünden und Feldern lassen sich nun sehr mächtige Datenstrukturen aufbauen. Um bei unserem bürokratischen Beispiel zu bleiben, könnte man dem Umstand Rechnung tragen, dass die gleiche Person zu verschiedenen Zeiten an verschiedenen Orten gelebt hat. Wir behalten die Datenstrukturen `Person`, `Adresse` und `Datum` bei und legen jeweils eine Akte für den Tatbestand an, dass sich eine Person in einem bestimmten Zeitraum an einer bestimmten Adresse einquartiert hat. Dabei sollen die Personen- und Adressdaten nicht jedesmal neu eingetragen werden, sondern wir verweisen einfach auf existierende Personen und Adressen.

```
TYPE
  VerweisAkte = RECORD
                  person:   REF Person;
                  wohnhaft: REF Adresse;
                  von, bis: Datum;
                END;
```

Obacht! Wenn wir

```
VAR
  a, b: Akte;
BEGIN
  a := b;
END;
```

schreiben, dann wird die komplette Akte `b` mit den Personendaten nach `a` kopiert. Verwendet man hingegen

```
VAR
  a, b: VerweisAkte;
BEGIN
  a := b;
END;
```

so findet man danach in `a.person` nur eine Kopie *des Verweises* `b.person`, also keinen Verweis auf eine Kopie von `b.person^`.
In diesem Fall erscheint dieses Verhalten sehr natürlich. Bei komplexeren Datenstrukturen muss man sich allerdings überlegen, welche Art von Kopie man benötigt. Betrachtet man eine Datenstruktur, die aus vielen Feldern und Datenverbünden durch Verweise zusammengesetzt ist, dann heißt eine Kopie der Datenstruktur

- *tiefe Kopie*, wenn alle Bestandteile der Datenstruktur vervielfältigt werden, so dass die Kopie völlig vom Original entkoppelt ist, und

- *flache Kopie*, wenn nur der Hauptbestandteil (die *Wurzel*) kopiert wird.

Nochmal: Die Deklaration

```
VAR
  p: REF Person;
```

legt lediglich einen *Verweis* auf eine Person an, nicht jedoch eine Person *und* einen Verweis! Zunächst verweist p auf gar nichts. Es wird ein Fehler zur Laufzeit ausgelöst, wenn man dennoch versucht, auf p^ zuzugreifen. Erst mit NEW kann man eine Person erzeugen, auf die man einen Verweis erhält. Speziell für Datenverbünde erlaubt NEW, dass man einzelne Elemente des Datenverbundes initialisiert. Das sieht etwa so aus:

```
p := NEW (REF Person, vorname  := "Klaus",
                      nachname := "Hülsensack");
```

So enthält p^.vorname bereits den Wert "Klaus". Wie bei Zeigern auf Felder darf man die Dereferenzierung bei Zeigern auf Datenverbünde weglassen, wenn man auf Elemente zugreift, also p.vorname statt p^.vorname tut es auch.
Hier das Komplettbeispiel:

```
MODULE Main;
(* $Id: KopieFlachTief.m3,v 1.2 2004/01/16 18:15:09 thielema Exp
   $ *)
(* Datenverbund, Verweise, tiefe und flache Kopie. *)

IMPORT IO, Fmt;

TYPE
  Geschlecht = {weib, mann};

  Datum = RECORD
            tag  : [1 .. 31];
            monat: [1 .. 12];
            jahr : CARDINAL;
          END;

  Person = RECORD
             vorname, nachname: TEXT;
             geschlecht       : Geschlecht;
             geburt           : Datum;
```

```
           END;

  Adresse = RECORD
              strasse: TEXT;
              haus   : CARDINAL;
              plz    : [00000 .. 99999];
              ort    : TEXT;
            END;

  Akte = RECORD
           person  : Person;
           wohnhaft: Adresse;
         END;

  VerweisAkte = RECORD
                  person  : REF Person;
                  wohnhaft: REF Adresse;
                  von, bis: Datum;
                END;

PROCEDURE PersonAlsText (READONLY p: Person; ): TEXT =
  BEGIN
    RETURN
      Fmt.FN(
        "%s %s %s,\ngeboren am %02s.%02s.%04s",
        ARRAY OF
          TEXT{
          ARRAY Geschlecht OF TEXT{"Frau", "Herr"}[p.geschlecht],
          p.vorname, p.nachname, Fmt.Int(p.geburt.tag),
          Fmt.Int(p.geburt.monat), Fmt.Int(p.geburt.jahr)});
  END PersonAlsText;

PROCEDURE AdresseAlsText (READONLY adr: Adresse; ): TEXT =
  BEGIN
    RETURN Fmt.F("%s %s\n%05s %s\n", adr.strasse,
                 Fmt.Int(adr.haus), Fmt.Int(adr.plz), adr.ort);
  END AdresseAlsText;

PROCEDURE AkteAlsText (READONLY akte: Akte; ): TEXT =
```

```
  BEGIN
    RETURN
      Fmt.F("%s ist wohnhaft in\n%s", PersonAlsText(akte.person),
            AdresseAlsText(akte.wohnhaft));
  END AkteAlsText;

PROCEDURE VerweisAkteAlsText (READONLY akte: VerweisAkte; ):
  TEXT =
  BEGIN
    RETURN Fmt.F("%s ist wohnhaft in\n%s",
                 PersonAlsText(akte.person^),
                 AdresseAlsText(akte.wohnhaft^));
  END VerweisAkteAlsText;

BEGIN
  (* Der Datenverbund 'Akte' enthält alle Unterdatenverbunde
     direkt.  Daher ist eine Kopie immer eine tiefe Kopie. *)
  VAR
    a: Akte;
    b := Akte{Person{"Klaus", "Hülsensack", Geschlecht.mann,
                     Datum{12, 3, 1968}},
              Adresse{
                "How-many-Road", 42, 01234, "Kleinkleckersdorf"}};
  BEGIN
    a := b;
    IO.Put("Tiefe Kopie:\n\n");
    IO.Put(AkteAlsText(a));
    b.person.nachname := "Flüssigbrot";
    b.wohnhaft.strasse := "Produktionsstraße";
    IO.Put("\nDie Akte enthält nun:\n" & AkteAlsText(a) & "\n\n");
  END;

  (* Der Datenverbund 'VerweisAkte' enthält nur Verweise auf
     Unterdatenverbunde.  Daher ist eine Kopie immer eine flache
     Kopie. *)
  VAR
    a: VerweisAkte;
    b := VerweisAkte{NEW(REF Person, vorname := "Klaus",
                         nachname := "Hülsensack",
```

```
                        geschlecht := Geschlecht.mann,
                        geburt := Datum{12, 3, 1968}),
                    NEW(REF Adresse, strasse := "How-many-Road",
                        haus := 42, plz := 01234,
                        ort := "Kleinkleckersdorf"),
                    Datum{24, 12, 2003}, Datum{31, 12, 2003}};
  BEGIN
    a := b;
    IO.Put("Flache Kopie:\n\n");
    IO.Put(VerweisAkteAlsText(a));
    b.person.nachname := "Flüssigbrot";
    b.wohnhaft.strasse := "Produktionsstraße";
    IO.Put("\nDie Akte enthält nun:\n" & VerweisAkteAlsText(a)
             & "\n\n");
  END;
END Main.
```

```
KopieFlachTief> LINUXLIBC6/prog
Tiefe Kopie:

Herr Klaus Hülsensack,
geboren am 12.03.1968 ist wohnhaft in
How-many-Road 42
01234 Kleinkleckersdorf

Die Akte enthält nun:
Herr Klaus Hülsensack,
geboren am 12.03.1968 ist wohnhaft in
How-many-Road 42
01234 Kleinkleckersdorf


Flache Kopie:

Herr Klaus Hülsensack,
geboren am 12.03.1968 ist wohnhaft in
How-many-Road 42
01234 Kleinkleckersdorf
```

```
Die Akte enthält nun:
Herr Klaus Flüssigbrot,
geboren am 12.03.1968 ist wohnhaft in
Produktionsstraße 42
01234 Kleinkleckersdorf
```

3.10.3 Felder von Feldern

Jetzt noch ein Beispiel dafür, wie man Zeiger und Felder sinnvoll kombinieren kann. Wir möchten die Werte des PASCALschen Dreieckes in eine Datenstruktur schreiben. Das PASCAL*sche Dreieck* besteht aus den Binomialkoeffizienten $\binom{n}{k}$. Für $k < 0$ und $n < k$ ist $\binom{n}{k} = 0$, deswegen kann man sich den Speicher für diese k schenken.
Zur Speicherung der ersten m Zeilen des PASCALschen Dreiecks bräuchte man also ein dreieckiges Feld. Doch wie erzeugt man so etwas?

1. Man nimmt einfach ein quadratisches Feld mit „Seitenlänge“ m und lässt fast die Hälfte der Einträge frei. Das ist auf jeden Fall Speicherverschwendung.

2. Man ordnet alle Zeilen hintereinander in einem eindimensionalen Feld an. Eine Routine wie

   ```
   PROCEDURE TriangleArrayGet (READONLY arr : ARRAY OF INTEGER;
                                          n, k: INTEGER; ): INTEGER =
     BEGIN
       RETURN arr[n * (n + 1) DIV 2 + k];
     END TriangleArrayGet;
   ```

   ermittelt daraus den richtigen Eintrag.

3. Man nimmt ein Feld von Verweisen auf Felder `ARRAY OF REF ARRAY OF INTEGER` und initialisiert alle Zeiger mit offenen Feldern der entsprechenden Zeilenlänge. Auf die Feldelemente greift man dann mit `a[n]^[k]` oder kurz `a[n][k]` oder sogar ganz kurz `a[n,k]` zu und es wird zur Laufzeit immer sichergestellt, dass nur auf Werte im Dreieck zugegriffen wird!

   ```
   MODULE Main;
   (* $Id: PascalDreieck.m3,v 1.1 2004/03/01 15:44:15 thielema Exp
      $ *)

   IMPORT IO, Fmt;
   ```

```
(*Der Typ eignet sich nicht nur für Dreiecke, sondern für jede
   Art von Matrix mit Flatterrand.*)
TYPE Pascal = REF ARRAY OF REF ARRAY OF CARDINAL;

PROCEDURE CreatePascal (n: CARDINAL; ): Pascal =
  VAR p := NEW(Pascal, n);
  BEGIN
    FOR i := 0 TO n - 1 DO
      p[i] := NEW(REF ARRAY OF CARDINAL, i + 1);
      FOR j := 0 TO i DO
        (*Die richtigen Werte zu berechnen ist Übungsaufgabe!
           :-]*)
        p[i, j] := 100 * i + j;
      END;
    END;
    RETURN p;
  END CreatePascal;

PROCEDURE FmtPascal (p: Pascal; ): TEXT =
  VAR t: TEXT := "";
  BEGIN
    FOR i := 0 TO LAST(p^) DO
      FOR j := 0 TO LAST(p[i]^) DO
        t := t & Fmt.Pad(Fmt.Int(p[i, j]), 10);
      END;
      t := t & "\n";
    END;
    RETURN t;
  END FmtPascal;

<* FATAL IO.Error *>
BEGIN
  IO.Put("Größe des Pascalschen Dreiecks: ");
  IO.Put(FmtPascal(CreatePascal(IO.GetInt())));
END Main.
```

3.10.4 Funktionenvariablen

Es ist in Modula-3 nicht möglich, während des Programmlaufes neue Prozeduren zu erzeugen. Das ist auch gut so, denn man braucht es in dieser Allgemeinheit nicht wirklich und die Abwesenheit dieser Technik erhöht die Übersicht, statische Sicherheit und Effizienz deutlich.
Dagegen ist es aber möglich Verweise auf Funktionen als Variablen oder Funktionsparameter zu definieren.

```
VAR
  a: PROCEDURE (x: LONGREAL; ): LONGREAL;
BEGIN
  a := Math.sin;
END;
```

Tatsächlich handelt es sich bei a und `Math.sin` nur um Verweise, denn es wird kein Programmcode kopiert, sondern lediglich die (Einsprung-)Adresse des Maschinencodes der Funktion `Math.sin`.
Ein kleines Programm möge zeigen, wie man auf diese Weise eine zusammengesetzte Funktion realisieren kann:

```
MODULE Main;
(* $Id: Funktionszeiger.m3,v 1.5 2004/02/04 15:51:28 thielema Exp
   $ *)
(* Zeiger auf Funktionen *)
(* vgl.  A.Willms und P.Benner *)

IMPORT IO, Fmt, Lex, Stdio;
IMPORT Math;

(* Konstante Funktion *)
PROCEDURE F1 (<* UNUSED *> x: LONGREAL; ): LONGREAL =
  BEGIN
    RETURN 0.0D0;
  END F1;

(* Funktion, bei der alle Ableitungen im Nullpunkt
   verschwinden *)
PROCEDURE F2 (x: LONGREAL; ): LONGREAL =
  BEGIN
    RETURN Math.exp(-1.0D0 / (x * x));
```

```
  END F2;

PROCEDURE ZsgFkt (f1, f2: PROCEDURE (x: LONGREAL; ): LONGREAL;
                  x     : LONGREAL;                              ):
  LONGREAL =
  BEGIN
    IF x <= 0.0D0 THEN RETURN f1(x); ELSE RETURN f2(x); END;
  END ZsgFkt;

VAR x: LONGREAL;

<* FATAL ANY *>
BEGIN
  IO.Put("Funktionsargument eingeben: ");
  x := Lex.LongReal(Stdio.stdin);

  IO.Put(Fmt.F("f(x) = %s\n", Fmt.LongReal(ZsgFkt(F1, F2, x))));
END Main.
```

3.11 Umgang mit Dateien

Nicht immer möchte man Daten über die Konsole in ein Programm eingeben. Oft soll ein Programm mit Ausgaben eines anderen Programmes gefüttert werden. Ebenso ist es nicht immer nützlich, neu erzeugte Daten auf der Konsole auszugeben. Man kommt also nicht um die Arbeit mit Dateien herum. Zwar kann man über die Shell ein Programm so starten, dass es als Standardein- und ausgabe Dateien zugewiesen bekommt. Das funktioniert allerdings nur, wenn es sich um jeweils nur eine Eingabe- und nur eine Ausgabedatei handelt. Und es funktioniert auch nicht, wenn das Programm die Dateien nicht nur von Anfang bis Ende einlesen, sondern auch in der Datei hin- und herspringen will.

Um die gleichen Programme sowohl für Tastatureingaben, wie auch für Dateien verwenden zu können, werden von den meisten Betriebssystem die Tastatureingaben in die Konsole wie eine endlose Datei behandelt. Auf der anderen Seite gibt es eine Datei, deren Inhalt, so wie er geschrieben wird, auf der Konsole erscheint.

Deswegen unterscheidet man auch in `Modula-3` nicht wirklich zwischen Dateien und der Konsole. Der wichtigste Unterschied ist, dass Dateien vor der ersten Benutzung geöffnet und nach der letzten Benutzung wieder geschlossen werden müssen. Dagegen sind die Dateien oder besser Ströme für die Ein- und Ausgabe auf der Konsole für

das Programm immer geöffnet. In `Modula-3` greift man auf sie über `Stdio.stdin` bzw. `Stdio.stdout` zu (Abschnitt B.6).
Auf der Konsole arbeitet man eigentlich immer mit darstellbaren Zeichen, dagegen benutzt man in Dateien durchaus auch andere Kodierungen.

- Textdateien: editierbarer „Klartext“, geordnete Zeichenfolgen in ASCII, meist zeilenweise strukturiert.
- Binärdateien: Die Daten werden in der gleichen kompakten Darstellung abgelegt, wie sie zum Speichern von Variablen und zur Berechnung benutzt werden. Derartige Dateien in einen Texteditor eingeladen, sehen ziemlich wirr aus.

Die Datentypen `Rd.T` und `Wr.T` repräsentieren Textein- bzw. Ausgabeströme. Die Module `Rd` (Abschnitt B.8) und `Wr` (Abschnitt B.9) sind sehr allgemein gehalten und enthalten Prozeduren, die auf jedem Textdatenstrom funktionieren.
Konkrete Textdatenströme werden zum Beispiel repräsentiert durch:

1. `FileRd` (B.10), `FileWr` (B.11)
   Lesen und Schreiben in Dateien, Standardein- und ausgabe eingeschlossen.
2. `TextRd`, `TextWr`
   Erlaubt es, Texte wie Dateien zu behandeln.

Binäre Dateien lassen sich mit den Modulen `File` und `FS` behandeln.
Für alle Dateien gilt:

1. Vor dem ersten Zugriff muss die Datei geöffnet werden.
   (a) Eine Datei zum Lesen wird mit einem Objekt vom Typ `FileRd.T` verwaltet. Durch `FileRd.Open` wird die Datei geöffnet und ein `FileRd.T` wird erzeugt und zurückgegeben.
   (b) Eine Datei zum Schreiben wird mit einem Objekt vom Typ `FileWr.T` verwaltet. Durch `FileWr.Open` wird der Inhalt der Datei gelöscht oder, falls nicht existent, wird eine Datei neu erzeugt und geöffnet. Durch `FileWr.OpenAppend` wird eine Datei, falls nicht existent erzeugt, und auf jeden Fall geöffnet. Folgende Ausgaben in diese Datei werden an das Ende der Datei angehängt.
2. Mit Befehlen wie `Wr.PutChar` und `Wr.PutText` schreibt man in eine Datei. Wie gewohnt kann man mit den Befehlen aus `Fmt` Daten vielfältig als Text aufbereiten und in die Textdatei schreiben.

   Mit Befehlen wie `Rd.GetChar`, `Rd.GetText`, `Rd.GetLine` oder auch den Routinen aus `Lex` liest man Daten aus einer Textdatei ein.

3. Bei einer einzulesenden Datei kann man mit `Rd.EOF` erfragen, ob das Ende der Datei erreicht wurde.

4. Nach dem letzten Zugriff muss eine geöffnete Datei mit `Rd.Close` bzw. `Wr.Close` wieder geschlossen werden. Durch einen Fehler beim Lesen oder Schreiben wird das Programm nicht weiter linear ausgeführt, deswegen ist es wichtig, die `Close`-Routinen immer im `FINALLY`-Zweig eines `TRY-FINALLY`s aufzurufen (siehe Abschnitt 3.12.2).

3.11.1 Beispiel zu Textdateien

```
MODULE Main;
(* $Id: Dateizugriff.m3,v 1.3 2004/01/16 18:15:09 thielema Exp
   $ *)
(* Beispiel fuer den Umgang mit Dateien: Lies einfache Textdatei
   ein und gib die Daten in einer HTML-Tabelle aus. *)

IMPORT FileRd, FileWr, Rd, Wr, Lex, Fmt, Stdio;
IMPORT OSError, FloatMode, Thread, AtomList, Atom;

PROCEDURE PutError (err: AtomList.T; ) =
  BEGIN
    WHILE err # NIL DO
      Wr.PutText(Stdio.stderr, Atom.ToText(err.head) & "\n");
      err := err.tail;
    END;
  END PutError;

VAR
  in : Rd.T;
  out: Wr.T;

<* FATAL Thread.Alerted, Wr.Failure *>
BEGIN
  TRY
    in := FileRd.Open("constants.txt");
    out := FileWr.Open("constants.html");
    TRY
      Wr.PutText(
        out,
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 3.2//EN\">\n");
      Wr.PutText(out, "<HTML>\n");
      Wr.PutText(out, "<HEAD></HEAD>\n");
      Wr.PutText(out, "<BODY>\n");
      Wr.PutText(out, "<TABLE BORDER>\n");
      TRY
```

```
        WHILE NOT Rd.EOF(in) DO
          VAR
            name  := Lex.Scan(in);
            value := Lex.LongReal(in);
          BEGIN
            Wr.PutText(
              out, Fmt.F("<TR><TD>%s</TD><TD>%s</TD></TR>\n",
                         name, Fmt.LongReal(value)));
          END;
          Lex.Skip(in);          (*Zeilenende ignorieren*)
        END;
      EXCEPT
      | Rd.Failure, Lex.Error, FloatMode.Trap =>
          Wr.PutText(
            Stdio.stderr,
            "Fehler in Eingabedatei "
              & "('tschuldigung für die ungenaue Fehlermeldung)\n");
      END;
      Wr.PutText(out, "</TABLE>\n");
      Wr.PutText(out, "</BODY>\n");
      Wr.PutText(out, "</HTML>\n");
    FINALLY
      Rd.Close(in);
      Wr.Close(out);
    END;
  EXCEPT
  | OSError.E (err) => PutError(err);
  | Wr.Failure (err) => PutError(err);
  | Rd.Failure (err) => PutError(err);
  END;
END Main.
```

3.12 Ausnahmebehandlung

Ein lästiges Übel beim Programmieren sind Fehler die beim Programmlauf auftreten können, gegen die man aber nichts unternehmen kann: Plötzlich ist der Speicher alle, ein Fenster passt nicht auf den Bildschirm, eine Datei lässt sich nicht öffnen, die eben noch da war, oder sie ist zu früh zu Ende oder enthält unsinnige Zeichen oder der Benutzer rastet aus oder oder oder.
Schon so eine einfache Routine wie `Lex.Int` kann auf vielfältige Weise fehlschlagen. Mit den bisher vorgestellten Mitteln würde man `Lex.Int` vielleicht so gestalten, dass es ein Element der Aufzählung

```
TYPE
  LexError = {ok, endOfFile, readProtected, notANumber};
```

zurückgibt. Diesen Rückgabewert müsste man bei jedem Aufruf von `Lex.Int` prüfen. Das artet schon beim schlichten Einlesen von drei Zahlenwerten aus einem Eingabestrom `rd` in solch eine Verzweigung aus:

```
err := Lex.Int(rd, a);
IF err = LexError.ok THEN
  err := Lex.Int(rd, b);
  IF err = LexError.ok THEN
    err := Lex.Int(rd, c);
    IF err = LexError.ok THEN
      Berechnung (a, b, c);
    END;
  END;
END;
IF err # LexError.ok THEN
  IO.Put (Fmt.F("Konnte eine der Zahlen nicht einlesen, Fehlernr. %s\n"
                Fmt.Int(ORD(err))));
END;
```

Die Routinen zur Fehlerausgabe `IO.Put`, `Fmt.F`, `Fmt.Int` können natürlich auch noch schiefgehen ...

3.12.1 `TRY-EXCEPT`-Konstrukt

Eben weil es so kompliziert ist, korrekte Fehlerbehandlungen auf diese Weise zu schreiben, verzichten etliche Programmierer ganz darauf ...
In `Modula-3` kann man die Fehlertests aus dem eigentlichen Programmfluss heraushalten. Routinen können Fehler auslösen, dadurch wird das Programm an dieser Stelle abgebrochen und es verzweigt in eine Fehlerbehandlungsroutine. Das obige Beispiel wird damit zu:

```
TRY
  a := Lex.Int(rd);
  b := Lex.Int(rd);
  c := Lex.Int(rd);
  Berechnung (a, b, c);
EXCEPT
| Lex.Error, FloatMode.Trap (typ) =>
  IO.Put ("Das waren nicht drei Zahlen.\n");
END;
```

Das allgemeine Schema der Ausnahmen ist: Wird eine Ausnahme ausgelöst, wird der gerade ausgeführte Anweisungsblock abgebrochen. Die Ausnahme „fällt“ dann so lange durch immer weiter außen liegende Fehlerbehandlungen (eingeleitet durch `EXCEPT`) bis in einem `EXCEPT`-Block die Ausnahme tatsächlich behandelt wird.
Das `TRY-EXCEPT`-Konstrukt sieht so aus:

```
TRY
  A;
EXCEPT
| Error1      => E1;
| Error2(var) => E2;
...
ELSE
  E3;
END;
```

Es wird versucht den Anweisungsblock `A` komplett auszuführen. Gelingt das, wird nach dem zu `TRY` gehörigen `END` fortgefahren. Tritt eine Ausnahmesituation ein, wird der Block `A` verlassen und die Ausnahme entsprechend der Liste nach `EXCEPT` behandelt.

- Wird `A` durch die Ausnahme `Error1` abgebrochen, wird danach `E1` ausgeführt. Nach dessen Beendigung geht es nach dem abschließenden `END` weiter.
- Die Ausnahme `Error2` gebe einen zusätzlichen Wert zurück. Wird `A` durch diese Ausnahme abgebrochen, wird die Variable `var` angelegt und mit diesem Wert initialisiert. Danach wird `E2` ausgeführt.
- Wird `A` durch eine andere als die aufgelisteten Ausnahmen abgebrochen, wird `E3` ausgeführt.

Aus einem `TRY-EXCEPT` können durchaus wieder Ausnahmen „herausfallen“, nämlich

1. wenn eine der Ausnahmebehandlungen selbst eine Ausnahme auslöst oder
2. wenn `ELSE` fehlt und eine Ausnahme auftritt, die von den anderen Ausnahmebehandlungen nicht abgedeckt wird.

Diese Technik der Fehlerbehandlung wird bewusst nicht „Fehlerbehandlung“ sondern *Ausnahmebehandlung* genannt, weil nicht alle derartigen Abbrüche automatisch Fehler sind. Mit einer Ausnahme kann man zum Beispiel eine Suche in den Tiefen einer Datenstruktur abbrechen und das gefundene Element zurückliefern.
Zwei Ausnahmen, die keine Fehler sind, sind uns sogar schon bekannt:

- EXIT bricht eine LOOP-Schleife ab,
- RETURN bricht eine PROCEDURE ab.

```
MODULE Main;
(* $Id: Ausnahmen.m3,v 1.3 2004/01/22 18:54:15 thielema Exp $ *)
(* Ausnahmebehandlung bei Lex.Int *)

IMPORT IO, Fmt, Stdio, Lex, Rd, Atom;
IMPORT Thread, FloatMode;

VAR a: INTEGER;

BEGIN
  IO.Put("Bitte geben Sie eine Zahl ein: ");
  LOOP
    TRY
      TRY
        a := Lex.Int(Stdio.stdin);
        EXIT;
      EXCEPT
      | Lex.Error =>
          IO.Put("Das ist keine Zahl!\n");
          (* Den Rest überspringen, sonst gibt es eine
             Endlosschleife. *)
          Lex.Skip(Stdio.stdin, Lex.NonBlanks);
      | FloatMode.Trap (typ) =>
          CASE typ OF
          | FloatMode.Flag.Underflow =>
              IO.Put("Zahl zu klein!\n");
          | FloatMode.Flag.Overflow,
              FloatMode.Flag.IntOverflow =>
              IO.Put("Zahl zu groß!\n");
          ELSE
            (*Sollten noch andere Fälle bei Lex.Int auftreten
               können?*)
            <* ASSERT FALSE *>
          END;
      END;
```

```
    EXCEPT
      (* Fehler von Lex.Skip auch noch abfangen *)
    | Rd.Failure (item) =>
        IO.Put("Folgende Fehler traten im Eingabestrom auf:\n");
        WHILE item # NIL DO
          IO.Put(Atom.ToText(item.head));
          item := item.tail;
        END;
    | Thread.Alerted =>
        IO.Put("Ich habe gar keinen Thread gestartet, "
                 & "der uns benachrichtigen könnte!\n");
    END;
  END;
  IO.Put(
    Fmt.F(
      "Nach unzähligen Versuchen ist es endlich gelungen, "
        & "die Zahl %s als Eingabe zu erhalten!\n", Fmt.Int(a)));
END Main.
```

Man sieht, dass wir durch die vorbildliche Fehlerbehandlung nun auch keine Compilerwarnungen mehr bekommen, die wir durch `<* FATAL ANY *>` unterdrücken müssten. . . . und die Übersicht ist dafür endgültig dahin. :-)
Besonders praktisch ist das Zusammenspiel von Ausnahmebehandlung und *Garbage Collector*: Eine Routine die Datenstrukturen aufbaut und dabei auf einen für sie unbehandelbaren Fehler trifft, kann sich darauf verlassen, dass die unvollständig aufgebauten Datenstrukturen automatisch und rückstandslos entsorgt werden.

3.12.2 `TRY-FINALLY`-Konstrukt

Oft möchte man eine Ausnahme gar nicht behandeln, sondern nur sicherstellen, dass bestimmte Dinge im Normal- wie im Ausnahmefall wieder in Ordnung gebracht werden. Beispielsweise sollte in jedem Fall eine Datei irgendwann geschlossen werden, wenn sie erfolgreich geöffnet wurde. (Siehe Abschnitt 3.11) Die Schöpfer der Standardbibliothek haben das Schließen von Dateien wohlweislich nicht dem *Garbage Collector* übertragen, denn dieser gibt Resourcen frei, wann er es für richtig hält. Wenn man aber zum Beispiel eine Datei nach dem Schreiben schließen und gleich wieder öffnen will, kann man darauf nicht warten.
Man braucht also ein Konstrukt, dass solchen Aufräumaktionen Platz bietet *ohne* die Ausnahme „zu schlucken“. Genau das bewirkt

```
TRY
  A
FINALLY
  B
END;
```

- es führt A aus und wenn das gelingt, auch B. Tritt in A eine Ausnahme auf, wird B ausgeführt und die Ausnahme erneut verhängt. Tritt dagegen in B eine Ausnahme auf, wird B abgebrochen und diese Ausnahme statt der bisher behandelten weitergegeben.

3.12.3 Ausnahmezustände selbst ausrufen

Wir haben gesehen, wie wir Fehler behandeln können, die von Routinen aus den Standardbibliotheken ausgelöst werden. Wir können aber auch selbst Ausnahmen definieren und auslösen, um Fehlersituationen anzuzeigen oder um aus einem anderen Grunde eine Operation abzubrechen.
Definiert werden Ausnahmen so:

```
EXCEPTION
  Error1;
  Error2 (TEXT);
```

Dabei ist Error1 eine Ausnahme ohne Argument und Error2 eine Ausnahme, die einen zusätzlichen Wert vom Typ TEXT zurückgibt.
Um den Überblick über mögliche Ausnahmen zu behalten, muss jede Prozedur ausweisen, welche Ausnahmen bei ihrer Ausführung auftreten können:

```
PROCEDURE F (x: LONGREAL; ): LONGREAL
  RAISES {FloatMode.Trap, NoConvergence}
```

zeigt an, dass F die Ausnahmen FloatMode.Trap und NoConvergence auslösen kann. Dagegen bedeutet

```
PROCEDURE F (x: LONGREAL; ): LONGREAL RAISES ANY
```

, dass die Prozedur prinzipiell jede Ausnahme auslösen kann. Damit sollte man sehr sparsam umgehen. Eigentlich wird es nur gebraucht, wenn man Funktionen als Eingaben erwartet, deren Ausnahmeliste man nicht einschränken möchte:

```
PROCEDURE F (f: PROCEDURE (x: LONGREAL; ): LONGREAL RAISES ANY;
             x: LONGREAL; ): LONGREAL RAISES ANY =
  BEGIN
    RETURN f(x);  (* Kann irgendeine Ausnahme auslösen *)
  END F;
```

```
MODULE Main;
(* $Id: Newton.m3,v 1.3 2004/01/16 18:15:09 thielema Exp $ *)
(* Newton-Verfahren: Zeiger auf Funktionen, Ausnahmebehandlung *)

IMPORT IO, Fmt, Lex, Stdio;
IMPORT Math;


EXCEPTION NoConvergence(LONGREAL); (* contains remaining error *)

PROCEDURE Newton (f, df  : PROCEDURE (x: LONGREAL; ): LONGREAL;
                  x, tol : LONGREAL;
                  maxIter: CARDINAL;                              ):
  LONGREAL RAISES {NoConvergence} =
  VAR fx: LONGREAL;
  BEGIN
    FOR n := 0 TO maxIter DO
      fx := f(x);
      IF ABS(fx) <= tol THEN RETURN x; END;
      x := x - fx / df(x);
    END;
    RAISE NoConvergence(fx);
  END Newton;

VAR y, tol: LONGREAL;

PROCEDURE F (x: LONGREAL; ): LONGREAL =
  BEGIN
    RETURN x + Math.log(x) - y;
  END F;

PROCEDURE DF (x: LONGREAL; ): LONGREAL =
  BEGIN
    RETURN 1.0D0 + 1.0D0 / x;
  END DF;

CONST maxIter = 20;

<* FATAL ANY *>
```

```
BEGIN
  IO.Put("Ich finde für jedes reelle y ein x, "
         & "so dass x + ln x = y gilt.\n"
         & "Jetzt brauche ich nur noch ein y: ");
  y := Lex.LongReal(Stdio.stdin);
  IO.Put("und den maximal erlaubten Fehler: ");
  tol := Lex.LongReal(Stdio.stdin);

  TRY
    IO.Put(
      Fmt.F("x = %s\n",
            Fmt.LongReal(Newton(F, DF, 1.0D0, tol, maxIter))));
  EXCEPT
  | NoConvergence (err) =>
      IO.Put(
        Fmt.F(
          "Auch nach %s Iterationen betrug der Fehler noch %s.\n",
          Fmt.Int(maxIter), Fmt.LongReal(err)));
  END;
END Main.
```

3.13 Module

Werden Programme größer, (und das werden sie nach kurzer Entwicklungszeit immer) reicht ein einfaches Hauptprogramm mit mehreren Prozeduren nicht mehr aus und man muss das Programm in mehrere *Module* aufteilen.
Ebenso kann es sein, dass man sehr allgemein gehaltene Prozeduren aus dem Hauptmodul auslagern möchte. Auch hier eignen sich eigenständige Module, um Prozeduren breiter zugänglich zu machen als nur innerhalb des Hauptmoduls.
Auch bei einfachsten Programmen machen wir schon von der Modularisierung Gebrauch, in dem wir Prozeduren aus den Modulen der Standardbibliothek verwenden. Nicht auszudenken, wenn alle beteiligten Prozeduren im Hauptmodul stehen müssten.
Modularisierung ist einer der zentralen Aspekte von `Modula-3` und daher auch der Ursprung des Namens dieser Sprache.

3.13.1 Einbinden von Modulen

Bis jetzt haben wir nur die einfachste, aber auch wichtigste Form des *Importieren*s verwendet:

```
IMPORT IO;
```

Bindet das Modul `IO` ein. Möchte man Objekte aus diesem Modul erreichen, muss man dem Namen des Objektes immer den Modulnamen voranstellen: `IO.Put` bezeichnet die Prozedur `Put` aus dem Modul `IO`. Diese Art genauere Bezeichnung von Prozedurnamen heißt *Qualifizieren*.
Dieses qualifizierte Importieren wirft keine Konflikte auf und benötigt von daher keine Verbesserungen. Allerdings gibt es noch manche Schreibvereinfachung:
Möglicherweise möchte man den Namen eines importierten Modules lieber für ein anderes Objekt vergeben, oder aber ein Modulname ist schlicht zu lang für die häufige Verwendung. Dann kann man das importierte Modul innerhalb des importierenden Moduls umbenennen:

```
IMPORT Random AS Rnd;
```

Statt `Random.T` muss man nun `Rnd.T` schreiben.
Jetzt kann es immer noch sein, dass man manche Variablen oder Prozeduren aus einem Modul so häufig verwenden möchte, dass dafür selbst die kürzeste Qualifizierung noch zu lang ist. Mit

```
FROM IO IMPORT Put;
```

importiert man nur die Prozedur `Put`. Sie kann nur mit `Put` aufgerufen werden, nicht aber mit `IO.Put`. Möchte man sich beide Varianten offenhalten, kann man auch `FROM IO IMPORT Put;` und `IMPORT IO;` zusammen verwenden.

3.13.2 Module selbst erstellen

Jedes Modul besteht aus zwei Dateien:

- Die Schnittstelle besteht aus allem, was man wissen muss, um das Modul verwenden zu können. Namen von Schnittstellendateien enden auf `.i3`. Der Programmtext beginnt mit „`INTERFACE` Modulname`;`“.
- Die eigentlichen Unterprogramme stehen in den Implementationsdateien, erkennbar an der Namensendung `.m3`. Der Programmtext beginnt mit „`MODULE` Modulname`;`“.

Alles was man in Schnittstellendateien schreibt, kann von anderen Modulen importiert, also benutzt werden. In Implementationsdateien kann man zum einen Typen genauer spezifizieren, die in der Schnittstelle nur umrissen wurden, man kann Prozeduren programmieren, deren Aufrufkonventionen in der Schnittstellendatei festgelegt wurde, man kann direkt auf globale Variablen der Schnittstelle zugreifen, man kann aber auch Objekte einführen, die nur in diesem Modul genutzt werden können.
Beispiel:
Datei `Meins.i3`

```
INTERFACE Meins;

PROCEDURE Supi (a: CARDINAL; ): TEXT;

VAR
  deins: INTEGER;

END Meins.
```

Datei `Meins.m3`

```
MODULE Meins;

IMPORT Fmt;

PROCEDURE Supi (a: CARDINAL; ): TEXT =
  BEGIN
    deins := a;
    RETURN Fmt.Int(a);
  END Supi;

END Meins.
```

Da verschiedene Module streng getrennt sind, kann der Übersetzer sie einzeln übersetzen. Bei einem großen Projekt nur diejenigen Module übersetzen zu müssen, die sich geändert haben, spart viel Zeit. Um ein Modul übersetzen zu können, müssen lediglich die Schnittstellen der importierten Module übersetzt vorliegen.

3.13.3 Pakete verwalten

Hat man ein Programm in mehrere Module zerlegt, ist für den Übersetzer noch nicht klar, wie alles wieder zusammengehört. Deswegen muss man die Module in einer

Struktur organisieren, die außerhalb der `Modula-3`-Programmtexte liegt. Dies sind die *Pakete*. Ein Paket enthält entweder genau ein ausführbares Programm, genau eine Funktionsbibliothek (auch aus mehreren Modulen aufgebaut) oder eine Ansammlung sonstiger Dateien.
Ein Paket ist wie folgt strukturiert:

`Paket`	Hauptverzeichnis des Paketes
`  src`	Programmtexte, Steuerdateien
`    Help.i3`	Beispiel: Schnittstelle für eigene Funktionen
`    Help.m3`	Beispiel: Implementation eigener Funktionen
`    Main.m3`	Hauptmodul
`    m3makefile`	*Steuerdatei*
`  LINUXLIBC6`	vom Compiler für Linux erzeugte Dateien
`  SOLgnu`	vom Compiler für SUNs Solaris erzeugte Dateien
`  NT386`	vom Compiler für Windows erzeugte Dateien

Die Steuerdatei `m3makefile` enthält Informationen darüber, welche anderen Pakete benötigt werden, welche Dateien im Paket welche Funktionen erfüllen, wie das ausführbare Programm heißen soll, usw. Aus diesen Informationen ermittelt der Übersetzer, welche Module von welchen abhängen und in welcher Reihenfolge welche Module übersetzt werden müssen. Wenn Beispielsweise Modul `A` Modul `B` importiert, und sich die Schnittstelle von `B` ändert, so muss auch `A` erneut übersetzt werden. Insbesondere behält sich der Compiler vor, überhaupt nichts zu tun, sofern kein Modul geändert wurde.
Die Steuerdatei `m3makefile` könnte in unserem Beispiel so aussehen:

`import("libm3")`	Einbinden der `Modula-3`-Standardbibliothek
`module("Help")`	Verwende `Help.i3` und `Help.m3`
`implementation("Main")`	Verwende nur `Main.m3`
`program("test")`	Das fertige Programm soll `test` heißen.

Eine Besonderheit des Compilers `cm3` ist, dass er zur Not auch ohne Steuerdatei auskommt. Bei einfachen Programmen funktioniert das ganz gut, bei größeren Projekten sollte man besser eine Steuerdatei von Hand erstellen.

3.14 Objekte

Objektorientiertes Programmieren ist eine Programmiertechnik, mit der sich ähnliche Arten von Objekten modellieren lassen, so dass man einerseits nicht einen Datentyp für alle Arten verwenden muss und doch von den Gemeinsamkeiten der Objekte profitieren kann.
Objektorientiertes Programmieren hat eine echte Begeisterungswelle losgetreten, die

dazu führte, dass selbst in die ältesten Programmiersprachen noch Objektdatentypen und dazugehörige Mechanismen eingebaut wurden.
Die großen Vorteile spielt das objektorientierte Programmieren vor allem bei großen Projekten aus. Daher ist es aber auch schwer, kleine und doch überzeugende Demonstrationsbeispiele zu finden.
Prinzipiell käme man durch Verwendung von Funktionenvariablen (Abschnitt 3.10.4) auch ohne neue Sprachelemente aus, allerdings sorgt eine direkte Unterstützung der Sprache für mehr Sicherheit.

3.14.1 Klassen

Objekte unterteilt man in *Klassen*, zum Beispiel könnte man die Klasse der geometrischen Figuren definieren. Weiter gibt es *Unterklassen*: Dreiecke bilden eine Unterklasse der geometrischen Figuren. Ein Dreieck ist eine spezielle Figur. Alles, was man mit Figuren machen kann (verschieben, rotieren, darstellen), kann man auch mit Dreiecken machen. Das, was man mit Objekten machen kann, nennt man *Methoden*. Setzt man Klassen mit Hilfe der Unterklassenrelation zueinander in Beziehung entsteht eine *Klassenhierarchie*.
Ein Dreieck ist in zweierlei Hinsicht etwas besonderes verglichen mit einer allgemeinen Figur:

1. Was die Methode „Verschieben“ für eine Figur bedeutet, kann man eigentlich nur erklären, wenn man einen konkreten Figurtyp wie das Dreieck betrachtet. Für die allgemeine Figurklasse definiert man lediglich, was zum „Verschieben“ benötigt wird (die *Schnittstelle*, etwa: Eingabe Objekt und Verschiebungsvektor, Ausgabe Objekt), aber man implementiert sie nicht. Die „geometrische Figur“ ist deshalb eine *abstrakte Klasse*. Die Verschiebung selbst programmiert man nur für geeignete Unterklassen von „Figur“, zum Beispiel für Dreiecke.

2. Ein Dreieck bringt neue Fähigkeiten mit, die sich für allgemeine geometrische Figuren nicht einmal definieren lassen. Beispielsweise kann man für ein Dreieck die Innenwinkel berechnen. Für einen Kreis hat eine solche Berechnung keinen Sinn und damit erst recht nicht für die gemeinsame Überklasse „Figur“.

3.14.2 Arbeiten mit Objekten

In der Standardbibliothek werden zahlreiche Dienste über Objekte zur Verfügung gestellt, beispielsweise Zufallszahlengeneratoren (siehe Abschnitt B.15). Ein *Zufallsgenerator* ist ein Objekt das auf Anfrage eine zufällige Zahl generiert. In Wirklich-

keit handelt es sich allerdings um eine Zahl die nach einem festen Schema berechnet wurde, dessen Ergebnisse aber irgendwie willkürlich aussehen.
Das Modul `Random` stellt die *abstrakte Klasse* `Random.T` zur Verfügung. In dieser wird festgelegt, was Zufallsgeneratoren leisten müssen:

```
TYPE
  T = OBJECT METHODS
    integer(min := FIRST(INTEGER);
      max := LAST(INTEGER)): INTEGER;
    real(min := 0.0e+0; max := 1.0e+0): REAL;
    longreal(min := 0.0d+0; max := 1.0d+0): LONGREAL;
    extended(min := 0.0x+0; max := 1.0x+0): EXTENDED;
    boolean(): BOOLEAN
  END;
```

Ein Objekt von einer Unterklasse von `Random.T` muss also in der Lage sein, Zufallszahlen verschiedener Arten zu erzeugen: ganze Zahlen, Fließkommazahlen, Wahrheitswerte. Für `Random.T` ist keine dieser Methoden implementiert, es hat also keinen Sinn, ein Objekt vom Typ `Random.T` zu erzeugen.
Zusätzlich gibt es eine Unterklasse `Random.Default`, welche alle geforderten Methoden implementiert.

```
TYPE
  Default <: T OBJECT METHODS
    init(fixed := FALSE): Default
  END;
```

Der Text `T OBJECT` bedeutet, dass `Default` eine Unterklasse von `T` ist. Details über die Implementation verrät uns `Random.Default` in dieser Definition nicht. Zum Beispiel muss dieser Zufallsgenerator irgendeinen Zustand speichern, der sich bei jeder Zufallszahlenabfrage ändert und der sicherstellt, dass nicht die gleiche Zahl immer wieder ausgegeben wird. Von diesem Zustand sehen wir in der Definition nichts. Dass es da noch etwas gibt, das wir nicht zu wissen brauchen, deutet die *Untertyp-Relation* `<:` an.
Wozu der ganze Aufwand mit Objekten, warum selbst für einfache Zufallsgeneratoren? Der Grund ist, dass man Zufallsgeneratoren auf vielerlei Weise programmieren kann, aber allen Zufallsgeneratoren ist die Schnittstelle gemein. Es ist mit Objekten leicht möglich, Zufallsgeneratoren auszuwechseln und sogar dieselben Routinen mit ganz verschiedenen Zufallsgeneratoren auszuprobieren. Das ist sinnvoll, denn Zufallsgeneratoren liefern, wie gesagt, nicht wirklich zufällige Zahlen. Deswegen kann es sein, dass ein bestimmter Generator für eine Anwendung nicht „zufällig“ genug

ist. In diesem Falle kann man andere Unterklassen von `Random.T` verwenden (oder erst schreiben) welche für die gewünschte Anwendung besser geeignet sind.
Man erzeugt Objekte mit `NEW` ganz analog zu Datenverbünden (Abschnitt 3.10.2). Beachte: `OBJECT`-Typen sind immer Verweise auf Objekte und von daher mit `REF RECORD` zu vergleichen und nicht mit `RECORD`.
Obwohl man Elemente von Objekten so wie bei Datenverbünden direkt beim Aufruf von `NEW` initialisieren kann, sollte man lieber von der `init`-Methode Gebrauch machen. Es ist Konvention, eine Objektklasse immer mit einer `init`-Methode auszustatten, welche das Objekt in einen wohldefinierten Ausgangszustand bringt, oder es in einen solchen zurückversetzt, falls es schon benutzt wurde.

3.14.3 Deklaration von Klassen

So programmierte man vor der Objektorientierung:

```
MODULE Main;
(* $Id: GeometrieRecord.m3,v 1.2 2004/02/29 22:39:17 thielema Exp
   $ *)

(* So würde man geometrische Objekte ohne Modula-3-Objekte
   modellieren. *)

IMPORT Fmt, IO, Math;

TYPE
  Figurtyp = {kreis, rechteck, dreieck};

  Punkt = RECORD x, y: LONGREAL;  END;

  Figur = RECORD
            typ: Figurtyp;
            p0: Punkt;          (*bei Kreis: Mittelpunkt, bei
                                   Dreieck und Rechteck ein
                                   Eckpunkt*)
            p1: Punkt;          (*Rechteck: gegenueberliegender
                                   Punkt, Dreieck: zweiter
                                   Eckpunkt*)
            p2: Punkt;          (*nur Dreieck: dritter
                                   Eckpunkt*)
            radius: LONGREAL;   (*nur Kreis*)
          END;

PROCEDURE Flaeche (f: REF Figur; ): LONGREAL =
  BEGIN
    CASE f.typ OF
    | Figurtyp.kreis =>
```

```
        RETURN f.radius * f.radius * FLOAT(Math.Pi, LONGREAL);
    | Figurtyp.rechteck =>
        RETURN (f.p0.x - f.p1.x) * (f.p0.y - f.p1.y);
    | Figurtyp.dreieck =>
        RETURN
          f.p0.x * (f.p1.y - f.p2.y) + f.p1.x * (f.p2.y - f.p0.y)
            + f.p2.x * (f.p0.y - f.p1.y);
    END;
  END Flaeche;

PROCEDURE Schwerpunkt (f: REF Figur; ): Punkt =
  BEGIN
    CASE f.typ OF
    | Figurtyp.kreis => RETURN f.p0;
    | Figurtyp.rechteck =>
        RETURN Punkt{(f.p0.x + f.p1.x) / 2.0D0,
                     (f.p0.y + f.p1.y) / 2.0D0};
    | Figurtyp.dreieck =>
        RETURN Punkt{(f.p0.x + f.p1.x + f.p2.x) / 3.0D0,
                     (f.p0.y + f.p1.y + f.p2.y) / 3.0D0};
    END;
  END Schwerpunkt;

BEGIN
  WITH s = Schwerpunkt(NEW(REF Figur, typ := Figurtyp.rechteck,
                           p0 := Punkt{1.0D0, 0.5D0},
                           p1 := Punkt{1.0D0, 5.0D0})) DO
    IO.Put(Fmt.F("Schwerpunkt von Rechteck: (%s, %s)\n",
                 Fmt.LongReal(s.x), Fmt.LongReal(s.y)));
  END;

  IO.Put(
    Fmt.F(
      "Fläche von Kreis: %s\n",
      Fmt.LongReal(
        Flaeche(
          NEW(REF Figur,
              typ := Figurtyp.dreieck, (*Falsch!  Sollte
                                         Figurtyp.kreis sein,
                                         aber der Übersetzer
                                         hat's nicht gemerkt!*)
              p0 := Punkt{1.0D0, 0.5D0}, radius := 2.0D0)))));
END Main.
```

So programmiert man mit Objekten:

```
MODULE Main;
(* $Id: GeometrieObjekt.m3,v 1.2 2004/02/29 22:39:17 thielema Exp
   $ *)
```

```
(* So kann man geometrische Objekte elegant mit Modula-3-Objekten
   modellieren. *)

IMPORT Fmt, IO, Math;

TYPE
  Punkt = RECORD x, y: LONGREAL;  END;

  Figur = OBJECT
          METHODS
            flaeche     (): LONGREAL;
            schwerpunkt (): Punkt;
          END;

TYPE
  Kreis = Figur OBJECT
            mitte : Punkt;
            radius: LONGREAL;
          OVERRIDES
            flaeche     := KreisFlaeche;
            schwerpunkt := KreisSchwerpunkt;
          END;

PROCEDURE KreisFlaeche (f: Kreis; ): LONGREAL =
  BEGIN
    RETURN f.radius * f.radius * FLOAT(Math.Pi, LONGREAL);
  END KreisFlaeche;

PROCEDURE KreisSchwerpunkt (f: Kreis; ): Punkt =
  BEGIN
    RETURN f.mitte;
  END KreisSchwerpunkt;


(*isoorientiertes Rechteck*)
TYPE
  Rechteck = Figur OBJECT
               p0, p1: Punkt;
             OVERRIDES
               flaeche     := RechteckFlaeche;
               schwerpunkt := RechteckSchwerpunkt;
             END;

PROCEDURE RechteckFlaeche (f: Rechteck; ): LONGREAL =
  BEGIN
    RETURN (f.p0.x - f.p1.x) * (f.p0.y - f.p1.y);
  END RechteckFlaeche;
```

```
PROCEDURE RechteckSchwerpunkt (f: Rechteck; ): Punkt =
  BEGIN
    RETURN Punkt{(f.p0.x + f.p1.x) / 2.0D0,
                 (f.p0.y + f.p1.y) / 2.0D0};
  END RechteckSchwerpunkt;


TYPE
  Dreieck = Figur OBJECT
              p0, p1, p2: Punkt;
            OVERRIDES
              flaeche     := DreieckFlaeche;
              schwerpunkt := DreieckSchwerpunkt;
            END;

PROCEDURE DreieckFlaeche (f: Dreieck; ): LONGREAL =
  BEGIN
    RETURN
      f.p0.x * (f.p1.y - f.p2.y) + f.p1.x * (f.p2.y - f.p0.y)
        + f.p2.x * (f.p0.y - f.p1.y);
  END DreieckFlaeche;

PROCEDURE DreieckSchwerpunkt (f: Dreieck; ): Punkt =
  BEGIN
    RETURN Punkt{(f.p0.x + f.p1.x + f.p2.x) / 3.0D0,
                 (f.p0.y + f.p1.y + f.p2.y) / 3.0D0};
  END DreieckSchwerpunkt;

VAR fig: Figur;

BEGIN
  fig := NEW(Rechteck, p0 := Punkt{1.0D0, 0.5D0},
             p1 := Punkt{1.0D0, 5.0D0});
  WITH s = fig.schwerpunkt() DO
    IO.Put(Fmt.F("Schwerpunkt von Rechteck: (%s, %s)\n",
                 Fmt.LongReal(s.x), Fmt.LongReal(s.y)));
  END;

  fig :=
    NEW(Kreis, mitte := Punkt{1.0D0, 0.5D0}, radius := 2.0D0);
  IO.Put(
    Fmt.F("Fläche von Kreis: %s\n", Fmt.LongReal(fig.flaeche())));
END Main.
```

Anhang A

Übersichten

A.1 Vorrangregeln

Die Operatoren besitzen folgende absteigend sortierte Bindungen:

x.a	Infix Punkt
f(x) a[i] T{x}	Argumentlisten beginnend mit (, [, {
p^	Postfix ^
+ -	Präfix-Arithmetik
* / DIV MOD	Infix-Arithmetik
+ - &	Infix-Arithmetik
= # < <= >= > IN	Infix-Relationen
NOT	Präfix NOT
AND	Infix AND
OR	Infix OR

Eine Operation a hat *Vorrang* vor b, oder auch: a bindet stärker als b, wenn X b Y a Z das gleiche bedeutet wie X b (Y a Z).
Alle Infix-Operatoren gleicher *Bindung* werden von links nach rechts ausgewertet (links-assoziativ). Mit Klammern kann man die Vorrangregeln umgehen. Hier ein paar Beispiele von ungeklammerten Ausdrücken und ihren Bedeutungen ausgedrückt durch vollständige Klammerung.

`M.F(x)`	`(M.F)(x)`	Elementzugriff vor Funktionsauswertung
`Q(x)^`	`(Q(x))^`	Funktionsauswertung vor Dereferenzierung `^`
`- p^`	`- (p^)`	Dereferenzierung `^` vor Negation `-`
`- a * b`	`(- a) * b`	Negation `-` vor Multiplikation `*`
`a * b - c`	`(a * b) - c`	Multiplikation `*` vor Subtraktion `-`
`x IN s - t`	`x IN (s - t)`	Subtraktion `-` vor Enthaltensein `IN`
`NOT x IN s`	`NOT (x IN s)`	Enthaltensein `IN` vor logischer Negation `NOT`
`NOT p AND q`	`(NOT p) AND q`	logische Negation `NOT` vor Konjunktion `AND`
`A OR B AND C`	`A OR (B AND C)`	Konjunktion `AND` vor Disjunktion `OR`

Anhang B

Wichtige Module der Standardbibliotheken m3core und libm3

B.1 libm3: Fmt

Pfad: libm3/src/fmtlex/Fmt.i3

```
(* Copyright (C) 1994, Digital Equipment Corporation           *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)
(* Last modified on Thu Jan  5 13:51:02 PST 1995 by detlefs    *)
(*      modified on Tue Mar 15 12:56:39 PST 1994 by heydon     *)
(*      modified on Fri Feb 18 13:12:30 PST 1994 by kalsow     *)
(*      modified on Tue Nov  9 08:37:38 PST 1993 by mcjones    *)
(*      modified on Thu Apr 29 16:32:36 PDT 1993 by muller     *)
(*      modified on Mon Feb 15 15:18:41 PST 1993 by ramshaw    *)

(* The "Fmt" interface provides procedures for formatting numbers and
   other data as text.
   \index{writing formatted data}
   \index{formatted data!writing}
*)

INTERFACE Fmt;

IMPORT Word, Real AS R, LongReal AS LR, Extended AS ER;

PROCEDURE Bool(b: BOOLEAN): TEXT;
(* Format "b" as {\tt \char'42TRUE\char'42} or  {\tt \char'42FALSE\char'42}. *)

PROCEDURE Char(c: CHAR): TEXT;
(* Return a text containing the character "c". *)

TYPE Base = [2..16];

PROCEDURE Int(n: INTEGER; base: Base := 10): TEXT;
```

```
PROCEDURE Unsigned(n: Word.T; base: Base := 16): TEXT;
(* Format the signed or unsigned number "n" in the specified base. *)

(* The value returned by "Int" or "Unsigned" never contains upper-case
   letters, and it never starts with an explicit base and underscore.
   For example, to render an unsigned number "N" in hexadecimal as a
   legal Modula-3 literal, you must write something like:
| "16_" & Fmt.Unsigned(N, 16)
*)

TYPE Style = {Sci, Fix, Auto};

PROCEDURE Real(
    x: REAL;
    style := Style.Auto;
    prec: CARDINAL := R.MaxSignifDigits - 1;
    literal := FALSE)
  : TEXT;
PROCEDURE LongReal(
    x: LONGREAL;
    style := Style.Auto;
    prec: CARDINAL := LR.MaxSignifDigits - 1;
    literal := FALSE)
  : TEXT;
PROCEDURE Extended(
    x: EXTENDED;
    style := Style.Auto;
    prec: CARDINAL := ER.MaxSignifDigits - 1;
    literal := FALSE)
  : TEXT;
(* Format the floating-point number "x". *)

(*

\paragraph*{Overview.}

   "Style.Sci" gives scientific notation with fields padded to fixed
   widths, suitable for making a table.  The parameter "prec"
   specifies the number of digits after the decimal point---that is,
   the relative precision.
   \index{scientific notation}

   "Style.Fix" gives fixed point, with "prec" once again specifying
   the number of digits after the decimal point---in this case, the
   absolute precision.  The results of "Style.Fix" have varying
   widths, but they will form a table if they are right-aligned (using
   "Fmt.Pad") in a sufficiently wide field.
   \index{fixed-point notation}
```

```
"Style.Auto" is not intended for tables.  It gives scientific
notation with at most "prec" digits after the decimal point for
numbers that are very big or very small.  There may be fewer than
"prec" digits after the decimal point because trailing zeros are
suppressed.  For numbers that are neither too big nor too small, it
formats the same significant digits---at most "prec+1" of them---in
fixed point, for greater legibility.

All styles omit the decimal point unless it is followed by at least
one digit.

Setting "literal" to "TRUE" alters all styles as necessary to make
the result a legal Modula-3 literal of the appropriate type.


\paragraph*{Accuracy.}

As discussed in the "Float" interface, the call "ToDecimal(x)"
converts "x" to a floating-decimal number with automatic precision
control~\cite{Steele,Gay}: Just enough digits are retained to
distinguish "x" from other values of type "T", which implies that
at most "T.MaxSignifDigits" are retained.  The "Real", "LongReal",
and "Extended" procedures format those digits as an appropriate
string of characters.  If the precision requested by "prec" is
higher than the automatic precision provided by "ToDecimal(x)",
they append trailing zeros.  If the precision requested by "prec"
is lower, they round "ToDecimal(x)" as necessary, obeying the
current rounding mode.  Because they exploit the "errorSign" field
of the record "ToDecimal(x)" in doing this rounding, they get the
same result that rounding "x" itself would give.

As a consequence, setting "prec" higher than "T.MaxSignifDigits-1"
in "Style.Sci" isn't very useful: The trailing digits of all of the
resulting numbers will be zero.  Setting "prec" higher than
"T.MaxSignifDigits-1" in "Style.Auto" actually has no effect at
all, since trailing zeros are suppressed.


\paragraph*{Details.}

We restrict ourselves at first to those cases where "Class(x)" is
either "Normal" or "Denormal".

In those cases, "Style.Sci" returns: a minus sign or blank, the
leading nonzero digit of "x", a decimal point, "prec" more digits
of "x", a character "'e'", a minus sign or plus sign, and
"T.MaxExpDigits" of exponent (with leading zeros as necessary).
```

```
When "prec" is zero, the decimal point is omitted.

"Style.Fix" returns: a minus sign if necessary, one or more digits,
a decimal point, and "prec" more digits---never any blanks.  When
"prec" is zero, the decimal point is omitted.

"Style.Auto" first formats "x" as in "Style.Sci", using scientific
notation with "prec" digits after the decimal point.  Call this
intermediate result "R".

If the exponent of "R" is at least "6" in magnitude, "Style.Auto"
leaves "R" in scientific notation, but condenses it by omitting all
blanks, plus signs, trailing zero digits, and leading zeros in the
exponent.  If this leaves no digits after the decimal point, the
decimal point itself is omitted.

If the exponent of "R" is at most "5" in magnitude, "Style.Auto"
reformats the digits of "R" in fixed point, first deleting any
trailing zeros and then adding leading or trailing zeros as
necessary to bridge the gap from the digits of "R" to the unit's
place.

For example, assuming the current rounding mode is "NearestElseEven":

| Fmt.Real(1.287e6,  Style.Auto, prec := 2) = "1.29e6"
| Fmt.Real(1.297e6,  Style.Auto, prec := 2) = "1.3e6"
| Fmt.Real(1.297e5,  Style.Auto, prec := 2) = "130000"
| Fmt.Real(1.297e-5, Style.Auto, prec := 2) = "0.000013"
| Fmt.Real(1.297e-6, Style.Auto, prec := 2) = "1.3e-6"
| Fmt.Real(9.997e5,  Style.Auto, prec := 2) = "1e6"
| Fmt.Real(9.997e-6, Style.Auto, prec := 2) = "0.00001"

"Style.Sci" handles zero by replacing the entire exponent field by
blanks, for example: {\tt \char'042\char'040 0.00\char'040
\char'040 \char'040 \char'040 \char'042}.  "Style.Fix" renders zero
with all digits zero; for example, "\char'042 0.00\char'042".
"Style.Auto" renders zero as "\char'042 0\char'042".  On IEEE
implementations, the value minus zero is rendered as a negative
number.

Also on IEEE implementations, "Style.Sci" formats infinities or
NaN's with a minus sign or blank, the string "\char'042
Infinity\char'042" or "\char'042 NaN\char'042", and enough trailing
blanks to get the correct overall width.  "Style.Fix" and
"Style.Auto" omit the blanks.  In "Style.Sci", if "\char'042
Infinity\char'042" doesn't fit, "\char'042 Inf\char'042" is used
instead.
```

```
   Setting "literal" to "TRUE" alters things as follows: Numbers that
   are rendered without a decimal point when "literal" is "FALSE" have
   a decimal point and one trailing zero appended to their digits.
   For the routines "Fmt.LongReal" and "Fmt.Extended", an exponent
   field of "d0" or "x0" is appended to numbers in fixed point and
   "'d'" or "'x'" is used, rather than "'e'", to introduce the
   exponents of numbers in scientific notation.  On IEEE
   implementations, the string "\char'042 Infinity\char'042" is
   replaced by "\char'042 1.0/0.0\char'042", "\char'042
   1.0d0/0.0d0\char'042", or "\char'042 1.0x0/0.0x0\char'042" as
   appropriate, and "\char'042 NaN\char'042" is similarly replaced by
   a representation of the quotient "0/0".  (Unfortunately, these
   quotient strings are so long that they may ruin the formatting of
   "Style.Sci" tables when "prec" is small and "literal" is "TRUE".)

*)

TYPE Align = {Left, Right};

PROCEDURE Pad(
    text: TEXT;
    length: CARDINAL;
    padChar: CHAR := ' ';
    align: Align := Align.Right): TEXT;
(* If "Text.Length(text) >= length", then "text" is returned
   unchanged.  Otherwise, "text" is padded with "padChar" until it has
   the given "length".  The text goes to the right or left, according
   to "align". *)

PROCEDURE F(fmt: TEXT; t1, t2, t3, t4, t5: TEXT := NIL)
  : TEXT;
(* Uses "fmt" as a format string. The result is a copy of "fmt" in
   which all format specifiers have been replaced, in order, by the
   text arguments "t1", "t2", etc. *)

(* A format specifier contains a field width, alignment and one of two
   padding characters. The procedure "F" evaluates the specifier and
   replaces it by the corresponding text argument padded as it would
   be by a call to "Pad" with the specified field width, padding
   character and alignment.

   The syntax of a format specifier is:
| %[-]{0-9}s
   that is, a percent character followed by an optional minus sign, an
   optional number and a compulsory terminating "s".

   If the minus sign is present the alignment is "Align.Left",
   otherwise it is "Align.Right". The alignment corresponds to the
```

```
   "align" argument to "Pad".

   The number specifies the field width (this corresponds to the
   "length" argument to "Pad"). If the number is omitted it defaults
   to zero.

   If the number is present and starts with the digit "0" the padding character
   is "'0'"; otherwise it is the space character. The padding character
   corresponds to the "padChar" argument to "Pad".

   It is a checked runtime error if "fmt" is "NIL" or the number of
   format specifiers in "fmt" is not equal to the number of non-nil
   arguments to "F".

   Non-nil arguments to "F" must precede any "NIL" arguments; it is a
   checked runtime error if they do not.

   If "t1" to "t5" are all "NIL" and "fmt" contains no format
   specifiers, the result is "fmt".

   Examples:
| F("%s %s\n", "Hello", "World") `returns` "Hello World\n".
| F("%s", Int(3))                 `returns` "3"
| F("%2s", Int(3))                `returns` " 3"
| F("%-2s", Int(3))               `returns` "3 "
| F("%02s", Int(3))               `returns` "03"
| F("%-02s", Int(3))              `returns` "30"
| F("%s", "%s")                   `returns` "%s"
| F("%s% tax", Int(3))            `returns` "3% tax"

   The following examples are legal but pointless:
| F("%-s", Int(3))                `returns` "3"
| F("%0s", Int(3))                `returns` "3"
| F("%-0s", Int(3))               `returns` "3"
*)

PROCEDURE FN(fmt: TEXT; READONLY texts: ARRAY OF TEXT)
  : TEXT;
(* Similar to "F" but accepts an array of text arguments. It is a
   checked runtime error if the number of format specifiers in "fmt"
   is not equal to "NUMBER(texts)" or if any element of "texts" is
   "NIL". If "NUMBER(texts) = 0" and "fmt" contains no format
   specifiers the result is "fmt". *)

(* Example:

| FN("%s %s %s %s %s %s %s",
|   ARRAY OF TEXT{"Too", "many", "arguments",
```

```
|     "for", "F", "to", "handle"})

   returns {\tt \char'42Too many arguments for F to handle\char'42}.
*)

END Fmt.

<*PRAGMA SPEC *>

<*SPEC Bool(b)                            ENSURES RES # NIL *>
<*SPEC Char(c)                            ENSURES RES # NIL *>
<*SPEC Int(n, base)                       ENSURES RES # NIL *>
<*SPEC Unsigned(n, base)                  ENSURES RES # NIL *>
<*SPEC Real(x, style, prec, literal)      ENSURES RES # NIL *>
<*SPEC LongReal(x, style, prec, literal) ENSURES RES # NIL *>
<*SPEC Extended(x, style, prec, literal) ENSURES RES # NIL *>
```

B.2 libm3: Lex

Pfad: `libm3/src/fmtlex/Lex.i3`

```
(* Copyright (C) 1994, Digital Equipment Corporation.         *)
(* All rights reserved.                                       *)
(* See the file COPYRIGHT for a full description.             *)
(* Last modified on Fri Feb 25 14:22:27 PST 1994 by kalsow    *)
(*      modified on Tue Nov 30 23:33:48 PST 1993 by heydon    *)
(*      modified on Wed Oct  6 09:09:31 PDT 1993 by mcjones   *)
(*      modified on Mon Feb 15 14:17:17 PST 1993 by ramshaw   *)
(*      modified on Sun Jun  3 14:25:23 PST 1991 by luca      *)

(* The "Lex" interface provides procedures for reading strings,
   booleans, integers, and floating-point numbers from an input
   stream.  Similar functionality on text strings is available
   from the "Scan" interface. *)

INTERFACE Lex;

IMPORT FloatMode, Rd, Word;
FROM Thread IMPORT Alerted;

EXCEPTION Error;

CONST
   Blanks = SET OF CHAR{
     ' ', '\t', '\n', '\r', '\013' (* vertical tab *), '\f'};
   NonBlanks = SET OF CHAR{'!' .. '~'};
```

```
(* Each of the procedures in this interface reads a specified prefix
   of the characters in the reader passed to the procedure, and leaves
   the reader positioned immediately after that prefix, perhaps at
   end-of-file.  Each procedure may call "Rd.UngetChar" after its
   final call on "Rd.GetChar". *)

PROCEDURE Scan(
    rd: Rd.T; READONLY cs: SET OF CHAR := NonBlanks): TEXT
  RAISES {Rd.Failure, Alerted};
(* Read the longest prefix of "rd" composed of characters in "cs" and
   return that prefix as a "TEXT".  *)

PROCEDURE Skip(
    rd: Rd.T; READONLY cs: SET OF CHAR := Blanks)
  RAISES {Rd.Failure, Alerted};
(* Read the longest prefix of "rd" composed of characters in "cs" and
   discard it.  *)

(* Whenever a specification of one of the procedures mentions skipping
   blanks, this is equivalent to performing the call "Skip(rd, Blanks)". *)

PROCEDURE Match(rd: Rd.T; t: TEXT)
  RAISES {Error, Rd.Failure, Alerted};
(* Read the longest prefix of "rd" that is also a prefix of "t".
   Raise "Error" if that prefix is not, in fact, equal to all of "t". *)

PROCEDURE Bool(rd: Rd.T): BOOLEAN RAISES {Error, Rd.Failure, Alerted};
(* Read a boolean from "rd" and return its value. *)

(* "Bool" skips blanks, then reads the longest prefix of "rd" that is
   a prefix of a "Boolean" in the following grammar:

| Boolean = "F" "A" "L" "S" "E" | "T" "R" "U" "E".

   The case of letters in a "Boolean" is not significant.  If the
   prefix read from "rd" is an entire "Boolean", "Bool" returns that
   boolean; else it raises "Error".  *)

PROCEDURE Int(rd: Rd.T; defaultBase: [2..16] := 10)
  : INTEGER RAISES {Error, FloatMode.Trap, Rd.Failure, Alerted};
PROCEDURE Unsigned(rd: Rd.T; defaultBase: [2..16] := 16)
  : Word.T RAISES {Error, FloatMode.Trap, Rd.Failure, Alerted};
(* Read a number from "rd" and return its value. *)

(* Each procedure skips blanks, then reads the longest prefix of "rd"
   that is a prefix of a "Number" as defined by the grammar below.  If
   "defaultBase" exceeds 10, then the procedure scans for a
   "BigBaseNum"; otherwise it scans for a "SmallBaseNum".  The effect
```

```
   of this rule is that the letters 'a' through 'f' and 'A' through
   'F' stop the scan unless either the "defaultBase" or the explicitly
   provided base exceeds 10.  "Unsigned" omits the scan for a "Sign".

| Number       = [Sign] (SmallBaseNum | BigBaseNum).
| SmallBaseNum = DecVal | BasedInt.
| BigBaseNum   = HexVal | BasedInt.
| BasedInt     = SmallBase "_" DecVal | BigBase "_" HexVal.
| DecVal       = Digit {Digit}.
| HexVal       = HexDigit {HexDigit}.
| Sign         = "+" | "-".
| SmallBase    = "2" | "3" | ... | "10".
| BigBase      = "11" | "12" | ... | "16".
| Digit        = "0" | "1" | ... | "9".
| HexDigit     = Digit | "A" | "B" | "C" | "D" | "E" | "F"
|                      | "a" | "b" | "c" | "d" | "e" | "f".

   If the prefix read from "rd" is an entire "Number" (as described
   above), the corresponding number is returned; else "Error" is
   raised.

   If an explicit base is given with an underscore, it is interpreted
   in decimal.  In this case, the digits in "DecVal" or "HexVal" are
   interpreted in the explicit base, else they are interpreted in the
   "defaultBase".

   Both procedures may raise "FloatMode.Trap(IntOverflow)".  They
   raise "Error" if some digit in the value part is not a legal digit
   in the chosen base. *)

PROCEDURE Real(rd: Rd.T): REAL
  RAISES {Error, FloatMode.Trap, Rd.Failure, Alerted};
PROCEDURE LongReal(rd: Rd.T): LONGREAL
  RAISES {Error, FloatMode.Trap, Rd.Failure, Alerted};
PROCEDURE Extended(rd: Rd.T): EXTENDED
  RAISES {Error, FloatMode.Trap, Rd.Failure, Alerted};
(* Read a real number from "rd" and return its value. *)

(* Each procedure skips blanks, then reads the longest prefix of "rd"
   that is a prefix of a floating-decimal number "Float" in the
   grammar:

| Float  = [Sign] FloVal [Exp].
| FloVal = {Digit} (Digit | Digit "." | "." Digit) {Digit}.
| Exp    = Marker [Sign] Digit {Digit}.
| Marker = ("E" | "e" | "D" | "d" | "X" | "x").

   where "Sign" and "Digit" are as defined above.  If the prefix read
```

```
   from "rd" is an entire "Float", that "Float" is converted to a
   "REAL", "LONGREAL", or "EXTENDED" using the routine "FromDecimal"
   in the appropriate instance of the "Float" generic interface; else
   "Error" is raised.  Note that the exponent of "Float" can be
   introduced with any of the six characters "'e'", "'E'", "'d'",
   "'D'", "'x'", or "'X'", independent of the target type of the
   conversion.

   On IEEE implementations, the syntax for "Float" is extended as
   follows:

| Float   = [Sign] FloVal [Exp] | [Sign] IEEEVal.
| IEEEVal = "I" "N" "F" "I" "N" "I" "T" "Y" | "I" "N" F"
|         | "N" "A" "N".

   The case of letters in an "IEEEVal" is not significant. The
   "FloatMode.Trap" exception may be raised with any of the arguments
   "Overflow", "Underflow", or "Inexact".

*)

END Lex.
```

B.3 libm3: Scan

Pfad: `libm3/src/fmtlex/Scan.i3`

```
(* Copyright (C) 1994, Digital Equipment Corporation.         *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)
(* Last modified on Fri Feb 25 14:22:24 PST 1994 by kalsow    *)
(*      modified on Tue Nov 30 23:33:48 PST 1993 by heydon    *)
(*      modified on Wed Oct  6 09:09:31 PDT 1993 by mcjones   *)
(*      modified on Mon Feb 15 14:17:17 PST 1993 by ramshaw   *)
(*      modified on Sun Jun  3 14:25:23 PST 1991 by luca      *)

(* The "Scan" interface provides procedures for reading strings,
   booleans, integers, and floating-point numbers from a text string.
   Similar functionality on readers is availble from the "Lex"
   interface. *)

INTERFACE Scan;

IMPORT Word, Lex, FloatMode;

(* Each of these procedures parses a string of characters and converts
   it to a binary value.  Leading and trailing blanks (ie. characters
   in "Lex.Blanks") are ignored.  "Lex.Error" is raised if the first
```

```
   non-blank substring is not generated by the corresponding "Lex"
   grammar or if there are zero or more than one non-blank substrings.
   "FloatMode.Trap" is raised as per "Lex". *)

PROCEDURE Bool(txt: TEXT): BOOLEAN
  RAISES {Lex.Error};

PROCEDURE Int(txt: TEXT; defaultBase: [2..16] := 10): INTEGER
  RAISES {Lex.Error, FloatMode.Trap};
PROCEDURE Unsigned(txt: TEXT; defaultBase: [2..16] := 16): Word.T
  RAISES {Lex.Error, FloatMode.Trap};

PROCEDURE Real(txt: TEXT): REAL
  RAISES {Lex.Error, FloatMode.Trap};
PROCEDURE LongReal(txt: TEXT): LONGREAL
  RAISES {Lex.Error, FloatMode.Trap};
PROCEDURE Extended(txt: TEXT): EXTENDED
  RAISES {Lex.Error, FloatMode.Trap};

END Scan.
```

B.4 m3core: Text

Pfad: m3core/src/text/Text.i3

```
(* Copyright (C) 1994, Digital Equipment Corporation.         *)
(* All rights reserved.                                       *)
(* See the file COPYRIGHT for a full description.             *)
(* Last modified on Fri Aug 11 12:50:04 PDT 1995 by detlefs   *)
(*      modified on Fri Sep 23 09:31:45 PDT 1994 by heydon    *)
(*      modified on Fri Mar 25 12:03:15 PST 1994 by kalsow    *)
(*      modified on Wed Nov  3 14:09:28 PST 1993 by mcjones   *)
(*      modified on Wed Oct  7 11:49:?? PST 1992 by muller    *)

(* A non-nil "TEXT" represents an immutable, zero-based sequence of
   characters.  "NIL" does not represent any sequence of characters,
   it will not be returned from any procedure in this interface, and
   it is a checked runtime error to pass "NIL" to any procedure in
   this interface.  *)

INTERFACE Text;

IMPORT TextClass, Word;

TYPE T = TEXT;

CONST Brand = TextClass.Brand;
```

```
PROCEDURE Length (t: T): CARDINAL;
(* Return the number of characters in "t". *)

PROCEDURE Empty (t: T): BOOLEAN;
(* Equivalent to "Length(t) = 0". *)

PROCEDURE Equal (t, u: T): BOOLEAN;
(* Return "TRUE" if "t" and "u" have the same length and
   (case-sensitive) contents. *)

PROCEDURE Compare (t1, t2: T): [-1..1];
(* Return -1 if "t1" occurs before "t2", 0 if "Equal(t1, t2)", +1 if
   "t1" occurs after "t2" in lexicographic order. *)

PROCEDURE Cat (t, u: T): T;
(* Return the concatenation of "t" and "u". *)

PROCEDURE Sub (t: T; start: CARDINAL;
               length: CARDINAL := LAST(CARDINAL)): T;
(* Return a sub-sequence of "t": empty if "start >= Length(t)" or
   "length = 0"; otherwise the subsequence ranging from "start" to the
   minimum of "start+length-1" and "Length(t)-1". *)

PROCEDURE Hash (t: T): Word.T;
(* Return a hash function of the contents of "t". *)

PROCEDURE HasWideChars (t: T): BOOLEAN;
(* Returns "TRUE" if "t" contains any "WIDECHAR" characters. *)

PROCEDURE GetChar     (t: T; i: CARDINAL): CHAR;
PROCEDURE GetWideChar (t: T; i: CARDINAL): WIDECHAR;
(* Return character "i" of "t".  It is a checked runtime error if "i
   >= Length(t)". *)

PROCEDURE SetChars     (VAR a: ARRAY OF CHAR;     t: T;  start: CARDINAL := 0);
PROCEDURE SetWideChars (VAR a: ARRAY OF WIDECHAR; t: T;  start: CARDINAL := 0);
(* For each "i" from 0 to "MIN(LAST(a), Length(t)-start-1)",
   set "a[i]" to "GetChar(t, i + start)".  *)

PROCEDURE FromChar     (ch: CHAR): T;
PROCEDURE FromWideChar (ch: WIDECHAR): T;
(* Return a text containing the single character "ch". *)

PROCEDURE FromChars     (READONLY a: ARRAY OF CHAR): T;
PROCEDURE FromWideChars (READONLY a: ARRAY OF WIDECHAR): T;
(* Return a text containing the characters of "a". *)

PROCEDURE FindChar     (t: T; c: CHAR;     start := 0): INTEGER;
```

```
PROCEDURE FindWideChar (t: T; c: WIDECHAR; start := 0): INTEGER;
(* If "c = t[i]" for some "i" in "[start~..~Length(t)-1]", return the
   smallest such "i"; otherwise, return -1. *)

PROCEDURE FindCharR(t: T; c: CHAR;  start := LAST(INTEGER)): INTEGER;
PROCEDURE FindWideCharR(t: T; c: WIDECHAR;  start := LAST(INTEGER)): INTEGER;
(* If "c = t[i]" for some "i" in "[0~..~MIN(start, Length(t)-1)]",
   return the largest such "i"; otherwise, return -1. *)

END Text.

(*
   The characters of a text may be "CHAR"s or "WIDECHAR"s.  A single
   text may contain both "CHAR"s and "WIDECHAR"s.  The characters of
   a text are converted between the types "CHAR" and "WIDECHAR" as
   needed.  Hence, client code may deal exclusively with either "CHAR"s
   or "WIDECHAR"s, or it may handle both character types.

   A "CHAR" is converted to a "WIDECHAR" by zero-extending its ordinal
   value.  For example, if "c" is a "CHAR", "VAL (ORD (c), WIDECHAR)"
   is the corresponding "WIDECHAR".

   A "WIDECHAR" is converted to a "CHAR" by dropping the high-order
   eight bits of the "WIDECHAR".  For example, if "c" is "WIDECHAR",
   "VAL (Word.And (c, 16_ff), CHAR)" is the corresponding "CHAR"
   value.
*)
```

B.5 libm3: ASCII

Pfad: `libm3/src/types/ASCII.i3`

```
(* Copyright (C) 1989, Digital Equipment Corporation           *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)

(* Last modified on Tue May 11 16:59:03 PDT 1993 by swart          *)
(*      modified on Thu Apr 22 09:56:55 PDT 1993 by mcjones        *)
(*      modified on Thu Nov  2 21:55:29 1989 by muller         *)
(*      modified on Fri Sep 29 15:46:46 1989 by kalsow         *)
(*      modified on Fri Jan 20 10:02:29 PST 1989 by glassman   *)
(*      modified on Wed May 27 23:11:56 1987 by mbrown         *)
(*      modified Mon May 13 20:11:50 1985 by Ellis             *)

INTERFACE ASCII;

(* Ascii Characters
```

```
   Char deals with individual characters.  It includes constant
   definitions for the character codes of exotic characters, such
   as Char.NL for new-line.  It classifies characters into groups,
   like digits or punctuation; each group is represented as a set
   of characters.  Finally, it provides mapping tables that translate
   lower-case letters into upper-case and the like.

   For systems with Unicode CHARs this interface can be used for
   classifying the subset whose values are in the range 0 to 255.

   Index: characters; punctuation; case, converting characters;
          characters, case conversion; upper-case, converting to lower;
          lower-case, converting to upper
*)


CONST
  NUL = '\000';    SOH = '\001';    STX = '\002';    ETX = '\003';
  EOT = '\004';    ENQ = '\005';    ACK = '\006';    BEL = '\007';
  BS  = '\010';    HT  = '\011';    NL  = '\012';    VT  = '\013';
  NP  = '\014';    CR  = '\015';    SO  = '\016';    SI  = '\017';
  DLE = '\020';    DC1 = '\021';    DC2 = '\022';    DC3 = '\023';
  DC4 = '\024';    NAK = '\025';    SYN = '\026';    ETB = '\027';
  CAN = '\030';    EM  = '\031';    SUB = '\032';    ESC = '\033';
  FS  = '\034';    GS  = '\035';    RS  = '\036';    US  = '\037';
  SP  = '\040';    DEL = '\177';

TYPE
  Range = ['\000'..'\377'];
    (* Characters which are representable in both 8 bit and unicode
       characters. *)

  Set = SET OF Range;

CONST
  All           = Set{FIRST(Range).. LAST(Range)};
  Asciis        = Set{'\000'.. '\177'};
  Controls      = Set{'\000'.. '\037', '\177'};
  Spaces        = Set{' ', '\t', '\n', '\r', '\f'};
  Digits        = Set{'0'.. '9'};
  Uppers        = Set{'A'.. 'Z'};
  Lowers        = Set{'a'.. 'z'};
  Letters       = Uppers + Lowers;
  AlphaNumerics = Letters + Digits;
  Graphics      = Asciis - Controls;
  Punctuation   = Graphics - AlphaNumerics;

VAR
```

```
  Upper   : ARRAY Range OF Range;
  Lower   : ARRAY Range OF Range;
  Control : ARRAY Range OF Range;
  (* These constant arrays implement character conversions (mappings):

        Upper[c]   = the upper-case equivalent of c if c is a letter, o.w. c
        Lower[c]   = the lower-case equivalent of c if c is a letter, o.w. c
        Control[c] = the control-shifted equivalent of c if c is in Graphics
                      (i.e. BitAnd( c, 037B )), o.w. c
    *)

END ASCII.
```

B.6 libm3: Stdio

Pfad: `libm3/src/rw/Stdio.i3`

```
(* Copyright (C) 1993, Digital Equipment Corporation           *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)
(* Last modified on Wed Feb  1 08:10:00 PST 1995 by kalsow     *)
(*      modified on Tue Nov  9 09:51:28 PST 1993 by mcjones    *)
(*      modified on Thu Jan 28 13:15:55 PST 1993 by mjordan    *)
(*      modified on Tue Feb 12 03:22:32 1991 by muller         *)

(* "Stdio" provides streams for standard input, standard output, and
   standard error. These streams correspond to file handles returned
   by the "GetStandardFileHandles" procedure in the "Process" interface.
   \index{standard I/O!streams}
*)

INTERFACE Stdio;

IMPORT Rd, Wr;

VAR
  stdin: Rd.T;
  stdout: Wr.T;
  stderr: Wr.T;
  bufferedStderr: Wr.T;

END Stdio.

(* The initialization of these streams depends on the underlying
   operating system.

   If the standard error stream is directed to a terminal, it will be
   unbuffered, so that explicit "Wr.Flush" calls are unnecessary for
```

```
   interactive programs.  A buffered version of the standard error
   stream is also provided, but programs should not use both "stderr"
   and "bufferedStderr".

   If the streams are directed to or from random-access files, they
   will be seekable.

   It is possible that "stderr" is equal to "stdout".  Therefore,
   programs that perform seek operations on "stdout" should take care
   not to destroy output data when writing error messages. *)
```

B.7 libm3: IO

Pfad: `libm3/src/rw/IO.i3`

```
(* Copyright 1993 by Digital Equipment Corp.                      *)
(* All rights reserved.                                           *)
(* See the file COPYRIGHT for a full description.                 *)
(* Last modified on Thu Sep 23 12:17:09 PDT 1993 by mcjones       *)
(*      modified on Fri Jun 18 13:27:27 PDT 1993 by wobber        *)
(*      modified on Sat Jan 26 14:38:00 PST 1993 by gnelson       *)

(* The "IO" interface provides textual input and output for simple
   programs.  For more detailed control, use the interfaces "Rd",
   "Wr", "Stdio", "FileRd", "FileWr", "Fmt", and "Lex".

   The input procedures take arguments of type "Rd.T" that specify
   which input stream to use.  If this argument is defaulted, standard
   input ("Stdio.stdin") is used.  Similarly, if an argument of type
   "Wr.T" to an output procedure is defaulted, "Stdio.stdout" is used.
   *)

INTERFACE IO;

IMPORT Rd, Wr;

PROCEDURE Put(txt: TEXT; wr: Wr.T := NIL);
(* Output "txt" to "wr" and flush "wr". *)

PROCEDURE PutChar(ch: CHAR; wr: Wr.T := NIL);
(* Output "ch" to "wr" and flush "wr". *)

PROCEDURE PutWideChar(ch: WIDECHAR; wr: Wr.T := NIL);
(* Output "ch" to "wr" and flush "wr". *)

PROCEDURE PutInt(n: INTEGER; wr: Wr.T := NIL);
(* Output "Fmt.Int(n)" to "wr" and flush "wr". *)
```

```
PROCEDURE PutReal(r: REAL; wr: Wr.T := NIL);
(* Output "Fmt.Real(r)" to "wr" and flush "wr". *)

PROCEDURE EOF(rd: Rd.T := NIL): BOOLEAN;
(* Return "TRUE" iff "rd" is at end-of-file. *)

EXCEPTION Error;

(* The exception "Error" is raised whenever a "Get" procedure
   encounters syntactically invalid input, including unexpected
   end-of-file. *)

PROCEDURE GetLine(rd: Rd.T := NIL): TEXT RAISES {Error};
(* Read a line of text from "rd" and return it. *)

(* A line of text is either zero or more characters terminated by a
   line break, or one or more characters terminated by an end-of-file.
   In the former case, "GetLine" consumes the line break but does not
   include it in the returned value.  A line break is either {\tt
   \char'42\char'134n\char'42} or {\tt
   \char'42\char'134r\char'134n\char'42}. *)

PROCEDURE GetChar(rd: Rd.T := NIL): CHAR RAISES {Error};
(* Read the next character from "rd" and return it. *)

PROCEDURE GetWideChar(rd: Rd.T := NIL): WIDECHAR RAISES {Error};
(* Read the next two bytes from "rd" and return it as a wide character. *)

PROCEDURE GetInt(rd: Rd.T := NIL): INTEGER RAISES {Error};
(* Read a decimal numeral from "rd" using "Lex.Int" and return its
   value. *)

PROCEDURE GetReal(rd: Rd.T := NIL): REAL RAISES {Error};
(* Read a real number from "rd" using "Lex.Real" and return its value.
   *)

PROCEDURE OpenRead(f: TEXT): Rd.T;
(* Open the file name "f" for reading and return a reader on its
   contents. If the file doesn't exist or is not readable, return
   "NIL". *)

PROCEDURE OpenWrite(f: TEXT): Wr.T;
(* Open the file named "f" for writing and return a writer on its
   contents.  If the file does not exist it will be created.  If the
   process does not have the authority to modify or create the file,
   return "NIL". *)
```

```
END IO.
```

B.8 libm3: Rd

Pfad: `libm3/src/rw/Rd.i3`

```
(* Copyright (C) 1989, Digital Equipment Corporation          *)
(* All rights reserved.                                       *)
(* See the file COPYRIGHT for a full description.             *)
(* Last modified on Mon Nov  8 17:21:08 PST 1993 by mcjones       *)
(*      modified on Tue Jul  6 13:05:03 PDT 1993 by wobber        *)
(*      modified on Tue Jun 15 09:42:56 1993 by gnelson       *)
(*      modified on Wed Apr 22 16:41:35 PDT 1992 by kalsow     *)
(*      modified on Mon Dec 24 01:10:09 1990 by muller        *)


(* An "Rd.T" (or ‘‘reader’’) is a character input stream.  The basic
   operation on a reader is "GetChar", which returns the source
   character at the ‘‘current position’’ and advances the current
   position by one.  Some readers are ‘‘seekable’’, which means that
   they also allow setting the current position anywhere in the
   source.  For example, readers from random access files are
   seekable; readers from terminals and sequential files are not.
   \index{character input stream}
   \index{input stream}
   \index{stream!input}
   \index{reader}

   Some readers are ‘‘intermittent’’, which means that the source of
   the reader trickles in rather than being available to the
   implementation all at once.  For example, the input stream from an
   interactive terminal is intermittent.  An intermittent reader is
   never seekable.

   Abstractly, a reader "rd" consists of

| len(rd)           ‘the number of source characters‘
| src(rd)           ‘a sequence of length "len(rd)+1"‘
| cur(rd)           ‘an integer in the range "[0..len(rd)]"‘
| avail(rd)         ‘an integer in the range "[cur(rd)..len(rd)+1]"‘
| closed(rd)        ‘a boolean‘
| seekable(rd)      ‘a boolean‘
| intermittent(rd)  ‘a boolean‘

   These values are not necessarily directly represented in the data
   fields of a reader object.  In particular, for an intermittent
   reader, "len(rd)" may be unknown to the implementation.  But in
   principle the values determine the state of the reader.
```

```
   The sequence "src(rd)" is zero-based: "src(rd)[i]" is valid for "i"
   from 0 to "len(rd"). The first "len(rd)" elements of "src" are the
   characters that are the source of the reader.  The final element is
   a special value "eof" used to represent end-of-file.  The value
   "eof" is not a character.

   The value of "cur(rd)" is the index in "src(rd)" of the next
   character to be returned by "GetChar", unless "cur(rd) = len(rd)",
   in which case a call to "GetChar" will raise the exception
   "EndOfFile".

   The value of "avail(rd)" is important for intermittent readers: the
   elements whose indexes in "src(rd)" are in the range
   "[cur(rd)..avail(rd)-1]" are available to the implementation and
   can be read by clients without blocking.  If the client tries to
   read further, the implementation will block waiting for the other
   characters.  If "rd" is not intermittent, then "avail(rd)" is equal
   to "len(rd)+1".  If "rd" is intermittent, then "avail(rd)" can
   increase asynchronously, although the procedures in this interface
   are atomic with respect to such increases.

   The definitions above encompass readers with infinite sources.  If
   "rd" is such a reader, then "len(rd)" and "len(rd)+1" are both
   infinity, and there is no final "eof" value.

   Every reader is a monitor; that is, it contains an internal lock
   that is acquired and held for each operation in this interface, so
   that concurrent operations will appear atomic.  For faster,
   unmonitored access, see the "UnsafeRd" interface.

   If you are implementing a long-lived reader class, such as a pipe
   or TCP stream, the index of the reader may eventually overflow,
   causing the program to crash with a bounds fault.  We recommend
   that you provide an operation to reset the reader index, which the
   client can call periodically. *)

INTERFACE Rd;

IMPORT AtomList;
FROM Thread IMPORT Alerted;

TYPE T <: ROOT;

EXCEPTION EndOfFile; Failure(AtomList.T);

(* Since there are many classes of readers, there are many ways that a
   reader can break---for example, the connection to a terminal can be
```

```
   broken, the disk can signal a read error, etc.  All problems of
   this sort are reported by raising the exception "Failure".  The
   documentation of a reader class should specify what failures the
   class can raise and how they are encoded in the argument to
   "Failure".

   Illegal operations cause a checked runtime error. *)

PROCEDURE GetChar(rd: T): CHAR
  RAISES {EndOfFile, Failure, Alerted};
(* Return the next character from "rd". More precisely, this is
   equivalent to the following, in which "res" is a local variable of
   type "CHAR": *)
(*
| IF closed(rd) THEN `Cause checked runtime error` END;
| `Block until "avail(rd) > cur(rd)"`;
| IF cur(rd) = len(rd) THEN
|   RAISE EndOfFile
| ELSE
|   res := src(rd)[cur(rd)]; INC(cur(rd)); RETURN res
| END
*)

PROCEDURE GetWideChar(rd: T): WIDECHAR
  RAISES {EndOfFile, Failure, Alerted};
(* Return the next wide character from "rd".  Two 8-bit bytes are
   read from "rd" and concatenated in little-endian order to
   form a 16-bit character.  That is, the first byte read will be the
   low-order 8 bits of the result and the second byte will be the
   high-order 8 bits. *)

(* Many operations on a reader can wait indefinitely.  For example,
   "GetChar" can wait if the user is not typing. In general these waits
   are alertable, so each procedure that might wait includes
   "Thread.Alerted" in its "RAISES" clause. *)

PROCEDURE EOF(rd: T): BOOLEAN RAISES {Failure, Alerted};
(* Return "TRUE" iff "rd" is at end-of-file. More precisely, this is
   equivalent to: *)
(*
| IF closed(rd) THEN `Cause checked runtime error` END;
| `Block until "avail(rd) > cur(rd)"`;
| RETURN cur(rd) = len(rd)
*)

(* Notice that on an intermittent reader, "EOF" can block. For example, if
   there are no characters buffered in a terminal reader, "EOF" must wait
   until the user types one before it can determine whether he typed the
```

```
   special key signalling end-of-file. If you are using "EOF" in an
   interactive input loop, the right sequence of operations is:
   \begin{enumerate}
   \item prompt the user;
   \item call "EOF", which probably waits on user input;
   \item presuming that "EOF" returned "FALSE", read the user’s input.
   \end{enumerate} *)

PROCEDURE UnGetChar(rd: T) RAISES {};
(* ‘‘Push back’’ the last character read from "rd", so that the next
   call to "GetChar" will read it again. More precisely, this is
   equivalent to the following: *)
(*
| IF closed(rd) THEN ‘Cause checked runtime error‘ END;
| IF cur(rd) > 0 THEN DEC(cur(rd)) END

   except there is a special rule: "UnGetChar(rd)" is guaranteed to
   work only if "GetChar(rd)" was the last operation on "rd".  Thus
   "UnGetChar" cannot be called twice in a row, or after "Seek" or
   "EOF". If this rule is violated, the implementation is allowed (but
   not required) to cause a checked runtime error. *)

PROCEDURE CharsReady(rd: T): CARDINAL RAISES {Failure};
(* Return some number of characters that can be read without
   indefinite waiting. The ‘‘end of file marker’’ counts as one
   character for this purpose, so "CharsReady" will return 1, not 0,
   if "EOF(rd)" is true. More precisely, this is equivalent to the
   following: *)
(*
| IF closed(rd) THEN ‘Cause checked runtime error‘ END;
| IF avail(rd) = cur(rd) THEN
|   RETURN 0
| ELSE
|   RETURN ‘some number in the range "[1~..~avail(rd) - cur(rd)]"‘
| END;
*)

(* Warning: "CharsReady" can return a result less than "avail(rd) -
   cur(rd)"; also, more characters might trickle in just as
   "CharsReady" returns. So the code to flush buffered input without
   blocking requires a loop:

| LOOP
|   n := Rd.CharsReady(rd);
|   IF n = 0 THEN EXIT END;
|   FOR i := 1 TO n DO EVAL Rd.GetChar(rd) END
| END;
*)
```

```
PROCEDURE GetSub(rd: T; VAR (*OUT*) str: ARRAY OF CHAR)
  : CARDINAL RAISES {Failure, Alerted};
(* Read from "rd" into "str" until "rd" is exhausted or "str" is
   filled. More precisely, this is equivalent to the following, in
   which "i" is a local variable: *)
(*
| i := 0;
| WHILE i # NUMBER(str) AND NOT EOF(rd) DO
|   str[i] := GetChar(rd); INC(i)
| END;
| RETURN i
*)

PROCEDURE GetWideSub(rd: T; VAR (*OUT*) str: ARRAY OF WIDECHAR)
  : CARDINAL RAISES {Failure, Alerted};
(* Read from "rd" into "str" until "rd" is exhausted or "str" is
   filled. More precisely, this is equivalent to the following, in
   which "i" is a local variable: *)
(*
| i := 0;
| WHILE i # NUMBER(str) AND NOT EOF(rd) DO
|   str[i] := GetWideChar(rd); INC(i)
| END;
| RETURN i
*)

PROCEDURE GetSubLine(rd: T; VAR (*OUT*) str: ARRAY OF CHAR)
  : CARDINAL RAISES {Failure, Alerted};
(* Read from "rd" into "str" until a newline is read, "rd" is
   exhausted, or "str" is filled. More precisely, this is equivalent
   to the following, in which "i" is a local variable: *)
(*
| i := 0;
| WHILE
|   i # NUMBER(str) AND
|   (i = 0 OR str[i-1] # '\n') AND
|   NOT EOF(rd)
| DO
|   str[i] := GetChar(rd); INC(i)
| END;
| RETURN i
*)

PROCEDURE GetWideSubLine(rd: T; VAR (*OUT*) str: ARRAY OF WIDECHAR)
  : CARDINAL RAISES {Failure, Alerted};
(* Read from "rd" into "str" until a newline is read, "rd" is
   exhausted, or "str" is filled. *)
```

```
(* Note that "GetLine" strips the terminating line break, while
   "GetSubLine" does not. *)

PROCEDURE GetText(rd: T; len: CARDINAL): TEXT
  RAISES {Failure, Alerted};
(* Read from "rd" until it is exhausted or "len" characters have been
   read, and return the result as a "TEXT".  More precisely, this is
   equivalent to the following, in which "i" and "res" are local
   variables: *)
(*
| res := ""; i := 0;
| WHILE i # len AND NOT EOF(rd) DO
|   res := res & Text.FromChar(GetChar(rd));
|   INC(i)
| END;
| RETURN res
*)

PROCEDURE GetWideText(rd: T; len: CARDINAL): TEXT
  RAISES {Failure, Alerted};
(* Read from "rd" until it is exhausted or "len" wide characters have been
   read, and return the result as a "TEXT".  More precisely, this is
   equivalent to the following, in which "i" and "res" are local
   variables: *)
(*
| res := ""; i := 0;
| WHILE i # len AND NOT EOF(rd) DO
|   res := res & Text.FromWideChar(GetChar(rd));
|   INC(i)
| END;
| RETURN res
*)

PROCEDURE GetLine(rd: T): TEXT
  RAISES {EndOfFile, Failure, Alerted};
(* If "EOF(rd)" then raise "EndOfFile".  Otherwise, read characters
   until a line break is read or "rd" is exhausted, and return the
   result as a "TEXT"---but discard the line break if it is present.
   A line break is either {\tt \char'42\char'134n\char'42} or {\tt
   \char'42\char'134r\char'134n\char'42} More precisely, this is
   equivalent to the following, in which "ch" and "res" are local
   variables: *)
(*
| IF EOF(rd) THEN RAISE EndOfFile END;
| res := ""; ch := '\000'; (* any char but newline *)
| WHILE ch # '\n' AND NOT EOF(rd) DO
|   ch := GetChar(rd);
```

```
|   IF ch = '\n' THEN
|     IF NOT Text.Empty(res) AND
|         Text.GetChar(res, Text.Length(res)-1) = '\r' THEN
|       res := Text.Sub(res, 0, Text.Length(res)-1)
|     END
|   ELSE
|     res := res & Text.FromChar(ch)
|   END
| RETURN res
*)

PROCEDURE GetWideLine(rd: T): TEXT
  RAISES {EndOfFile, Failure, Alerted};
(* If "EOF(rd)" then raise "EndOfFile".  Otherwise, read wide characters
   until a line break is read or "rd" is exhausted, and return the
   result as a "TEXT"---but discard the line break if it is present.
   A line break is either {\tt \char'42\char'134n\char'42} or {\tt
   \char'42\char'134r\char'134n\char'42}. *)

PROCEDURE Seek(rd: T; n: CARDINAL) RAISES {Failure, Alerted};
(* This is equivalent to: *)
(*
| IF closed(rd) OR NOT seekable(rd) THEN
|   `Cause checked runtime error`
| END;
| cur(rd) := MIN(n, len(rd))
*)

PROCEDURE Close(rd: T) RAISES {Failure, Alerted};
(* Release any resources associated with "rd" and set "closed(rd) :=
   TRUE".  The documentation of a procedure that creates a reader
   should specify what resources are released when the reader is
   closed.  This leaves "rd" closed even if it raises an exception,
   and is a no-op if "rd" is closed. *)

PROCEDURE Index(rd: T): CARDINAL RAISES {};
(* This is equivalent to: *)
(*
| IF closed(rd) THEN `Cause checked runtime error` END;
| RETURN cur(rd)
*)

PROCEDURE Length(rd: T): INTEGER RAISES {Failure, Alerted};
(* This is equivalent to: *)
(*
| IF closed(rd) THEN
|   `Cause checked runtime error`
| END;
```

```
| RETURN len(rd)

   If "len(rd)" is unknown to the implementation of an intermittent
   reader, "Length(rd)" returns -1.  *)

PROCEDURE Intermittent(rd: T): BOOLEAN RAISES {};
PROCEDURE Seekable(rd: T): BOOLEAN RAISES {};
PROCEDURE Closed(rd: T): BOOLEAN RAISES {};
(* Return "intermittent(rd)", "seekable(rd)", and "closed(rd)",
   respectively. These can be applied to closed readers. *)

END Rd.
```

B.9 libm3: Wr

Pfad: `libm3/src/rw/Wr.i3`

```
(* Copyright (C) 1989, Digital Equipment Corporation           *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)
(* Last modified on Mon Nov  8 17:21:09 PST 1993 by mcjones    *)
(*      modified on Tue Jul  6 13:05:58 PDT 1993 by wobber     *)
(*      modified on Sat Feb 29 08:19:34 PST 1992 by kalsow     *)
(*      modified on Mon Dec 24 01:09:54 1990 by muller         *)

(* A "Wr.T" (or ``writer'') is a character output stream.  The basic
   operation on a writer is "PutChar", which extends a writer's
   character sequence by one character. Some writers (called
   ``seekable writers'') also allow overwriting in the middle of the
   sequence.  For example, writers to random access files are
   seekable, but writers to terminals and sequential files are not.
   \index{character output stream}
   \index{output stream}
   \index{stream!output}
   \index{writer}

   Writers can be (and usually are) buffered. This means that
   operations on the writer don't immediately affect the underlying
   target of the writer, but are saved up and performed later.  For
   example, a writer to a disk file is not likely to update the disk
   after each character.

   Abstractly, a writer "wr" consists of:

| len(wr)       `a non-negative integer`
| c(wr)         `a character sequence of length "len(wr)"`
| cur(wr)       `an integer in the range "[0..len(wr)]"`
| target(wr)    `a character sequence`
```

```
| closed(wr)    ‘a boolean‘
| seekable(wr)  ‘a boolean‘
| buffered(wr)  ‘a boolean‘

  These values are generally not directly represented in the data
  fields of a writer object, but in principle they determine the
  state of the writer.

  The sequence "c(wr)" is zero-based: "c(wr)[i]" is valid for "i"
  from 0 through "len(wr)-1".  The value of "cur(wr)" is the index of
  the character in "c(wr)" that will be replaced or appended by the
  next call to "PutChar".  If "wr" is not seekable, then "cur(wr)" is
  always equal to "len(wr)", since in this case all writing happens
  at the end.

  The difference between "c(wr)" and "target(wr)" reflects the
  buffering: if "wr" is not buffered, then "target(wr)" is updated to
  equal "c(wr)" after every operation; if "wr" is buffered, then
  updates to "target(wr)" can be delayed.  For example, in a writer
  to a file, "target(wr)" is the actual sequence of characters on the
  disk; in a writer to a terminal, "target(wr)" is the sequence of
  characters that have actually been transmitted.  (This sequence may
  not exist in any data structure, but it still exists abstractly.)

  If "wr" is buffered, then the assignment "target(wr) := c(wr)" can
  happen asynchronously at any time, although the procedures in this
  interface are atomic with respect to such assignments.

  Every writer is a monitor; that is, it contains an internal lock
  that is acquired and held for each operation in this interface, so
  that concurrent operations will appear atomic.  For faster,
  unmonitored access, see the "UnsafeWr" interface.

  If you are implementing a long-lived writer class, such as a pipe
  or TCP stream, the index of the writer may eventually overflow,
  causing the program to crash with a bounds fault.  We recommend
  that you provide an operation to reset the writer index, which the
  client can call periodically.

  It is useful to specify the effect of several of the procedures in
  this interface in terms of the action "PutC(wr, ch)", which outputs
  the character "ch" to the writer "wr":

| PutC(wr, ch) =
|   IF closed(wr) THEN ‘Cause checked runtime error‘ END;
|   IF cur(wr) = len(wr) THEN
|     ‘Extend "c(wr)" by one character, incrementing "len(wr)"‘
|   END;
```

```
|   c(wr)[cur(wr)] := ch;
|   INC(cur(wr));

   "PutC" is used only in specifying the interface; it is not a real
   procedure.

   Like "PutC", "PutWC" is used to specify how wide characters are written.
   Wide characters are written in little-endian order, the low-order
   8-bits first, followed by the high-order 8-bits:

| PutWC(wr, wch) =
|   PutC(wr, VAL (Word.Extract (ORD (wch), 0, 8), CHAR));
|   PutC(wr, VAL (Word.Extract (ORD (wch), 8, 8), CHAR));

*)

INTERFACE Wr;

IMPORT AtomList, OSConfig;
FROM Thread IMPORT Alerted;

TYPE T <: ROOT;

EXCEPTION Failure(AtomList.T);

(* Since there are many classes of writers, there are many ways that a
   writer can break---for example, the network can go down, the disk
   can fill up, etc.  All problems of this sort are reported by
   raising the exception "Failure".  The documentation of each writer
   class should specify what failures the class can raise and how they
   are encoded in the argument to "Failure".

   Illegal operations (for example, writing to a closed writer) cause
   checked runtime errors. *)

CONST EOL = OSConfig.LineSep;
(* End of line. *)

(* On POSIX, "EOL" is {\tt \char'42\char'134n\char'42}; on Win32,
   "EOL" is {\tt \char'42\char'134r\char'134n\char'42}. *)

PROCEDURE PutChar(wr: T; ch: CHAR) RAISES {Failure, Alerted};
(* Output "ch" to "wr".  More precisely, this is equivalent to: *)
(*
| PutC(wr, ch); IF NOT buffered(wr) THEN Flush(wr) END
*)

PROCEDURE PutWideChar(wr: T; ch: WIDECHAR) RAISES {Failure, Alerted};
```

```
(* Output "ch" to "wr".  More precisely, this is equivalent to: *)
(*
| PutWC(wr, ch); IF NOT buffered(wr) THEN Flush(wr) END
*)

(* Many operations on a writer can wait indefinitely.  For example,
   "PutChar" can wait if the user has suspended output to his
   terminal.  These waits can be alertable, so each procedure that
   might wait includes "Thread.Alerted" in its raises clause. *)

PROCEDURE PutText(wr: T; t: TEXT) RAISES {Failure, Alerted};
(* Output "t" to "wr".  More precisely, this is equivalent to: *)
(*
| FOR i := 0 TO Text.Length(t) - 1 DO
|   PutC(wr, Text.GetChar(t, i))
| END;
| IF NOT buffered(wr) THEN Flush(wr) END

   except that, like all operations in this interface, it is atomic
   with respect to other operations in the interface. (It would be
   wrong to write "PutChar" instead of "PutC", since "PutChar" always
   flushes if the writer is unbuffered.)
*)

PROCEDURE PutWideText(wr: T; t: TEXT) RAISES {Failure, Alerted};
(* Output "t" to "wr".  More precisely, this is equivalent to: *)
(*
| FOR i := 0 TO Text.Length(t) - 1 DO
|   PutWC(wr, Text.GetChar(t, i))
| END;
| IF NOT buffered(wr) THEN Flush(wr) END
*)

PROCEDURE PutString(wr: T; READONLY a: ARRAY OF CHAR)
  RAISES {Failure, Alerted};
(* Output "a" to "wr".  More precisely, other than the fact that this
   is atomic, it is equivalent to: *)
(*
| FOR i := FIRST(a) TO LAST(a) DO PutC(wr, a[i]) END;
| IF NOT buffered(wr) THEN Flush(wr) END
*)

PROCEDURE PutWideString(wr: T; READONLY a: ARRAY OF WIDECHAR)
  RAISES {Failure, Alerted};
(* Output "a" to "wr".  More precisely, other than the fact that this
   is atomic, it is equivalent to: *)
(*
| FOR i := FIRST(a) TO LAST(a) DO PutWC(wr, a[i]) END;
```

```
| IF NOT buffered(wr) THEN Flush(wr) END
*)

PROCEDURE Seek(wr: T; n: CARDINAL) RAISES {Failure, Alerted};
(* Set the current position of "wr" to "n".  This is an error if "wr"
   is closed. More precisely, this is equivalent to: *)
(*
| IF wr.closed OR NOT seekable(wr) THEN
|   `Cause checked runtime error`
| END;
| cur(wr) := MIN(n, len(wr))
*)

PROCEDURE Flush(wr: T) RAISES {Failure, Alerted};
(* Perform all buffered operations.  That is, set "target(wr) :=
   c(wr)".  It is a checked runtime error if "wr" is closed. *)

PROCEDURE Close(wr: T) RAISES {Failure, Alerted};

(* Flush "wr", release any resources associated with "wr", and set
   "closed(wr) := TRUE".  The documentation for a procedure that
   creates a writer should specify what resources are released when
   the writer is closed.  This leaves "closed(wr)" equal to "TRUE"
   even if it raises an exception, and is a no-op if "wr" is closed.
   *)

PROCEDURE Length(wr: T): CARDINAL RAISES {Failure, Alerted};
PROCEDURE Index(wr: T): CARDINAL RAISES {};
PROCEDURE Seekable(wr: T): BOOLEAN RAISES {};
PROCEDURE Closed(wr: T): BOOLEAN RAISES {};
PROCEDURE Buffered(wr: T): BOOLEAN RAISES {};
(* These procedures return "len(wr)", "cur(wr)", "seekable(wr)",
   "closed(wr)", and "buffered(wr)", respectively. "Length" and
   "Index" cause a checked runtime error if "wr" is closed; the other
   three procedures do not. *)

END Wr.
```

B.10 libm3: FileRd

Pfad: `libm3/src/rw/FileRd.i3`

```
(* Copyright (C) 1989, Digital Equipment Corporation           *)
(* All rights reserved.                                         *)
(* See the file COPYRIGHT for a full description.               *)
(* Last modified on Wed Dec 15 15:06:26 PST 1993 by mcjones    *)
(*      modified on Sat Aug  3 00:54:38 1991 by kalsow         *)
(*      modified on Fri Aug 17 01:56:52 1990 by muller         *)
```

```
(* A "FileRd.T", or file reader, is a reader on a "File.T".
   \index{buffered file I/O}
   \index{file!buffered I/O}
*)

INTERFACE FileRd;

IMPORT Rd, File, OSError, Pathname;

TYPE
  T <: Public;
  Public = Rd.T OBJECT METHODS
    init(h: File.T): T RAISES {OSError.E}
  END;
(* If "r" is a file reader and "h" is a file handle, the call
   "r.init(h)" initializes "r" so that reading "r" reads characters
   from "h", and so that closing "r" closes "h". *)

(* If "h" is a regular file handle, "r.init(h)" causes "r" to be a
   nonintermittent, seekable reader and initializes "cur(r)" to
   "cur(h)".

   For any other file handle "h", "r.init(h)" causes "r" to be an
   intermittent, nonseekable reader and initializes "cur(r)" to zero.

   If a subsequent reader operation on "r" raises "Rd.Failure", the
   associated exception argument is the "AtomList.T" argument
   accompanying an "OSError.E" exception from a file operation on "h".
   *)

PROCEDURE Open(p: Pathname.T): T RAISES {OSError.E};
(* Return a file reader whose source is the file named "p".  If the file
   does not exist, "OSError.E" is raised with an implementation-defined
   code. *)

(* The call "Open(p)" is equivalent to

| RETURN NEW(T).init(FS.OpenFileReadonly(p))

*)

END FileRd.
```

B.11 libm3: FileWr

Pfad: `libm3/src/rw/FileWr.i3`

```
(* Copyright (C) 1989, 1992, Digital Equipment Corporation     *)
(* All rights reserved.                                         *)
(* See the file COPYRIGHT for a full description.               *)
(* Last modified on Wed Dec 15 15:06:14 PST 1993 by mcjones     *)
(*      modified on Mon Feb 24 11:32:41 PST 1992 by muller      *)
(*      modified on Sat Aug  3 00:45:49 1991 by kalsow          *)

(* A "FileWr.T", or file writer, is a writer on a "File.T".
   \index{buffered file I/O}
   \index{file!buffered I/O}
*)

INTERFACE FileWr;

IMPORT Wr, File, OSError, Pathname;

TYPE
  T <: Public;
  Public = Wr.T OBJECT METHODS
    init(h: File.T; buffered: BOOLEAN := TRUE): T
      RAISES {OSError.E}
  END;
(* If "w" is a file writer and "h" is a file handle, the call
   "w.init(h)" initializes "w" so that characters output to "w" are
   written to "h" and so that closing "w" closes "h". *)

(* If "h" is a regular file handle and "b" is a Boolean, "w.init(h, b)"
   causes "w" to be a buffered seekable writer and initializes "cur(w)"
   to "cur(h)".

   For any other file handle "h", "w.init(h, b)" causes "w" to be
   a nonseekable writer, buffered if and only if "b" is "TRUE", and
   initializes "cur(w)" to zero.

   If a subsequent writer operation on "w" raises "Wr.Failure", the
   associated exception argument is the "AtomList.T" argument
   accompanying an "OSError.E" exception from a file operation on "h".
   *)

PROCEDURE Open(p: Pathname.T): T RAISES {OSError.E};
(* Return a file writer whose target is the file named "p".  If the
   file does not exist, it is created.  If the file exists, it is
   truncated to a size of zero. *)

(* The call "Open(p)" is equivalent to the following:

| RETURN NEW(T).init(FS.OpenFile(p))
```

```
*)

PROCEDURE OpenAppend(p: Pathname.T): T RAISES {OSError.E};
(* Return a file writer whose target is the file named "p".  If the
   file does not exist, it is created.  If the file exists, the writer is
   positioned to append to the existing contents of the file. *)

(* The call "OpenAppend(p)" is equivalent to the following:

| WITH h = FS.OpenFile(p, truncate := FALSE) DO
|   EVAL h.seek(RegularFile.Origin.End, 0);
|   RETURN NEW(T).init(h)
| END

*)

END FileWr.
```

B.12 libm3: File

Pfad: `libm3/src/os/Common/File.i3`

```
(* Copyright (C) 1993, Digital Equipment Corporation. *)
(* All rights reserved. *)
(* See the file COPYRIGHT for a full description. *)
(* Last modified on Fri Jul 15 13:45:59 PDT 1994 by mcjones *)

(* A "File.T", or {\em file handle}, is a source and/or sink of bytes.
   File handles provide an operating-system independent way to perform
   raw I/O.  For buffered I/O, use the "FileRd" and "FileWr"
   interfaces instead.  A file handle is created using "OpenFile" or
   "OpenFileReadonly" in the "FS" interface.
   \index{file!handle}
   \index{unbuffered file I/O}
   \index{file!unbuffered I/O}
   \index{I/O!unbuffered}
*)

INTERFACE File;

IMPORT Atom, OSError, Time;

TYPE
  T <: Public;
  Public = OBJECT METHODS
    read(VAR (*OUT*) b: ARRAY OF Byte;
      mayBlock: BOOLEAN := TRUE): INTEGER RAISES {OSError.E};
    write(READONLY b: ARRAY OF Byte) RAISES {OSError.E};
```

```
    status(): Status RAISES {OSError.E};
    close() RAISES {OSError.E}
  END;
  Byte = BITS 8 FOR [0 .. 255];
  Status = RECORD
    type: Type;
    modificationTime: Time.T;
    size: CARDINAL
  END;
  Type = Atom.T;

END File.

(* Formally, a file handle "h" has the components:

| type(h)       `an atom, the type of file`
| readable(h)   `a boolean`
| writable(h)   `a boolean`
| src(h)        `(a "REF" to) a sequence of bytes`
| srcCur(h)     `an integer in the range "[0..len(src(h))]"`
| srcEof(h)     `a boolean`
| snk(h)        `(a "REF" to) a sequence of bytes`
| snkCur(h)     `an integer in the range "[0..len(snk(h))]"`

   The "src..." components are meaningful only if "readable(h)".  The
   sequence "src(h)" is zero-based: "src(h)[i]" is valid for "i" from
   "0" to "len(src(h))-1".  For some subtypes of "File.T", the
   sequence "src(h)" can grow without bound.

   The "snk..." components are meaningful only if "writable(h)".  The
   sequence "snk(h)" is zero based: "snk(h)[i]" is valid for "i" from
   "0" to "len(snk(h))-1".

   For full details on the semantics of a file handle, consult the
   interface defining the particular subtype, for example, "Pipe.T",
   "Terminal.T", or "RegularFile.T".  In the case where no exceptions
   are raised, the methods of the subtypes of "File.T" obey the
   following specifications:

   The call

| h.read(b, mayBlock)

   is equivalent to

| IF NOT readable(h) OR NUMBER(b) = 0 THEN
|   `Cause checked runtime error`
| END;
```

```
| IF srcCur(h) = len(src(h)) AND NOT srcEof(h) THEN
|   IF NOT mayBlock THEN RETURN -1 END;
|   `Block until "srcCur(h) < len(src(h)) OR srcEof(h)"`
| END;
| IF srcCur(h) = len(src(h)) THEN RETURN 0 END;
| `Choose "k" such that:`
|   1 <= k <= MIN(NUMBER(b), len(src(h))-srcCur(h));
| FOR i := 0 TO k-1 DO
|   b[i] := src(h)[srcCur(h)];
|   INC(srcCur(h))
| END;
| RETURN k
```

\index{non-blocking read}

A result of zero always means end of file. The meaning of a subsequent read after end of file has been reached is undefined for a "File.T" but may be defined for a particular subtype.

The call

```
| h.write(b)
```

is equivalent to

```
| IF NOT writable(h) THEN `Cause checked runtime error` END;
| FOR i := 0 TO NUMBER(b)-1 DO
|   IF snkCur(h) = len(snk(h)) THEN
|     `Extend "snk(h)" by one byte`
|   END;
|   snk(h)[snkCur(h)] := b[i]
|   INC(srcCur(h))
| END;
```

The "read" and "write" methods are not alertable because it isn't possible to alert a thread blocked in a Win32 "ReadFile" or "WriteFile" system call.

The call

```
| h.status()
```

returns a result whose "type" field contains "type(h)". See the documentation for each subtype of "File.T" for more details, including the values of the "modificationTime" and "size" fields of the result, if any.

The call

```
| h.close()

   is equivalent to

| readable(h) := FALSE;
| writable(h) := FALSE

   Additionally, it releases any subtype-specific resources used by
   "h".  Every file handle should be closed.

   Clients should assume that file handles are unmonitored and should
   avoid concurrent accesses to a file handle from multiple threads.
   A particular subtype of "File.T" may provide a stronger
   specification with respect to atomicity.

*)
```

B.13 m3core: Float

Pfad: `m3core/src/float/Common/Float.ig`

```
(* Copyright (C) 1991, Digital Equipment Corporation          *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)
(* Last modified on Thu Dec  9 11:29:00 PST 1993 by mcjones    *)
(*      modified on Thu Apr 29 13:58:11 PDT 1993 by muller     *)
(*      modified on Mon Feb 15 15:17:13 PST 1993 by ramshaw    *)

(* The generic interface "Float" provides access to the floating-point
   operations required or recommended by the IEEE floating-point
   standard.  Consult the standard to resolve any fine points in the
   specification of the procedures.  Non-IEEE implementations that
   have values similar to NaNs and infinities should explain how those
   values behave in an implementation guide. (NaN is an IEEE term
   whose informal meaning is ‘‘not a number’’.) *)

GENERIC INTERFACE Float(R);

IMPORT FloatMode;

TYPE T = R.T;

PROCEDURE Scalb(x: T; n: INTEGER): T RAISES {FloatMode.Trap};
(* Return $\hbox"x"\cdot 2^{\hbox"n"}$. *)

PROCEDURE Logb(x: T): T RAISES {FloatMode.Trap};
(* Return the exponent of "x".  More precisely, return the unique
```

```
   integer $n$ such that the ratio $\hbox"ABS(x) / Base"^{n}$ is in
   the half-open interval "[1..Base)", unless "x" is denormalized, in
   which case return the minimum exponent value for "T". *)

PROCEDURE ILogb(x: T): INTEGER;
(* Like "Logb", but returns an integer, never raises an exception, and
   always returns the $n$ such that $\hbox"ABS(x) / Base"^{n}$ is in
   the half-open interval "[1..Base)", even for denormalized numbers.
   Special cases: it returns "FIRST(INTEGER)" when "x" = 0.0,
   "LAST(INTEGER)" when "x" is plus or minus infinity, and zero when
   "x" is NaN.  *)

PROCEDURE NextAfter(x, y: T): T RAISES {FloatMode.Trap};
(* Return the next representable neighbor of "x" in the direction
   towards "y".  If "x = y", return "x". *)

PROCEDURE CopySign(x, y: T): T;
(* Return "x" with the sign of "y". *)

PROCEDURE Finite(x: T): BOOLEAN;
(* Return "TRUE" if "x" is strictly between minus infinity and plus
   infinity.  This always returns "TRUE" on non-IEEE implementations.
   *)

PROCEDURE IsNaN(x: T): BOOLEAN;
(* Return "FALSE" if "x" represents a numerical (possibly
   infinite) value, and "TRUE" if "x" does not represent a
   numerical value.  For example, on IEEE implementations, returns
   "TRUE" if x is a NaN, "FALSE" otherwise. *)

(*
\index{NaN (not a number)}
*)


PROCEDURE Sign(x: T): [0..1];
(* Return the sign bit "x".  For non-IEEE implementations, this is
   the same as "ORD(x >= 0)"; for IEEE implementations,
   "Sign(-0) = 1" and "Sign(+0) = 0". *)

PROCEDURE Differs(x, y: T): BOOLEAN;
(* Return "(x < y OR y < x)".  Thus, for IEEE implementations,
   "Differs(NaN,x)" is always "FALSE"; for non-IEEE implementations,
   "Differs(x,y)" is the same as "x # y". *)

PROCEDURE Unordered(x, y: T): BOOLEAN;
(* Return "NOT (x <= y OR y <= x)". Thus, for IEEE implementations,
   "Unordered(NaN, x)" is always "TRUE"; for non-IEEE implementations,
```

```
   "Unordered(x, y)" is always "FALSE".
*)

PROCEDURE Sqrt(x: T): T RAISES {FloatMode.Trap};
(* Return the square root of "T".  This must be correctly rounded if
   "FloatMode.IEEE" is "TRUE". *)

TYPE IEEEClass =
  {SignalingNaN, QuietNaN, Infinity, Normal, Denormal, Zero};

PROCEDURE Class(x: T): IEEEClass;
(* Return the IEEE number class containing "x".  On non-IEEE systems,
   the result will be "Normal" or "Zero". *)

PROCEDURE FromDecimal(
    sign: [0..1];
    READONLY digits: ARRAY OF [0..9];
    exp: INTEGER): T RAISES {FloatMode.Trap};
(* Convert from floating-decimal to type "T". *)

(* \index{floating-point!conversion from decimal}
   \index{decimal conversion!to floating-point}

   Let "F" denote the nonnegative, floating-decimal number

| digits[0] . digits[1] ... digits[LAST(digits)] * 10^exp
| = sum(i, digits[i] * 10^(exp - i))

   The result of "FromDecimal" is the number "(-1)^sign * F", rounded
   to a value of type "T".

   The procedure "FromDecimal" is a floating-point operation, just
   like "+" and "*", in the sense that it rounds its ideal result
   correctly, observing the current rounding mode, and it sets flags
   and raises traps by the usual rules.  On IEEE implementations, it
   returns minus zero when "F" is sufficiently small and "sign=1". *)

TYPE DecimalApprox = RECORD
    class: IEEEClass;
    sign: [0..1];
    len: [1..R.MaxSignifDigits];
    digits: ARRAY[0..R.MaxSignifDigits-1] OF [0..9];
    exp: INTEGER;
    errorSign: [-1..1]
  END;

PROCEDURE ToDecimal(x: T): DecimalApprox;
(* Convert from type "T" to floating-decimal. *)
```

```
(* \index{floating-point!conversion to decimal}
   \index{decimal conversion!from floating-point}

   Let "D" denote "ToDecimal(x)".  Then, "D.class = Class(x)" and
   "D.sign = Sign(x)".  The other fields are defined only when
   "D.class" is either "Normal" or "Denormal".  In those cases, the
   values "D.len", "D.digits[0]" through "D.digits[D.len-1]", and
   "D.exp" encode a floating-decimal number "F" with the property that
   "(-1)^D.sign * F" approximates "x" in a sense discussed below.  The
   encoding is such that

| F = digits[0] . digits[1] ... digits[len - 1]  *  10^exp
|   = sum(i, digits[i] * 10^(exp - i))

   and

| ABS(x) = F * (1 + errorSign * epsilon)

   where "epsilon" is small and positive.  In particular, "D.errorSign"
   is "+1", "0", or "-1" according as "ABS(x)" is larger than, equal
   to, or smaller than "F".

   The current rounding mode determines the sense in which the
   floating-decimal number "(-1)^sign * F" approximates "x", but in a
   slightly subtle way.  Define the opposite of a directed rounding
   mode by reversing the direction, as follows:

|      Opp(TowardPlusInfinity) := TowardMinusInfinity
|     Opp(TowardMinusInfinity) := TowardPlusInfinity
|              Opp(TowardZero) := AwayFromZero

   Note that "AwayFromZero" isn't actually a rounding mode, but it is
   clear what it would mean if it were.  For all other rounding modes
   "M", we define "Opp(M) = M".  If the current rounding mode is "M",
   the call "ToDecimal(x)" returns a floating-decimal number that
   "FromDecimal" would convert, under rounding mode "Opp(M)", back to
   "x".  Among all such numbers, the returned value has as few digits
   as possible.  This implies that both "D.digits[0]" and
   "D.digits[D.len-1]" are nonzero.  If there is a tie for having the
   fewest digits, the tying number closest to "x" wins.  If there is
   also a tie for being closest to "x", it must be a two-way tie and
   the number whose last digit is even wins.

   Unlike "FromDecimal", "ToDecimal" never sets a "FloatMode.Flag" and
   never raises "FloatMode.Trap".

   The idea of converting to decimal by retaining just as many digits
```

```
   as are necessary to convert back to binary exactly was popularized
   by Guy L.~Steele Jr.\ and Jon L White~\cite{Steele}.  David M.~Gay
   pointed out the importance, in this context, of demanding that the
   conversion to binary handle mid-point cases by a known
   rule~\cite{Gay}.  For example, in IEEE double precision, the
   floating-decimal number "1e23" is precisely halfway between two
   adjacent floating-binary numbers.  If conversion to binary were
   allowed to go either way in such a mid-point case, conversion to
   decimal would have to avoid producing the simple number "1e23",
   producing instead either "1.0000000000000001e23" or
   "9.999999999999999e22".  We believe the idea of combining the
   Steele/White style of automatic precision control with directed
   rounding by using opposite rounding modes, as above, is new with
   Lyle Ramshaw. *)

END Float.
```

B.14 libm3: Math

Pfad: `libm3/src/arith/POSIX/Math.i3`

```
(* Copyright (C) 1989, Digital Equipment Corporation          *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)

(* Last modified on Wed Aug 18 20:33:57 PDT 1993 by heydon     *)
(*      modified on Fri Nov  3 14:14:31 PDT 1989 by muller     *)
(*      modified on Fri Oct 20 11:16:20 PDT 1989 by kalsow     *)
(*      modified on Fri Jan 20 12:42:01 PDT 1989 by glassman   *)
(*      modified on Thu May 21 17:29:38 PDT 1987 by rovner     *)
(*      modified on Sun Jun 22 11:05:15 PDT 1986 by violetta   *)

INTERFACE Math;

(* An interface to the C math library

   Programs that call any of these routines must be linked
   with the math library "-lm".

   The detailed semantics of these procedures are defined by
   your local C math library.  To learn the full story about
   any of these functions (e.g. their domains, ranges and
   accuracies), see the appropriate man page.

   Index: floating point, C math interface;
          C programming, interface to C math library
*)
```

```
(*---- miscellaneous useful constants ----*)

CONST
  Pi     = 3.14159265358979323846264338 33;
  LogPi  = 1.14472988584940017414342735 14;
  SqrtPi = 1.77245385090551602729816748 33;
  E      = 2.71828182845904523536028747 14;
  Degree = 0.01745329251994329576923690 7684;  (* One degree in radians *)


(*---- Exponential and Logarithm functions ----*)

<*EXTERNAL*> PROCEDURE exp (x: LONGREAL): LONGREAL;
(* returns E^x. *)

<*EXTERNAL*> PROCEDURE expm1 (x: LONGREAL): LONGREAL;
(* returns (E^x)-1, even for small x. *)

<*EXTERNAL*> PROCEDURE log (x: LONGREAL): LONGREAL;
(* returns the natural logarithm of x (base E). *)

<*EXTERNAL*> PROCEDURE log10 (x: LONGREAL): LONGREAL;
(* returns the base 10 logarithm of x. *)

<*EXTERNAL*> PROCEDURE log1p (x: LONGREAL): LONGREAL;
(* returns log(1+x), even for small x. *)

<*EXTERNAL*> PROCEDURE pow (x, y: LONGREAL): LONGREAL;
(* returns x^y. *)

<*EXTERNAL*> PROCEDURE sqrt (x: LONGREAL): LONGREAL;
(* returns the square root of x. *)


(*---- Trigonometric functions ----*)

<*EXTERNAL*> PROCEDURE cos (x: LONGREAL): LONGREAL;
(* returns the cosine of x radians. *)

<*EXTERNAL*> PROCEDURE sin (x: LONGREAL): LONGREAL;
(* returns the sine of x radians. *)

<*EXTERNAL*> PROCEDURE tan (x: LONGREAL): LONGREAL;
(* returns the tangent of x radians. *)

<*EXTERNAL*> PROCEDURE acos (x: LONGREAL): LONGREAL;
(* returns the arc cosine of x in radians. *)
```

```
<*EXTERNAL*> PROCEDURE asin (x: LONGREAL): LONGREAL;
(* returns the arc sine of x in radians. *)

<*EXTERNAL*> PROCEDURE atan (x: LONGREAL): LONGREAL;
(* returns the arc tangent of x in radians. *)

<*EXTERNAL*> PROCEDURE atan2 (y, x: LONGREAL): LONGREAL;
(* returns the arc tangent of y/x in radians. *)


(*---- Hyperbolic trigonometric functions ----*)

<*EXTERNAL*> PROCEDURE sinh (x: LONGREAL): LONGREAL;
(* returns the hyperbolic sine of x. *)

<*EXTERNAL*> PROCEDURE cosh (x: LONGREAL): LONGREAL;
(* returns the hyperbolic cosine of x. *)

<*EXTERNAL*> PROCEDURE tanh (x: LONGREAL): LONGREAL;
(* returns the hyperbolic tangent of x. *)

<*EXTERNAL*> PROCEDURE asinh (x: LONGREAL): LONGREAL;
(* returns the inverse hyperbolic sine of x *)

<*EXTERNAL*> PROCEDURE acosh (x: LONGREAL): LONGREAL;
(* returns the inverse hyperbolic cosine of x *)

<*EXTERNAL*> PROCEDURE atanh (x: LONGREAL): LONGREAL;
(* returns the inverse hyperbolic tangent of x *)


(*---- Rounding functions ----*)

<*EXTERNAL*> PROCEDURE ceil (x: LONGREAL): LONGREAL;
(* returns the least integer not less than x.
   Note: use the builtin Modula-3 function CEILING. *)

<*EXTERNAL*> PROCEDURE floor (x: LONGREAL): LONGREAL;
(* returns the greatest integer not greater than x.
   Note: use the builtin Modula-3 function FLOOR. *)

<*EXTERNAL*> PROCEDURE rint (x: LONGREAL): LONGREAL;
(* returns the nearest integer value to x.
   Note: the Modula-3 function ROUND may  be appropriate. *)

<*EXTERNAL*> PROCEDURE fabs (x: LONGREAL): LONGREAL;
(* returns the absolute value of x.
   Note: use the builtin Modula-3 function ABS. *)
```

```
(*---- Euclidean distance functions ----*)

<*EXTERNAL*> PROCEDURE hypot (x, y: LONGREAL): LONGREAL;
(* returns sqrt (x*x + y*y). *)

<*EXTERNAL*> PROCEDURE cabs (z: Complex): LONGREAL;
TYPE Complex = RECORD x, y: LONGREAL END;
(* returns sqrt (z.x*z.x + z.y*z.y) *)


(*---- Floating point representations ----*)

<*EXTERNAL*> PROCEDURE frexp (x: LONGREAL;  VAR exp: INTEGER): LONGREAL;
(* returns a value y and sets exp such that x = y * 2^exp,
    where ABS(X) is in the interval [0.5, 1). *)

<*EXTERNAL*> PROCEDURE ldexp (x: LONGREAL; exp: INTEGER): LONGREAL;
(* returns x * 2^exp. *)

<*EXTERNAL*> PROCEDURE modf (x: LONGREAL; VAR(*OUT*) i: LONGREAL): LONGREAL;
(* splits the argument "x" into an integer part "i" and a fractional part "f"
   such that "f + i = x" and such that "f" and "i" both have the same sign as
   "x", and returns "f". Although "i" is a LONGREAL, it is set to an integral
   value. *)

(*---- Error functions ----*)

<*EXTERNAL*> PROCEDURE erf (x: LONGREAL): LONGREAL;
(* returns the "error" function of x. *)

<*EXTERNAL*> PROCEDURE erfc (x: LONGREAL): LONGREAL;
(* returns 1.0 - erf(x), even for large x. *)


(*---- Gamma function ----*)

<*EXTERNAL*> PROCEDURE gamma (x: LONGREAL): LONGREAL;
<*EXTERNAL*> VAR signgam: INTEGER;
(* returns log(ABS(Gamma(ABS(x)))).  The sign of Gamma(ABS(X))
   is returned in signgam. *)


(*---- Bessel functions ----*)

<*EXTERNAL*> PROCEDURE j0 (x: LONGREAL): LONGREAL;
(* returns the zero-order Bessel function of first kind on x. *)
```

```
<*EXTERNAL*> PROCEDURE j1 (x: LONGREAL): LONGREAL;
(* returns the first-order Bessel function of first kind on x. *)

<*EXTERNAL*> PROCEDURE jn (n: INTEGER;  x: LONGREAL): LONGREAL;
(* returns the n th-order Bessel function of first kind on x. *)

<*EXTERNAL*> PROCEDURE y0 (x: LONGREAL): LONGREAL;
(* returns the zero-order Bessel function of second kind on x. *)

<*EXTERNAL*> PROCEDURE y1 (x: LONGREAL): LONGREAL;
(* returns the first-order Bessel function of second kind on x. *)

<*EXTERNAL*> PROCEDURE yn (n: INTEGER;  x: LONGREAL): LONGREAL;
(* returns the n th-order Bessel function of second kind on x. *)

(*---- Modulo functions ----*)

<*EXTERNAL*> PROCEDURE fmod (x, y: LONGREAL): LONGREAL;
(* returns the remainder of dividing x by y.
   Note: use the built-in Modula-3 function MOD. *)

<*EXTERNAL*> PROCEDURE drem (x, y: LONGREAL): LONGREAL;
<*EXTERNAL*> PROCEDURE remainder (x, y: LONGREAL): LONGREAL;
(* returns remainder "r = x - n*y", where "n = ROUND(x/y)".
   Note: the Modula-3 functions MOD and ROUND may be appropriate. *)

END Math.
```

B.15 libm3: Random

Pfad: `libm3/src/random/Common/Random.i3`

```
(* Copyright (C) 1989, Digital Equipment Corporation           *)
(* All rights reserved.                                        *)
(* See the file COPYRIGHT for a full description.              *)
(* Created September 1989 by Bill Kalsow                       *)
(* Based on Random.def by Mark R. Brown                        *)
(* Last modified on Tue Dec  7 17:33:37 PST 1993 by mcjones    *)
(*      modified on Thu Oct 21 08:12:21 PDT 1993 by kalsow     *)
(*      modified on Tue Jan 30 11:02:59 1990 by muller         *)
(*      modified on Thu Jan 25 21:30:53 PST 1990 by stolfi     *)

(* A "Random.T" (or just a generator) is a pseudo-random number
   generator.
   \index{pseudo-random number}
*)
```

```
INTERFACE Random;

TYPE
  T = OBJECT METHODS
    integer(min := FIRST(INTEGER);
      max := LAST(INTEGER)): INTEGER;
    real(min := 0.0e+0; max := 1.0e+0): REAL;
    longreal(min := 0.0d+0; max := 1.0d+0): LONGREAL;
    extended(min := 0.0x+0; max := 1.0x+0): EXTENDED;
    boolean(): BOOLEAN
  END;
  Default <: T OBJECT METHODS
    init(fixed := FALSE): Default
  END;
END Random.

(* Individual generators are unmonitored, and all the operations
   have side effects.

   The methods provided by a generator "rand" are:

   The call "rand.integer(a, b)" returns a uniformly distributed
   "INTEGER" in the closed interval "[a..b]".

   The call "rand.real(a, b)" returns a uniformly distributed "REAL"
   in the half-open interval "[a..b)".

   The call "longreal" and "extended" are like "real", but return
   values of the specified types.

   The call "rand.boolean()" returns a random "BOOLEAN" value.

   It is a checked runtime error if "min > max" on any call.

   "NEW(Default).init()" creates and initializes a generator (see
   below for implementation details).  If "fixed" is "TRUE", a
   predetermined sequence is used.  If "fixed" is "FALSE", "init"
   chooses a random seed in such a way that different sequences result
   even if "init" is called many times in close proximity.

\paragraph*{Example.} A good pseudo-random permutation of an array "a"
   can be generated as follows:

| WITH rand = NEW(Random.Default).init() DO
|   FOR i := FIRST(a) TO LAST(a) - 1 DO
|     WITH j = rand.integer(i, LAST(a)) DO
|       `Exchange "a[i]" and "a[j]"`
```

```
|     END
|   END
| END

\paragraph*{SRC Modula-3 implementation details.} The object returned
   by a call of "New(Default).init" uses an additive generator based
   on Knuth's Algorithm 3.2.2A (see \cite{Knuth:Vol2}).

*)
```

Index

HILBERT-Matrix, 99
PASCALsche Dreieck, 105

absolute Pfade, 12
Absolutwert, 81
abstrakte Klasse, 122, 123
aktuellen Verzeichnis, 11
Arcus Kosinus, 81
Arcus Sinus, 81
Arcus Tangens, 81
Assembler, 32
Ausnahmebehandlung, 113

Betriebssystem, 5
Bindung, 128
Bourne again Shell, 18

Call by reference, 62, 69
Call by value, 62, 67

Datei, 11
Datentypen, 43
Datenverbund, 95
Dekadischer Logarithmus, 82
deklariert, 43

Exponentialfunktion, 81

Feld, 84
flache Kopie, 101
Funktion, 61

Garbage Collector, 98, 115
globale Variable, 64

Heimverzeichnis, 13

Importieren, 119

Jokerzeichen, 19

Klasse, 122
Klassenhierarchie, 122
kompiliert, 32
Kosinus, 81
Kosinus Hyperbolicus, 81

Lazy evaluation, 56
Literale, 43
Logarithmus Naturalis, 82
logische Operatoren, 54
lokale Variable, 64

Manual Pages, 9
Maschinensprache, 32
Methode, 122
Module, 118
Modulofunktion, 81

natürlicher Logarithmus, 82
Negationsoperator, 55

Obere Gaußklammer, 81
Objektorientiertes Programmieren, 121
offenes Feld, 87
ordinale Typen, 49

Pakete, 121
Pfad, 12
Pipe, 19

Platzhalter, 19
Potenzfunktion, 82
Prozedur, 61

Quadratwurzel, 82
Qualifizieren, 40, 119
Quota, 10

Rekursion, 71
Rekursionsverankerung, 71
rekursiv, 71
relative Pfade, 12

Schleife, 28, 72
Schnittstelle, 122
Shell, 18
Shell-Skripte, 18
Signatur, 61
Sinus, 82
Sinus Hyperbolicus, 82
Steuerdatei, 121

Tangens, 82
Tangens Hyperbolicus, 82
Textliteral, 38
TicTacToe, 87
tiefe Kopie, 100

Untere Gaußklammer, 81
Unterklasse, 122
Unterprogramm, 61
Untertyp-Relation, 123
Unterverzeichnisse, 11

Vergleichsoperatoren, 54
Verkettung, 91
Verweis, 11
Verzeichnis, 11
Vorrang, 128

Wildcards, 19
Writer, 67
Wurzel, 11, 101

Zeiger, 97
Zufallsgenerator, 122