

Inhaltsverzeichnis

1	Einführung in UNIX	6
1.1	Etwas über Betriebssysteme	6
1.1.1	Aufbau eines Computers	6
1.1.2	UNIX	7
1.2	Die Etikette im Rechnerpool des Fachbereichsnetzes	8
1.3	An- und Abmelden beim System	8
1.4	Aufbau von UNIX-Kommandos	9
1.5	Editoren	10
1.6	Drucken	11
1.7	Verzeichnisse und Dateien	12
1.7.1	Dateien und Pfade	12
1.7.2	Grundgerüst des Verzeichnisbaumes	13
1.7.3	Navigation im Verzeichnisbaum	14
1.7.4	Ausgabe von Inhaltsverzeichnissen	16
1.7.5	Zugriffsrechte für Dateien und Verzeichnisse	17
1.7.6	Numerische und symbolische Form der Rechteänderung	18
1.8	UNIX-Shells	19
1.8.1	Sonderzeichen	19
1.8.2	Shell-Variablen	20
1.8.3	Kommandos	20
1.8.4	Platzhalter in Dateinamen	21
1.8.5	Umleitung	21
1.8.6	Vorverarbeitung der Kommandos	22
1.9	Prozesse	24
1.10	Arbeiten auf entfernten Rechnern	24
1.11	Weitere nützliche Befehle	25
1.12	Versionsverwaltung	26
1.12.1	Übersicht über verbreitete Versionsverwaltungssysteme	27
1.12.2	Arbeiten mit einem Darcs-Projekt	28
1.12.3	Austausch zwischen Darcs-Projekten	30

2	Algorithmen	32
2.1	Sequenzen	33
2.1.1	Zubereitung einer Tasse Pulverkaffee	33
2.1.2	Verfeinerung	33
2.2	Auswahl (Selektion)	34
2.2.1	Kaffeekochen mit Entscheidung	34
2.3	Schleifen (Wiederholung, Iteration)	35
2.3.1	Kaffeekochen mit Wiederholung	35
2.3.2	Maximumsbestimmung	36
2.3.3	EUKLIDischer Algorithmus zur Berechnung des größten gemeinsamen Teilers	37
2.3.4	Berechnung von Potenzen x^n	37
2.4	Funktionen	38
2.4.1	Zeichnen zweier konzentrischer Quadrate	38
2.5	Vom Algorithmus zum Programm	39
3	Einführung in die Modula-3-Programmierung	41
3.1	Überblick	41
3.1.1	Was ist Modula-3?	41
3.1.2	Literatur	42
3.1.3	Das nullte und kürzeste Modula-3-Programm	42
3.1.4	Ein erstes Programm: Ausgabe mit <code>IO.Put</code>	43
3.1.5	Projektstruktur	45
3.1.6	Unterprogramme und Funktionen	46
3.1.7	Bezeichner	46
3.2	Variablen und ordinale Typen	47
3.2.1	Variable und ihr Datentyp, ganze Zahlen	47
3.2.2	Weitere Beispiele für formatierte Ausgabe	50
3.2.3	Ausdrücke	52
3.2.4	Gleitkommazahlen	54
3.2.5	Aufzählungen	57
3.2.6	Wahrheitswerte	59
3.2.7	Unterbereiche	62
3.2.8	Mengen	63
3.2.9	Was ist der richtige Typ für mich?	64
3.2.10	Typumwandlungen	65
3.2.11	Eingabe mit <code>Lex</code>	65
3.3	Anweisungen zur Auswahl	67
3.3.1	IF -Anweisungen	67

3.3.2	CASE -Anweisung	70
3.4	Funktionen und Prozeduren	72
3.4.1	Definition	72
3.4.2	Aufruf	73
3.4.3	Beispiel: Call by value	74
3.4.4	Beispiel: Funktion	75
3.4.5	Beispiel: Funktion mit mehreren Eingaben	76
3.4.6	Bezugsrahmen von Variablen	77
3.4.7	Programm mit lokalen Variablen	77
3.4.8	Programm mit globalen und lokalen Variablen	78
3.4.9	Funktionsparameter sind lokale Variablen	80
3.4.10	Call by reference	81
3.4.11	Rekursive Funktionen	83
3.4.12	Konzeption von Funktionen	84
3.5	Schleifen	86
3.5.1	WHILE - und REPEAT -Schleife	86
3.5.2	LOOP -Schleife	86
3.5.3	FOR -Schleife	87
3.5.4	Beispiel: Berechnung von π nach Leibniz	88
3.6	Felder (Arrays)	90
3.6.1	Eindimensionale Felder	90
3.6.2	Mehrdimensionale Felder	93
3.6.3	Felder variabler Größe	93
3.6.4	Feldabschnitte	95
3.6.5	Felder und Mengen	95
3.7	Texte	96
3.7.1	Der Datentyp Zeichen (CHAR)	96
3.7.2	Der Datentyp TEXT	98
3.7.3	Ein- und Ausgabe von Texten	99
3.7.4	Ein Beispiel für Text-Verarbeitung: Test auf Bereichsüberschreitung	100
3.8	Datenverbände	102
3.9	Zeiger (Pointer)	104
3.9.1	Operationen auf Zeigern	105
3.9.2	Zeiger auf offene Felder	106
3.9.3	Zeiger auf Datenverbände	109
3.9.4	Felder von Feldern	114
3.9.5	Funktionsvariablen	116
3.10	Umgang mit Dateien	118

3.10.1	Beispiel zu Textdateien	120
3.11	Ausnahmebehandlung	122
3.11.1	TRY-EXCEPT -Struktur	123
3.11.2	TRY-FINALLY -Struktur	126
3.11.3	Ausnahmezustände selbst ausrufen	127
3.11.4	Konzeption von Ausnahmen	129
3.12	Module	130
3.12.1	Einbinden von Modulen	131
3.12.2	Module selbst erstellen	131
3.12.3	Paketverwaltung	135
3.12.4	Generische Module	137
3.13	Objekte	142
3.13.1	Klassen	142
3.13.2	Arbeiten mit Objekten	143
3.13.3	Deklaration von Klassen	144
3.13.4	Beispiel: geometrische Figuren	145
3.13.5	Gewissensfrage: Generisches Modul oder Objektklasse	150
A	Übersichten	152
A.1	Einige Grundsätze zur Programmierung	152
A.2	Vorrangregeln	154
A.3	Steuerzeichen in Textliteralen	155
A.4	Mathematische Funktionen	155
B	Wichtige Module der Standardbibliotheken <code>m3core</code> und <code>libm3</code>	161
B.1	<code>libm3: Fmt</code>	161
B.2	<code>libm3: Lex</code>	166
B.3	<code>libm3: Scan</code>	169
B.4	<code>m3core: Text</code>	169
B.5	<code>libm3: ASCII</code>	171
B.6	<code>libm3: Stdio</code>	173
B.7	<code>libm3: IO</code>	173
B.8	<code>libm3: Rd</code>	175
B.9	<code>libm3: Wr</code>	181
B.10	<code>libm3: FileRd</code>	185
B.11	<code>libm3: FileWr</code>	186
B.12	<code>libm3: File</code>	187
B.13	<code>m3core: Float</code>	189
B.14	<code>libm3: Math</code>	193

[B.15 libm3: Random](#) 196

Kapitel 1

Einführung in UNIX

1.1 Etwas über Betriebssysteme

1.1.1 Aufbau eines Computers

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:

- UNIX und seine Derivate (BSD, Solaris, LINUX, MacOS X ...)
- MacOS, RiscOS (Archimedes), AmigaOS, TOS (Atari) ...
- MS-DOS, Windows XP, Windows 2000, ...

1.1.2 UNIX

UNIX:

- Viele Programme laufen gleichzeitig (multi-tasking)
- Mehrbenutzerbetrieb, Netzwerk (multi-user-system)
- Rechnerunabhängigkeit, Portabilität
- leistungsfähige Benutzerschnittstellen (Shells)
- größtenteils in C programmiert

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 es eine Reihe graphischer Bedienoberfläche für UNIX-Betriebssysteme: KDE, Gnome, CDE, FVWM, ...

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

1.2 Die Etikette im Rechnerpool des Fachbereichsnetzes

- Transponderkarte 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 <mailto:service@informatik.uni-bremen.de>.
- 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.
- Ruhe bewahren, die Nachbarn möchten sich auch auf ihre eigenen Aufgaben konzentrieren.
- Abmelden nicht vergessen.

Weitere Informationen:

<http://www.informatik.uni-bremen.de/t/info/text/miteinander.html>

1.3 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 Beenden verlässt man zunächst KDE (Schalter unten rechts oder Menü). Damit ist man aber noch nicht endgültig abgemeldet. Deswegen muss man anschließend noch die UNIX-Sitzung verlassen.

-> exit

Am besten ist es, KDE gleich so zu starten: -> X.kde ; exit dann wird nach dem Beenden von KDE auch gleich die ganze Sitzung beendet.

Wenn man den Rechner nur kurzzeitig (!) verlässt, sollte man ihn zur Sicherheit sperren: Mausklick links, Menüpunkt „Lock Screen“, oder Schalter 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 ggf. 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

1.4 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 1.8) eingegeben

werden. Ein Terminal öffnet man über das KDE-Menü, man schließt es mit
-> `exit`

Kommandos besitzen unterschiedliche *Parameter* 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 ``-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

1.5 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.

1.6 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 (zurzeit 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, etwa:

Rank	owner (class)	Job	Files	Total size	(Time)
active	benner	79	titel.ps	132 438	bytes
1st	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 Druckkontingent kann man sich anschauen mit

```
-> pacc
```

1.7 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* oder *Schublade*, 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.

1.7.1 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 1.7.3).

```
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
```

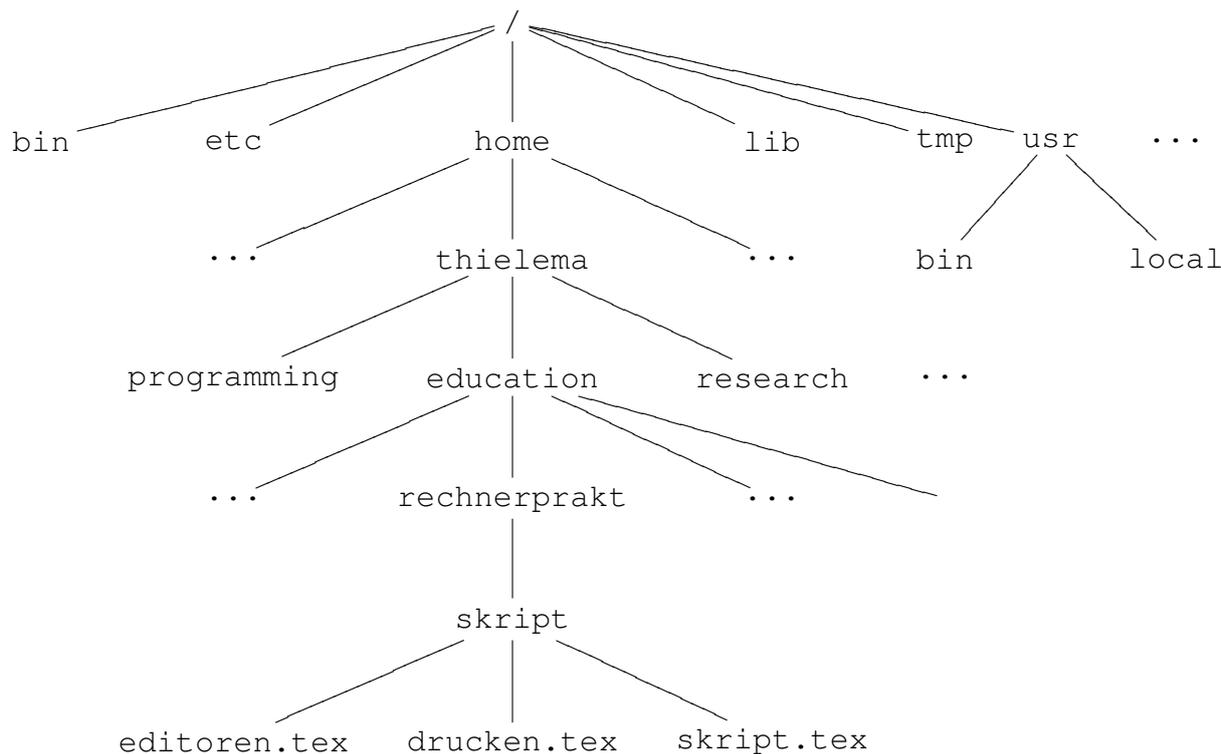
trotzdem ist der Punkt nicht überflüssig, denn wenn ein Kommando ein Verzeichnis als Argument erwartet, dann kann man mit einem einfachen Punkt das aktuelle Verzeichnis übergeben.

- 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
```

1.7.2 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 1.8) mit dem Pfad des Heimverzeichnisses als Wert. Nach dem Einloggen befindet man sich im eigenen Heimverzeichnis.

1.7.3 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 verzeichnis	make directory Anlegen von neuen Verzeichnissen
-> rmdir verzeichnis	remove directory Löschen von leeren Verzeichnissen

1.7.4 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 oder des Verzeichnisses,
- die Gruppe des Besitzers,
- die Größe der Datei (in Bytes),
- das Datum der letzten Änderung und
- den Namen der Datei oder 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 selbsterklärende Namen verwenden, damit man selbst (und vielleicht auch ein anderer) vermuten kann, was die Datei enthält. Oft sind Dateinamen als `name.endung` 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 von einem Übersetzer aus einer .c-Datei erzeugt,
name.tex	T _E X-Datei,
name.ps	Postscript-Datei,
name.doc	Word-Dokument,
name.html	Datei im HTML-Format,
.name	versteckte Datei: Systemdatei, Einstellungen, oder ähnliches

1.7.5 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. In der Ausgabe von `ls -l` erkennt man ein Verzeichnis 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. Die Zugriffsrechte kann man in *numerischer* oder *symbolischer* Form angeben.

```
chmod rechte dateiname change mode
```

1.7.6 Numerische und symbolische Form der Rechteänderung

Numerische Form

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 aller anderen
2	write	Schreibberechtigung aller anderen
1	execute	Ausführungsberechtigung aller anderen

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

-> `chmod 644 datei1`

die Rechte für diese Datei entsprechend.

Symbolische Form

Die symbolische Form der Rechteänderungen besteht aus Benutzerklasse, Operation und Zugriffsrecht:

[who] op permission

Dabei bedeuten die einzelnen Ausdrücke das Folgende:

who:	u	Eigentümer
	g	Gruppe
	o	andere
	a	Alle (Voreinstellung, falls „who“ weggelassen)
op:	+	Rechte hinzufügen
	-	Rechte wegnehmen
	=	nur diese Rechte einräumen und alle anderen widerrufen
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`

1.8 UNIX-Shells

Öffnet man ein Terminal-Fenster, dann läuft im Hintergrund eine *Shell*, die die eingegebenen Kommandos interpretiert und dann ausführt oder eine Fehlermeldung her ausgibt. 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 Merkmale der *bash*:

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

1.8.1 Sonderzeichen

Eine Reihe von Zeichen werden von der Shell besonders behandelt.

Solche Sonderzeichen sind z.B.

<code>;</code>	Kommandos hintereinander ausführen
<code>&&</code>	Kommandos hintereinander ausführen bis Fehler auftritt
<code>\$</code>	Shell-Variable abrufen Abschnitt 1.8.2
<code><</code>	Eingabeumleitung aus Datei, siehe Abschnitt 1.8.5
<code>></code>	Ausgabeumleitung in Datei
<code> </code>	Ausgabeweiterleitung zu anderem Programm
<code>&</code>	Programm vom Terminal abkoppeln, siehe Abschnitt 1.9
<code>`cmd`</code>	Ausgabe von Kommando <code>cmd</code> als Argument benutzen
<code>␣</code>	Leerzeichen
Zeilenende	Kommando starten.

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

```
-> echo \$PATH \; \& \> \| \\  
$PATH ; & > | \  
-> rm datei1 datei2 datei3 \  
-
```

```
> datei4 datei5
```

Setzt man einen Befehl oder einen Teil davon in einfache Hochkommata, wird dieser Text genauso, also ohne Ersetzungen übernommen.

```
-> echo \\' &<>\' a
\&<>\a
```

1.8.2 Shell-Variablen

Die Shell verwaltet eine Reihe von *Variablen*. Eine Variable hat einen Namen und einen Wert. Der Wert einer Variablen kann abgefragt werden, indem man dem Namen der Variablen ein \$ voranstellt, etwa \$HOME. Existiert die Variable nicht, gibt es leider keine Fehlermeldung, sondern als Wert wird der leere Text genommen. Schreibfehler können dadurch nur schwer aufgedeckt werden.

Mit `set` bekommt man eine Liste von definierten Variablen mit ihrem aktuellen Wert. Mit `export` setzt man den Wert einer Variablen.

```
-> export BLA=fasel
-> echo $BLA
fasel
```

Einige interessante Shell-Variablen:

HOME	Heimverzeichnis
HOST	Rechnername
PATH	Liste aller Verzeichnisse mit Kommandos
PS1	Format der Eingabeaufforderung (Prompt), z.B. \u@\h:\w>
PWD	aktuelles Verzeichnis
USER	Benutzername

1.8.3 Kommandos

Zu jedem UNIX-Kommando existiert eine entsprechende ausführbare Datei. Es gibt aber auch Shell-interne Kommandos, wie `export`. Eine ausführbare Datei ist in der Regel ein Maschinenprogramm, 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 sich aus ein paar festverdrahteten Orten wie `/bin`, `/usr/bin` und den in der Shellvariablen `PATH` aufgezählten Verzeichnissen zusammensetzt. Man kann sich diesen Pfad anzeigen lassen mit

```
-> echo $PATH
/usr/local/cm3/bin:/usr/lib/qt3/bin:/opt/kde3/bin
```

Die verschiedenen Verzeichnisse werden also mit Doppelpunkten getrennt.
Ein weiteres Verzeichnis kann man dem Pfad z.B. mit

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

hinzufügen, was zu einem neuen Wert der Variablen `PATH` führt.

```
-> echo $PATH
/usr/local/cm3/bin:/usr/lib/qt3/bin:/opt/kde3/bin:/home/name/bin
```

Nun schaut die Shell auch in `/home/name/bin` nach, ob dort eine ausführbare Datei des eingegebenen Namens existiert.

Erfahrene UNIX-Nutzer schreiben und benutzen sogenannte *Shell-Skripte*. Shell-Skripte bestehen aus einer Abfolge mehrerer UNIX-Kommandos. Sie werden (im Gegensatz zu Modula-3-Programmen) für die Ausführung nicht in Maschinenprogramme übersetzt.

1.8.4 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 beliebig viele Zeichen (keins und eins eingeschlossen).
- [] steht für ein Zeichen aus einem begrenzten Zeichenvorrat.

Beispiele:

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

1.8.5 Umleitung

Bis jetzt haben wir Kommandos immer über die Tastatur ein- und das Ergebnis immer auf dem Bildschirm ausgegeben. Man kann aber sowohl Ein- wie Ausgabe 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.

1.8.6 Vorverarbeitung der Kommandos

Noch bevor das eigentliche Kommando (Programm) 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 kann.

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/thielema
```

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 Automaten erlaubt, können sich auch schnell rächen. Beispiele:

1. 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.

2. Wir wollen eine Reihe von Dateien in ein anderes Verzeichnis verschieben, vergessen aber, das Zielverzeichnis anzugeben.

```
mv *.tex
```

Möglicherweise gibt es nur die Dateien `datei1.tex` und `datei2.tex`, die zu diesem Muster passen. Dann wird der Befehl

```
mv datei1.tex datei2.tex
```

ausgeführt, mit dem Ergebnis, dass `datei2.tex` unwiederbringlich durch `datei1.tex` ersetzt wird.

1.9 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 diverse vom System ausgelöste Prozesse, ohne dass man es merkt.

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. Startet man ein Programm ohne `&`, kann man es mit `<CTRL><z>` anhalten und dann mit dem Kommando `bg` unabhängig laufen lassen („in den Hintergrund verlagern“ = background). Mit `fg` wird es wieder in den Vordergrund geholt und blockiert die Shell.

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 möglicherweise Ressourcen nicht wieder freigibt.

1.10 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. 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
```

Viel einfacher geht es aber, wenn man `ssh` gleich mit der Option `-X` startet.

1.11 Weitere nützliche Befehle

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

```
find . -name '*.m3'
grep suchmuster datei
which kommando
quota
```

Suche nach Dateinamen

Suche nach Text in Datei

Suche nach dem Pfad von Kommandos

Zeigt an, ob man mehr Plattenplatz als erlaubt beansprucht. Bei Überschreitung der vorgegebenen harten Grenze kann man sein Benutzerkonto praktisch nicht mehr benutzen, da immer Platz für temporäre Dateien gebraucht wird.

<code>du -sk</code>	Berechnet den Speicherplatz, der vom Verzeichnis inklusiv Unterverzeichnis belegt wird.
<code>gzip datei</code>	Komprimiert <code>datei</code> , Ausgabe <code>datei.gz</code>
<code>gunzip datei.gz</code>	Dekomprimiert Datei
<code>tar -cvf archiv.tar ordner</code>	Legt Datei-Archiv an
<code>tar -tf archiv.tar</code>	Zeigt den Inhalt des tar-Archivs
<code>tar -xvf archiv.tar</code>	Entpackt Archiv
<code>scp nutzer@rechner:datei .</code>	Datei zwischen Nutzern oder Rechnern austauschen
<code>slogin rechner -l nutzer</code>	Anmelden auf einem anderen Rechner
<code>sftp rechner</code>	secure file transfer protocol – Dateien zwischen Rechnern austauschen
<code>a2ps -r -P drucker datei</code>	ASCII-Text im Querformat drucken
<code>grp -setup boerse</code>	Anlegen der Gruppe <code>boerse</code>
<code>grp -show boerse</code>	Anzeigen der Daten der Gruppe
<code>grp -invite boerse nutzer</code>	Einladen von Teilnehmern
<code>chgrp boerse BOERSE</code>	Ändern der Gruppenzugehörigkeit vom Verzeichnis <code>BOERSE</code>
<code>chmod 770 BOERSE</code>	Nur Mitglieder der Gruppe dürfen in das Verzeichnis
<code>chmod g+s BOERSE</code>	Alle angelegten Dateien gehören der Gruppe <code>boerse</code>

1.12 Versionsverwaltung

Ein Projekt, wie ein Programm oder ein \LaTeX -Dokument, entwickelt sich über die Zeit. Dabei entstehen viele Zwischenstände, die man festhalten will. Beispielsweise möchte man den Zustand wiederherstellen, in dem man ein Programm an andere weitergegeben hat, um so einen Fehler im Programm reproduzieren zu können. Oder man möchte einen Programmabschnitt wiederherstellen, den man bereits entfernt hat, nun aber wiederhaben möchte.

Ohne die Unterstützung durch eine Versionsverwaltung müsste man hin und wieder Kopien des Projektes anlegen. Der Datenumfang wächst auf diese Weise beträchtlich und wenn man kein konsequentes Schema beibehält, ist irgendwann nicht mehr klar, welche Dateien den aktuellen Stand widerspiegeln und welche veraltet sind.

Eine Versionsverwaltung gibt ein für alle Projektmitarbeiter einheitliches Konzept

zur Organisation der Projektversionen vor und unterstützt diese Organisation durch eine Reihe von Programmen.

Ein übermäßiges Anwachsen der Versionsdaten dämmen programmgestützte Systeme in der Regel dadurch ein, dass nur die Unterschiede zwischen zwei Versionen gespeichert werden. Die Ordnung wird dadurch sichergestellt, dass jede Datei nur einmal geführt wird und die unterschiedlichen Versionen zusammen mit einer Bemerkung getrennt archiviert werden.

1.12.1 Übersicht über verbreitete Versionsverwaltungssysteme

Es gibt bereits eine unüberschaubare Vielzahl von Versionsverwaltungssystemen: <http://better-scm.berlios.de/comparison/comparison.html>. Hier eine Auswahl frei verfügbarer Systeme:

- Revision Control System (RCS)

<http://www.gnu.org/software/rcs/rcs.html>

Verwaltet die Geschichte jeder Datei einzeln. Speichert nur Unterschiede zwischen Versionen und zwar speichert es die aktuelle Version immer vollständig ab und dazu eine Beschreibung, wie aus der aktuellen Version die vorhergehende Version konstruiert werden kann. Arbeitet ein Projektmitarbeiter an einer Datei, sind alle anderen ausgeschlossen. Umbenennungen einer Datei lassen sich nicht verwalten. Benennt man eine Datei um, können ältere Versionen des Projektes nicht ohne weiteres wieder hergestellt werden. Alternativ kann man die Datei formal löschen und unter neuem Namen wieder anmelden. Dann geht aber der Zusammenhang zwischen den beiden Dateien verloren.

- Concurrent Version System (CVS)

<https://www.cvshome.org/>

CVS verwendet das RCS-Dateiformat. Jeder Mitarbeiter hält eine Kopie des Projektes (aber nicht der Versionsgeschichte) und es kann jeder an seinen Kopien ungestört arbeiten. Bei mehreren Projektmitarbeitern kann jeder an seiner Kopie der gleichen Datei arbeiten. CVS hilft dabei, die Änderungen aller Mitarbeiter in einer Datei zu vereinigen und bittet den Benutzer um Hilfe, wenn die gleiche Textstelle auf verschiedene Weise geändert wurde.

Die Versionsdateien werden zentral verwaltet. Zum Einbringen neuer Änderungen braucht man häufig einen Netzzugang.

- Subversion (SVN)

<http://subversion.tigris.org/>

Subversion ist in der Bedienung an CVS angelehnt. Dateien werden nicht mehr

einzelnen behandelt, sondern ein Projekt als Ganzes. Somit ist es auch möglich Verschiebungen und Umbenennungen von Dateien und Verzeichnissen zu verwalten.

- David's advanced revision control system (Darcs)

<http://darcs.net/>

Darcs ist eine aktuelle (und noch nicht ganz ausgereifte) Entwicklung. Es ist eine in der funktionalen Programmiersprache Haskell (<http://www.haskell.org/>) geschriebene Versionsverwaltung. Es geht gänzlich andere Wege als RCS, CVS und Subversion und verwaltet nicht verschiedene Versionen eines Projektes sondern Unterschiede zwischen Projektversionen. Dies erlaubt es, eine Projektkopie zu erstellen, in der manche Entwicklungsschritte ausgelassen sind. Es können ebenfalls Verschiebungen und Umbenennungen von Dateien und Verzeichnissen verwaltet werden und darüber hinaus auch Umbenennungen von Bezeichnern in Programmtexten.

Jeder Projektmitarbeiter besitzt sowohl eine Kopie der Arbeitsdateien als auch der Versionsdateien. Der Speicheraufwand ist dadurch größer, aber die Entwicklung wird nicht behindert, wenn der Netzzugang gestört ist. Man kommt sogar ohne zentralen Rechner aus, der die Versionsdateien speichert.

1.12.2 Arbeiten mit einem Darcs-Projekt

Die folgenden Kommandos sind das grundlegende Handwerkszeug, um mit `darcs` ein Projekt auf einem Rechner verwalten zu können.

- `darcs initialize`
Dieser Befehl legt im aktuellen Verzeichnis das Verzeichnis `_darcs` an, in welches später unter anderem die Projektänderungen gespeichert werden.
- `darcs add datei1 datei2 ...`
Weist `darcs` an, die Dateien `datei1`, `datei2`, ... zu verwalten. Nicht-verwaltete Dateien werden von `darcs` ignoriert, Änderungen an ihnen werden nicht gespeichert. Es sollten nur diejenigen (Text-)Dateien verwaltet werden, an denen man selbst schreibt. Von Übersetzern erzeugte Dateien sollten nicht verwaltet werden, denn diese sind rechnerabhängig. Beispiel: Von \LaTeX -Projekten sollten die `*.tex`-Dateien verwaltet werden, nicht aber die `*.dvi`-, `*.pdf`- oder `*.dvi`-Dateien. Von Modula-3-Projekten sollte nur das `src`-Verzeichnis mit Inhalt verwaltet werden, die vom Übersetzer angelegten Verzeichnisse, wie `LINUXLIBC6` und `SOLgnu` dagegen nicht.

- `darcs record -m "Bemerkungen zu Änderung"`
Hiermit kann man den aktuellen Projektstand sichern. Jeder Projektstand entspricht einer Version. Jede Version sollte ein lauffähiges Programm oder ein übersetzbares (L^AT_EX-)Dokument darstellen. Andernfalls bekommen die anderen Projektmitarbeiter eine unbenutzbare Version, wenn sie sich diese Version abholen.
Den Projektstand sollte man nach jeder Änderung sichern, die für sich genommen einen Sinn ergibt.
- `darcs whatsnew -s`
Zeigt an, welche Dateien seit dem letzten gespeicherten Projektstand geändert wurden.
- `darcs revert`
Geht alle Änderungen seit dem letzten gespeicherten Projektstand durch, und fragt welche zurückgenommen werden sollen.
- `darcs changes`
Gibt eine Liste aller bisherigen Versionen zusammen mit den Bemerkungen aus.
- `darcs unrecord`
Löscht den letzten Projektstand lässt aber alle Projektdateien unverändert. Das ist nützlich, wenn man merkt, dass man sich bei der Bemerkung zur Version verschrieben hat, oder dass Dateien vergessen wurden.
- `darcs unpull`
Löscht den letzten Projektstand und setzt die Dateien auf den vorigen Stand zurück. Damit sind alle Änderungen, die für den letzten Projektstand vorgenommen wurden verloren. Dieses Kommando ist vor allem dann nützlich, wenn man auf einer Kopie des `darcs`-Projektes arbeitet und zu alten Versionen zurückkehren will.

Ändert man den Inhalt von Dateien oder löscht sie, so erkennt `darcs record` dies und konserviert diese Änderungen. Es gibt aber andere Arten von Änderungen, die sich nicht erkennen lassen und die deswegen ausdrücklich über `darcs` abgewickelt werden müssen.

- `darcs mv alt neu`
Verschiebt oder ändert den Namen eines Verzeichnisses oder einer Datei von `alt` in `neu`. Die Änderung an den Dateien wird von `darcs` selbst vorgenommen und bis zum nächsten `darcs record` zwischengespeichert.

- `darcs replace alt neu datei`
Ersetzt in der Datei `datei` alle Wörter `alt` durch `neu`. Groß- und Kleinschreibung werden unterschieden, es werden nur ganze Wörter und keine Wortbestandteile ersetzt und es wird nur ersetzt, wenn es `neu` in dem Text noch nicht gibt. (Diese Maßnahme ist grob gesprochen notwendig, damit die Ersetzung auch umgekehrt werden kann.)

Natürlich kann man auch mit einem Texteditor Wörter ersetzen. Tut man dies, so registriert `darcs record` lediglich die Änderungen, die sich dadurch in jeder Zeile ergeben. Es kann aber nicht erkennen, dass ein Wort überall ersetzt wurde. Benutzt man hingegen `darcs replace` so wird die Wortersetzung registriert und entsprechend vorgenommen, wenn diese Änderung an andere Mitarbeiter geschickt wird.

1.12.3 Austausch zwischen Darcs-Projekten

In diesem Abschnitt wenden wir uns Kommandos zu, mit denen wir `darcs`-Projekte übertragen können und Änderungen an Projekten an verschiedenen Standorten untereinander austauschen können.

Man kann mit `darcs` verschiedene Modelle verfolgen.

1. Ein zentraler vernetzter Rechner verwaltet das Referenzprojekt. Jeder Mitarbeiter muss seine Änderungen an dieses Projekt senden, und die Änderungen der anderen Mitarbeiter dort abholen.
2. Wenn es nur wenige Projektmitarbeiter gibt, oder ein Entwickler auf verschiedenen Rechnern arbeitet, die nur gelegentlich vernetzt sind, dann kann man auch auf einen zentralen Rechner verzichten, und die Änderungen direkt an die anderen Projektmitarbeiter oder die anderen Rechner übermitteln.

Die folgenden Kommandos erlauben verschiedene Arten des Austausches.

- `darcs get url`
Dieses Kommando holt eine Kopie eines kompletten `darcs`-Projektes auf den eigenen Rechner. Die Adresse `url` kann eine HTTP-Adresse sein, die Adresse auf einem Rechner mit SSH-Zugang oder auch ein Pfad zu einem `darcs`-Projekt auf dem eigenen Rechner.

Beispiele:

- `darcs get $HOME/darcs/haskore/`
Projekt auf dem eigenen Rechner

- `darcs get nutzer@rechner:/home/nutzer/haskore/`
Projekt auf einem anderen Rechner mit SSH-Zugang
- `darcs get http://cvs.haskell.org/darcs/haskore/`
Projekt auf einem Web-Server
- `darcs pull url`
Innerhalb eines `darcs`-Projektverzeichnisses aufgerufen holt dieses Kommando Änderungen an dem Projekt unter der Adresse `url` ab. Nachdem man einmal von dieser Adresse Änderungen geholt hat, braucht man im folgenden `url` nicht mehr anzugeben.
Möglicherweise hat ein anderer Mitarbeiter an der gleichen Datei Änderungen vorgenommen wie man selbst. Dies bezeichnet man als einen *Versionskonflikt*. Ein Versionsverwaltungssystem ist nicht in der Lage dieses Problem selbständig zu beheben. Es markiert daher in den betreffenden Textdateien die Stellen, an denen zwei unvereinbare Änderungen vorgenommen wurden. Bei `darcs` wird eine Konfliktstelle mit `v v v v v` eingeleitet und mit `^ ^ ^ ^ ^` abgeschlossen. Bevor man weiterarbeiten kann, muss man nach Rücksprache mit dem Urheber der konkurrierenden Änderung die beiden Änderungen zusammenbringen und die Konfliktstellenmarkierung entfernen.
- `darcs push url`
Schickt Änderungen eines `darcs`-Projektes an das `darcs`-Projekt unter der Adresse `url`. Das Kommando `push` hat fast den gleichen Effekt, wie wenn man sich in das Verzeichnis `url` begibt und dort `darcs pull` aufruft. Ein wesentlicher Unterschied ist, dass `push` den Versand der Projektänderungen ablehnt, wenn sich dadurch Konflikte ergeben. In diesem Falle muss man erst mit `pull` die Änderungen von dem fremden Standort abholen, die entstehenden Konflikte in der eigenen Projektkopie beheben, diese als neue Änderung konservieren und dann erneut verschicken.
- `darcs send url`
Verschickt per E-Mail Änderungen eines `darcs`-Projektes bezogen auf das `darcs`-Projekt unter der Adresse `url`. Funktioniert der E-Mail-Versand nicht direkt, kann man auch eine Änderungsdatei `datei` anlegen
`darcs send -output=datei url`
und diese per E-Mail verschicken.
- `darcs apply datei`
Mit diesem Kommando kann der Empfänger der Änderungsdatei die Änderungen in seine Projektkopie einarbeiten.

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. Der Algorithmus selbst 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. Diese Methode 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

- Berechenbarkeit: Kann es einen Algorithmus geben, der das gegebene Problem löst?
- Korrektheit: Löst der Algorithmus die gegebene Aufgabe korrekt?
- Komplexität: Vergleich von Algorithmen nach Zeit- und Speicherbedarf

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 parallelem Algorithmus),
- 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 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 <i>Fülle Kessel</i> | 2.1 <i>Öffne Kaffeeglas</i> |
| 1.2 <i>Schalte Kessel an</i> | 2.2 <i>Entnimm einen Löffel Kaffee</i> |
| 1.3 <i>Warte, bis es kocht</i> | 2.3 <i>Kippe Löffel in die Tasse</i> |
| 1.4 <i>Schalte Kessel aus</i> | 2.4 <i>Schließe Kaffeeglas</i> |

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, wenn das Kaffeeglas leer ist und was ist zu tun, wenn mehrere Tassen Kaffee gekocht werden sollen?

2.2 Auswahl (Selektion)

Man benötigt ein Konstrukt, das 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 Kaffeekochen mit Entscheidung

Verfeinerung von Schritt 2.1 durch Auswahl

2.1.1 *Nimm Kaffeeglas aus dem Fach*

2.1.2 **FALLS** *Kaffeeglas leer ist*
DANN *Nimm 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 *Nimm Kaffeeglas aus dem Fach*

2.1.2 **FALLS** *Kaffeeglas leer ist*
FALLS *Weiteres Kaffeeglas vorhanden*
DANN *Nimm neues Kaffeeglas*
SONST *Brich 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 – *Schleifen*. Beliebte sind folgende drei Arten:

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

Achtung: Man muss die Abbruchbedingung so formulieren, dass die Schleife in jedem Fall nach endlich vielen Durchläufen beendet wird. Notfalls muss man eine maximale Anzahl an Schritten vorgeben. Ebenso ist zu vermeiden, dass die Schleife weiter durchlaufen wird, obwohl die Schritte in der Schleife nicht mehr ausführbar sind.

2.3.1 Kaffeekochen mit Wiederholung

Schritt 2.1 mit Wiederholungsschleife

2.1.1 WIEDERHOLE

Nimm Kaffeeglas aus dem Fach
BIS *Kaffeeglas gefüllt oder Gläser alle*

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

2.3.2 Maximumsbestimmung

Eingabe: Liste von Zahlen

Ausgabe: Größte Zahl in der Liste

Setze erste Zahl der Liste als bislang größte Zahl

WIEDERHOLE

Lies 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 *weitere Zahl in Liste* **FÜHRE AUS**

Lies 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 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:

$$\text{ggT}(x, y) = \text{ggT}(y, x)$$

$$\text{ggT}(x, 0) = x$$

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 Berechnung von Potenzen x^n

Eingabe: Reelle Zahl x , natürliche Zahl n

Ausgabe: x^n

Übernimm die Werte von x und n

Setze Produkt auf 1

FÜR k **VON** 1 **BIS** n

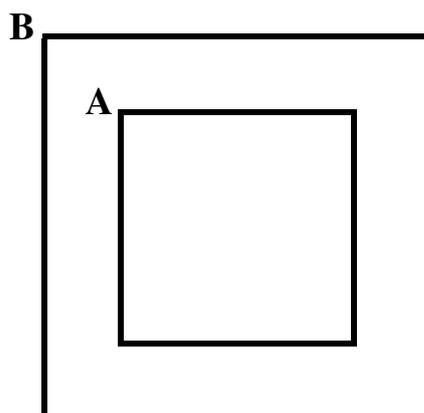
Multipliziere Produkt mit x

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
<code>forward(x)</code> <code>right(α)</code> <code>down(Y)</code> <code>up</code>	Bewege Stift um x vorwärts Drehe Stift um α nach rechts Senke Stift auf den Punkt Y Hebe Stift vom Papier ab
<code>ZeichneQuadrat(x, Y)</code>	<code>down(Y)</code> WIEDERHOLE vier mal <code>forward(x)</code> <code>right($\pi/2$)</code> <code>up</code>

2.4.1 Zeichnen zweier konzentrischer Quadrate

`ZeichneQuadrat(10, A)`

`ZeichneQuadrat(15, B)`

2.5 Vom Algorithmus zum Programm

Computer haben eine eigene, vom Typ der CPU abhängige *Maschinsprache*. Unter *Assembler* versteht man die mnemotechnische Umsetzung der Maschinsprache: 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.

- imperative Sprachen
Pascal, Modula-2, FORTRAN, C, Basic, Algol, Perl, PHP, ...
- imperative objektorientierte Sprachen
Modula-3, Ada, C++, Java, Python, ...
- funktionale Programmiersprachen
Haskell, ML, OCaml, LISP, ...
- Programmiersprachen für spezielle Anwendungen
SQL (Datenbanken), HPC Fortran (Parallelrechnen) oder MATLAB (lineare Algebra)

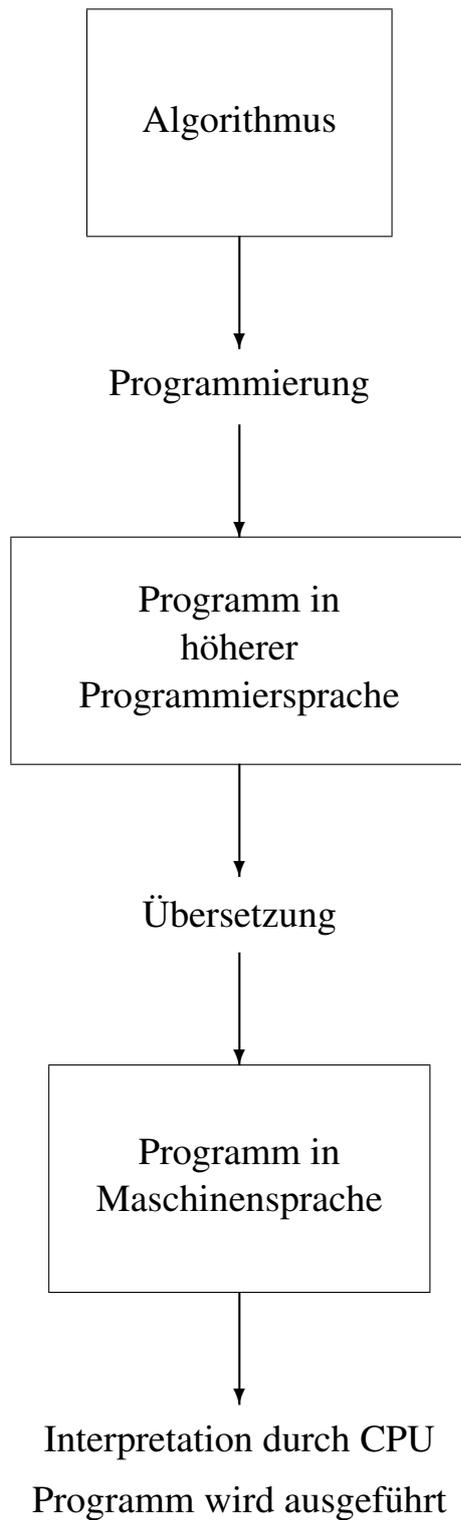
Höhere Programmiersprachen sind in der Regel rechnerunabhängig, ihre Beherrschung erfordert die Kenntnis von

- 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. Das entsprechende Programm muss in die Maschinsprache des Prozessors übersetzt (*kompiliert*) werden, so dass ein ausführbares Programm, also eine Folge von Befehlen in Maschinsprache entsteht.

- Compiler: Der gesamte Quelltext wird übersetzt und ein ablauffähiges Programm erzeugt.
- Interpreter: Es wird je ein Befehl übersetzt und abgearbeitet.
- Mischformen: Just-In-Time-Compiler, Byte-Code-Interpreter

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



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.
- ... eine imperative Programmiersprache: Programme bestehen aus aneinandergereihten Anweisungen.
- ... eine strukturierte Programmiersprache: Es gibt kein GOTO, dafür Unterprogramme, Schleifen, Abfragen, etc.
- ... sehr streng, fördert guten Programmierstil, deckt viele Fehler noch vor Programmstart auf und ist auf Grund seiner Modularisierung sehr gut für große Projekte geeignet.
- ... eine moderne objektorientierte Programmiersprache: Unterstützt werden Objektklassen (nur einfaches Erben), Ausnahmen (Exceptions), automatische Speicherverwaltung (Garbage-Collector), Programmierung allgemeiner Datenstrukturen (Templates), nebenläufige Ausführung (Threads).
- ... eine Systemprogrammiersprache: Es lassen sich sehr effiziente maschinennahe Programme schreiben, z.B. Gerätetreiber oder Betriebssysteme, aber systemnahe und höhere Programmteile sind streng getrennt.
- ... trotzdem 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. den Befehl `IO.Put`, der von `Modula-3` zusammen mit anderen Befehlen zur Ein-/Ausgabe in dem Modul `IO` zur Verfügung gestellt wird. Mit dem Befehl **IMPORT** wird das Modul für das Programm sichtbar gemacht.

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 werden Fehler gemeldet? 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 für einen Zeilenumbruch eingebaut werden. Dies geschieht mit dem Steuerzeichen `\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. (Komplette Übersicht in Abschnitt [A.3](#))

3.1.5 Projektstruktur

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 [3.12](#))

Die Hierarchie der Projektkomponenten sieht damit von oben nach unten so aus:

Ebene	Beispiel
Paket	
- Programm	cm3 (der Übersetzer)
- Bibliothek	m3core, libm3
- sonstiges	Sprachunterstützung für Texteditor
Modul	IO, Fmt (enthalten in libm3)
Unterprogramm	Put, GetLine (enthalten in IO)

3.1.6 Unterprogramme und Funktionen

Unterprogramme fassen häufig benötigte Arbeitsschritte zusammen. Befehle wie `IO.Put` sind nicht in der `Modula-3`-Sprachdefinition vorgesehen und nicht im Übersetzer integriert. Sie sind in einer Standardbibliothek zusammengefasst, also in regulären Modulen abgelegt und meist selbst in `Modula-3` geschrieben. Wie das geht, lernen wir in Abschnitt 3.4.

Die Arbeitsweise eines Unterprogrammes lässt sich durch *Parameter* steuern. Für `IO.Put` ist der Steuerparameter der auszugebende Text. Der Wert, der `IO.Put` übergeben wird heißt *Argument*.

Ein Unterprogramm kann auch einen Wert zurückgeben. In diesem Falle nennt man es auch *Funktion*. Ein Beispiel dafür ist `Math.sin`.

- Unterprogrammaufrufe enthalten immer `()`. In diesen Klammern stehen die Werte, die an die Funktion übergeben werden.
- Der Aufruf eines Unterprogrammes ohne Rückgabewert entspricht einer Anweisung. Daher wird er mit einem Semikolon abgeschlossen.

```
IO.Put("Autofensterklorollnhäkelmütz");
```

- Der Aufruf einer Funktion entspricht dem Funktionswert. Der Ausdruck `x - Math.cos(0.0D0)` ist also gleichbedeutend mit `x - 1.0D0`. Die Schreibweise der Zahlen wird in Abschnitt 3.2.4 eingehend behandelt.
- Führt eine Funktion neben der Berechnung des Funktionswertes noch andere Arbeitsschritte aus und sind diese Arbeitsschritte wichtiger als der Funktionswert, so kann man den Funktionswert mit **EVAL** ignorieren.

```
EVAL Math.cos(0.0D0);
```

Die Verwendung von **EVAL** ist allerdings ein Indiz für eine schlecht konzipierte Funktion. (Abschnitt 3.4.12)

3.1.7 Bezeichner

Funktionsnamen, aber auch Variablen-, Konstanten-, Typennamen und Namen anderer `Modula-3`-Objekte nennt man *Bezeichner*.

- Bezeichner müssen mit einem Buchstaben anfangen und dürfen nur Buchstaben, Ziffern, Unterstriche enthalten, sie enthalten also keine Interpunktionszeichen und sind mit Wörtern natürlicher Sprachen vergleichbar.

- Groß- und Kleinschreibung werden unterschieden, z.B. sind `main`, `Main`, `mAin`, `MAIN` unterschiedliche Namen.
- `Modula-3`-Schlüsselwörter sind keine Bezeichner, aber Bezeichner dürfen Schlüsselwörter enthalten. So ist `BEGINNER` erlaubt, obwohl es **BEGIN** und **IN** enthält. Aber solch ein Bezeichner ist kein guter Stil, denn durchgehend großgeschriebene Wörter sehen aus wie Schlüsselwörter und könnten tatsächlich in Spracherweiterungen neue Schlüsselwörter sein.
- Beispiele für erlaubte Bezeichner:
`a`, `A`, `a0`, `in`, `center`, `Newton2`
- Beispiele für verbotene Bezeichner:
`0a`, **IN**, `center.x`, `Newton-2`

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 umgehend „dumme“ Fehler im Programm auf!

Man muss unterscheiden:

- Objekt (Datum): 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: Variable, die immer das gleiche Objekt enthält.

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 Textschnipsel, die ein Objekt eines bestimmten Typs repräsentieren. Zum Beispiel ist `'A'` ein Literal für ein Zeichen, es 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 (einfachgenaue) *Gleitkommazahl* mit Wert 12. Sobald man einer Zahl Nachkommastellen verpasst, betrachtet der Compiler sie als Gleitkommazahl. (Abschnitt 3.2.4)

Variablen müssen vor ihrer Benutzung *deklariert* werden. Damit wird zum einen Speicher für die Variable reserviert und zum anderen wird ein Name (Abschnitt 3.1.7) 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. (Abschnitt 3.2.7)

- `c` wird mit einer verkürzten Schreibweise deklariert und initialisiert. Das ist immer dann sinnvoll, wenn sich aus dem Initialwert der passende Typ ergibt, hier ergibt sich aus **TRUE** der Typ **BOOLEAN** (Wahrheitswert, Abschnitt 3.2.6).

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 3.6) 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.2). 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

$$a \text{ **DIV** } b * b + a \text{ **MOD** } b = a.$$

So verhalten sich die Operationen bei negativen Zahlen:

$$\begin{array}{ll} 13 \text{ **DIV** } 4 = 3 & 13 \text{ **MOD** } 4 = 1 \\ -13 \text{ **DIV** } 4 = -4 & -13 \text{ **MOD** } 4 = 3 \\ 13 \text{ **DIV** } -4 = -4 & 13 \text{ **MOD** } -4 = -3 \\ -13 \text{ **DIV** } -4 = 3 & -13 \text{ **MOD** } -4 = -1 \end{array}$$

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 keine Konstante 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 (*Überlauf*, *Overflow*)?

Es wird ein Fehler ausgelöst, der das Programm abbricht, wenn er nicht behandelt wird. (Siehe Abschnitt 3.11)

Problem: Die meisten Modula-3-Compiler bauen auf Codeerzeugern von C-Compilern auf. Da C keine Überlaufabfrage kennt, können diese Codeerzeuger ebenfalls keinen Überlaufstest ins Programm einbauen und so bleiben Überläufe über die größte mit **INTEGER** darstellbare Zahl hinaus unerkannt!

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 ;

3.2.4 Gleitkommazahlen

Außer ganzen Zahlen werden auch gebrochene Zahlen unterstützt. Diese umfassen einen Bereich mehrerer Größenordnungen aber stets gleichbleibender relativer Genauigkeit. Intern werden rationale Zahlen auf dem Computer im *Gleitkommaformat*

in der Form

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

mit Ziffern $z_i \in \{0, \dots, B - 1\}$ mit $z_1 \neq 0$ und einem beschränkten Exponenten e mit $e \in \mathbb{Z}$ dargestellt. Aus Effizienzgründen wählt man meistens die Basis $B = 2$.

Man könnte natürlich auch ein Zahlenformat der Form

$$\pm z_n z_{n-1} \dots z_0 . z_{-1} \dots z_m$$

so genannte *Festkommazahlen* verwenden, dieses könnte aber nicht mehrere Größenordnungen abdecken.

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-Gleitkommazahlen*, 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 Gleitkommaformat 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. Schließlich gibt es eine ganze mathematische Disziplin, die numerische Mathematik, welche sich intensiv solchen Problemen widmet.

Konkret gibt es in Modula-3 drei Datentypen für Gleitkommavariablen:

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

Die Zahlen hinter E, D und X sind die Exponenten der Zehnerpotenzen in der Exponentialschreibweise. Also kann die Zahl 123,456 eingegeben werden als 123.456, 123.456E0, 123.456D0 oder 123.456X0 (je nach gewünschter Genauigkeit) oder, da $1,23456 \cdot 10^2$ die gleiche Zahl bezeichnet, auch 1.23456E2,

1.23456D2 oder 1.23456X2. Den Ausdruck $123456 \cdot 10^{-3}$ setzt man entsprechend in 123456.0E-3, 123456.0D-3 oder 123456.0X-3 um.

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 (oder eben $0.5D0$).

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.

Es gibt noch ein paar andere häufig benötigte mathematische Funktionen, die direkt in der Sprache Modula-3 integriert sind. In der folgenden Tabelle steht `Number` für einen Zahlentyp wie **INTEGER**, **CARDINAL**, $[-10..10]$, **REAL**, **LONGREAL**, **EXTENDED** und `Float` für einen Gleitkommazahlentyp wie **REAL**, **LONGREAL**, **EXTENDED**.

ABS (x: Number;): Number	Berechnet den Absolutbetrag von x.
MAX (x, y: Number;): Number	Ermittelt die größere der beiden Zahlen.
MIN (x, y: Number;): Number	Ermittelt die kleinere der beiden Zahlen.
FLOAT (x: Number; T: Type := REAL ;) : T	Konvertiert eine Zahl in eine Gleitkommazahl des Typs T .
FLOOR (x: Float;): INTEGER	Rundet eine Gleitkommazahl nach unten.
CEILING (x: Float;): INTEGER	Rundet eine Gleitkommazahl nach oben.
ROUND (r: Float;): INTEGER	Rundet eine Gleitkommazahl zur nächsten ganzen Zahl.
TRUNC (r: Float;): INTEGER	Rundet eine Gleitkommazahl in Richtung 0.

Weitere spezielle Gleitkommafunktionen sind in Abschnitt [A.4](#) aufgeführt.

Formatierung

Mit den Routinen aus dem Modul `Fmt` (Abschnitt [B.1](#)) kann eine Gleitkommazahl im Festkommaformat (z.B. `x=123.456`) oder in Exponentialschreibweise (z.B. `x=0.123456e3`) ausgegeben werden, wobei angegeben werden kann, wieviele Ziffern nach dem Dezimalpunkt verwendet werden sollen. Einige Beispiele:

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

3.2.5 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};
Modelfarbe = {Mauve, Gruen, Schaumolweiss,
              Rotbraungrau};
Status      = {Pause, MaustasteLosgelassen,
              MaustasteGedrueckt};
BOOLEAN   = {FALSE, TRUE};

```

(* nur zur Veranschaulichung *)

Der Typ **BOOLEAN** ist bereits vordefiniert und wird von Operatoren wie **AND** und **OR** verwendet. (Siehe Abschnitt [3.2.6](#)) Den Aufzählungselementen werden intern aufsteigend ganze Zahlen bei 0 beginnend zugeordnet. Zum Beispiel entspricht `Grundfarbe.Blau` dem Wert 2. Von diesem Wissen sollte man allerdings so wenig wie möglich Gebrauch machen.

Genau wie ganze Zahlen kann man Aufzählungselemente miteinander vergleichen, kann man Aufzählungen in **FOR**-Schleifen durchlaufen (Abschnitt [3.5.3](#)), aus ihnen

Mengen bilden (Abschnitt 3.2.8) und sie als Indexmenge für Felder (Abschnitt 3.6) verwenden.

Aufzählungselemente sind also wie Ganzzahlkonstanten mit dem Unterschied, dass man mit ihnen nicht rechnen kann und dass sie unverträglich mit ganzen Zahlen und den Elementen anderer 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 3.2.10) 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, indem man auf ein konstantes Feld mit dem Aufzählungswert als Index zugreift. (siehe Abschnitt [3.6](#))

3.2.6 Wahrheitswerte

Wahrheitswerte sind vom Typ **BOOLEAN** und können falsch (**FALSE**) oder wahr (**TRUE**) sein. Bedingungen, wie sie in Auswahlanweisungen oder Schleifen verwendet werden, sind immer Wahrheitswerte. (Abschnitt [3.3.1](#))

Der besseren Lesbarkeit wegen sollte man **BOOLEAN**-Variablen immer positive Bezeichnungen geben, etwa `lampOn`, `showSplashScreen`, `playJingle`, `textIsComprehensible` statt `lampOff`, `hideSplashScreen`, `skipJingle`, `textIsIncomprehensible`. Um zu erschließen, was es bedeutet, wenn eine Variable wie `lampOff` den Wert **FALSE** hat, braucht der Leser nämlich deutlich länger. Das wird noch verschlimmert, wenn viele **BOOLEAN**-Variablen definiert sind, bei denen diese Orientierung uneinheitlich ist.

Relationen

Wahrheitswerte erhält man bei *Vergleichen* und *logischen Verknüpfungen*, wie zum Beispiel der Verneinung.

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

Wichtig ist, dass die Operanden immer verträglich sein müssen. Wenn wie in Abschnitt [3.2.5](#) 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 1.6 2005/02/24 15:18:01
   thielema Exp $ *)

IMPORT IO, Fmt;

CONST
  (* Wer hat recht? *)
  grundschueler = 5 * 7 = 35;
  abiturient    = 5 * 7 = 36;

BEGIN
  IO.Put (
    Fmt.F (
      "Die Antwort des Grundschülers ist %s\n"
      & "Die Antwort des Abiturienten ist %s\n",
      Fmt.Bool(grundschueler), Fmt.Bool(abiturient)));
END Main.

```

Logische Operatoren

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

```

Wegen des Vorrangs von = vor **NOT** 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

Die Vorrangreihenfolge ist:

1. Arithmetische Operationen (+, *, ...)
2. Relationen (<, =, ...)
3. Logische Operationen (**NOT**, **AND**, ...)

Eine vollständige Liste gibt es in Abschnitt [A.2](#).

Intervalltest

Beispiel: Wie programmiert man den Test, ob eine Zahl in einem Intervall enthalten ist?

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:

$$x \in [20, 40] \iff 20 \leq x \text{ AND } x \leq 40$$

$$x \notin [20, 40] \iff x < 20 \text{ OR } 40 < x$$

Verzögerte Auswertung

Verknüpfungen mit **AND** und **OR** werden von links nach rechts abgearbeitet. Die Auswertung endet, sobald der Wert des Ausdrucks feststeht (*Verzögerte Auswertung* oder auch *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 Programmablauf abhängen. Beispiel: Wenn $y = 0$ dann führt der Ausdruck $x \text{ MOD } y$ zum Programmabbruch wegen einer Division durch 0. Wenn in einer konkreten Situation im Programm $y=0$ als „nicht teilbar“ zu werten ist, dann ist für die Relation „ x ist teilbar durch y “ der korrekte Ausdruck $y \neq 0 \text{ AND } x \text{ MOD } y = 0$, denn der rechte Operand von **AND** wird nur ausgewertet, wenn $y \neq 0$. Dagegen führt der Ausdruck $x \text{ MOD } y = 0 \text{ AND } y \neq 0$ nach wie vor unter Umständen zum Programmabbruch.

3.2.7 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 Programmablaufes 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)];

(* nur zur Veranschaulichung *)

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.

3.2.8 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'};

TYPE
  Lottozahlen = [1 .. 49];
  Lottoziehung = SET OF Lottozahlen;

CONST
  Glueckskombination =
    Lottoziehung {1, 4, 9, 16, 25, 36};
```

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		$x \text{ OR } y$
-	Mengendifferenz		$x \text{ AND NOT } y$
*	Schnitt		$x \text{ AND } y$
/	symmetrische Differenz	$A/B = (A-B) + (B-A)$	$x \# y$
Relationen			
IN	Element von	$a \text{ IN } A$ - a enthalten in A	
\leq	Teilmenge	$A \leq B = A-B = \{ \}$	NOT $x \text{ OR } y$
$<$	echte Teilmenge	$A < B = A \leq B \text{ AND } A \# B$	
\geq	Obermenge	$A \geq B = B-A = \{ \}$	$x \text{ OR NOT } y$
$>$	echte Obermenge	$A > B = A \geq B \text{ AND } A \# B$	

Hinweis: ‚#‘ steht für ‚ungleich‘ (Abschnitt 3.2.6), x steht für $e \text{ IN } A$, y steht für $e \text{ IN } B$ und der Ausdruck in der Spalte „Logik“, Abschnitt „Operationen“ ist äquivalent zum Ausdruck $e \text{ IN } A \text{ op } B$, wobei *op* die betrachtete Operation ist.

3.2.9 Was ist der richtige Typ für mich?

Bei so vielen verschiedenen Typen fragt man sich zurecht, welcher Typ wofür am besten geeignet ist.

REAL, LONGREAL	Physikalische Größen
CARDINAL	Anzahlen
INTEGER	Ganze Zahlen, Werte mit denen man natürlicherweise rechnen kann
BOOLEAN	Antworten auf ja/nein Fragen, ein/aus-Schalter
SET	Antworten auf einen Satz von ja/nein Fragen, auch Optionen genannt. Alternative zu mehreren BOOLEAN -Werten.
Aufzählung	Eine Auswahl an Alternativen, die möglicherweise eine natürliche Ordnung haben ($<$, $>$), aber mit denen man nicht rechnen kann, etwa addieren oder multiplizieren. Beispiel: Wochentage, Monate, Kartentypen beim Skat, Programmstatus. Alternative zu BOOLEAN falls logische Operationen auf dem Typ keinen Sinn haben. Alternative zu SET falls sich die Optionen gegenseitig ausschließen.

3.2.10 Typumwandlungen

Modula-3 ist sehr streng, was Typen anbetrifft. Ganze Zahlen, Gleitkommazahlen, 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 Gleitkommazahlen etwas andere Eigenschaften. Auslöschungsfehler und Rundungsfehler können nur Gleitkommazahlen 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)

3.2.11 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 1.3 2005/02/28 21:36:03 thielema
   Exp $ *)

IMPORT IO, Fmt, Lex, Stdio;
```

```
VAR
  z1: CARDINAL;
  z2: INTEGER;

(* Wir können Eingabefehler noch nicht behandeln. *)
<* 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 [3.11](#) sehen.

Seltsame Effekte bei der Eingabe mit `Lex.Int`

Liest man Werte von der Standardeingabe mit `Lex.Int` und Verwandten ein, gibt es einige merkwürdige Effekte.

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

Wir konnten die drei Fragen nach Zahlen also gleich in der ersten Zeile mit der Eingabe `10 10 -42` befriedigen. Das gleiche erreichen wir mit:

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

Es ist `Lex.Int` einerlei, ob die Standardeingabe eine Datei, Benutzereingaben oder die Ausgabe eines anderen Programmes (hier `echo`) ist. Das besondere an der Eingabe über die Konsole ist lediglich, dass die eingegebenen Zeichen erst „abgeschickt“ werden, wenn `<RETURN>` gedrückt wird. So lange das nicht passiert ist, kann man die Eingabe noch korrigieren. Die Funktion `Lex.Int` bekommt vom Betriebssystem nur die fertige die Zeichenfolge zu sehen. Darin betrachtet es Leerzeichen und Zeilenvorschübe als gleichwertige Trennzeichen und beachtet ansonsten nur Ziffern. Möchte man sicherstellen, dass pro Zeile immer genau eine Zahl steht, also jede Zahleneingabe mit `<RETURN>` abgeschlossen wird, so sollte man statt `Lex.Int (Stdio.stdin)` auf `Scan.Int (IO.GetLine (Stdio.stdin))` zurückgreifen.

3.3 Anweisungen zur Auswahl

3.3.1 IF-Anweisungen

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

```
FALLS Bedingung
DANN
  Anweisungen1
SONST
  Anweisungen2
```

```
IF Bedingung
THEN
  Anweisungen1
ELSE
  Anweisungen2
END
```

Beispiel: Maximumsbestimmung

```
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 enthält 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.

Ganz nebenbei bemerkt, berechnet man in Modula-3 das Maximum dreier Zahlen am besten überhaupt nicht mit Verzweigungen, sondern rein funktional mit **MAX** (x , **MAX** (y , z)).

3.3.2 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 3.1.6)

3.4.1 Definition

Prozeduren und *Funktionen* sind *Unterprogramme*, also Teile eines großen Programmes aber nicht eigenständig lauffähig. Die Reihenfolge von Funktionsdefinition und Funktionsaufruf im Programmtext ist egal!

```
MODULE Main;
(* $Id: AusgabeFunktion.m3,v 1.3 2005/01/23 11:38:09
   thielema Exp $ *)

IMPORT IO;

PROCEDURE Text1 () =
  BEGIN
    IO.Put("Mein erstes Modula-3-Programm.\n");
  END Text1;

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

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

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 Datenverbänden (Abschnitt 3.8) zusammenfassen. Der Typ des Rückgabewertes wird mit Doppelpunkt getrennt hinter die Parameterliste geschrieben.

Solche Prozeduren heißen auch Funktionsprozeduren oder kurz Funktionen. Mit dem **RETURN**-Befehl werden bei solchen Prozeduren die Werte zurückgegeben.

```
PROCEDURE Name (var1, var2: TypA; ): 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 kommasetrennter 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. Die übergebenen Werte heißen *Argumente*, wohingegen die lokalen Variablen, in die diese Werte kopiert werden, *Parameter* heißen. Die Kopiermethode 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

```
MODULE Main;
(* $Id: QuadratFunktion.m3,v 1.4 2005/02/28 21:36:28
   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 ganze 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 ganze 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 1.2 2005/01/23 11:38:09
   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 größere 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 Variablen*.

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, derartige reine 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 3.10, 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 Funktionsparameter sind lokale Variablen

Hier ein Beispiel für unsinnige Programmierung, welches demonstriert, dass Funktionsparameter auch lokale Variablen sind.

```

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 der Parameter `x` der Funktion `Quadrat` nichts mit der globalen Variablen `x` zu tun hat, und dass der Parameter `x` innerhalb der Funktion ohne Wirkung nach außen verändert werden kann.

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 Variableninhalts übergeben, sondern ein Verweis auf die Variable. Insbesondere darf man an solche Parameter auch keine Ausdrücke übergeben. Das Verfahren heißt neu-deutsch *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 1.3 2005/01/23 16:45:01
   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 der Übergabe initialisieren?
- Wird die übergebene Variable von der Funktion beschrieben, also muss sich der Aufrufer selbst Kopien des Inhalts 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 3.6) 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.

Übergabeart	Erklärung	Anwendung
VALUE	Call by value (Übergabe einer Kopie)	kleine Datenmengen wie ordinale Werte, Zahlen, kleine Mengen, Zeiger (insbesondere auf Objekte)
READONLY	Call by reference (Übergabe eines Verweises mit Schreibverbot)	große Datenmengen wie Felder, große Datenverbünde
VAR	Call by reference (Übergabe eines Verweises mit uneingeschränktem Zugriff)	Prinzipiell für Aktualisierungen gedacht, übersichtlicher sind aber immer Rückgabewerte (RETURN), gegebenenfalls mit Datenverbänden oder Zeigern auf solche. (Siehe Abschnitte 3.8 und 3.9.)

3.4.11 Rekursive Funktionen

Eine mächtige Programmieretechnik ist die *Rekursion*.

Eine Funktion heißt *rekursiv*, wenn sie sich selbst aufruft. Ebenso nennt man mehrere Funktionen *rekursiv*, wenn sie sich gegenseitig aufrufen.

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 $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \dots \cdot n$

```

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 $n \rightarrow n-1 \rightarrow n-2 \rightarrow \dots$) 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.

3.4.12 Konzeption von Funktionen

- Benötigt man eine Funktionalität, die von so allgemeinem Nutzen ist, dass dafür schon einmal jemand eine Funktion geschrieben haben könnte, sollte man danach in den vorhandenen Bibliotheken suchen. Bibliotheksfunktionen zu verwenden statt selbst zu schreiben hat den Vorteil, dass Fehler nur einmal korrigiert werden müssen. Viele Programmierer können davon profitieren.
- 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).
- Funktionsnamen sind Bezeichner und müssen nach diesen Regeln aufgebaut werden. (Abschnitt 3.1.7) Der Name der Funktion sollte mit einem Großbuchstaben beginnen, aber nicht nur aus Großbuchstaben bestehen. Wortbestandteile im Namen sollten durch Großbuchstaben markiert werden. Etwa `PutText`.
- Funktionen sollten sortenrein in Modulen abgelegt werden. Der Name der Funktion sollte keine Information des Modulnamens wiederholen. Zum Beispiel ist `Sound.Play` dem Namen `Sound.PlaySound` vorzuziehen. Das hat neben der kompakteren Schreibweise auch den Vorteil, dass sich ähnliche Module leichter austauschen lassen, etwa wenn sie für verschiedene Datentypen eine passende `Play`-Funktion anbieten.
- Eine *reine Funktion* hat nur **READONLY**- und **VALUE**-Parameter, liest oder verändert keine globalen Variablen und gibt einen Wert zurück. Eine reine Funktion kann folglich keine nicht-reine Funktion aufrufen und ihr Funktionswert hängt allein von den übergebenen Argumenten ab. Der Effekt solcher

Funktionen lässt sich vom Aufrufer am besten abschätzen: Die Funktion bezieht nur die Informationen ein, die man ihr gibt und sie verändert keine Variablen im Kontext des Aufrufers. Reine Funktionen sind daher das anzustrebende Ideal beim Funktionsentwurf. Die Gründe, die hin und wieder dagegen sprechen, sind in der Regel Ineffizienz oder dass man auf Unterprogramme des Betriebssystems zurückgreifen muss, die keine reinen Funktionen sind.

- Unterprogramme müssen globale oder per Referenz erhaltene Objekte ändern, um überhaupt einen Effekt zu haben. Unterprogramme ohne Rückgabewerte sind daher mit Skepsis zu betrachten.
- Kompliziertes aus Einfachem zusammensetzen und nicht umgekehrt Einfaches als Spezialfall des Komplizierten auffassen.

Schlechtes Beispiel:

```
PROCEDURE SetColor (b : BOOLEAN;  
                    i : INTEGER;  
                    x : LONGREAL; );  
  
(* Wenn 'b', so setze auf Wert von 'i',  
   sonst auf Wert von 'x'. *)
```

Besser:

```
PROCEDURE SetColorInt      (i : INTEGER; );  
PROCEDURE SetColorLongReal (x : LONGREAL; );
```

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
  Anweisungen
```

```
WHILE bedingung DO
  Anweisungen
END;
```

```
WIEDERHOLE
  Anweisungen
BIS Bedingung erfüllt
```

```
REPEAT
  Anweisungen
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.

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

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

FORher:

```
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**

.

3.5.4 Beispiel: Berechnung von π nach Leibniz

Von Leibniz stammt die Reihenentwicklung

$$\frac{\pi}{4} = \arctan 1$$

$$= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \mp \dots$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n}{2 \cdot n + 1},$$

die wir zur Berechnung einer Näherung von π benutzen wollen. Eine Näherung auf diesem Wege zu finden, ist zwar nicht besonders effizient aber dafür sehr einfach.

```

MODULE Main;
(* $Id: ApproxPi.m3,v 1.4 2005/03/02 11:57:52 thielema
   Exp $ *)
(* Approximation von Pi mit Verfahren von Leibniz *)

IMPORT IO, Fmt, Lex, Stdio;

VAR
  max      : CARDINAL;
  zaehler : LONGREAL := 4.0D0;
  nenner  : LONGREAL := 1.0D0;
  pi      : LONGREAL := 0.0D0;

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

  FOR n := 0 TO max - 1 DO
    pi := pi + zaehler / nenner;
    zaehler := -zaehler;
    nenner := nenner + 2.0D0;
  END;
  IO.Put(Fmt.F("n = %8s, pi = %s\n", Fmt.Int(max),
    Fmt.LongReal(pi)));
END Main.

```

```
ApproxPi> LINUXLIBC6/prog
```

```
Wieviele Reihenglieder sollen berechnet werden? 200
```

```
n =      200,  pi = 3.1365926848388144
```

```
ApproxPi> LINUXLIBC6/prog
```

```
Wieviele Reihenglieder sollen berechnet werden? 20000
```

```
n =      20000,  pi = 3.1415426535898208
```

```
ApproxPi> LINUXLIBC6/prog
```

```
Wieviele Reihenglieder sollen berechnet werden? 2000000
```

```
n = 2000000,  pi = 3.1415921535897184
```

3.6 Felder (Arrays)

3.6.1 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 die vollständige Wertetabelle einer Funktion. Die repräsentierte Funktion bildet einen Wert vom Indextyp auf einen Wert vom Wertetyp ab.

Noch ein Beispiel, welches auf dem Typen Grundfarbe aus dem Abschnitt 3.2.5 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};
```

Dann gilt

```
orange[Grundfarbe.Gruen] = 0.5D0
lila  [Grundfarbe.Blau]  = 1.0D0
```

Das folgende Beispiel beantwortet die in Abschnitt 3.2.5 aufgeworfene Frage nach der Konvertierung eines Aufzählungswertes in einen Text.

```

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;

```

Wichtig: Obwohl ein Feld mit genau einem Element genau so viel Information enthält, wie eine Variable vom Elementtyp, sind beide Dinge grundsätzlich verschieden. Beispiel:

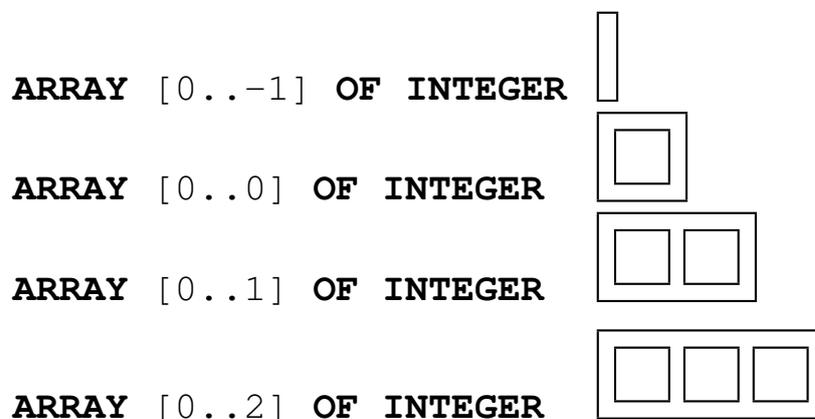
```

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.



3.6.2 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 3.9.2. 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;
```

3.6.3 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 (Abschnitt 3.12.4).

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 3.9.2) und Funktionsparameter können selbst offene Felder sein.

Beispiel: Durchschnitt einer beliebigen Anzahl 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 3.7.2 verwendet, der zwei Texte aneinanderhängt.

3.6.4 Feldabschnitte

In `Modula-3` ist es möglich, einen Abschnitt eines Feldes wie ein Feld zu behandeln. Dabei wird nicht der Inhalt des Feldes in ein neues kleineres Feld kopiert, es wird lediglich die Sicht auf einen Teil des Feldes eingeschränkt. Dieser Abschnitt kann daher auch beschrieben werden, falls das Gesamtfeld beschreibbar ist.

Die hierzu benötigte Funktion heißt **SUBARRAY**. Der Ausdruck **SUBARRAY**(*x*, *start*, *size*) entspricht dem Abschnitt des Feldes *x* der beim Index *start* beginnt und *size* Elemente umfasst.

```

VAR
  a := ARRAY OF CARDINAL{0, 1, 2, 3, 4, 5, 6, 7, 8};
  b: ARRAY [0..2] OF CARDINAL;
BEGIN
  b := SUBARRAY (a, 0, 3);
  SUBARRAY (a, 6, 3) := b;
END;

```

3.6.5 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.

3.7 Texte

3.7.1 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 1.5 2005/03/01 13:58:18
   thielema Exp $ *)
(* Beispielprogramm zu CHAR-Variablen und ASCII.i3 *)

IMPORT IO, Fmt, Stdio, ASCII, Text;
```

```
VAR
  line: TEXT;
  z    : CHAR;

<* FATAL ANY *>
BEGIN
  LOOP
    IO.Put("\nBitte ein paar Zeichen eingeben"
           & " oder <RETURN> für Abbruch: ");
    line := IO.GetLine(Stdio.stdin);
    IF Text.Empty(line) THEN EXIT END;

    FOR i := 0 TO Text.Length(line) - 1 DO
      z := Text.GetChar(line, i);

      IO.Put(
        Fmt.F(
          "\nDer ASCII-Code des Zeichens '%s' ist %s.\n",
          Fmt.Char(z), Fmt.Int(ORD(z))));

      IF z IN ASCII.Digits THEN
        IO.Put("Das Zeichen ist eine Ziffer.\n");
      ELSIF z IN ASCII.Uppers THEN
        IO.Put(
          Fmt.F(
            "Das Zeichen ist ein Großbuchstabe\nund "
            & "der passende Kleinbuchstabe ist '%s'\n",
            Fmt.Char(ASCII.Lower[z])));
      ELSIF z IN ASCII.Lowers THEN
        IO.Put(
          Fmt.F(
            "Das Zeichen ist ein Kleinbuchstabe\nund "
            & "der passende Großbuchstabe ist '%s'\n",
            Fmt.Char(ASCII.Upper[z])));

      END;
    END;
  END;
  IO.Put(
    "Danke für die Eingaben und schönen Tag noch!\n");
```

```
END Main.
```

Bitte ein paar Zeichen eingeben oder <RETURN> für Abbruch: A6 b^G

Der ASCII-Code des Zeichens 'A' ist 65.
Das Zeichen ist ein Großbuchstabe
und der passende Kleinbuchstabe ist 'a'

Der ASCII-Code des Zeichens '6' ist 54.
Das Zeichen ist eine Ziffer.

Der ASCII-Code des Zeichens ' ' ist 32.

Der ASCII-Code des Zeichens 'b' ist 98.
Das Zeichen ist ein Kleinbuchstabe
und der passende Großbuchstabe ist 'B'

Der ASCII-Code des Zeichens '' ist 7.

3.7.2 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.

Textlitterale sind in Anführungszeichen eingeschlossene Zeichenketten wie "Das ist ein Text."

Es gibt nur eine festverdrahtete Operation für Texte: Die *Verkettung*, repräsentiert durch das Symbol &. Textlitterale 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 [3.9](#)). 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 3.6.1): Die Literale `'A'` und `"A"` sind völlig verschieden. Das erste ist ein Zeichen, das zweite ein Text, der zufälligerweise nur ein Zeichen enthält.

3.7.3 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 kniffliger. 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 1.5 2005/03/02 11:58:38
   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:\n");
text :=
  Lex.Scan(
    Stdio.stdin, SET OF CHAR{'a' .. 'z', 'A' .. 'Z'});
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:
Oberpostbriefmarkenstempelautomatenmechaniker
brpstbrfmrknstmplmtnmchnkr

```

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.

3.7.4 Ein Beispiel für Text-Verarbeitung: Test auf Bereichsüberschreitung

```

MODULE Main;
(* $Id: TestGanzzahltext.m3,v 1.6 2005/03/02 11:59:10
  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 erfuehlt *)
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. *)
  RETURN
    Text.Compare(Text.Sub(s, i, 5), "32768") < 0;
END Is16BitInt;

VAR text: TEXT;

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

  IF Is16BitInt(text) THEN
    VAR wert := Scan.Int(text);
    BEGIN
      IO.Put(
        Fmt.F("Der eingelesene Text \"%s\""
          & " entspricht dem Zahlenwert %s.\n",

```

```

                                text, Fmt.Int(wert));
    END;
    EXIT
ELSE
    IO.Put(
        "Wert außerhalb des gueltigen Bereichs!\n\n");
    END;
END;
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 [3.11](#).

3.8 Datenverbände

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 Datenverbände (**RECORD**). Die Syntax für die Elementliste eines *Datenverbandes* 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 eigenen Namen (Person) ein. Damit können wir bereits eine erste kleine Datenbank aufbauen, unbestritten eines der beliebtesten Beispiele zur Demonstration von Datenverbänden.

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.vorname.

Ebenso kann man Datenverbände variablen Inhalts anlegen und erst recht Felder von Datenverbänden

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 Datenverbände 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;
```

3.9 Zeiger (Pointer)

Felder und Datenverbände sind in gewisser Weise starr. Die Elemente eines Feldes gehören fest zu einem Feld, die Elemente eines Datenverbandes gehören fest zum Datenverbund. Es ist so nicht möglich, dass sich zwei Datenverbände dieselben Daten teilen. Es ist auch nicht möglich, an ein Feld Elemente anzuhängen, mittendrin einzufügen oder zu löschen.

Das alles erlauben *Zeiger*. Mit Zeigern lässt sich ein Geflecht, also ein *gerichteter Graph* erzeugen, in dem einfache Daten, Felder und Datenverbände Knoten sind und Zeiger gerichtete Kanten.

Mit der Deklaration

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

legt man eine Variable *p* an, die nicht selbst eine ganze Zahl enthält, sondern nur auf eine solche Variable verweist (eine Referenz, daher **REF**).

Wichtig: Diese Deklaration erzeugt **nur** den Verweis und keinen Speicherplatz fuer eine Zahl, auf die gezeigt werden könnte.

3.9.1 Operationen auf Zeigern

Was fängt man mit so einem Verweis an? Als allererstes kann man dem Verweis folgen, also zu dem Objekt gehen, auf das verwiesen wird. Dieser Vorgang nennt sich *Dereferenzieren*. Wenn p wie oben vereinbar ist, dann bezeichnet p^{\wedge} die ganze Zahl, auf die p zeigt.

Da aber p nicht initialisiert wurde, zeigt sie noch nicht auf etwas sinnvolles. Wir müssen erst ein Objekt erstellen, auf das der Zeiger zeigen kann. Dafür gibt es die **NEW**-Funktion. Diese reserviert Speicher für ein Feld, Datenverbund oder Objekt und gibt einen Verweis darauf zurück.

```

VAR
  p: REF INTEGER := NEW (REF INTEGER) ;
BEGIN
  p^ := 3;

```

Hier wird die Zeigervariable p mit Typ **REF INTEGER** angelegt. **NEW** reserviert Speicher für eine ganze Zahl und gibt einen Verweis darauf zurück. Mit diesem Verweis wird p initialisiert. Auch hier kann die Typangabe hinter p entfallen, denn der Typ ergibt sich aus dem Typ des von **NEW** erzeugten Objektes.

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*.

Kann man auch einen Verweis auf ein schon existierendes Objekt bekommen? Im Prinzip gibt es dafür die **ADR**-Operation. Die **ADR**-Operation (*Referenzieren*) ist im Wesentlichen die Umkehrung der \wedge -Operation (*Dereferenzieren*).

Im Normalfall (sprich: in sicheren Modulen) ist das Referenzieren verboten. Dieses Verbot ist nötig, denn mit **ADR** kann man sich einen Verweis auf eine lokale Variable besorgen, der noch weiterbesteht, obwohl die zugehörige Variable 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 (**NEW**).

Es gibt den speziellen Zeigerwert **NIL**. Dieser zeigt auf nichts, und kann dazu verwendet werden, anzuzeigen, dass derzeit kein Objekt von diesem Typ verfügbar ist.

Es gibt den allgemeinen Zeigertyp **REFANY**. Dieser kann dazu verwendet werden, um Daten verschiedener Typen zu verwalten. Man sollte möglichst wenig Gebrauch von ihm machen, denn er senkt die statische Sicherheit und benötigt Typentests zu Laufzeit.

```

VAR
  a: REF INTEGER;
  b: REFANY;
BEGIN
  b := a;
  a := b;
END

```

In diesem Beispiel hat `a` einen spezielleren Zeigertyp als `b`. Die erste Zuweisung klappt daher immer. Für die zweite Anweisung muss das Programm zur Laufzeit überprüfen, ob `b` tatsächlich auf eine ganze Zahl zeigt, es könnte schließlich auf alles mögliches zeigen.

Weiß man, dass `b` auf eine ganze Zahl zeigt, und möchte man diese Zahl wissen, so kann man den Zeigertyp mit **NARROW**(`b`, **REF INTEGER**) spezialisieren. Das heißt mit **NARROW**(`b`, **REF INTEGER**)[^] erfährt man die Zahl, auf die verwiesen wird. Zeigt `b` nicht auf eine Zahl, löst **NARROW** einen Fehler aus.

3.9.2 Zeiger auf offene Felder

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 als 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 wiederum die Dereferenzierung (^) bemüht. Die Variable z enthält den Verweis auf ein offenes Feld, z[^] steht für 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:

```

TYPE
  Vector = ARRAY OF LONGREAL;

PROCEDURE Scale (READONLY x: Vector;
                  k: LONGREAL; ):

  REF Vector =

  VAR z := NEW(REF Vector, 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.
```

3.9.3 Zeiger auf Datenverbände

Aus Zeigern, Datenverbänden und Feldern lassen sich nun sehr mächtige Datenstrukturen aufbauen. Um auf unser bürokratisches Beispiel zurückzukommen, 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 Datenverbänden 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^{\wedge} zuzugreifen. Erst mit **NEW** kann man eine Person erzeugen, auf die man einen Verweis erhält. Speziell für Datenverbände erlaubt **NEW**, dass man einzelne Elemente des Datenverbundes initialisiert. Das sieht etwa so aus:

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

So enthält $p^{\wedge}.vorname$ bereits den Wert "Klaus". Wie bei Zeigern auf Felder braucht man das \wedge nicht mitzuschreiben, wenn man auf Elemente des verwiesenen Datenverbundes zugreift. Deswegen ist $p.vorname$ statt $p^{\wedge}.vorname$ auch erlaubt.

Hier das Komplettbeispiel:

```
MODULE Main;
```

```
(* $Id: KopieFlachTief.m3,v 1.3 2005/03/02 15:35:59
   thielema Exp $ *)
```

```
(* Datenverbund, Verweise, tiefe und flache Kopie. *)
```

```
IMPORT IO, Fmt;
```

```
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;

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

PROCEDURE FmtPerson (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 FmtPerson;

PROCEDURE FmtAdresse (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 FmtAdresse;

PROCEDURE FmtAkte (READONLY akte: Akte; ): TEXT =
BEGIN
    RETURN Fmt.F("%s ist wohnhaft in\n%s",
                FmtPerson(akte.person),
                FmtAdresse(akte.wohnhaft));
END FmtAkte;

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

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.Kerl, Datum{12, 3, 1968}},
            Adresse{"How-many-Road", 42, 01234,
                  "Kleinkleckersdorf"}};

    BEGIN
        a := b;
        IO.Put("Tiefe Kopie:\n\n");
        IO.Put(FmtAkte(a));

        b.person.nachname := "Flüssigbrot";
        b.wohnhaft.strasse := "Produktionsstraße";
        IO.Put("\nDie Akte enthält nun:\n");
    
```

```

    IO.Put (FmtAkte(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.Kerl,
            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(FmtVerweisAkte(a));

    b.person.nachname := "Flüssigbrot";
    b.wohnhaft.strasse := "Produktionsstraße";
    IO.Put("\nDie Akte enthält nun:\n");
    IO.Put(FmtVerweisAkte(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

3.9.4 Felder von Feldern

Jetzt noch ein Beispiel dafür, wie man Zeiger und Felder sinnvoll kombinieren kann. Wir möchten die Werte des PASCALSchen Dreiecks in eine Datenstruktur schreiben. Das PASCALSche Dreieck besteht aus den Binomialkoeffizienten $\binom{n}{k}$.

$$\begin{aligned}\binom{n}{k} &= \frac{n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-k+1)}{1 \cdot 2 \cdot 3 \cdot \dots \cdot k} \\ &= \prod_{i=0}^{k-1} \frac{n-i}{i+1} \\ &= \frac{n!}{k! \cdot (n-k)!} \\ \binom{n+1}{k} &= \binom{n}{k-1} + \binom{n}{k}\end{aligned}$$

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 1.5 2005/03/02 18:45:11
   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
      (* neue Zeile in Dreieck einhängen *)
      p[i] := NEW(REF ARRAY OF CARDINAL, i + 1);
      p[i, 0] := 1;
      FOR j := 1 TO i - 1 DO
        p[i, j] := p[i - 1, j - 1] + p[i - 1, j];
      END;
      p[i, i] := 1;
    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]^(j)) DO
        t := t & Fmt.Pad(Fmt.Int(p[i, j]), 6);
      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.

```

```

PascalDreieck> LINUXLIBC6/prog
Größe des Pascalschen Dreiecks: 7
 1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10  5  1
1  6 15 20 15  6  1

```

3.9.5 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`.

Auf diese Weise ist es möglich eine Funktion unter einem neuen Namen sichtbar zu machen.

```
CONST Sinus = 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.
```

3.10 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 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 eine Eingabe- und eine Ausgabedatei handelt. Es funktioniert auch nicht, wenn das Programm die Dateien nicht 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 vielen Betriebssystemen 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 streng 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“, darstellbare ASCII-Zeichen, 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` (Abschnitt B.10), `FileWr` (Abschnitt 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 `BinIO`, `FastBinIO`, `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` oder `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-FINALLYs** aufzurufen (siehe Abschnitt 3.11.2).

3.10.1 Beispiel zu Textdateien

```
MODULE Main;
(* $Id: Dateizugriff.m3,v 1.4 2005/01/23 16:45:01
   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 "
            & "ungenauere 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.

```

3.11 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 ...

3.11.1 TRY-EXCEPT-Struktur

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 =>
  IO.Put("Das war keine Zahl.\n");
| FloatMode.Trap(typ) =>
  IO.Put("Zahl außerhalb des Bereichs.\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.

Die **TRY-EXCEPT**-Struktur 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 1.4 2005/01/23 11:38:09 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
            < * ASSERT FALSE,
              "Unerwartete Ausnahme in Lex.Int" * >
          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 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.

3.11.2 TRY-FINALLY-Struktur

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 3.10) Die Schöpfer der Standardbibliothek haben das Schließen von Dateien wohlweislich nicht dem *Garbage Collector* übertragen, denn dieser gibt Ressourcen 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 eine Struktur, die 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.

3.11.3 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 abubrechen.

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;
```

Das folgende Beispielprogramm realisiert das Newton-Verfahren für beliebige Funktionen. Das Newton-Verfahren ist ein Iterationsverfahren, welches für eine

Funktion näherungsweise eine Nullstelle berechnet. Das Programm ist gleichzeitig ein schönes Beispiel für Funktionsvariablen. (Abschnitt 3.9.5) Die Ausnahme NoConvergence wird verwendet, um das Scheitern des Iterationsalgorithmus und den verbleibenden Näherungsfehler zu übermitteln.

```

MODULE Main;
(* $Id: Newton.m3,v 1.4 2005/01/23 16:45:01 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.

```

3.11.4 Konzeption von Ausnahmen

- Der Name einer Ausnahme sollte mit einem Großbuchstaben beginnen.
- Man sollte zwischen Benutzungsfehlern einer Funktion und unvorhersehbaren Problemen unterscheiden. Eine Funktion zur Vektoraddition sollte im Fal-

le ungleich großer Operanden keine Ausnahme verhängen, denn gleich große Vektoren lassen sich vom Aufrufer sehr leicht sicherstellen. Es hat sicher keinen Sinn, wenn der Aufrufer vom Übersetzer damit konfrontiert wird, dass die Vektoraddition eine Ausnahme wegen ungleich großer Vektoren auslösen könnte, obwohl der Aufrufer sehr einfach die gleiche Länge sicherstellen kann. Ungleich große Vektoren sollte man nicht als unverhersehbares Ereignis betrachten, sondern als unsachgemäße Benutzung der Funktion für die Vektoraddition. Entsprechend sollte man in der Vektoraddition nicht

```
IF NUMBER(x) # NUMBER(y) THEN
    RAISE VectorSizesMismatch;
END;
```

sondern

```
< * ASSERT NUMBER(x) = NUMBER(y),
    "Vector sizes mismatch" * >
```

schreiben. Das Pragma `ASSERT` bricht das Programm unwiderruflich ab, falls die getestete Bedingung verletzt ist.

Schwieriger sieht die Lage bei Überläufen aus. Im allgemeinen kann der Aufrufer einer Funktion nicht vorhersehen, ob das Ergebnis im darstellbaren Bereich bleibt. Hier ist eine Ausnahme dem `ASSERT` vorzuziehen.

3.12 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, indem 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.

3.12.1 Einbinden von Modulen

Bis jetzt haben wir nur die einfachste, aber auch wichtigste Form des *Importierens* 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 Typen 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;` kombinieren.

3.12.2 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 wurden, 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;

CONST
  strengGeheim = 23;

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

BEGIN
END Meins.
```

Module sind nach einem festen Schema aufgebaut:

- Kopf
 - Schnittstellen: **INTERFACE** Meins;
 - Implementationen: **MODULE** Meins;

– Implementationen, welche etwas zu fremden Schnittstellen beisteuern:
MODULE Meins **EXPORTS** Deins;

- Einbinden anderer Module (**IMPORT**)
- Deklarationen von Typen, Konstanten, Variablen, Ausnahmen und Prozeduren
- Hauptprogramm bei Implementationen (alles ab **BEGIN**)
- Ende (**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.

Mit der **EXPORTS**-Abwandlung kann man eine Implementation in mehrere Teile zerlegen.

Datei Monument.i3

```
INTERFACE Monument;  
  
PROCEDURE A();  
  
PROCEDURE B();  
  
END Monument.
```

Datei MonumentTeilA.m3

```
MODULE MonumentTeilA EXPORTS Monument;  
  
PROCEDURE A() =  
  BEGIN  
  END A;  
  
BEGIN  
END Monument.
```

Datei MonumentTeilB.m3

```
MODULE MonumentTeilB EXPORTS Monument;  
  
PROCEDURE B() =  
  BEGIN
```

```
END B;
```

```
BEGIN
```

```
END Monument.
```

Mit der gleichen Strategie kann man zwei verschiedene Implementationen für die gleiche Schnittstelle anbieten und von der Projektverwaltung die jeweils angenehmste einsetzen lassen.

Gestaltung

- Erwartungsgemäß sind auch Modulnamen Bezeichner (Abschnitt 3.1.7) und müssen nach diesen Regeln gebildet werden. Der Modulname sollte mit einem Großbuchstaben beginnen. Modulname und Dateinamen sollten (auch in Groß- und Kleinschreibung) übereinstimmen. Beispiel: Die Datei `Month.i3` beginnt mit **INTERFACE** `Month;`.
- In jedem Modul sollte höchstens ein (Haupt-)Datentyp implementiert werden. Mehrere Hilfstypen sind aber durchaus sinnvoll. Ein Modul sollte den Namen des implementierten Datentyps tragen. Dieser Typ wird einfach **T** genannt, so dass sich dieser leicht in qualifizierter Form verwenden lässt. Beispiel: `Month.T` für einen Monat. Generische Module (Abschnitt 3.12.4) benötigen einheitliche Schnittstellen als Parameter. Für diese Module ist die Konvention ganz besonders wichtig.
- Aus diesem Verwendungsmuster ergibt sich, dass man Modulen wie auch Typen am besten Wörter in der Einzahl gibt. Beispiel: `Month` statt `Months`.
- Mit **EXPORTS** sollte sparsam umgegangen werden, denn man verliert leicht die Übersicht, welche Schnittstellen und Implementationen zusammengehören.
- Die Schnittstelle sollte minimalistisch sein. Der Benutzer des Moduls sollte nicht mit Details der Realisierung belästigt werden. Alles was der Benutzer zur Lösung seiner Aufgaben nicht braucht, gehört in den Implementationsteil. (Nicht in diesem Skript: Mit der Untertypenrelation `<:` kann man den inneren Aufbau von Datenstrukturen verheimlichen und mit *teilweiser Aufdeckung* (**REVEAL**) in den Implementationsteil oder andere Schnittstellen verlegen.)

Auf ein Objekt, das im Implementationsteil deklariert wird, wie im obigen Beispiel `strengGeheim`, kann nur innerhalb des Implementationsteils zu-

gegriffen werden. Kein anderes Modul kann dieses einsehen. Damit kann man unkontrollierten Zugriff von außen unterbinden.

- Während globale Variablen schlechter Stil sind, sind globale Variablen in Schnittstellen ganz schlechter Stil. Sie können von allen Nutzern des Moduls gelesen und sogar geändert werden. Bei Fehlern, die auf solche Variablen zurückgehen, ist der Kreis der Verdächtigen entsprechend groß.
- Globale Variablen in Implementationen sind zwar für andere Module unsichtbar, aber es sind immer noch globale Variablen! Sie könnten dafür genutzt werden, einen Zustand vor dem Benutzer zu verheimlichen, so dass ein zweimaliger Aufruf des gleichen Unterprogrammes zu verschiedenen Ergebnissen führt. Deswegen sollten auch globale Variablen dieser Art vermieden werden. Man kann sie vermeiden, indem man dem Aufrufer ein Objekt zurückgibt, dessen Inhalt der Aufrufer nicht kennt (*BlackBox*). Dieses Objekt muss der Aufrufer bei Folgeaufrufen immer übergeben. Dieses Vorgehen hat folgenden Vorteil: Der Benutzer hat genaue Kontrolle über den Zustand der Berechnungen, ohne den Zustand manipulieren können. Er kann mehrere solcher Objekte anfordern und parallel benutzen, ja sogar gleichzeitig (*nebenläufige Programmierung*).

3.12.3 Paketverwaltung

Ein Paket erstellen

Hat man ein Programm in mehrere Module zerlegt, ist für den Übersetzer noch nicht klar, wie alles 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	<i>Steuerdatei</i>
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 ("libc3")           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.

Umgang mit mehreren Paketen

Solange man es mit einem Paket für ein ausführbares Programm zu tun hat, ist alles ganz einfach: Erst `cm3` und danach das fertige Programm aufrufen, etwa `LINUXLIBC6/prog`. Hat man aber eine Bibliothek geschrieben, so möchte man die darin enthaltenen Module irgendwie nutzen, entweder für weitere Bibliotheken oder aber für ausführbare Programme. Möchte man eine Bibliothek allen Nutzern eines Rechners zugänglich machen und besitzt Root-Rechte für diesen Rechner, so kann man die Bibliothek mit `cm3 -ship` an einem zentralen Ort (nämlich dort, wo auch alle Standardbibliotheken herumliegen) installieren. Besitzt man keine Root-Rechte oder möchte man mit einer noch nicht fertiggestellten Version der Bibliothek experimentieren, dann muss man den Nutzern der Bibliothek sagen, wo die Experi-

mentiversion liegt.

Angenommen wir entwickeln die Bibliothek `woerterbuch` und wollen sie im Programmprojekt `rechtschreibkontrolle` verwenden. Beide Projekte liegen im Verzeichnis `$HOME/modula3`. Wäre die Bibliothek `woerterbuch` fertig, würden wir einfach `cm3 -ship` im Verzeichnis `$HOME/modula3/woerterbuch` aufrufen und in der Datei `$HOME/modula3/rechtschreibkontrolle/src/m3makefile` die Zeile

```
import ("woerterbuch")
```

hinzufügen.

Wollen wir aber die Testversion ohne Installation nutzen, dann müssen wir zusätzlich die Datei `$HOME/modula3/rechtschreibkontrolle/src/m3overrides` mit dem Inhalt

```
override("woerterbuch", $HOME & "/modula3")
```

anlegen. Ruft man im Verzeichnis `rechtschreibkontrolle` das Kommando `cm3` auf, wird die Datei `m3overrides` ignoriert und daher wird die Bibliothek `woerterbuch` dort gesucht, wo auch die Standardbibliotheken installiert sind. Ruft man aber `cm3 -override` auf, so wird auch `m3overrides` beachtet und die `override`-Anweisung bewirkt, dass die Bibliothek `woerterbuch` nicht an dem zentralen Platz gesucht wird, sondern im Verzeichnis des Benutzers unter `modula3`.

3.12.4 Generische Module

Häufig möchte man die gleiche Funktion für verschiedene Typen haben. Beispielsweise ist die Vektoraddition sowohl für **REAL** als auch für **LONGREAL** sinnvoll. Ein anderes Beispiel ist die Speicherung einer variablen Datenmenge in einer Liste. Dabei sollen Listen aus Texten genau so möglich sein wie Listen aus Zahlen. Eine naheliegende Möglichkeit ist es, die Implementation für Listen immer wieder zu kopieren und mit Suchen und Ersetzen im Texteditor auf den neuen Typen anzupassen. Diese Vorgehensweise ist aufwändig und hat den Nachteil, dass sich Fehler in der Listenimplementation unkontrolliert ausbreiten. Wird der Fehler in einer Kopie beseitigt, existiert er noch in allen anderen.

Um dieses Problem zu umgehen, kann man einen Programmteil als *generisches Modul* (nicht „energisches Modul“! :-] – manchmal auch *Template* genannt) anlegen. Wir wollen uns aber zunächst vorgefertigten generischen Modulen der Standardbibliothek zuwenden.

Datenstrukturen für Standardtypen

Es gibt bereits einige generische Module für dynamische Datenstrukturen.

List	einfach verkettete Listen, am besten für Stapel zu gebrauchen (Was man zuletzt darauf gelegt hat, kann man zuerst wieder herunternehmen.)
PQueue	Prioritätenwarteschlangen (Es wird das aus der Warteschlange genommen, was höchste Priorität hat.)
Sequence	dynamische Felder, auch als Warteschlangen geeignet (Was zuerst eingefügt wird, kann auch zuerst wieder entfernt werden.)
Set	schwachbesetzte Mengen, also Mengen die verglichen mit dem Grundbereich relativ wenig Elemente enthalten
SortedTable	Wörterbücher für Schlüssel (Stichwörter) mit einer linearen Ordnung
Table	Wörterbücher, einfache Zuordnungen (Funktionen)

Wendet man ein derart allgemein gehaltenes Modul auf spezielle Typen an, so nennt man dies *ausprägen*. Alle aufgezählten Module sind bereits für die vier Typen `Atom.T` (eine spezielle Textform), **TEXT**, **INTEGER** und **REFANY** (Abschnitt 3.9.1) ausgeprägt.

Ohne weiteren Aufwand können wir so zum Beispiel mit Folgen von Texten arbeiten. Das folgende Beispiel erwartet die Eingabe einer beliebigen Anzahl von Wörtern. Die Wörter werden anschließend in umgekehrter Reihenfolge ausgegeben. Der Benutzer muss die Anzahl der eingegebenen Wörter nicht im Voraus angeben und so können wir kein offenes Feld anlegen. Wir sind auf eine dynamische Datenstruktur angewiesen.

```

MODULE Main;
(* $Id: Reverse.m3,v 1.1 2005/03/02 18:44:11 thielema
   Exp $ *)

IMPORT IO, Stdio, Text, TextList;

VAR list: TextList.T := NIL;

< * FATAL ANY * >

BEGIN
  IO.Put("Bitte geben Sie mehrere Zeilen "
        & "mit jeweils einem Wort ein.\n")

```

```

        & "Eingabe bitte mit Leerzeile beenden.\n");
LOOP
    VAR line := IO.GetLine(Stdio.stdin);
    BEGIN
        IF Text.Empty(line) THEN EXIT END;
        (* neues Wort auf den Stapel legen *)
        list := TextList.Cons(line, list);
    END;
END;

IO.Put (
    "Hier ihre Wörter in umgekehrter Reihenfolge:\n");
(* Zahlen rückwärts vom Stapel nehmen *)
WHILE list # NIL DO
    IO.Put(list.head & "\n");
    (* nächstes Element vom Stapel holen *)
    list := list.tail;
END;
END Main.

```

Generische Module ausprägen

Um ein generisches Modul ausprägen zu können, braucht man Module, die den Typen enthalten, für den man das Modul ausprägen will. Es reicht nicht einen Typ ohne Modul zu haben, denn generische Module haben als Parameter Module nicht Typen. Auch aus dieser Sicht lohnt es sich, für jeden wichtigen Typen ein eigenes Modul zu führen. Die Parameter-Module müssen gewisse Bedingungen erfüllen, die das generische Modul an sie stellt. In der Regel muss das Parameter-Modul als Haupttyp einen Typen mit dem Namen **T** definieren. Weiterhin muss es meistens eine Textkonstante namens **Brand** mit dem Modulnamen als Wert zur Verfügung stellen. Generische Module wie `PQueue`, die eine lineare Ordnung der verwalteten Daten benötigen, verlangen eine Funktion namens `Compare`, die zwei Elemente vergleicht.

Am Beispiel der Aufzählung aller Monate wollen wir demonstrieren, wie ein Parametermodul aussehen muss und wie wir damit einen Listentyp für Monate ableiten können.

```
Month.i3
```

```
INTERFACE Month;
```

CONST

```
Brand = "Month";
```

TYPE

```
T = {January, February, March, April, May, June, July,
      August, September, October, November, December};
```

```
PROCEDURE Equal(k1, k2: T; ): BOOLEAN;
```

```
END Month.
```

Month.m3

```
MODULE Month;
```

```
PROCEDURE Equal(k1, k2: T; ): BOOLEAN =
```

```
  BEGIN
```

```
    RETURN k1 = k2;
```

```
  END Equal;
```

```
END Month.
```

Wenn wir nun im `m3makefile` die Zeile

```
List("Month", "Month")
```

einfügen, dann steht uns das ausgeprägte Modul `MonthList` zur Verfügung.

Ein bisschen Fingerspitzengefühl beim Ausprägen neuer Module ist geboten. Ein für spezielle Zwecke ausgeprägtes Modul ist statisch sicher und leicht zu handhaben. Aber jedes ausgeprägte Modul verursacht neuen Maschinencode und das obwohl viele Operationen sogar mit dem gleichen Code erledigt werden könnten. Es ist ein schwer lösbares Problem für den Übersetzer, unnötige Codevervielfältigungen zu erkennen und zu vermeiden. Man kann das Vervielfältigen selbst verhindern, indem man auf eine spezielle Ausprägung wie hier für Monate verzichtet und stattdessen auf `IntList` und Konvertierungen mit **ORD** und **VAL** zurückgreift. Dies führt allerdings zu Programmtexten mit häufigen Konvertierungen und größerer Fehleranfälligkeit.

Generische Module entwerfen

Generische Module sind in `Modula-3` höchst einfach umgesetzt. Im Prinzip ist ein generisches Modul ein Modul, bei dem ein paar **IMPORT**-Anweisungen variabel ge-

halten sind. Möchte man eine Datenstruktur für Listen aus Texten entwerfen, beginnt die Schnittstellendatei wahrscheinlich so:

TextList.i3

```
INTERFACE TextList;

IMPORT Text;

TYPE
  T = REF RECORD
    value: Text.T;
    next: T;
  END;
```

In einem generischen Modul wird nicht fest `Text` importiert, sondern das zu importierende Modul wird austauschbar gemacht, es wird zum Parameter `Elem` des generischen Moduls `List`.

List.ig

```
GENERIC INTERFACE List (Elem);

TYPE
  T = REF RECORD
    value: Elem.T;
    next: T;
  END;
```

Achtung: Generische Module können nicht übersetzt werden! Erst ein ausgeprägtes Modul kann übersetzt und damit auf syntaktische Richtigkeit getestet werden. Um nun zu einem speziellen Modul für Textlisten zu kommen, müssen wir ein Modul schreiben, welches Listen mit Texten ausprägt.

TextList.i3

```
INTERFACE TextList = List (Text) END TextList.
```

Man muss diese Schnittstelle aber nicht von Hand anlegen. Die Zeilen

```
build_generic_intf ("TextList", "List", ["Text"], VISIBLE)
build_generic_impl ("TextList", "List", ["Text"])
```

im `m3makefile` erzeugen sowohl Schnittstelle als auch Implementation der Ausprägung. Besonders vorbildlich ist es, kleine Funktionen für das Modula-3-Projektssystem `quake` im Stile der Funktionen in `list.tmpl` anzulegen.

3.13 Objekte

Objektorientiertes Programmieren ist eine Programmier Technik, 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 älteste Programmiersprachen wieder ausgegraben wurden, um ihnen Objektdatentypen und dazugehörige Mechanismen zu verpassen.

Die großen Vorteile spielt das objektorientierte Programmieren vor allem bei großen Projekten aus. Daher ist es schwer, kleine und doch überzeugende Demonstrationsbeispiele zu finden.

Prinzipiell käme man durch Verwendung von Funktionenvariablen (Abschnitt 3.9.5) auch ohne neue Sprachelemente aus, allerdings sorgt eine direkte Unterstützung der Sprache für mehr Sicherheit.

3.13.1 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 Oberklasse „Figur“.

3.13.2 Arbeiten mit Objekten

In der Standardbibliothek werden zahlreiche Dienste über Objekte zur Verfügung gestellt, beispielsweise Zufallszahlengeneratoren (siehe Abschnitt B.15). Ein *Zufalls-generator* ist ein Objekt, das auf Anfrage eine Zahl zufällig auswählt. In Wirklichkeit handelt es sich allerdings um eine Zahl, die nach einem festen Schema berechnet wurde, dessen Ergebnisse aber ausreichend 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, Gleitkommazahlen, 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 Datenverbänden (Abschnitt 3.9.3). 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 Datenverbänden direkt beim Aufruf von **NEW** initialisieren kann, sollte man lieber von der `init`-Methode Gebrauch machen. Es ist Konvention, eine Klasse 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.

3.13.3 Deklaration von Klassen

Eine Klasse **T** wird wie folgt deklariert.

TYPE

T = Oberklasse **OBJECT**

`x, y: INTEGER;`

METHODS

`init (x, y: INTEGER;) : T := Init;`

`abstrakt (z: INTEGER;);`

OVERRIDES

`umgebogen := Umgebogen;`

END;

PROCEDURE `Init (SELF: T; x, y: INTEGER;) : T =`

BEGIN

`SELF.x := x;`

`SELF.y := y;`

RETURN `SELF;`

END `Init;`

PROCEDURE `Umgebogen (SELF: T;) : INTEGER =`

BEGIN

RETURN `SELF.x * SELF.y;`

END `Umgebogen;`

Dabei ist der Typ **T** der Objekttyp (synonym: die Klasse). Der Typ `Oberklasse` bezeichnet die Oberklasse von **T**. Jedes Objekt vom Typ **T** ist also immer auch ein Objekt vom Typ `Oberklasse`. Das schließt übrigens ein, dass eine Variable vom Typ `Oberklasse` ebenfalls ein Objekt (strenggenommen nur einen Verweis darauf) des spezielleren Typs **T** enthalten kann.

Die Deklarationen `x, y: INTEGER;` nach **OBJECT** bezeichnen Variablen, die jedes Objekt enthält. Bis hierhin unterscheidet sich **OBJECT** noch nicht wesentlich von **RECORD**.

Im Abschnitt **METHODS** werden nun neue Methoden für diese Klasse vereinbart. Die Methode `init` hat die Parameter `x` und `y` und gibt ein Objekt vom Typ **T** zurück. Die Methode wird durch das Unterprogramm `Init` verwirklicht. Der Typ des Unterprogrammes `Init` und der der Methode `init` müssen übereinstimmen bis auf den zusätzlichen ersten Parameter von `Init`. Ist `obj` vom Typ **T**, so entspricht der Aufruf `obj.init(a,b)` dem Aufruf `Init(obj,a,b)`.

Auch die Methode `abstrakt` wird neu eingeführt, allerdings wird sie noch nicht implementiert. Bei einem Objekt vom Typ **T** (keine Unterklasse!) kann sie also nicht aufgerufen werden. Sie ist allein dazu da, um eine einheitliche Schnittstelle für die Implementation dieser Methode in Unterklassen vorzugeben.

Im Abschnitt **OVERRIDES** können nun Methoden der Oberklasse überschrieben werden. So wie das Unterprogramm `Umgebogen` deklariert ist, muss es in der Oberklasse bereits eine Methode mit der Signatur `umgebogen(): INTEGER` geben. Diese wird nun von unserer neuen Klasse **T** mit Leben gefüllt, falls sie in Oberklasse eine abstrakte Methode ist, oder andernfalls ersetzt.

3.13.4 Beispiel: geometrische Figuren

Nun ein ausführlicheres Beispiel zur Verwendung von Klassen bei geometrischen Objekten.

So programmierte man vor der Objektorientierung:

```
MODULE Main;
(* $Id: GeometrieRecord.m3,v 1.3 2005/01/23 11:38:09
   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:
                          gegenüberliegender
                          Punkt, Dreieck:
                          zweiter Eckpunkt *)
    p2: Punkt;           (* nur Dreieck:
                          dritter Eckpunkt *)
    radius: LONGREAL;  (* nur Kreis *)
  END;

PROCEDURE Flaechе (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 Flaechе;

```

```

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 1.3 2005/01/23 11:38:09
   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.
```

3.13.5 Wissensfrage: Generisches Modul oder Objektklasse

Sowohl generische Module als auch objektorientierte Programmierung stellen Techniken der Abstraktion dar: Wir wollen ein Programm in gewissen Grenzen unabhängig von den verwendeten Typen machen. Ist eine der beiden Techniken der anderen überlegen? Kann man auf eine von beiden ganz verzichten?

Vorteil der generischen Module ist die hohe statische Sicherheit und die höhere Ausführungsgeschwindigkeit (es müssen zur Laufzeit keine Entscheidungen über tatsächlich aufzurufende Programmteile getroffen werden), der Nachteil ist der höhere Speicherverbrauch für das ausführbare Programm. Beim Programmieren mit Klassen ist es genau anders herum. Darüberhinaus folgen Klassen Vererbungshierarchien, generische Module hingegen nicht.

An welchem Kriterium soll man nun die Entscheidung festmachen? Ganz pragmatisch kann man sagen: Benötigt man die Umlenkung zur Implementation einer Funk-

tion zur Laufzeit, dann kommen nur Klassen in Frage. Ist es dagegen zum Zeitpunkt der Übersetzung völlig klar, welcher Typ eingesetzt werden soll, so sind generische Module die bessere Wahl.

Selbstverständlich kann man generische Module mit Objekttypen kombinieren, um so beide Varianten von Typenabstraktion zu trennen (und die Entscheidung für das richtige Konzept zu erschweren :-). Man kann zum einen den Haupttyp eines generischen Moduls als Objekt vereinbaren, man kann aber auch ein generisches Modul mit einem Objekttypen ausprägen.

Anhang A

Übersichten

A.1 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 jedes Moduls steht ein Kommentar mit
 - Autor, Datum
 - Kurzbeschreibung / Zweck / Methode
 - evtl. Referenzen oder Versionsnummer

Versionskontrollsysteme wie CVS können diesen auch automatisch erzeugen und aktuell halten.

- Kommentare sind wichtig, aber nicht jeder Kommentar ist gut
 - Kommentare können komplizierte Anweisungen erklären. Noch besser ist es aber komplizierte Anweisungen zu vermeiden.
 - Kommentare brauchen nicht die Sprachbeschreibung zu wiederholen. Beispielsweise sagt der Kommentar in

```
a := b; (* b wird nach a geschrieben *)
```

dem Leser wenig neues. Besser wäre

```
a := b; (* Zählung beginnt wieder von vorn *)
```

.

- Manchmal ist es auch gut, auf naheliegende aber unzulässige Vereinfachungen hinzuweisen.
- 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.
- Benutzereingaben müssen daraufhin überprüft werden, ob sie den gestellten Anforderungen genügen. Gleiches gilt für Parameter von Funktionen.
- Typen so wählen, dass sie die Realität möglichst genau modellieren.
- Invarianten, die nicht durch geeignete Typen sichergestellt werden können, durch zusätzliche Abfragen (`ASSERT`-Pragma) testen.
- Warnungen des Übersetzers ernst nehmen. Nach Ursachen suchen, statt die Warnungen zu unterdrücken. Mit Kommentar begründen, falls man doch eine Warnung unterdrückt.
- Bevor man im Texteditor mit „Kopieren“ und „Einfügen“ Programmtext vervielfältigt, sollte man sich überlegen, ob es nicht besser wäre, den Programmabschnitt in eine Prozedur zu verwandeln. Das hat mehrere Vorteile:
 - Der Leser des Programmes muss sich nur einmal in den Programmabschnitt hineindenken.
 - Fehler, die in diesem Programmabschnitt enthalten sind, müssen nur einmal bereinigt werden.
 - Ist die Prozedur einmal erstellt, lässt sie sich sehr leicht an weiteren Stellen einsetzen.
- Sonderbehandlung für spezielle Werte einer Wertemenge eines Typen vermeiden. Statt

```

RECORD
  name: TEXT;
    (* vorangestelltes '*' bedeutet Einzelstück *)
  price: CARDINAL;
    (* 0 bedeutet unverkäuflich *)
END;

```

besser

```

RECORD
  oddPiece: BOOLEAN;
  name: TEXT;
  saleable: BOOLEAN;
  price: CARDINAL;
END;

```

deklarieren, denn über die Bedeutung der Spezialwerte verliert man schnell den Überblick und wenn man nachträglich Spezialwerte einführt oder entfernt, kann es Funktionen geben, die darauf reagieren müssten, die man aber vergessen hat.

A.2 Vorrangregeln

Die Operatoren besitzen folgende absteigend sortierte Bindungen:

$x.a$	Infix Punkt
$f(x) \ a[i] \ \mathbf{T}\{x\}$	Argumentlisten beginnend mit $(, [, \{$
p^{\wedge}	Postfix \wedge
$+ \ -$	Präfix-Arithmetik
$* \ / \ \mathbf{DIV} \ \mathbf{MOD}$	Infix-Arithmetik
$+ \ - \ \&$	Infix-Arithmetik
$= \ \# \ < \ \leq \ \geq \ > \ \mathbf{IN}$	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
$Q(x)^\wedge$	$(Q(x))^\wedge$	Funktionsauswertung vor Dereferenzierung $^\wedge$
$-p^\wedge$	$-(p^\wedge)$	Dereferenzierung $^\wedge$ vor Negation $-$
$-a * b$	$(-a) * b$	Negation $-$ vor Multiplikation $*$
$a * b - c$	$(a * b) - c$	Multiplikation $*$ vor Subtraktion $-$
$x \text{ IN } s - t$	$x \text{ IN } (s - t)$	Subtraktion $-$ vor Enthaltensein IN
NOT $x \text{ IN } s$	NOT $(x \text{ IN } s)$	Enthaltensein IN vor logischer Negation NOT
NOT $p \text{ AND } q$	(NOT $p)$ AND q	logische Negation NOT vor Konjunktion AND
$A \text{ OR } B \text{ AND } C$	$A \text{ OR } (B \text{ AND } C)$	Konjunktion AND vor Disjunktion OR

A.3 Steuerzeichen in Textliteralen

Steuerzeichen	Name	Ausgabe
<code>\f</code>	FF (formfeed)	Seitenvorschub wird ausgelöst
<code>\n</code>	NL (newline)	Cursor geht zum Anfang der nächsten Zeile
<code>\r</code>	CR (carriage return)	Cursor geht zum Anfang der aktuellen Zeile
<code>\t</code>	HT (horizontal tab)	Cursor geht zur nächsten horizontalen Tabulatorposition
<code>\"</code>		doppelte Anführungszeichen oben: "
<code>\'</code>		Anführungszeichen oben: '
<code>\\</code>		Rückwärtsschrägstrich (Backslash): \
<code>\nnn</code>		ACSII-Zeichen mit Nummer <i>nnn</i> im Oktalsystem.

A.4 Mathematische Funktionen

Es gibt für jeden Fließkommazahlentyp ein Modul welches formatspezifische Manipulationen erlaubt. Die Module heißen RealFloat, LongFloat, ExtendedFloat.

PROCEDURE Scalb($x: \mathbf{T}; n: \mathbf{INTEGER}$): \mathbf{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 .
<pre> TYPE IEEEClass = {SignalingNaN, QuietNaN, Infinity, Normal, Denormal, Zero}; PROCEDURE Class(x: T): IEEEClass; </pre>	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 eingebauten Modula-3-

Funktionen oder mit denen aus LongFloat. In diesem Falle sollten letztere verwendet werden.

Funktion	Funktionsweise
Pi	π , also halber Umfang des Einheitskreises
LogPi	$\ln \pi$
SqrtPi	$\sqrt{\pi}$
E	EULERSches e
Degree	Radianen pro Grad
acos	<i>Arkuskosinus</i> : Umkehrfunktion des Kosinus im Bogenmaß im Intervall $[0, \pi]$. $w = \text{acos}(a) \iff w = \arccos(a)$
asin	<i>Arkussinus</i> : Umkehrfunktion des Sinus im Bogenmaß im Intervall $[-\pi/2, \pi/2]$. $w = \text{asin}(a) \iff w = \arcsin(a)$
atan	<i>Arkustangens</i> : Umkehrfunktion des Tangens im Bogenmaß. $w = \text{atan}(a) \iff w = \arctan(a), w \in [-\pi/2, \pi/2]$
atan2	Winkel zwischen x -Achse und der Strecke $(0, 0) - (a, b)$. $w = \text{atan2}(a, b) \iff w = \arg(a + i \cdot b), w \in [-\pi, \pi]$ falls $w \in [-\pi/2, \pi/2]$ dann $w = \arctan(\frac{a}{b})$
ceil	<i>Obere Gaußklammer</i> : Nächstgrößere Ganzzahl $w = \text{ceil}(a) \iff w = \lceil a \rceil$
cos	<i>Kosinus</i> : Berechnung des Kosinus mit Argument im Bogenmaß $w = \text{cos}(a) \iff w = \cos(a)$
cosh	<i>Kosinus Hyperbolicus</i> : Berechnung des Kosinus Hyperbolicus mit Argument im Bogenmaß $w = \text{cosh}(a) \iff w = \cosh(a)$
exp	<i>Exponentialfunktion</i> : $w = \text{exp}(a) \iff w = e^a$
expm1	<i>Exponentialfunktion</i> verringert um eins: Diese Implementation ist numerisch günstiger bei Exponenten nahe null. $w = \text{exp}(a) - 1 \iff w = e^a - 1$
fabs	<i>Absolutwert</i> : $w = \text{fabs}(a) \iff w = a $
floor	<i>Untere Gaußklammer</i> : Nächstkleinere Ganzzahl $w = \text{floor}(a) \iff w = \lfloor a \rfloor$
fmod	<i>Modulofunktion</i> : Analogon zu $a \bmod b$ für rationale Zahlen $w = \text{fmod}(a, b) \iff w = a \bmod b$

frexp	Aufspaltung von a in $a = f \cdot 2^i$, $f \in [0.5, 1)$ wird zurückgegeben, i in b gespeichert $f = \text{frexp}(a, b) \iff a = f \cdot 2^b$
ldexp	Umkehrfunktion zu frexp. $w = \text{ldexp}(a, b) \iff w = a \cdot 2^b$
log	<i>Logarithmus Naturalis</i> : Logarithmus zur Basis e $w = \log(a) \iff w = \ln(a) = \log_e(a)$
log10	<i>Dekadischer Logarithmus</i> : Logarithmus zur Basis 10 $w = \log_{10}(a) \iff w = \lg(a)$
log1p	<i>natürlicher Logarithmus vom um eins erhöhten Argument</i> : numerisch günstiger bei Argumenten nahe 1 $w = \log_{1p}(a) \iff w = \ln(a + 1)$
modf	Aufspaltung einer Zahl a in Ganzzahlanteil i und Rest $f = a - i$, $f \in [0, 1)$ wird zurückgegeben, $i \in \mathbb{Z}$ in b gespeichert. $f = \text{modf}(a, b) \iff a = b + f$
pow	<i>Potenzfunktion</i> : $w = \text{pow}(a, b) \iff w = a^b$
sin	<i>Sinus</i> : Berechnung des Sinus mit Argument im Bogenmaß $w = \sin(a) \iff w = \sin(a)$
sinh	<i>Sinus Hyperbolicus</i> : Berechnung des Sinus Hyperbolicus mit Argument im Bogenmaß $w = \sinh(a) \iff w = \sinh(a)$
sqrt	<i>Quadratwurzel</i> : $w = \text{sqrt}(a) \iff w = \sqrt{a}$
tan	<i>Tangens</i> : Berechnung des Tangens mit Argument im Bogenmaß $w = \tan(a) \iff w = \tan(a)$
tanh	<i>Tangens Hyperbolicus</i> : 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 1.4 2005/01/23 11:38:09 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
```

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 + Lower;
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  $x \cdot 2^n$ . *)

PROCEDURE Logb(x: T): T RAISES {FloatMode.Trap};
(* Return the exponent of "x". More precisely, return the unique
integer $n$ such that the ratio  $ABS(x) / Base^n$  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  $ABS(x) / Base^n$  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 "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". 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)] * 10exp
| = 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 "1.00000000000000001e23" or
"9.9999999999999999e22". 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      = 3.1415926535897932384626433833D0;
LogPi   = 1.1447298858494001741434273514D0;
SqrtPi  = 1.7724538509055160272981674833D0;
E       = 2.7182818284590452353602874714D0;
Degree  = 0.017453292519943295769236907684D0;  (* 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 loglp (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

```

```
| 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

Numbers written in *italic* refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in **roman** refer to the pages where the entry is used.

Überlauf,	54	Garbage Collector,	105 , 126	Ordner,	12
HILBERT-Matrix,	107	generisches Modul,	137	Overflow,	54
PASCALSche Dreieck,	114	gerichteter Graph,	104	Pakete,	135
3D-TicTacToe,	106	Gleitkommaformat,	54	Parameter,	10 , 46 , 73
absolute Pfade,	13	Gleitkommazahl,	48	Pfad,	13
Absolutwert,	157	globale Variable,	77	Pipe,	22
abstrakte Klasse,	142 , 143	Heimverzeichnis,	14	Platzhalter,	21
aktuellen Verzeichnis,	12	IEEE-Gleitkommazahlen,	55	Potenzfunktion,	158
Algorithmus,	32	Importieren,	131	Prozedur,	72
Argument,	46 , 73	Jokerzeichen,	21	Prozess,	32
Arkuskosinus,	157	Klasse,	142	Prozessor,	32
Arkussinus,	157	Klassenhierarchie,	142	Quadratwurzel,	158
Arkustangens,	157	kompiliert,	39	Qualifizieren,	45 , 131
Assembler,	39	Kosinus,	157	Quota,	11
Ausnahmebehandlung,	124	Kosinus Hyperbolicus,	157	Raummühle,	106
ausprägen,	138	Lazy evaluation,	62	Referenzieren,	105
Betriebssystem,	6	Lebensdauer,	78	reine Funktion,	84
Bezeichner,	46	Literale,	48	Rekursion,	83
Bezugsrahmen,	78	Logarithmus Naturalis,	158	Rekursionsverankerung,	84
Bindung,	154	logischen Verknüpfungen,	59	rekursiv,	83
BlackBox,	135	lokale Variable,	77	relative Pfade,	13
Bourne again Shell,	19	Manual Pages,	10	Schleife,	35 , 86
Call by reference,	73 , 81	Maschinensprache,	39	Schnittstelle,	142
Call by value,	73 , 79	Methode,	142	schrittweise Verfeinerung,	32
Datei,	12	Module,	130	Schublade,	12
Datentypen,	48	Modulofunktion,	157	Shell,	19
Datenverbund,	102	natürlicher Logarithmus,	158	Shell-Skripte,	21
Dekadischer Logarithmus,	158	nebenläufige Pro-		Sichtbarkeitsbereich,	78
deklariert,	48	grammierung,	135	Signatur,	72
Dereferenzieren,	105	Negationsoperator,	60	Sinus,	158
Exponentialfunktion,	157	Obere Gaußklammer,	157	Sinus Hyperbolicus,	158
Feld,	90	Objektorientiertes Pro-		Steuerdatei,	136
Festkommazahlen,	55	grammieren,	142	Tangens,	158
flache Kopie,	110	offenes Feld,	93 , 94	Tangens Hyperbolicus,	158
Funktion,	46 , 72	Optionen,	10	teilweiser Aufdeckung,	134
Gültigkeitsbereich,	78	ordinale Typen,	57	Template,	137
				Textliteral,	43
				TicTacToe,	93

tiefe	Kopie,	110	Vergleich,	59	Wahrheitswerte,	59
Untere	Gaußklammer,	157	Verkettung,	98	Wildcards,	21
Unterklasse,		142	Versionskonflikt,	31	Writer,	80
Unterprogramm,	46,	72	Verweis,	12	Wurzel,	12, 110
Untertyp-Relation,		143	Verzögerte Auswertung,	62		
Unterverzeichnisse,		12	Verzeichnis,	12	Zeiger,	104
Variablen,		20	Vorrang,	154	Zufallsgenerator,	143