

FH Braunschweig/Wolfenbüttel

Hochschule für Technik, Sozial- und Wirtschaftswissenschaften

Fachbereich Informatik

Moderne Programmiertechniken – Eine Einführung mit Oberon

R. Rüdiger

Unterlagen zu den Vorlesungen *Informatik I* und *II*
08/01

Postanschrift: Fachbereich Informatik, FH Braunschweig/Wolfenbüttel, Salzdahlumer Str.
46/48, D-38302 Wolfenbüttel

Vorwort

Das vorliegende Skript ist entstanden aus einem Text, der ursprünglich als Vorlage gedient hat für das Anschreiben des Tafeltextes. Dementsprechend sind große Teile provisorisch wie z.B. eine Reihe von mit der Hand gezeichneten Bildskizzen und wie der nicht im Detail ausgeführte Text. Auch inhaltlich kann und soll dieses Skript nicht mit sorgfältig ausgearbeiteten Lehrbüchern wetteifern; das Studium von Lehrbuchliteratur erübrigt sich daher in keiner Weise. Die Hauptaufgabe des Skripts – jedenfalls entsprechend dem derzeitigen Stand – besteht eigentlich nur darin, das etwas lästige und auch nicht sehr sinnvolle Anschreiben von längeren Quelltexten an die Tafel zu vermeiden, so daß die dadurch freiwerdende Zeit anders – und besser – genutzt werden kann.

Die beiden Vorlesungen *Informatik I* und *II* bilden eine inhaltliche Einheit. Das gesamte Skript besteht aus den Teilen I bis IV, wobei die Teile I und II ungefähr die Vorlesung *Informatik I* sowie die Teile III und IV ungefähr die Vorlesung *Informatik II* abdecken.

Wolfenbüttel, im September 1998

R. Rüdiger

Inhaltsverzeichnis

Literatur	2
Einleitung	3
I Grundlagen	9
1 Oberon: Eine Übersicht anhand von Beispielen	11
1.1 Das Oberon-System	12
1.2 Notation zur Syntaxbeschreibung: EBNF	14
1.3 Erste Beispiele zur Sprache Oberon	16
1.4 Syntax der Sprache Oberon	22
1.5 Weitere Beispiele zur Sprache Oberon	24
2 Grundbegriffe der Informatik. Algorithmen	37
2.1 Komponenten eines Rechners	37
2.2 Programmiersprachen	40
2.3 Betriebssysteme	43
2.4 Algorithmen	44
II Strukturierte Programmierung. Prozedurale Abstraktion	55
3 Elemente der Programmierung	57
3.1 Symbole und Grundtypen	57
3.2 Rechnerinterne Darstellung ganzzahliger Werte	66
3.3 Deklarationen, Ausdrücke und Wertzuweisungen	67
3.4 Kontrollstrukturen	87
4 Prozedurale Abstraktion. Konkrete Datenstrukturen	91
4.1 Allgemeines zur Rolle der Abstraktion	91
4.2 Prozeduren	91
4.3 Strukturierte Datentypen	105
4.4 Konkrete Datenstrukturen	126

III	Modularisierung und Abstrakte Datentypen	135
5	Modularisierung. Einfache Datenabstraktion	137
5.1	Die Rolle von Modulen in der Softwaretechnik	137
5.2	Aufbau eines Moduls. Modulzustand	139
5.3	Intermodulare Beziehungen	143
5.4	Abstrakte Datenstrukturen	145
6	Abstrakte Datentypen und Module	149
6.1	Prioritätswarteschlange als ADT	149
6.2	Wert- und Referenzsemantik	152
6.3	Zeigertypen und dynamische Variablen	153
6.4	Dynamische Datenstrukturen	158
IV	Jenseits von Pascal und Modula	173
7	Objektorientierung	175
7.1	Informelle Einführung	175
7.2	Technische Umsetzung in Oberon-2	177
7.3	Zusammenspiel der OO-Konzepte: Beispiele	193
8	Erweiterbare Systeme	205
8.1	Up-Calls	205
8.2	Simulationspaket nach Reiser/Wirth ([RW92])	206
8.3	Strukturmodelle erweiterbarer Systeme. Graphik-Editor	213

Abbildungsverzeichnis

1.1	Oberon-Display	12
1.2	MS-DOS-Delete-Kommando	16
1.3	MS-DOS-Set-Kommando	16
1.4	Oberon-System.Free-Kommando	17
1.5	Parametrisierung einer geometrischen Darstellung	18
1.6	Ausgabeformat	19
1.7	Aufbau von Modul	22
1.8	IMPORT-Liste	23
1.9	DeclarationSequence (vollständig)	23
1.10	ProcedureDeclaration (vollständig)	24
1.11	Geschachtelte Struktur von Modulen	25
1.12	StatementSequence	26
1.13	AssignmentStatement	26
1.14	Aufbau eines Arrays	31
1.15	Import-Beziehungen	36
2.1	Komponenten eines Rechners	37
2.2	Diskettenaufbau: Prinzip	38
2.3	Belegung eines Bytes	39
2.4	Virtuelle Oberon-Maschine	44
2.5	Acht-Damen-Problem	47
2.6	Simulation von Raktenstart und -landung	48
2.7	rekursive Berechnung der Fibonacci-Zahlen	52
2.8	Busy Beaver	53
3.1	Syntax von Namen	58
3.2	Syntax von Zahlen	58
3.3	Syntax von real	58
3.4	Syntax von integer	59
3.5	Syntax von ScaleFactor	59
3.6	Syntax von Zeichenkonstanten	60
3.7	Syntax von Zeichen	60
3.8	Syntax von Zeichenketten	61
3.9	zur binären Zahlendarstellung	66
3.10	Binäre Zahlendarstellung: Zweierkomplementdarstellung	67
3.11	Syntax von Deklarationsfolge	69
3.12	Syntax von Ausdrücken	70
3.13	Datentypen von Zahlen	71

3.14	zu Booleschen Variablen	75
3.15	Natürliche Zahlen	78
3.16	Schnitt von Mengen: $x \in A, x \in B$	78
3.17	Vereinigung von Mengen: $A \cup B$	78
3.18	Differenz von Mengen: $A \setminus B$	79
3.19	Leerer Durchschnitt von Mengen: $A \cap B = \emptyset$	79
3.20	Enthaltensein von Mengen: $A \subset B$	79
4.1	Sichtbarkeitsbereiche für Namen	96
4.2	mathematischer Begriff <i>Funktion</i>	99
4.3	Simpsonsche Regel	100
4.4	Umkehrung einer Zeichenkette	104
4.5	Darstellung der Struktur ARRAY	107
4.6	Zeichenkette in ARRAY n OF CHAR	114
4.7	Darstellung von Records	120
4.8	Prioritätswarteschlange	127
4.9	Heapstruktur	130
4.10	Vollständiger Baum: Abbildung auf Array	130
4.11	Prioritätswarteschlange: Knoten einfügen	131
4.12	Prioritätswarteschlange: Knoten löschen	131
5.1	Das Phasenmodell, vgl. [Eng88, S. 546]	138
5.2	Computersystem und Benutzermaschine	139
5.3	zur hierarchischen Modularisierung	139
5.4	Tabellenzugriff. Anwendung des Geheimnisprinzips	140
5.5	Blockstruktur in Modul	140
5.6	Geschichtete Struktur zur Nachbildung lokaler Module	141
5.7	Modulgraph	143
5.8	Import-Beziehungen von Modulen	145
6.1	Ansprechen von Objekten	152
6.2	Ausschnitt aus Arbeitsspeicher	153
6.3	„geerdete“ Zeiger	154
6.4	Dynamische Variablen	154
6.5	mit zugewiesenen Werten	155
6.6	Zuweisung von dynamischen Variablen	155
6.7	Zuweisung von Zeigern	156
6.8	Lineare Liste	161
6.9	Darstellungen von Zeigern	162
6.10	Initialisierung von p : p := NIL;	163
6.11	Schritt 1.1: NEW(q);	163
6.12	Schritt 1.2: q.key := 3;	164
6.13	Schritt 1.3: q.next := p;	164
6.14	Schritt 1.4: p := q;	165
6.15	Schritt 2.1: NEW(q);	165
6.16	Schritt 2.2: q.key := 7;	166
6.17	Schritt 2.3: q.next := p;	166
6.18	Schritt 2.4: p := q;	167

6.19	Geordneter binärer Baum (Binärer Suchbaum)	169
7.1	Spezialisierung und Generalisierung	176
7.2	Spezialisierung und Generalisierung, andere Darstellung	176
7.3	Vererbungsbeziehungen in einem Graphikeditor	177
7.4	Vererbungsbeziehungen, anders dargestellt	178
7.5	angestrebte Datenstruktur für den Graphikeditor	178
7.6	Basistyp Person	179
7.7	Erweiterter Typ Student	179
7.8	Mehrfache Typerweiterung	180
7.9	Typerweiterung und Zuweisung (1)	180
7.10	Typerweiterung und Zuweisung (2)	180
7.11	Typerweiterung und Zuweisung (3)	181
7.12	Instanz eines erweiterten Record-Typs	181
7.13	Zuweisung bei Zeigern verschiedener Typen	182
8.1	Up-Call und Down-Call	206
8.2	Importbeziehungen der Module des Simulationspaketes	207

Literaturverzeichnis

- [Eng88] H. Engesser, editor. *Duden Informatik*. Dudenverlag, 1988.
- [Gut94] J. Gutknecht. Oberon system 3: Vision of a future software technology. *Software — Concepts and Tools*, 15:26–33, 1994.
- [Knu73a] Donald E. Knuth. *The Art of Computer Programming. Fundamental Algorithms*, volume 1. Addison-Wesley, 2nd edition, 1973.
- [Knu73b] Donald E. Knuth. *The Art of Computer Programming. Sorting and Searching*, volume 3. Addison-Wesley, 1st edition, 1973.
- [Knu81] Donald E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*, volume 2. Addison-Wesley, 2nd edition, 1981.
- [Mey90] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.
- [MLK95] J.R. Mühlbacher, B. Leisch, and U. Kreuzeder. *Programmieren mit Oberon-2 unter Windows*. Hanser, 1995.
- [Mö93] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, 1993.
- [Nau63] Revised report on the algorithmic language Algol 60. Naur, N. (ed.): *Comm. ACM*, 6:1–7; *Computer Journal*, 5:349–367; *Numerical Mathematics*, 4:420–453, 1963.
- [Rei91] M. Reiser. *The Oberon System. User Guide and Programmer's Manual*. Addison-Wesley, 1991.
- [Rü97a] R. Rüdiger. Aspekte der Auswahl von Programmiersprachen für die Lehre. Ein Diskussionbeitrag. Argumente für Oberon als erste Sprache im Informatik-Studium. Technical report, Institut für Verteilte Systeme, FH Braunschweig/Wolfenbüttel, 1997.
- [Rü97b] R. Rüdiger. Programmierung Erweiterbarer Systeme. Eine Kurzeinführung in Oberon. Technical report, Institut für Verteilte Systeme, FH Braunschweig/Wolfenbüttel, 1997.
- [RW92] M Reiser and N. Wirth. *Programming in Oberon. Steps beyond Pascal and Modula-2*. Addison-Wesley, 1992.
- [RW94] M. Reiser and N. Wirth. *Programmieren in Oberon. Das neue Pascal*. Addison-Wesley, 1994.

- [Swi93] R. Switzer. *EIFFEL. An Introduction*. Prentice Hall, 1993.
- [WG92] N. Wirth and J. Gutknecht. *Project Oberon. The Design of an Operating System and Compiler*. Addison-Wesley, 1992.
- [Wir71] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.
- [Wir77] N. Wirth. *Compilerbau. Eine Einführung*. Teubner, 1977.
- [Wir85] N. Wirth. *Programmieren in Modula-2*. Springer-Verlag, 1985.
- [Wir96] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996.

Einleitung

Inhalt und Konzept der Vorlesung

Eine einführende Vorlesung zur Informatik zu konzipieren und zu halten ist eine interessante aber keine besonders leichte Aufgabe. Einige wesentliche Gründe dafür sind die folgenden:

- Die Informationstechnik entwickelt sich extrem schnell weiter. Es ist daher permanent zu überprüfen, ob die Inhalte der Lehrveranstaltung noch ausreichend aktuell sind. Sie dürfen aber auf der anderen Seite nicht so stark abhängig von Neuerungen sein, daß sie innerhalb von wenigen Semestern ihren Wert bereits wieder verloren haben.
- Es soll während des Studiums, also während einer Zeitspanne von ca. 4 Jahren, eine Basis für ca. 40 Jahre Berufstätigkeit gelegt werden. Es gibt heute keine sichtbaren Gründe, daß sich die Informationstechnik in den kommenden Jahrzehnten langsamer entwickeln wird als in den vergangenen Jahrzehnten; viele Prognosen sagen eher das Gegenteil voraus.
- Meistens gibt es unter den Studierenden ein extrem weites Spektrum an Vorkenntnissen. Das geht von „praktisch noch nie am Rechner gesessen“ bis zu bereits vorhandener umfangreicher Berufserfahrung in der Informationstechnik.
- Es gibt nicht ganz selten eine realitätsferne Erwartungshaltung, was ein Informatikstudium leisten sollte. Insbesondere der Abstraktionsgrad, der erforderlich ist, wird häufig grob unterschätzt.

Das Studium insgesamt ist in zwei Abschnitte unterteilt. Im ersten Studienabschnitt werden im wesentlichen Grundlagen und Konzepte vermittelt, gewissermaßen „indirektes Wissen“ wie z.B. die Fähigkeit, sich selbst weiteres Wissen anzueignen; beliebte Schlagwörter in diesem Zusammenhang sind „Das Lernen lernen“ oder „LLL“ („Lebens-Langes Lernen“). Im zweiten Studienabschnitt geht es zwar auch weiterhin um Konzepte, aber als ein weiterer wichtiger Aspekt tritt hier auch die direkte Vermittlung berufsbezogenen Detailwissens hinzu. Es wird dann z.B. unmittelbar berücksichtigt, welche Variante einer Entwicklungsumgebung im späteren potentiellen beruflichen Umfeld mit einiger Wahrscheinlichkeit genutzt werden wird.

Eine konkrete produktbezogene Schulung bereits in den ersten Semestern wäre wenig sinnvoll, da praktisch alle Produkte in 4 bis 5 Jahren ihr Aussehen vollständig verändert haben. Außerdem sind gute professionelle Produkte nicht unbedingt auch didaktisch geeignet.

Die grundsätzlichen konkreten Ziele der Vorlesungen *Informatik I* und *II* kann man kurz folgendermaßen zusammenfassen:

- Es geht um grundlegende Begriffe der Informatik, insbesondere die Fragen
- was ist ein *Algorithmus*, was sind *Datenstrukturen* und wie sieht deren Zusammenspiel aus?
- Was bedeutet Programmentwicklung heute? Stichworte sind: Abstraktionsstufenmodell, prozedurale Abstraktion, konkrete Datenstrukturen, abstrakte Datenstrukturen, abstrakte Datentypen (ADTs), Objektorientierte Programmierung (OOP), Erweiterbarkeit. Das Fachgebiet *Softwaretechnik* (*Softwareengineering*) bildet einen größeren Rahmen, in dem diese Begriffe auftreten.
- Um über Konzepte dieser Art zu sprechen muß eine Notation, eine Sprache (eine „Programmier“sprache), zur Verfügung stehen. Wie geht man daran, diese zu lernen? Wesentlich dabei sind: das grundsätzliche Vorgehen, die Gewöhnung an eine Genauigkeit in der Nomenklatur und im Denken. So sollen im konkreten Kontext die richtigen Begriffe verwendet werden, nicht mehr oder weniger selbst gewählte oder freizügig aus anderen Bereichen übertragene.
- Es soll trainiert werden, wie man sich den Umgang mit einem neuen, für den Lernenden noch unbekannten System aneignet.

Der Begriff *Programmieren* bedarf einiger Erläuterungen. Er wird ähnlich wie der Begriff *Rechnen* in äußerst verschiedener Weise genutzt. Rechnen kann z.B. den Umgang mit dem kleinen Einmaleins bedeuten, aber auch – wenn ein Mathematiker davon spricht – den Umgang mit wesentlich abstrakteren Gebilden, z.B. der Algebra, wie etwa Vektoren, Matrizen, Tensoren und anderen Dingen oder auch dem Lösen von Differentialgleichungen. So wird *Programmieren* auf der einen Seite im Sinne von *Schreiben von Programmcode* verwendet, einer evtl. sehr elementaren Tätigkeit, wenn das Konzept für das Programm fertig vorliegt. Auf der anderen Seite gibt es aber auch unter den berühmtesten Informatikern solche, die ihren Beruf als den des „Programmierers“ umschreiben. Wer sich informieren möchte, was „Programmieren“ in diesem letzteren Sinne bedeutet, schaue nach in den drei Büchern von D.E. Knuth ([Knu73a, Knu81, Knu73b]), die auch heute noch eine Basis für wesentliche Teile der Informatik bilden und deren Titel bereits das Wort *Programming* enthalten.

Um der obigen Zweiteilung des Studiums mit den beschriebenen Schwerpunkten gerecht zu werden, hat sich der Fachbereich Informatik im WS 94/95 entschieden, als erste Programmiersprache die Sprache *Oberon* als ein Beispiel einer modernen Programmiersprache einzusetzen. Sie wird verwendet auf der Basis eines Betriebssystems, ebenfalls *Oberon* genannt. Letzteres ist allerdings nicht Lehr- und Prüfungsgegenstand.

Einige Vorteile der Sprache Oberon im Zusammenspiel mit dem System Oberon sind:

- Die Sprache ist extrem kompakt. Dieses bedeutet einen erheblichen Zeitgewinn,
- der dazu genutzt werden kann, möglichst tiefgehend bereits in den ersten zwei Semestern einige der oben angesprochenen Begriffe einzuführen.
- Oberon ist ein Mitglied der Algol/Pascal-Sprachfamilie, die sich als didaktisch besonders gut erwiesen hat.

- Die starke Divergenz in den Vorkenntnissen der Studierenden wird wenigstens etwas ausgeglichen durch eine etwas unkonventionelle Sprache.

Ein Nachteil von Oberon ist der, daß es in Industrieunternehmen kaum verbreitet ist und daß Oberon sozusagen „mit dem Rest der Welt nichts zu tun haben will“ – allerdings aus voller Absicht der Autoren Niklaus Wirth und Jürg Gutknecht heraus. Man kann durchaus darüber geteilter Meinung sein, ob dieser letztere Punkt im Rahmen des Konzeptes für den ersten Studienabschnitt, wie es oben beschrieben worden ist, wirklich ein Nachteil ist. Weiteres zu den Konzepten und zu den bisherigen Erfahrungen mit Oberon ist zusammengefaßt in [Rü97b, Rü97a].

Es gibt stets eine gewisse Tendenz – auch unter etablierten Informatikern –, nach der endgültigen, richtigen, wahren, für alle Probleme am besten geeigneten und ästhetisch schönsten Sprache zu suchen. Um es deutlich zu sagen: niemand meint, übrigens auch der Schöpfer der Sprache Oberon, Niklaus Wirth, nicht, das wäre Oberon. Die allgemeine Meinung heute ist die, daß es eine solche Sprache nicht gibt und aller Voraussicht auch nicht geben wird. Der Wert einer Sprache hängt außerordentlich stark ab von dem Kontext, in dem sie verwendet wird. In Betrieben sind das häufig auch wirtschaftliche Zwänge, so daß dann rein wissenschaftliche Aspekte der Informatik kaum eine Rolle spielen.

Um dieses auch gegenüber den Studierenden deutlich zu machen, soll daher bereits im zweiten Semester zusätzlich eine weitere und zur Zeit außerordentlich aktuelle Sprache eingeführt werden, nämlich *Java*. Die Hoffnung, die sich mit einer zusätzlich eingeführten Sprache verbindet, ist hauptsächlich die, daß das Wissen über Konzepte der oben beschriebenen Art nicht ausschließlich nur mit den Konstrukten einer einzigen speziellen Sprache verknüpft wird. Erfahrungsgemäß bedeutet eine solche Verknüpfung, wenn sie sich zu sehr verfestigt, ein späteres erhebliches berufliches Handikap. Eine solche Einführung entspricht auch den Anforderungen der beruflichen Praxis, wo es selbstverständlich ist, daß sehr schnell und im wesentlichen in Eigeninitiative eine neue Sprache erlernt wird. Natürlich bedeutet es auch eine erhebliche Anforderung an die Studierenden, sich bereits in einem so frühen Stadium des Studiums mit einer weiteren Sprache auseinanderzusetzen. Für die Lehre an einer Hochschule sind die Anforderungen der Berufswelt aber vorgegeben und nicht frei wählbar; ein zu vorsichtiges und „rücksichtsvolles“ Heranführen an die Informatik würde angesichts heutiger Anforderungen einfach nur Praxisferne bedeuten.

Java gleicht äußerlich der Sprache C++, strukturell in einiger Hinsicht Oberon, ist aber in anderer Hinsicht davon wiederum grundlegend unterschiedlich. Diese strukturellen Unterschiede zu verstehen ist ein wesentliches Ziel.

Es ist vielleicht nicht ganz überflüssig zu erwähnen, was in den Vorlesungen *Informatik I* und *II* ausdrücklich nicht erreicht werden soll und für wen die Vorlesungen nicht geeignet sind. Es gibt gelegentlich durch eine falsche Erwartungshaltung durchaus auch Enttäuschungen.

- Es wird – wie bereits ausgeführt – keine Produktschulung in den *ersten* Semestern geboten. Das ist durchaus ein Nachteil für alle diejenigen, die ihr Studium frühzeitig abbrechen müssen oder wollen,
- ebenso auch für diejenigen, die nebenher bereits beruflich im Informatikbereich arbeiten und das Studium von Anfang an nur als „Begleitmaßnahme“ sehen. Von

wirklichem Wert ist das Studium nur dann, wenn es auch wie vorgesehen zuende geführt wird.

- Es werden keine detaillierten Kenntnisse zu konkreter Hardware vermittelt, etwa zu Grafikkarten, Festplatten usw..
- Es werden in den ersten Semestern keine Kenntnisse der Art vermittelt wie z.B. *Wie schreibt man Linux- oder Windows 3.1/95/98/xx/NT-Anwendungen?*. Das folgt aber durchaus in den höheren Semestern und dann auch mit der aktuellen Versionsnummer.
- Auch die folgende Frage wird nicht beantwortet werden: *Wie lauten die 20 wichtigsten MS-DOS-Kommandos?* Letzteres ist übrigens ein Beispiel dafür, daß in einem Informatikstudium unterstellt wird, daß sich die Studierenden viele Dinge selbst aneignen. Wenn etwas nicht ausdrücklich erwähnt bzw. behandelt wird, so heißt das noch lange nicht, daß es unwichtig wäre. Das gilt z.B. typischerweise – um das obige DOS-Beispiel etwas zu aktualisieren – für den Umgang mit den sog. Browsern, mit denen man sich im WWW bewegen und informieren kann. Ein weiteres Beispiel dieser Art ist etwa die Textverarbeitung. Hier müssen kurze Anleitungen reichen bzw. entsprechende Spezialkurse, die an verschiedenen Stellen angeboten werden, u.a. vom Rechenzentrum der Hochschule.

Literaturhinweise

Seit 1991 sind eine ganze Reihe von Büchern zu Oberon erschienen.

- 1991:** Reiser beschreibt in dem ersten zu Oberon erschienenen Buch ([Rei91]) das Oberon-System; dieses Buch kann man als Referenz und Einführung in die Benutzung der Standardbibliothek ansehen. Das Buch setzt eine Kenntnis der Sprache Oberon-1 voraus. Es enthält gleichzeitig sehr viele vollständige Beispiele, wie das System zu benutzen ist.
- 1992:** Das Buch [RW92] von Reiser und Wirth enthält die (englische) Sprachbeschreibung von Oberon, insbesondere auch eine sehr kurze Einführung in Oberon-2. Dieses Buch stellt faktisch die verbindliche Sprachdefinition von Oberon dar.
- 1992:** Die Ergebnisse des gesamten Oberon-Projektes sind in dem umfangreichen Band [WG92] von den Oberon-Autoren Wirth und Gutknecht zusammengefaßt. Dieser Projektbericht enthält außer einer Diskussion vieler Design-Entscheidungen auch die vollständigen Quellen des Oberon-Systems und des Compilers.
- 1993:** Mössenböck hat in [Mö93] die Technik der OOP anhand von Oberon-2 beschrieben. Das Buch enthält auch eine kurze Einführung in Oberon-2, ist aber nach dem ausdrücklichen Bekunden des Autors keine Einführung in Oberon-2. Es enthält außer vielen Beispielen für einen sinnvollen Einsatz von OOP auch ein vollständiges Mini-Oberon-System, genannt Oberon-0, welches im ETH-Oberon-System lauffähig ist und ein größeres Beispiel eines erweiterbaren Systems darstellt (ETH: Eidgenössische Technische Hochschule).

- 1994:** [RW94] ist die deutsche Übersetzung von [RW92]. Dieser Band kann so betrachtet werden, daß hier die (verbindlichen) deutschen Begriffskorrespondenzen zu den englischen Begriffen festgelegt sind.
- 1995:** [MLK95] ist eine eher elementare Einführung in die Sprache Oberon-2, die hier eingebettet ist in die Entwicklungsumgebung *Oberon/Pow!* unter Windows. Durch die andersartige Einbettung ist auch die Sprache substantiell geändert. Es gibt u.a. z.B. ausdrücklich deklarierte Hauptprozeduren sowie eine vordefinierte Prozedur *Dispose*, da kein automatischer Garbage Collector vorhanden ist.
- 1996:** [Wir96] ist eine Neufassung des klassischen Bändchens von Wirth über Compilerbau [Wir77] auf der Basis der Sprache Oberon. Auch hier ist ein lauffähiger Compiler im Quellcode vollständig abgedruckt. Der in dem Buch wiedergegebene Quellcode steht auf dem oben angegebenen ftp-Server ebenfalls zur Verfügung.

Geschichtliche Entwicklung

Im folgenden sind stichwortartig einige geschichtliche Daten zur Entwicklung der Algol/Pascal-Sprachfamilie (Algol-60, Pascal, Modula, Oberon), insbesondere zur Entwicklung von Oberon zusammengestellt. Die Angaben zu Oberon sind überwiegend dem Projektbericht [WG92, p. 1] entnommen.

- 1960:** Entwurf von Algol 60 (Algorithmic Language) ([Nau63])
- 1970:** Pascal ([Wir71])
- 1979:** Modula-2 ([Wir85])
- 1985:** Beschluß zum Entwurf eines neuen Betriebssystems (Wirth, Gutknecht) (Ende 1985)
- 1985:** Ceres-1 mit 32-bit-Prozessor NS 32032
- 1986:** Definition des Systems und Beginn der Programmierarbeiten (Anfang 1986)
- 1987:** Erste Version von Oberon (Mitte 1987)
- 1988:** System zugänglich für größeren Benutzerkreis (Mitte/Ende 1988)
- 1988:** Ceres-2 mit Prozessor NS 32532
- 1989:** Oberon wird anstelle von Modula-2 als einführende Programmiersprache an ETH eingesetzt.
- 1989:** Start von Projekt: Portierung des Systems auf kommerziell erhältliche Workstations
- 1990:** Ceres-3 (low cost-Workstation) mit Prozessor NS 32GX32
- 1992:** Projektbericht zum Oberon-System und Compiler ([WG92])
- 1994:** Weiterentwicklung des Oberon-Systems zum *System-3* mit Gadgets ([Gut94])

Teil I

Grundlagen

Kapitel 1

Oberon: Eine Übersicht anhand von Beispielen

Der Name *Oberon* steht für zweierlei, nämlich

1. für ein Betriebssystem und
2. für eine Programmiersprache.

Oberon ist entstanden in den Jahren seit 1985 aus der Unzufriedenheit einer Informatik-Gruppe in der ETH Zürich um Niklaus Wirth und Jürg Gutknecht mit den heutigen (hochkomplexen) Fenstersystemen.

(N. Wirth: Autor von Pascal und von Modula-2, eine der herausragenden Persönlichkeiten in der Informatik)

Besondere Eigenschaften von Oberon:

- Oberon ist ein erweiterbares System, d.h., das System kann vom Benutzer weitgehend an seine Bedürfnisse angepaßt werden: z.B. wird nicht unterschieden zwischen Benutzer- und Systemkommandos
- wesentliche Neuerung: sog. nicht-modale Texte: Texte auf dem Bildschirm werden nicht unterschiedlich behandelt abhängig davon, ob sie z.B. in einem Dokument auftreten oder z.B. in einem Systemausgabebereich; konkrete Anwendung: Aufruf von Kommando aus Dokument heraus; Hypertextfähigkeit, im Prinzip für weltweiten Zugriff auf Daten im WWW
- Texte und Graphiken sind integriert.
- Die formale Beschreibung der Sprache Oberon ist extrem kurz und einfach.
- Das ganze System ist (gemessen an den Möglichkeiten) extrem klein.
- Oberon ist eine Weiterentwicklung von Pascal und fast gleich zu Modula-2. Gegenüber Modula-2 ist wenig, aber sehr wichtiges hinzugefügt und einiges weggelassen.
- Oberon ist eine *objektorientierte* Sprache. In einiger Hinsicht geht Oberon über OOP hinaus (*Extensible Programming*).

Grundsätzlicher Aufbau der Vorlesung:

- Kurze Übersicht über das *System* Oberon
- Kurze Einführung in die *Sprache* anhand von erläuterten Beispielen (damit die Übungen beginnen können)
- allgemeine Einführung in Grundbegriffe der Informatik
- systematische Einführung in die Sprache Oberon (in mehreren Kapiteln)
- grundlegende Strukturen und Algorithmen der Informatik unter Verwendung von Oberon als sprachlichem Hilfsmittel
- Objektorientierte Programmierung mit den Sprachmitteln von Oberon.

1.1 Das Oberon-System

nach Reiser ([Rei91]):

Grundbegriffe

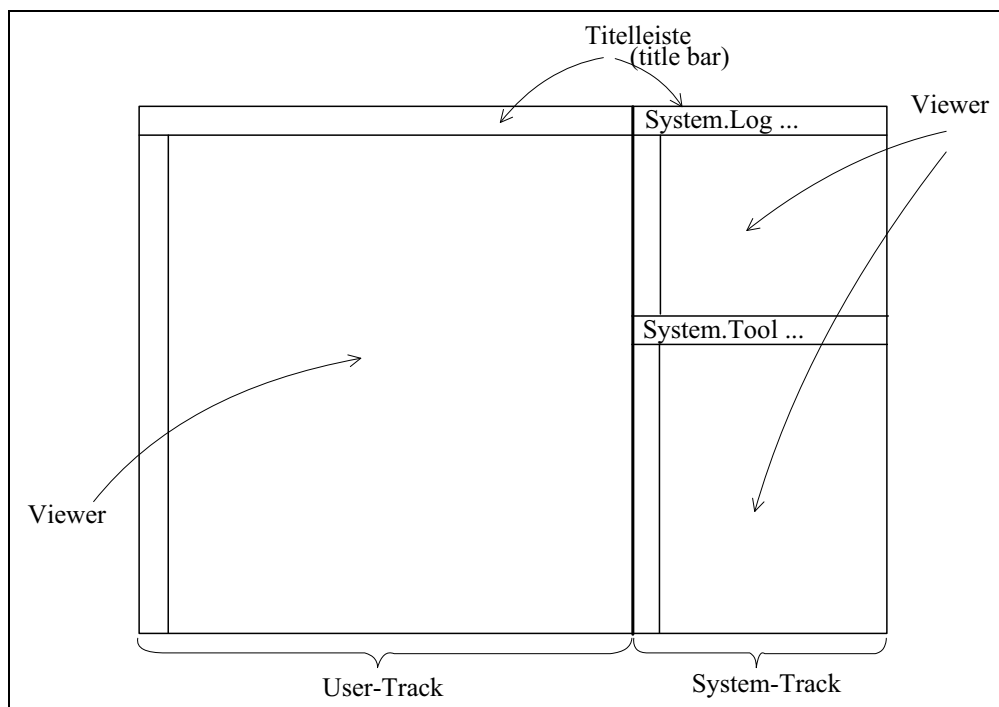


Abbildung 1.1: Oberon-Display

s. Abbildung 1.1

Erläuterungen zum verteilten Text *Getting Started with Oberon for Windows*:

- Special Keys: Bedeutung bestimmter Tasten (Oberon ist ursprünglich nicht für Personal Computer (PCs) geschrieben)

- Basic Commands: Caret setzen, Kommando ausführen, ...
- Scrolling (in scroll bar to the left of the text)
- Viewer Layout (Moving the Window)
- Advanced Commands: Texte kopieren, Schriftarten, Schriftfarbe umsetzen
- Parameter Conventions: Mechanismen der Parameterübergabe an Kommandos
 - `Mod.Cmd`: Kommando z.B. ohne Parameter; Bsp: `System.Time`
 - `Mod.Cmd~`: Kommando erwartet Liste von Namen; Liste wird beendet durch `~`, z.B. `System.DeleteFiles~`
 - `Mod.Cmd*`: Markierter Viewer (Markierung setzen mit Taste F1) wird als Parameter genommen; z.B. `Compiler.Compile*`: Text in Fenster wird übersetzt
 - `Mod.Cmd^`: Kommando bezieht sich auf die letzte Auswahl (selection), z.B. `Edit.Open^`
 - `!Mod.Cmd`: riskantes Kommando: vor Ausführen: Blank hinter `!` einfügen
- Editing: einige Editor-Kommandos (Datei öffnen zum Schreiben, Ändern, ..., Datei sichern auf der Festplatte)

Einige wichtige Dateibezeichner:

genereller Aufbau: `FileName.FileType`, wobei `FileType` z.B.:

- `Mod`: Oberon Quelltext (Modul)
- `Text`: Textfile, erzeugt z.B. mit Standard-Editor
- `Doc`
- `Graph`: graphische Dokumente
- `Pict`
- `Obj`: Oberon Modul (Objektfile)
- `Sym`: Symbolfile (Erklärung später)

Beispiele zu Kommandos

Jedes Kommando kann man sich in einer Datei (z.B. in `Name.Tool`) notieren. Wenn man diese Datei sichert, dann steht das Kommando jederzeit zum Aufruf zur Verfügung (auch — falls entsprechend vorgesehen — mit zahlreichen speziellen Parametern)

Hinweis: Die Parameterübergabe in den folgenden Beispielen kann auch einfacher erfolgen: Einzelheiten in den Übungen

Datei öffnen

`Edit.Open Test.Mod`

Wirkung: Die Datei `Test.Mod` wird zum Bearbeiten geöffnet

Datei sichern

`Edit.Store` (von Menü in Titelzeile aus)

Wirkung: Die Datei `Test.Mod` wird auf der Festplatte gesichert

Datei kopieren

`System.CopyFiles Test.Mod => Test1.Mod ~`

Wirkung: weiteres Exemplar der Datei `Test.Mod` erzeugen: Name der neuen Datei `Test1.Mod`

Datei löschen

`System.DeleteFiles Test1.Mod ~`

ACHTUNG: `~` nicht vergessen: es werden alle Dateien gelöscht bis `~` auftritt

Wirkung: Datei `Test1.Mod` wird gelöscht

Datei umbenennen

`System.RenameFiles Test.Mod => Test2.Mod ~`

Wirkung: Datei `Test.Mod` erhält den neuen Namen `Test2.Mod`

Übersicht über vorhandene Tools verschaffen

`System.Directory *.Tools`

Wirkung: Es wird eine Liste aller im System vorhandenen Dateien mit Dateityp `Tool` ausgegeben.

Oberon-System verlassen

`System.Quit` (Alternative: `Alt F4`)

Allgemeiner Hinweis: Es gibt Methoden herauszufinden, welche Kommandos z.B. der allgemeinen Form `System.xxx` es überhaupt gibt (Einzelheiten später).

1.2 Notation zur Syntaxbeschreibung: EBNF

Durch die Syntaxbeschreibung einer Sprache wird die Grammatik dieser Sprache festgelegt. Genaue Definitionen: in Theoretischer Informatik (Vorlesung Informatik III)

In jedem Betriebssystem, in jeder Programmiersprache tritt die Frage auf: welche Konstrukte sind syntaktisch zulässig (d.h. von der Grammatik her erlaubt)

Die Frage, welche sinnvoll sind, ist eine weitere Frage.

Semantik: Bedeutungslehre: hier: Beschreibung durch umgangssprachlichen Text.

z.B. in MS-DOS:

- `DIR` (erlaubt)
- `DEL TEST.TXT` (erlaubt)(*)
- `DEL TEST1.TXT TEST2.TXT TEST3.TXT` (nicht erlaubt)
- `DEL A:\TEXTE\TEST.TXT` (erlaubt, evtl. gleiche Wirkung wie (*))

Frage: Wie kann man das systematisch beschreiben? In Praxis werden zwei Methoden verwendet (für sog. *kontextfreie* Grammatiken; Definition davon und Beispiele für nicht-kontextfreie Grammatiken: in Theoretischer Informatik):

1. textuell
2. graphisch

im Detail:

1. **textuell** (sog. EBNF (**E**xtended **B**ackus-**N**aur **F**orm); Einführung der BNF erfolgte im Zusammenhang mit der Programmiersprache ALGOL 60);

wenn A und B z.B. Parameter eines Kommandos sind, dann bedeutet:

- `[A]`: A ist optional, d.h., es tritt genau 0-mal oder genau 1-mal auf
- `{A}`: A tritt beliebig häufig auf, also entweder 0-mal oder 1-mal oder 2-mal ...
- `A|B`: Alternative: entweder A oder B: genau eines der beiden
- Symbole sind kombinierbar: z.B. bedeutet `[A|B]`: entweder nichts oder A|B, also A oder B, insgesamt also entweder nichts oder A oder B
- Symbole, die „wörtlich“ übernommen werden: Terminalsymbole; werden in der Syntaxbeschreibung in geeigneter Weise kenntlich gemacht, i.a. durch Großschreibung (z.B. `BEGIN` in der Sprache Oberon) oder durch Großschreibung des ersten Buchstabens (z.B. `System.Free` im System Oberon) und durch " " bei Sonderzeichen, z.B. `;"`.
- Beispiele:
 - MS-DOS-Delete-Kommando = `DEL [Laufwerk":"] [Pfad] Dateiname ["/p"]`.
 - MS-DOS-Set-Kommando = `SET [Variable "=" [Zeichenfolge]]`.
Schreibweise mit " " ist etwas unübersichtlich. Daher wird gelegentlich bereits vorher festgelegt, welche der Symbole Terminalsymbole sind; " " wird dann weggelassen.
Im Oberon-**System** werden die folgenden Terminalsymbole so behandelt:
`~ * ^ => /d /s "`
 - `System.Free-Kommando = System.Free { moduleName [ * ] } ( ~|^ )`.
Die Klammern (und) sind sog. *Metasymbole*, gehören also der *Metasprache* an.
Schreibweise ist nützlich, wenn eine textuelle Beschreibung erforderlich ist (z.B. bei direkter Verwendung auf einem Rechner oder Übertragung über ein Rechnernetz); Schreibweise ist für Menschen aber immer noch unübersichtlich

für direkte Lesbarkeit: graphische Darstellung geeigneter

2. **graphisch** (Syntaxdiagramm)

MS-DOS-Delete-Kommando: s. Abbildung 1.2

MS-DOS-Set-Kommando: s. Abbildung 1.3

Oberon-**System.Free**-Kommando: s. Abbildung 1.4

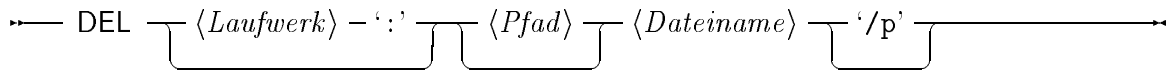
MS-DOS-Delete Kommando:

Abbildung 1.2: MS-DOS-Delete-Kommando

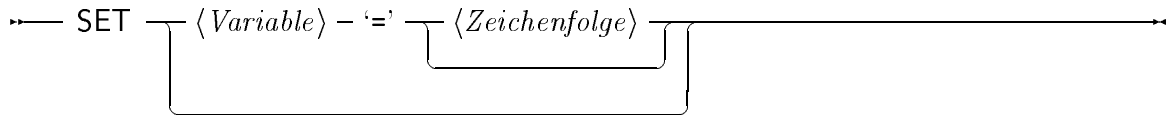
MS-DOS-Set-Kommando:

Abbildung 1.3: MS-DOS-Set-Kommando

Terminalsymbole werden meistens durch *Kreis* oder *Oval* charakterisiert, Nicht-Terminalsymbole durch *rechteckigen Kasten*

Einzelheiten zur Syntax der **Sprache** Oberon: später

zum Begriffspaar Syntax / Semantik:

- *Syntax*: Lehre vom Satzbau, hier: Regeln, die den formalen Aufbau einer Programmier- oder Kommandosprache festlegen, allgemein: einer sog. formalen Sprache
- *Semantik*: Bedeutungslehre: hier: Beschreibung durch umgangssprachlichen Text: eine Formalisierung ist erheblich aufwendiger und in der Praxis kaum verbreitet.

Es folgt eine lose (nichtsystematische) Zusammenstellung diverser Oberon-Programme mit typischen Eigenschaften. Systematische Erläuterungen werden in den folgenden Kapiteln gegeben.

1.3 Erste Beispiele zur Sprache Oberon

Eingabe und Ausgabe

1.3.1 PROGRAMMBEISPIEL (AUSGABE EINES TEXTES)

```

1  MODULE Hallo;
2  (* Wirkung des Moduls: Ausgabe von Hallo *)
3  IMPORT Out;
4
5  PROCEDURE Start*;
6  BEGIN
7    Out.String(" Hallo "); Out.Ln
8  END Start;
9
10 BEGIN
11 END Hallo.
```

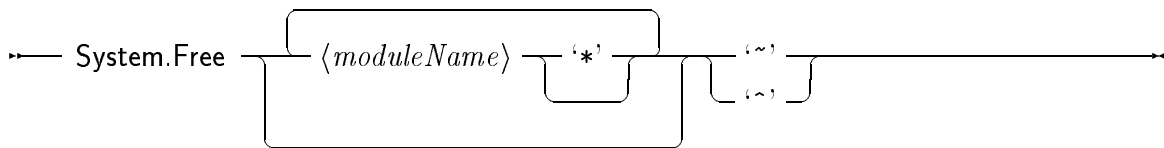
Oberon-System.Free-Kommando:

Abbildung 1.4: Oberon-System.Free-Kommando

```

12
13 ModulAufruf durch:
14   Hallo.Start ~
15 Ausgabe:
16   Hallo

```

1.3.2 ERLÄUTERUNGEN

- In `(* ... *)` steht ein Kommentar, d.h. ein beliebiger Text, der vom Oberon-Compiler (Übersetzer) nicht zur Kenntnis genommen wird.
- **Hallo** und **Start** sind frei gewählte Namen: **Hallo** ist der Name des Moduls, **Start** der Name einer Prozedur.
- **MODULE**, **BEGIN**, **END**, **IMPORT** und **PROCEDURE** sind sogenannte *reservierte Wörter* (*reserved words*): diese können nur in vordefiniertem Zusammenhang verwendet werden; sie stehen syntaktisch auf der gleichen Stufe wie z.B. „;“ oder „.“.
- Groß / Kleinschreibung: wird in der Sprache Oberon — wie im Oberon-System — unterschieden (anders als etwa in Pascal), z.B. gelten **BEGIN**, **Begin**, **BeGiN** als unterschiedlich, nur **BEGIN** ist ein reserviertes Wort. Alle reservierten Wörter werden mit Großbuchstaben geschrieben.
- **Out** ist der Name eines sog. Bibliotheksmoduls. **IMPORT** legt fest, daß **Out** im Modul **Hallo** verwendet werden kann.
- Die konkrete Verwendung von **Out** geschieht in diesem Beispiel durch **Out.String(...)**: Ausgabe einer Zeichenkette auf dem Bildschirm. **Out.String(...)** ist *nicht* Bestandteil der Sprache Oberon. Wie in allen modernen Sprache wird die Ein- und Ausgabe durch Bibliotheksmodule gemacht (so z.B. auch C)
- Die Kombination **Out.String** wird auch als *qualifizierter Name* bezeichnet: **String** wird durch **Out** qualifiziert, d.h. genau beschrieben.
- **Out.Ln** bedeutet: Zeilenvorschub (neue Zeile)
- Alle Aktionen des Programms (hier: Ausgabe der Zeichenkette) stehen in den folgenden Beispielen zunächst innerhalb der Prozedur **Start**.
- Der ***** hinter **Start** bedeutet: das Programm kann gestartet werden durch den „Aufruf“ **Hallo.Start** (d.h. konkret durch Anklicken mit MM). Es wird dann in diesem Beispiel der Code ausgeführt, der innerhalb der Prozedur **Start** steht.

- Der Text `Hallo.Start` darf an beliebiger Stelle auf dem Bildschirm stehen.
- `Hallo.Start` ist ein Beispiel für ein benutzerdefiniertes Kommando.

1.3.3 PROGRAMMBEISPIEL (EINGABE – AUSGABE: MULTIPLIKATION ZWEIER GANZER ZAHLEN)

```

1  MODULE Multiplikation;
2  (* Das Modul liest zwei ganze Zahlen ein und gibt das Produkt dieser Zahlen aus. *)
3  IMPORT In, Out;
4  CONST f = 5;
5
6  PROCEDURE Start*;
7    VAR x, y: INTEGER;
8  BEGIN
9    In.Open; In.Int(x); In.Int(y);
10   Out.Int(x*y, f); Out.Ln
11 END Start;
12
13 BEGIN
14 END Multiplikation.
15
16 ModulAufruf durch:
17   Multiplikation.Start 8 5 ~
18 Ausgabe:
19   40

```

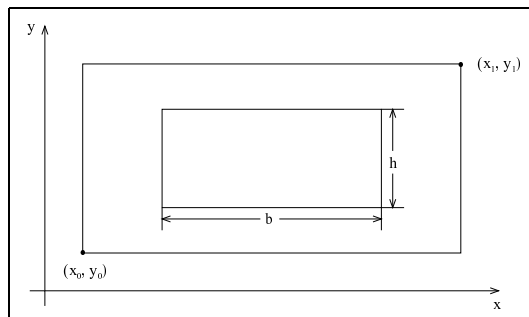


Abbildung 1.5: Parametrisierung einer geometrischen Darstellung

1.3.4 ERLÄUTERUNGEN

- `CONST` und `VAR` sind weitere reservierte Wörter.
- `CONST`: überall im Programm wird *zur Compilezeit* anstelle von `f` der Wert 5 gesetzt; `CONST` ist insbesondere dann sinnvoll, wenn eine Konstante mehrmals auftritt oder mehrere voneinander abhängige Konstanten auftreten. Bsp.: ein beliebig auf dem Bildschirm plaziertes Rechteck mit einem dazu symmetrisch liegenden Rechtecke (s. Abbildung 1.5):

Anzahl der freien Parameter ist 6, festgelegt durch

`CONST x0 = ...; y0 = ...; ...` Anzahl der auftretenden Zahlenwerte ist $8 \times 2 = 16$.

- **VAR:** wird verwendet zur Deklaration von Variablen, hier der Variablen **x** und **y**: es wird Speicherplatz bereitgestellt für zwei Variablen, die ganzzahlige Werte aufnehmen können. Mit der Deklaration sind die konkreten Werte **nicht** festgelegt, es gibt also keine *default*-Initialisierung.
- Jede Variable, genauer: jeder Name, muß vor der ersten Verwendung deklariert werden.
- **INTEGER:** Kennzeichnung des Typs (genauer: Datentyps) *ganze Zahl*; ist *vordefinierter Name (predefined identifier)*, ist *kein* reserviertes Wort
- **VAR x: INTEGER;** entspricht der mathematischen Schreibweise $x \in \mathbb{Z}$, wenn $\mathbb{Z} = \{0, \pm 1, \pm 2, \dots\}$
- **In.Open:** Eingabe eröffnen
- **In.Int(x):** Eingabe des ganzzahligen Wertes **x**
- **Out.Int(x,f):** Ausgabe des ganzzahligen Wertes **x** im Format **f**
- *Ausgabeformat (f = 5):*

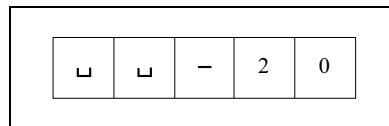


Abbildung 1.6: Ausgabeformat

(s. Abbildung 1.6)

Bemerkung: freies Layout in Oberon-Programmen zulässig, also auch z.B.

1.3.5 PROGRAMMBEISPIEL (LAYOUT EINES OBERON-PROGRAMMS)

```

1  MODULE QuellCode; IMPORT
2    Out;  PROCEDURE
3      Show
4  *;  BEGIN  Out .
5    Real  (
6      3.14,
7    15 ); Out.Ln; END
8    Show
9    ;
10
11  BEGIN END
12      QuellCode
13      .
14  ModulAufruf durch:
15    QuellCode.Show ~
16  Ausgabe:
17    3.1400001E+00
```

1.3.6 ERLÄUTERUNGEN

- Durch geeignete Wahl der textuellen Gestaltung werden Programme *gut* lesbar gemacht. Die gute Lesbarkeit stellt einen auch wirtschaftlich bedeutenden Faktor der Softwareentwicklung dar. Programmtexte müssen also auch und gerade für Menschen gut lesbar sein!

Der obige Programmtext stellt ein Beispiel für ein syntaktisch und semantisch richtiges, aber vollkommen unlesbares Programm dar.

Ausdrücke und Zuweisungen

1.3.7 PROGRAMMBEISPIEL (LÖSUNG VON QUADRATISCHEN GLEICHUNGEN)

$$x^2 + 2ax + b = 0$$

Lösungen:

$$x_{1,2} = -a \pm \sqrt{a^2 - b} \quad \text{falls } a^2 - b > 0$$

```

1  MODULE QuadratGl;
2  (*
3   Die Prozedur 'Loesen' im Modul 'QuadratGl' liest die Koeffizienten
4   a und b der quadratischen Gleichung  $x^2 + 2ax + b = 0$  ein
5   und bestimmt die Lösungen x1 und x2 für den Fall, daß die
6   Diskriminante positiv ist (wenn also genau zwei verschiedene
7   reelle Lösungen existieren).
8   Das Modul ist insofern unfertig, als die Fälle mit genau einer
9   oder keiner reellen Lösung nicht behandelt werden.
10 *)
11  IMPORT In, Out, Math;
12  CONST f1 = 12; f2 = 18;
13
14  PROCEDURE Loesen*;
15    VAR a, b, D, W, x1, x2: REAL;
16  BEGIN
17    In.Open; In.Real(a); In.Real(b);
18    D := a*a - b;
19    IF D >= 0 THEN
20      W := Math.sqrt(a*a - b);
21      x1 := -a + W; x2 := -a - W;
22
23      Out.String("Die Wurzeln der Gleichung mit den Koeffizienten ");
24      Out.Real(a, f1); Out.String(" und "); Out.Real(b, f1);
25      Out.String(" lauten:"); Out.Ln;
26      Out.Real(x1, f2); Out.Real(x2, f2); Out.Ln; Out.Ln;
27    ELSE
28      Out.String("keine zwei reellen Loesungen"); Out.Ln
29    END
30  END Loesen;
31
32  BEGIN

```

```

33  END QuadratGl.
34
35  (*ModulAufruf durch: *)
36  QuadratGl.Loesen 4.0 6.0 ~
37  (* Ausgabe: *)
38  Die Wurzeln der Gleichung mit den Koeffizienten 4.0000E+00 und 6.0000E+00 lauten:
39  -8.3772230E-01 -7.1622777E+00

```

1.3.8 ERLÄUTERUNGEN

- Durch `IMPORT Math` werden einige mathematische Funktionen importiert, u.a. die hier benötigte Quadratwurzel `sqrt()` (engl. *square root*).
- `REAL` ist ein Datentyp, der *reelle* Zahlen (näherungsweise) repräsentiert, z.B. 1.0, 1.5, $\sqrt{2} = 1.4142\dots$, $\pi = 3.1415\dots$
- `In.Real()`, `Out.Real()`: Eingabe- und Ausgabeprozeduren aus den Bibliotheksmodulen `In` und `Out` für `REAL`-Werte.
- `W := ...`; bzw. `x1 := -a + W`; bedeutet: in den durch `W` bzw. `x1` definierten Speicherplatz wird der Wert der rechten Seite geschrieben. `:=` ist **ein** Symbol (*special symbol*)
- Wenn `r` vom Typ `REAL` z.B. ist, so ist es auch sinnvoll zu schreiben:
`r := r + 3.0;`

Das ist **keine** mathematische Gleichung, sondern eine *Zuweisung* (*assignment statement*). Vergleich mit der mathematischen Gleichung $r = r + 3.0$!

1.3.9 PROGRAMMBEISPIEL (BERECHNUNG DER WERTE EINER FUNKTION)

Gesucht sind Funktionswerte der sog. Standard- (oder Gauß-) Normalverteilung

$$y = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

```

1  MODULE FunktionsWert;
2
3  (* Berechnung des Funktionswertes der (Gauß-) Standard-Normalverteilung
4     zum (einzulesenden) REAL-Wert x *)
5
6  IMPORT In, Out, m := Math;
7  CONST c = 15; pi = m.pi;
8  (* oder auch: m.pi direkt unten verwenden*)
9
10 PROCEDURE Berechnen*;
11   VAR x, y: REAL;
12 BEGIN
13   In.Open; In.Real(x);
14   y := (1/m.sqrt(2 * pi)) * m.exp(-x*x/2);
15   Out.Real(y, c); Out.Ln;
16 END Berechnen;

```

```

17
18 BEGIN
19 END FunktionsWert.
20
21 ModulAufruf durch:
22   FunktionsWert.Berechnen 2.5 ~
23 Ausgabe:
24   1.7528301E-02

```

1.3.10 ERLÄUTERUNGEN

- **m := Math:** anstelle des Modulnamens **Math** wird **m** verwendet und **Math** nicht mehr. Vorteile: nicht nur Abkürzung sondern auch leichtere Austauschbarkeit des Moduls **Math**: zum Austausch läßt man **m** überall im Modul stehen und verändert nur **m := Math**.
- Mathematische Ausdrücke werden wie in der Mathematik geschrieben; Operationen $+$, $-$, $*$, $/$. Besonderheit: $*$ muß geschrieben werden. **ab** würde als eigener Name, nicht als Produkt **a*b** interpretiert. Regel: $*$, $/$ vor $+$, $-$; Änderung von Vorrangregeln durch Setzen von Klammern (wie in der Mathematik)

1.4 Syntax der Sprache Oberon

Aufbau eines Moduls

Module:

$\rightarrow$ MODULE - $\langle \text{ident} \rangle$ - ';' - $\underbrace{\langle \text{ImportList} \rangle}_{\text{ImportList}}$ - $\underbrace{\langle \text{DeclSeq} \rangle}_{\text{DeclSeq}}$ - $\underbrace{\text{BEGIN} - \langle \text{StatSeq} \rangle}_{\text{StatSeq}}$ - $\underbrace{\text{END} - \langle \text{ident} \rangle - ';' }_{\text{ident}}$ $\rightarrow$

Abbildung 1.7: Aufbau von Modul

Aufbau von Modul: s. Abbildung 1.7

1.4.1 ERLÄUTERUNGEN

- Modul ist die größte syntaktische Einheit; eine Datei kann mehrere Module enthalten.
- **ident:** Name des Moduls, wird hinter **END** wiederholt. Aus der syntaktischen Beschreibung geht *nicht* hervor, daß die beiden Namen gleich sein müssen.
- **ImportList:** enthält alle Namen derjenigen Module, aus denen z.B. Prozeduren verwendet werden sollen (genauer: importiert werden grundsätzlich Namen).
- **DeclSeq** (DeclarationSequence, Folge von Deklarationen): enthält Beschreibung aller im Programm verwendeten Größen (analog zur Mathematik: z.B. $x \in \mathbb{R}$, $i \in \mathbb{N}$)
- **StatSeq** (StatementSequence, Folge von Anweisungen): enthält durchzuführende Aktionen, in Oberon an dieser Stelle nur solche mit einer sehr speziellen Bedeutung (Einzelheiten später)

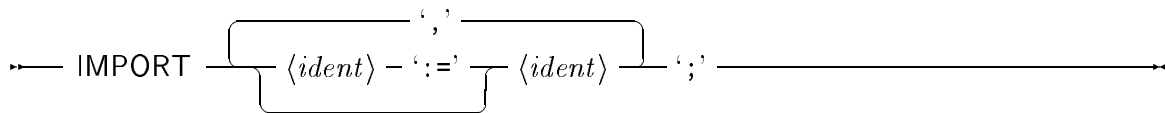
IMPORT-Liste*ImportList:*

Abbildung 1.8: IMPORT-Liste

s. Abbildung 1.8

1.4.2 ERLÄUTERUNGEN

- Semikolon ist Teil des syntaktischen Konstrukts, anders als bei Anweisungen

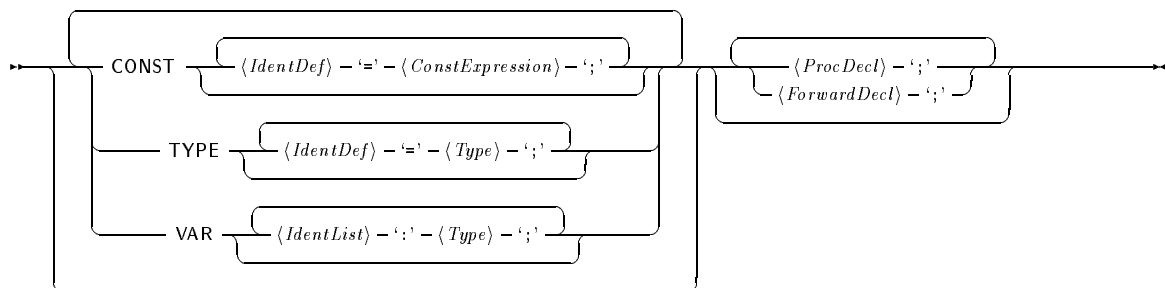
Deklarationen*DeclSeq:*

Abbildung 1.9: DeclarationSequence (vollständig)

s. Abbildung 1.9

1.4.3 ERLÄUTERUNGEN

- Diagramm ist in der Form bereits vollständig
- hinter **CONST** steht =, nicht :=; **keine** Zuweisung
- Als „Test“ für Präzision der obigen Beschreibung: Z.B. ist

```
MODULE Test;
  VAR

BEGIN END Test.
```

syntaktisch richtig, nicht dagegen obiger Programmtext mit **VAR** ;

ProcDecl:

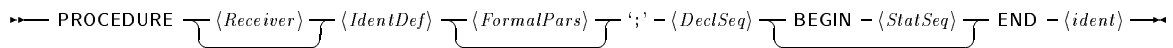


Abbildung 1.10: ProcedureDeclaration (vollständig)

Prozedur-Deklaration

s. Abbildung 1.10

1.4.4 ERLÄUTERUNGEN

- Diagramm ist vollständig.
- Fundamentale Eigenschaft aller Sprachen der Pascal-Familie (im Unterschied z.B. zu C):

die Kombination `DeclSeq BEGIN StatSeq END` tritt geschachtelt auf. Sie ist Bestandteil eines jeden Moduls und innerhalb eines Moduls Bestandteil einer jeden Prozedur, eventuell innerhalb einer Prozedur *P* wiederum Bestandteil einer Prozedur (die innerhalb von *P* deklariert ist) und so fort.

Schachtelungsstruktur von Oberon-Modulen: s. Abbildung 1.11

Anweisungsfolge

s. Abbildung 1.12

1.4.5 ERLÄUTERUNGEN

- Anders als z.B. in der Sprache C ist das Semikolon nicht Teil einer Anweisung sondern Trennsymbol zwischen Anweisungen
- Anweisungen legen die Aktionen fest, die ausgeführt werden sollen. Im Mittelpunkt jeder Programmierereinführung müssen daher zunächst die Anweisungen stehen.

Beispiel für eine Anweisung: die Zuweisung

s. Abbildung 1.13

1.5 Weitere Beispiele zur Sprache Oberon

Strukturierte Anweisungen

WHILE-Anweisung: Durchschnittsberechnung

Durchschnitt: $d = (a_1 + a_2 + \dots + a_n) / n = \frac{1}{n} \sum_{i=1}^n a_i$, wobei $a_i \in \mathbb{R}$ für $1 \leq i \leq n$, $n \in \mathbb{N}$.
Berechnungsschema:

0. setze: $s := 0$;
1.
 - Lies x ein ($x = a_1$)
 - $s := s + x$; ($s = a_1$)

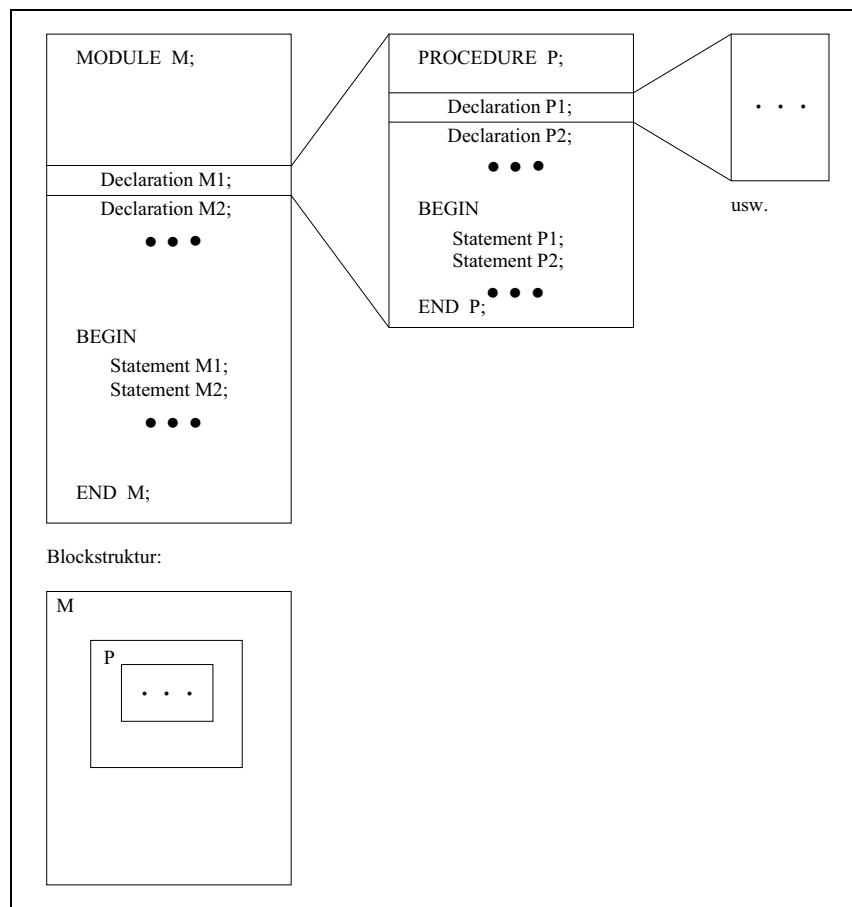


Abbildung 1.11: Geschachtelte Struktur von Modulen

2. • Lies x ein ($x = a_2$)
 • $s := s + x$; ($s = a_1 + a_2$)
3. • Lies x ein ($x = a_3$)
 • $s := s + x$; ($s = a_1 + a_2 + a_3$)

usw.

- n . • Lies x ein ($x = a_n$)
 • $s := s + x$; ($s = a_1 + a_2 + \dots + a_n$)

$s := s/n$; (Durchschnitt)

1.5.1 PROGRAMMBEISPIEL (DURCHSCHNITTSBERECHNUNG)

```

1  MODULE DurchschnittsBer;
2  (* Modul arbeitet nur dann einwandfrei, wenn mindestens ein Wert eingelesen wird. *)
3  IMPORT In, Out;
4
5  PROCEDURE Exec*;
6  VAR s, x: REAL; n: INTEGER;
```

StatSeq:

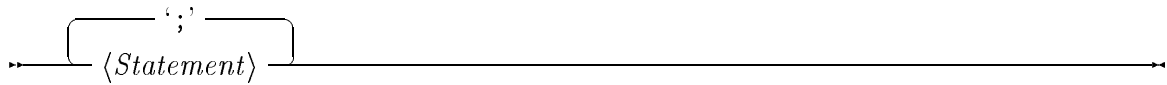


Abbildung 1.12: StatementSequence

AssignmentStatement:

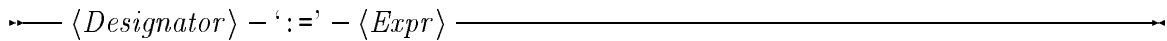


Abbildung 1.13: AssignmentStatement

```

7   BEGIN
8     In.Open; s := 0; n := 0; In.Real(x);
9     WHILE In.Done DO
10      s := s + x; In.Real(x); INC(n)
11    END;
12    Out.Real(s/n, 15); Out.Ln
13  END Exec;
14
15  END DurchschnittsBer.
16
17  Beispielauf:
18  DurchschnittsBer.Exec 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0 ~
19  Ergebnis:
20  5.5000000E+00

```

1.5.2 ERLÄUTERUNGEN

- allgemeine Struktur der WHILE-Anweisung:

WHILE *Bedingung* DO *Anw1*; *Anw2*; ... END

- Bedeutung: Führe die Anweisungen *Anw1*; *Anw2*; ... zwischen DO und END solange aus, **wie** *Bedingung* erfüllt ist: z.B.:

```

i := 5;
WHILE i > 0 DO i := i - 1 END

```

wird genau 5 mal durchlaufen.

- In.Done: ist Antwort (Ja / Nein) auf die Frage: War die letzte Leseoperation erfolgreich? [RW92, S. 88]
- /: normale Division (Resultat vom Typ REAL)
- INC(n): Wirkung ist $n := n + 1$ (increment)
- analog: DEC(n): Wirkung ist $n := n - 1$ (decrement)
- diskutieren: warum eigene Symbole?

REPEAT-Anweisung

REPEAT-Anweisung: ist analog zur WHILE-Anweisung

Struktur:

REPEAT *Anw1*; *Anw2*; ... UNTIL *Bedingung*

Bedeutung: Führe *Anw1*, *Anw2*, ... solange aus, **bis** *Bedingung* erfüllt ist.

Beispiel:

```
i := 0;
REPEAT
  i := i + 1
UNTIL i > 5;
```

wird genau 6 mal durchlaufen

Grundlegender Unterschied zwischen WHILE-Anweisung und REPEAT-Anweisung:

REPEAT : Bedingung wird **nach** jedem Durchlauf geprüft; daher: Durchlaufen stets mindestens 1 mal (*postchecked*)

WHILE : Bedingung wird **vor** jedem Durchlauf geprüft; daher: Durchlaufen eventuell 0 mal (*prechecked*)

Beispiel (als Übungsaufgabe):

Berechnung der Fakultätsfunktion:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

z.B.: $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

FOR-Anweisung: Tabellierung von Funktionswerten

Das folgende Modul erstellt eine Funktionstabelle der Funktion cosh, definiert durch

$$\cosh x = \frac{1}{2}(e^x + e^{-x}),$$

für die Werte cosh x mit

$$x = x_0 + \frac{x_1 - x_0}{N} i, \text{ wobei } i = 0, 1, 2, \dots, N$$

1.5.3 PROGRAMMBEISPIEL (TABELLIERUNG VON FUNKTIONSWERTEN)

```
1  MODULE FktTabell;
2  IMPORT In, Out, Math;
3  CONST f = 15;
4
5  PROCEDURE Erstellen*;
6    VAR i, N: INTEGER; x, y, x0, x1, a: REAL;
7  BEGIN
8    In.Open; In.Real(x0); In.Real(x1); In.Int(N);
9    a := (x1-x0)/N;
10   FOR i := 0 TO N BY 1 DO      (* BY 1 kann entfallen *)
```

```

11      x := x0 + a*i;
12      y := 0.5*(Math.exp(x) + Math.exp(-x));
13      Out.Real(x, f); Out.String("": "");Out.Real(y,f);Out.Ln;
14      END;
15      Out.Ln;
16      END Erstellen;
17
18      BEGIN
19      END FktTabell.
20
21      Beispielaufruf durch:
22      FktTabell.Erstellen 0.0 10.0 5 ~
23      Ausgabe:
24      0.0000000E+00:      1.0000000E+00
25      2.0000000E+00:      3.7621956E+00
26      4.0000000E+00:      2.7308233E+01
27      6.0000000E+00:      2.0171564E+02
28      8.0000000E+00:      1.4904791E+03
29      1.0000000E+01:      1.1013233E+04

```

1.5.4 ERLÄUTERUNGEN

- Struktur der FOR-Anweisung (zunächst nur beispielhaft):
`FOR i := 0 TO N BY k DO Anw1; Anw2; ... END`
- Wirkung(vgl. [RW92, S. 273]): Führe *Anw1*, *Anw2*, ... wiederholt aus, zunächst mit $i = 0$, dann mit $i = k$, dann mit $i = 2*k$, usw., solange wie $i \leq N$ erfüllt ist.
- genaue Beschreibung der Wirkung: ist **nicht** einfach, entgegen dem ersten Anschein (vgl. Übungsaufgabe)
- Vorteil der FOR-Schleife: sie terminiert garantiert
- In einer FOR-Schleife (wie in jeder anderen Schleife auch) sollten keine Berechnungen durchgeführt werden, die vom Schleifenindex nicht abhängen.

IF-Anweisung: die Signum-Funktion

Signum-Funktion in der Mathematik:

$$\operatorname{sgn} x = \begin{cases} -1 & \text{falls } x < 0 \\ 0 & \text{falls } x = 0 \\ +1 & \text{falls } x > 0 \end{cases}$$

im folgenden Modul: anstelle des jeweiligen Zahlenwertes -1, 0 bzw. +1: erläuternder Text

1.5.5 PROGRAMMBEISPIEL (SIGNUM-FUNKTION)

```

1  MODULE SignumFkt;
2  (* Schema zur Bestimmung des Vorzeichens einer reellen Zahl *)
3  IMPORT In, Out;

```

```

4  CONST c = 15;
5
6  PROCEDURE Test*;
7    VAR x: REAL;
8  BEGIN
9    In.Open; In.Real(x);
10   IF x > 0 THEN Out.Real(x, c); Out.String(" ist groesser als 0.")
11   ELSIF x < 0 THEN Out.Real(x, c); Out.String(" ist kleiner als 0.")
12   ELSE Out.Real(x, c); Out.String(" ist gleich 0.")
13   END;
14   Out.Ln
15 END Test;
16
17 BEGIN
18 END SignumFkt.
19
20 (* Beispielaufruf durch: *)
21   SignumFkt.Test 1.234 ~
22 (* Ausgabe: *)
23   1.2340000E+00 ist groesser als 0.

```

1.5.6 ERLÄUTERUNGEN

- allgemeine Struktur der IF-Anweisung:

```

IF      Bedingung1 THEN Anw1; Anw2; ...
ELSIF   Bedingung2 THEN Anw3; Anw4; ...
ELSIF   ...
ELSE
                                Anw5; Anw6; ...
END

```

diskutieren: Semikolon vor ELSE?

- Bedeutung: Wenn *Bedingung1* erfüllt ist, dann führe *Anw1*, *Anw2*, ... aus, wenn *Bedingung1* nicht erfüllt ist, *Bedingung2* dagegen erfüllt ist, dann führe *Anw3*, *Anw4*, usw., andernfalls *Anw5*, *Anw6*, ...

Strukturierte Datentypen: Beispiel ARRAY

Problemstellung

Wie kann man sehr viele Variablen eines bestimmten Typs einführen, z.B.: 10 000 Variable vom Typ REAL; offenbar **nicht** durch

```
VAR x1, x2, x3, x4, ..., x10000: REAL;
```

„...“ müßte ausgeschrieben werden! (Deklaration als sog. Array)

Anwendung in Mathematik: **Vektor**

Einführung entweder geometrisch oder abstrakt als sog. Tripel (oder 3-Tupel) von reellen Werten: (u_1, u_2, u_3) , allgemeiner $(u_1, u_2, \dots, u_n)$ mit $n \in \mathbb{N}$.

fundamentale Operationen mit Vektoren: Multiplikation eines Vektors mit reeller Zahl c :

$$c\mathbf{u} = (cu_1, cu_2, \dots, cu_n)$$

Addition von Vektoren („komponentenweise“)

$$\mathbf{u} + \mathbf{v} = (u_1 + v_1, u_2 + v_2, \dots, u_n + v_n)$$

Die Verallgemeinerung auf beliebiges n hat eine Bedeutung z.B. im Zusammenhang mit der Anwendung auf lineare Gleichungssysteme.

Beispiel: Skalarprodukt von Vektoren:

Formel für Skalarprodukt von Vektoren:

$$\mathbf{u} \cdot \mathbf{v} = u_1v_1 + u_2v_2 + \dots + u_nv_n = \sum_{i=1}^n u_iv_i$$

1.5.7 PROGRAMMBEISPIEL (SKALARPRODUKT VON VEKTOREN)

```

1  MODULE Skalarprodukt;
2  IMPORT Out;
3  CONST n = 3; f = 15;
4  VAR
5    i: INTEGER; p: REAL;
6    u, v: ARRAY n OF REAL;
7
8  PROCEDURE Exec*;
9  BEGIN i := 0;
10  (* Die Arrays u und v werden mit (willkürlichen) Beispielwerten belegt.
11     In wirklichen Anwendungen könnte man die Werte z.B. durch In.Real(u[i]) einlesen. *)
12    WHILE i < n DO u[i] := i; v[i] := i*i; INC(i) END;
13    p := 0; i := n;
14    (* Es folgt die Berechnung des Skalarprodukts *)
15    WHILE i > 0 DO DEC(i); p := u[i] * v[i] + p END;
16    Out.Real(p, f); Out.Ln
17  END Exec;
18
19  BEGIN
20  END Skalarprodukt.
21
22  (* Beispielaufruf durch:*)
23  Skalarprodukt.Exec ~
24  (* Ausgabe: *)
25  9.0000000E+00
```

1.5.8 ERLÄUTERUNGEN

- VAR u: ARRAY n OF REAL;

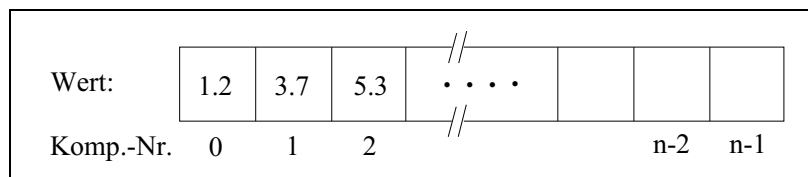


Abbildung 1.14: Aufbau eines Arrays

- **ARRAY**: Datenstruktur zur Zusammenfassung einer festen Anzahl (hier: n) von Größen (Komponenten) *gleichen* Typs (einer *Aneinanderreihung*), hier des Typs **REAL**:

Abbildung 1.14

- Die einzelnen Komponenten des **ARRAYs** u werden angesprochen als $u[0]$, $u[1]$, ..., $u[n-1]$.
- $u[i]$ für $0 \leq i \leq n-1$ wird ansonsten genau wie eine Variable vom Typ **REAL** behandelt.
- Ein weiterer fundamental wichtiger strukturierter Datentyp in Oberon ist der Typ **RECORD**.

Prozeduren

Selbstdefinierte Prozeduren: Erweiterung des Moduls Math

zwei Beispiele: Tangens und allgemeine Potenzfunktion a^x mit $a, x \in \mathbb{R}$:
mathematische Definitionen:

$$\tan x = \frac{\sin x}{\cos x}$$

$$a^x = e^{x \ln a} \quad \text{mit } 0 < a \in \mathbb{R}$$

1.5.9 PROGRAMMBEISPIEL (ERWEITERUNG DES MODULS Math)

```

1  MODULE MathX1;
2  (* eigene Erweiterung des Moduls Math *)
3  IMPORT Out, m := Math;
4  CONST f = 18;
5
6  PROCEDURE tan(x: REAL): REAL;
7  BEGIN
8    RETURN m.sin(x)/m.cos(x)
9  END tan;
10
11 PROCEDURE pow(a, x: REAL): REAL;
12   (*  $a^x$ , definiert für  $a > 0$  *)
13 BEGIN
14   RETURN m.exp(x*m.ln(a))
15 END pow;
16
```

```

17  PROCEDURE Show*;
18  BEGIN
19    Out.Real(tan(0.23), f); Out.String("    ");
20    Out.Real(pow(1.2, 2.0), f); Out.Ln;
21  END Show;
22
23  BEGIN
24  END MathX1.
25
26  Beispielaufruf durch:
27    MathX1.Show ~
28  Ausgabe:
29    2.3414336E-01      1.4400001E+00

```

1.5.10 ERLÄUTERUNGEN

- Außer **Show** werden zwei weitere Prozeduren deklariert: **tan** und **pow**
- Die Prozeduren sind sog. Funktionsprozeduren, d.h. solche mit Rückgabewert; beide Prozeduren werden in der Prozedur **Show** aufgerufen (mit Beispielwerten zur Demonstration)
- typischer (beispielhafter) Aufbau einer Funktionsprozedur:

```

PROCEDURE f(x: REAL): REAL;
  VAR i, k: INTEGER;      (* sog. lokale Variablen *)
  ...
BEGIN
  ...
  RETURN ... (* RETURN, gefolgt von Ausdruck *)
  ...
END f

```

- **x** ist der *zu übergebende* Parameter, hier: vom Typ **REAL**, Funktionsprozedur selbst: hier ebenfalls **REAL**-wertig
- Aktiviert wird die Prozedur **f** durch einen *Prozeduraufruf* **f(x)**.
- Anstelle des Prozeduraufrufs hätte der Ausdruck hinter **RETURN** auch nach **Show** kopiert werden können.

Vorteil der Deklaration in einer Prozedur: diese kann immer wieder aufgerufen werden, ohne daß der entsprechende Prozedurcode wiederholt werden müßte.

Fakultätsfunktion

1.5.11 PROGRAMMBEISPIEL (BERECHNUNG DER FAKULTÄT)

```

1  MODULE FakultaetsBer; (* Variante: nichtrekursive Berechnung *)
2  IMPORT In, Out;
3  CONST c = 5;
4

```

```

5  PROCEDURE f(n: INTEGER): LONGINT;
6  (* Voraussetzung: n >= 0 *)
7    VAR k: LONGINT;
8  BEGIN
9    IF (n=0) OR (n=1) THEN RETURN 1
10   ELSE (* also n >= 2 *)
11     k := n;
12     WHILE n >= 3 DO DEC(n); k := n*k END;
13     RETURN k
14   END;
15 END f;
16
17 PROCEDURE Start*;
18   VAR n: INTEGER;
19 BEGIN In.Open; In.Int(n);
20   Out.Int(n, c); Out.String("! ist "); Out.Int(f(n), c); Out.Ln;
21 END Start;
22
23 BEGIN
24 END FakultaetsBer.
25
26 Beispielaufruf durch:
27   FakultaetsBer.Start 5 ~
28 Ausgabe:
29   5! ist 120

```

1.5.12 ERLÄUTERUNGEN

- **LONGINT**: Datentyp wie **INTEGER**, aber mit größerem Wertebereich; hat aber nichts direkt zu tun mit **c** in **Out.Int(n, c)**!
- **OR**: Verknüpfung von logischen Ausdrücken:
A OR B:
entweder **A** oder **B** (oder evtl. beide) z.B. $(x < 0) \text{ OR } (x > 1)$
- **RETURN**: kann auch mehrmals auftreten; liefert nicht nur Rückgabewert, sondern beendet auch die Prozedur zur Laufzeit.

Fakultätsfunktion rekursiv

Andere mathematische Definition von $n!$:

$$f(n) = n! = \begin{cases} 1 & \text{für } n = 0 \\ n f(n-1) = n(n-1)! & \text{für } n \geq 1 \end{cases}$$

Diese Funktion ist ein beliebtes Lehrbuchbeispiel für rekursive Funktionen. In einem wirklich genutzten Programm würde man die Fakultätsfunktion allerdings kaum so programmieren. Sie dient hauptsächlich deswegen als Beispiel, weil die Rekursivität hier besonders durchsichtig ist.

1.5.13 PROGRAMMBEISPIEL (FAKULTÄT REKURSIV)

```

1  MODULE FakultaetsBer; (* rekursive Berechnung *)
2  IMPORT In, Out;
3  CONST c = 5;
4
5  PROCEDURE f(n: INTEGER): LONGINT;
6  BEGIN
7    IF n = 0 THEN RETURN 1
8    ELSE RETURN n*f(n-1) (* mathematisch:  $f(n) = n \cdot f(n-1)$  *)
9    END;
10 END f;
11
12 PROCEDURE Start*; (* wie im vorangehenden Beispiel *)
13   VAR n: INTEGER;
14   BEGIN In.Open; In.Int(n);
15     Out.Int(n, c); Out.String("! ist "); Out.Int(f(n), c); Out.Ln;
16   END Start;
17
18 BEGIN
19 END FakultaetsBer.
20
21 Beispielaufruf durch:
22   FakultaetsBer.Start 12 ~
23 Ausgabe:
24   12! ist 479001600

```

1.5.14 ERLÄUTERUNGEN

- rekursive Prozedurdeklaration: die Deklaration von **f** enthält **f** selbst analog zur mathematischen Definition $f(n) = n \cdot f(n-1)$
- Es muß dafür gesorgt werden, daß die Rekursion terminiert; eine nicht kontrolliert terminierende Rekursion (z.B. durch Speicherüberlauf) ist stets ein Fehler.
- Rekursive Algorithmen: fundamental wichtig für die gesamte Informatik (Beispiele folgen)

Module

- allgemeine Erläuterung: *Modularisierung* bedeutet: Aufteilung eines größeren und komplexen Systems in kleinere überschaubare Bestandteile, die abgeschlossene, sinnvolle Einheiten bilden und für sich unabhängig voneinander getestet werden können;
- *Modularisierung* ist allgemeines Prinzip heutiger Technik; Bsp.: Rechnerhardware, elektronische Geräte (Fernseher, ...)
- Problem mit Modul **MathX** in bisheriger Form: kann (in der Form noch) nicht in eigenen neuen Modulen verwendet werden
bisher: nur innerhalb von **MathX** können die Prozeduren **tan** und **pow** aufgerufen und verwendet werden

jetzt: Verwendung von **MathX** (in modifizierter Form) in eigenem Modul, z.B. einem Modul **TestMath**: alle Prozedurnamen in **MathX**, die von **TestMath** aus verwendbar sein sollen, werden mit ***** versehen (sog. *Exportmarke*)

in **TestMath** wird (u.a.) **IMPORT MathX** eingeführt; Aktivierung der importierten Prozeduren wie bisher

1.5.15 PROGRAMMBEISPIEL (ERWEITERUNG DES MODULS **Math**)

jetzt von **2** Modulen:

```

1  MODULE MathX2;
2  (* Erweiterung des Moduls Math zum Importieren in eigene Module *)
3
4  IMPORT m := Math;
5
6  PROCEDURE tan*(x: REAL): REAL;
7  BEGIN
8    RETURN m.sin(x)/m.cos(x)
9  END tan;
10
11 PROCEDURE pow*(a, x: REAL): REAL;
12 (* ax *)
13 BEGIN
14   RETURN m.exp(x*m.ln(a))
15 END pow;
16
17 (* ... und weitere eigene Prozeduren ... *)
18
19 END MathX2.

1  MODULE Test;
2  (* Testmodul zum Testen der Schnittstelle von MathX2 *)
3  IMPORT Out, MathX2;
4  CONST f = 15;
5
6  PROCEDURE Start*;
7  VAR a, x: REAL;
8  BEGIN
9    x := 3.1415/2; Out.Real(MathX2.tan(x), f); Out.String(" ");
10   a := 3; x := 3; Out.Real(MathX2.pow(a, x), f); Out.Ln
11 END Start;
12
13 BEGIN
14 END Test.

15
16 (* Beispielaufruf durch: *)
17 Test.Start ~
18 (* Ausgabe: *)
19 2.1584891E+04      2.7000002E+01
```

1.5.16 ERLÄUTERUNGEN

- Bedeutung von *: Alle Namen, die mit * versehen sind, sind von anderen (externen) Modulen aus sichtbar und in dem Fall dann von allen.
- Diagramm zum Import: s. Abbildung 1.15

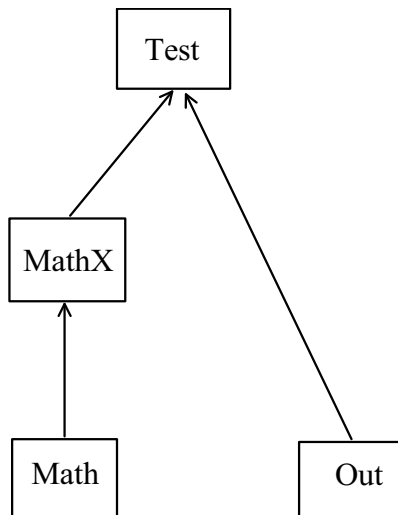


Abbildung 1.15: Import-Beziehungen

- Durch das Kommando `Watson.ShowDef ^ MathX` (im Oberon System 3) erhält man die sog. *Modulschnittstelle* von `MathX`:

```
DEFINITION MathX;
```

```
    PROCEDURE pow (a, x: REAL): REAL;
    PROCEDURE tan (x: REAL): REAL;
```

```
END MathX.
```

- Wenn eine Änderung der Schnittstelle zulässig sein soll, so muß der Compiler aufgerufen werden mit

```
Compiler.Compile \s:
```

Bedeutung: Das sog. *Symbol-File*, welches Informationen über die Schnittstelle enthält, darf geändert werden.

- Oberon unterstützt ebenso wie Modula-2 die sog. *separate Compilation* (im Unterschied zur *unabhängigen Compilation*): bereits der Compiler überprüft die Schnittstellen zu den importierten Modulen.

Kapitel 2

Grundbegriffe der Informatik. Algorithmen

2.1 Komponenten eines Rechners

Darstellung der Komponenten eines Rechners

(Gumm/Sommer: *Einführung in die Informatik*, Addison-Wesley, 1994, S. 40)

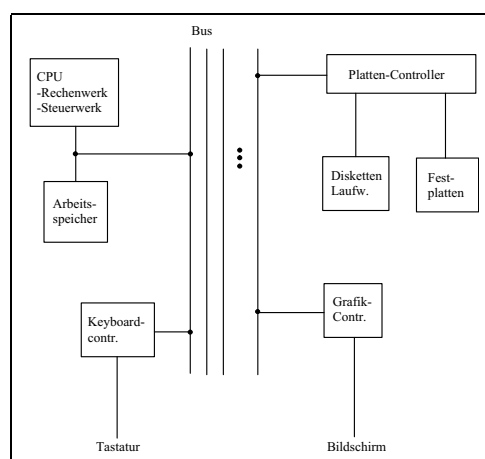


Abbildung 2.1: Komponenten eines Rechners

s. Abbildung 2.1

Erläuterungen

Interner Speicher

(= Arbeitsspeicher)

Zentralprozessor (CPU) kann nur auf diesen direkt zugreifen;

Zugriffszeit = mittlere Zeit, mit der die Zentraleinheit auf Daten zugreifen kann; Größenordnung ab einige Nanosekunden aufwärts: 1 Nanosekunde ($1 \text{ ns} = 10^{-9} \text{ s}$)

Kapazität: Anzahl Bytes, die gespeichert werden können (Größenordnung: MB)

Einheiten:

1 KByte = 2^{10} = 1 024 Byte
 1 Byte = 8 Bit (Bit = **b**inary **d**igit)
 1 MByte = 1 024 KByte
 1 GByte = 1 024 MByte
 also: $K = 2^{10} = 1024$ (abweichend von Physik)
 Unterscheidung von:

- **RAM: Random Access Memory:** (Schreib- und Lesespeicher mit wahlfreiem Zugriff); Speicherinhalt kann beliebig überschrieben werden
- **ROM: Read Only Memory** (Nur-Lese-Speicher, geschützt vor Überschreiben); z.B. zur Aufnahme eines Teils des Betriebssystems (etwa: Atari-TOS)

Externer Speicher

zum quasi-permanenten Aufbewahren größerer Datenmengen;

- Vorteil: billiger für langfristiges Speichern
- Nachteil: Zugriffszeiten im ms-Bereich ($1 \text{ ms} = 10^{-3} \text{ s}$); Rechenwerk kann hierauf nicht direkt zugreifen (wäre auch sinnlos wegen der großen Zugriffszeit)

Ausführungen:

- Magnetband (heute z.B. als Kassette mit Robotersystem)
- Diskette, Magnetplatte, Magnetwechselplatte
- CD-Speicher, CD-ROM

Prinzipieller Aufbau einer Diskette:

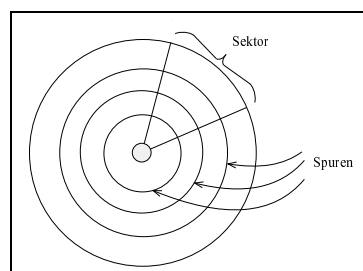


Abbildung 2.2: Diskettenaufbau: Prinzip

s. Abbildung 2.2

Anzahl Spuren und Sektoren sehr unterschiedlich; z. B.:

bei 3,5 Zoll-Disketten: 9 Sektoren, 80 Spuren;

entsprechendes MS-DOS-Kommando:

`format A: /N:9 /T:80`

Vergleich von Speichermedien

Medium	Kapazität Untergrenze	Kapazität Obergrenze	Mittlere Zugriffszeit	Erreichbare Datentransferrate in MByte s ⁻¹
Register			5 ns	3000
Cache	10 k	256 k	15 ns	1500
Arbeitsspeicher	1 M	1 G	50 ns	500
Flashkarten				
Festplatten	100 M	5 G	5 ms	10
Optische Platten	100 M	5 G	150 ms	1
Disketten	320 k	3 M	100 ms	0.2

Bit und Byte

Bit: ist Speichergrundeinheit mit zwei verschiedenen Zuständen (üblicherweise 0, 1 genannt)

Byte: 8-Bit-Reihe mit $2^8 = 256$ verschiedenen Zuständen:

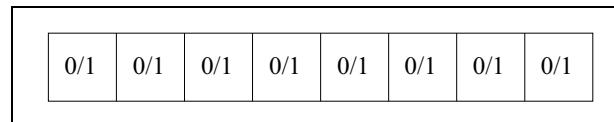


Abbildung 2.3: Belegung eines Bytes

s. Abbildung 2.3

Dadurch können dargestellt werden z.B.

- positive Zahlen $0, +1, +2, \dots, +127 (= 2^7 - 1)$ **und** negative Zahlen $-1, -2, \dots, -128 (= -2^7)$
- **oder:** nur nicht-negative Zahlen $0, +1, +2, \dots, +255 (= 2^8 - 1)$

weitere Einzelheiten: später

Rechnerkern (Central Processing Unit (CPU))

zur eigentlichen Befehlsausführung

im wesentlichen zwei Teile:

- **Steuerwerk:** nimmt Befehle aus dem Arbeitsspeicher und steuert den Arbeitsablauf
- **Rechenwerk:** führt arithmetische Operationen konkret aus (z.B. Additionen, Multiplikationen)

Eingabeeinheiten

Funktion: macht Daten rechnerlesbar; wichtigste Beispiele: Tastatur und Maus

weitere:

- Farb- und Schwarz/Weiß-Scanner
- Mikrophon
- Datenhandschuh
- Stift (Pen)

Ausgabeeinheiten

wichtigste Beispiele:

- Bildschirm (z.B. mit 24 Zeilen zu je 80 Zeichen)
- Drucker (z.B. Matrixdrucker, Tintenstrahldrucker, Laserdrucker)
- Plotter (zur Ausgabe von Zeichnungen)
- Lautsprecher
- Videorekorder
- Modem

2.2 Programmiersprachen

einige allgemeine Bemerkungen zu Programmiersprachen

Maschinen- und Assemblersprachen

Maschinensprachen

oder auch: Maschinencode

alle Daten und Befehle bestehen nur aus Binärzeichen: z.B.:

```
0011 1001
1111 0011
0000 1010
. . . . .
```

- Vorteil: kein Übersetzen
- extrem zeitaufwendiges Programmieren (nachträgliche Änderungen, Fehlersuche sehr aufwendig)
- schwer erlern- und lesbar

wird in der Form heute *nicht mehr* benutzt

Assemblersprachen

in Symbolen dargestellte Sprachelemente;
Beispiel:

```
ADD R1, R2
```

„Addiere den Inhalt des Registers R2 zum Inhalt von R1.“

- Vorteile:
 - Maschinennah, schnelles Übersetzen in Maschinencode
 - Code ist (eventuell (!)) effizient
- Nachteil: ähnlich wie bei Maschinensprachen, insbesondere ist Sprache vom Prozessor abhängig (also: Programm von DOS-Rechner läuft nicht auf Atari)

Benutzung auch noch heute teilweise

Problemorientierte Programmiersprachen

meistens auch als „Programmiersprachen“ schlechthin angesehen

- Vorteile:
 - kurze Programme
 - leicht erlernbar
 - Programme sind leicht auf andere Rechner übertragbar (gute *Portabilität*)
- Nachteile:
 - i. a. nicht ganz so effizient wie Assembler
 - (möglicherweise sehr) entfernt von Hardware

Verbreitete (sog. „klassische“) Programmiersprachen

- FORTRAN: **form**ula **tran**slator: Entwicklung Ende der 50er Jahre für naturwissenschaftliche und ingenieurmäßige Anwendungen; Weiterentwicklung bis heute; nach wie vor bedeutungsvoll und zwar auch (und gerade) auf modernen Höchstleistungsrechnern (sog. Vektorrechnern)
- COBOL: **com**mon **busi**ness **ori**ented **lan**guage: ebenfalls Ende der 50er Jahre entwickelt: Anwendungen hauptsächlich im kaufmännischen Bereich
- BASIC: **be**ginners **all** purpose **sym**bolic **in**struction **co**de: 1963 - 1965; für Programmieranfänger gedacht; aus methodischen Gründen jedoch ungeeignet
- C: verbunden mit Betriebssystem UNIX; heute im betrieblichen Bereich eine der wichtigsten Sprachen überhaupt; didaktisch jedoch nicht besonders geeignet (viele schwierige Details, die für Anfänger nicht erkennbar sind)

- Pascal: benannt nach B. Pascal (1623 - 1662); entwickelt von N. Wirth in ETH Zürich, seit 1971
inzwischen genormt durch ISO: ISO-Pascal (endgültig: 1983); in Deutschland durch DIN (Deutsches Institut für Normung)

Zahlreiche neue Sprachen (heute vielleicht 1000, weit verbreitet davon: 20):

- Modula-2
- Ada
- Smalltalk
- C++: Weiterentwicklung von C mit Möglichkeiten der sog. *Objektorientierung* (für Informatiker heute praktisch zwingend)
- Eiffel: Sprache von B. Meyer; sog. objektorientierte Sprache, die die Schwächen von C++ vermeidet
- Sprachen, die in der Industrie eine Rolle spielen:

• FORTRAN • COBOL		• C • C++ • Smalltalk • Eiffel
--	--	---

Übersetzen problemorientierter Sprachen

Übersetzen bedeutet: Umwandeln des geschriebenen Programms („Quellcodes“) in eine für den Rechner verständliche Form

2 Möglichkeiten: durch

- **Compiler:** Arbeitsweise: vollständig übersetzen, dann laden und ausführen
 - Vorteile:
 - * schnell in der Ausführung (weil nicht bei jeder Ausführung wieder übersetzt werden muß)
 - Nachteile:
 - * größerer Arbeitsspeicher erforderlich
 - * unter Umständen schwierige Fehlersuche
- oder durch **Interpreter:** Arbeitsweise: Jeder Befehl wird einzeln übersetzt und sofort ausgeführt
 - Vorteile:
 - * weniger Arbeitsspeicher-Kapazität erforderlich (weil nicht das ganze Programm in den Arbeitsspeicher passen muß)
 - * leichtere Fehlersuche
 - Nachteile:
 - * relativ langsam in der Ausführung

2.3 Betriebssysteme

engl. Operating Systems: Programme zur Steuerung und Überwachung des Rechenbetriebs; befinden sich teilweise ständig im Arbeitsspeicher, teilweise auf der Festplatte
heute auch wichtig: Betriebssysteme für Rechnernetze

Aufgabe von Betriebssystemen: Benutzer soll sich um zahlreiche Details nicht kümmern müssen, z. B.:

- An welche Stelle des Speichers wird sein Programm geladen?
- Details bei Ablage von Daten auf Hintergrundspeicher
- Schutz vor unbefugtem Zugriff auf Daten

Beispiel für **Funktionen** des Betriebssystems:

- Einlesen eines Zeichens von der Tastatur
- Ausgeben eines Zeichens auf den Bildschirm
- Laden eines Programms in den Speicher

Komponenten eines Betriebssystems:

- **Organisationsprogramme:** werden benötigt zur
 - Speicherverwaltung
 - Prozessorverwaltung (Zuteilung von Prozessor an zu bearbeitende Programme)
 - Geräteverwaltung (Druckeransteuerung)
 - heute insbesondere auch zur Kommunikation zwischen Rechnern (in Rechnernetzen)
- **Übersetzungsprogramme** (werden aber nicht unbedingt als Teil eines Betriebssystems angesehen)
 - übersetzen Programme, geschrieben in höheren Programmiersprachen, in direkt auf Rechner ausführbare Programme
- **Dienstprogramme:** lösen Standardanwendungsprobleme, z. B.
 - Sortieren
 - Dateiverwaltung
 - Programm laden und starten
 - Programm „binden“ (Programme aus Bibliothek und eigene)
 - Editor
 - Debugger (zur Fehlersuche und -behebung in Programmen)
 - Abrechnungsprogramme (Accounting)

Kommunikation mit Rechner auf Betriebssystemebene: **Kommandosprache**

Beispiele von Betriebssystemen:

- **Großrechner:** IBM: MVS, VM/SP; Siemens: BS2000; heute auch: Unix
- **mittlere Rechner** (z. B. **Workstations**): herstellerspezifische; heute überwiegend: Unix,
- **PCs:** MS-DOS, Windows, Windows-NT, OS/2, heute auch: Unix (z.B. Solaris, Linux)

Trend geht seit geraumer Zeit zu Bedienoberflächen, die komplexe Betriebssystemfunktionen mittels graphischer Symbole zur Verfügung stellen

Reale und virtuelle Maschinen

- Frage: Was geschieht „wirklich“ auf einem Rechner, wenn z.B. ein Oberon- (oder Pascal- oder C-) Programm abläuft? (Frage überhaupt sinnvoll?)
- heutiges Bild: virtuelle Maschine (= Oberon-Maschine), Schichtenmodell (s. Abbildung 2.4)

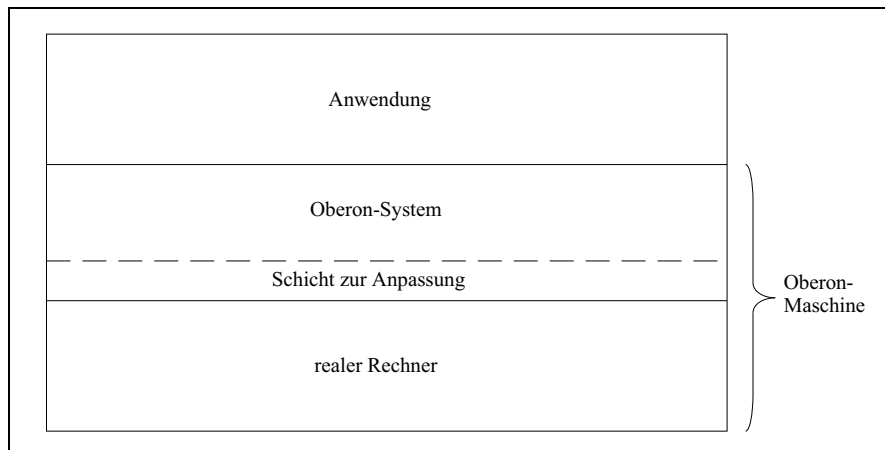


Abbildung 2.4: Virtuelle Oberon-Maschine

- Vorteil dieser Betrachtungsweise: (theoretisch) Unabhängigkeit von Details der Realisierung durch Hardware (etwa: IBM-PC, Atari, Amiga, IBM oder Siemens-Großrechner usw.)
- Die Bildung von logischen Schichten zur Gliederung komplexer Systeme ist eine grundlegende Methode der Informatik. Das bekannteste Beispiel ist das 7-Schichten-Modell der ISO für die offene Kommunikation zwischen Rechensystemen (ISO: **I**nternational **S**tandards **O**rganization), das sog. ISO/OSI-Modell

2.4 Algorithmen

Grundlegendes

Algorithmen sind präzise beschriebene und automatisch durchführbare Verarbeitungsvorschriften.

Name *Algorithmus*: von persisch arabischem Mathematiker Ibn Mûsâ Al-Chwârismi (9. Jahrhundert), Buch über die *Regeln der Wiedereinsetzung und Reduktion* (Eigenschaften von Algorithmen (dazu: Informatik-Duden [Eng88, S. 25]):

1. Algorithmen lösen stets eine ganze Klasse von Problemen; Problemauswahl durch Parameter
2. Algorithmen sind in der Regel *determiniert*, d.h., sie liefern bei gleichen Eingabewerten und unter gleichen Startbedingungen die gleichen Endwerte. (Es gibt aber auch nichtdeterminierte Algorithmen, z.B. stochastische Algorithmen.)
3. Beschreibung eines Algorithmus: endlich (*statische Finitheit*); belegter Speicherplatz zu jeder Zeit: ebenfalls endlich (*dynamische Finitheit*)
4. Resultat nach endlich vielen Schritten (*Terminierung*); Ausnahmen: z.B.: Betriebssystem, Editor
5. Algorithmus heißt *deterministisch*, wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht (vgl. aber nicht-deterministische endliche Automaten (Vorlesung Theoretische Informatik))
6. Formalisierung des Algorithmusbegriffs: u.a. durch sog. *Turing-Maschinen* (abstraktes Maschinenmodell)

Beispiele für Algorithmen

allgemein bekannt: z.B.: Addition, Subtraktion, Multiplikation ganzer Zahlen:

$$\begin{array}{r} 345 \\ + 678 \\ \hline 1023 \end{array}$$

oder: Fakultätsfunktion $n! = n(n-1)!$ für $n \geq 1$

Weitere Beispiele:

Euklidischer Algorithmus

„klassisches“ Beispiel eines Algorithmus (s. [Eng88, S. 219]):

dient zur Bestimmung des größten gemeinsamen Teilers (ggT) zweier gegebener natürlicher Zahlen a und b

z.B.: $a = 693$, $b = 147$

$$\begin{array}{rclcl} 693 & = & 4 \cdot 147 & + & 105 \\ 147 & = & 1 \cdot 105 & + & 42 \\ 105 & = & 2 \cdot 42 & + & 21 \\ 42 & = & 2 \cdot 21 & + & 0 \end{array}$$

allgemein:

$$\begin{array}{rclcl} a & = & q_0 \cdot b & + & r_2 \quad \text{mit } r_2 < b \\ b & = & q_1 \cdot r_2 & + & r_3 \quad \text{mit } r_3 < r_2 \\ r_2 & = & q_2 \cdot r_3 & + & r_4 \quad \text{mit } r_4 < r_3 \\ \dots & & \dots & & \dots \\ r_{n-2} & = & q_{n-2} \cdot r_{n-1} & + & r_n \quad \text{mit } r_n < r_{n-1} \\ r_{n-1} & = & q_{n-1} \cdot r_n & + & 0 \end{array}$$

Der letzte Rest $\neq 0$ ist der ggT von a und b .

2.4.1 PROGRAMMBEISPIEL (EUKLIDISCHER ALGORITHMUS)

```

1
2  MODULE MODggT;
3    (* Implementierung des Euklidischen Algorithmus: *)
4    (* ggT1 verwendet die Operation MOD,*)
5    (* ggT2 verwendet nur die Differenzbildung. *)
6
7    IMPORT In, Out;
8
9    PROCEDURE ggT1*;
10     VAR a, b, r: LONGINT;
11     BEGIN
12       In.Open; In.LongInt(a); In.LongInt(b);
13       REPEAT
14         r := a MOD b; a := b; b := r
15       UNTIL r = 0;
16       Out.Int(a, 10); Out.Ln
17     END ggT1;
18
19     PROCEDURE ggT2*;
20     VAR a, b, r: LONGINT;
21     BEGIN
22       In.Open; In.LongInt(a); In.LongInt(b);
23       WHILE a # b DO
24         IF a > b THEN a := a - b ELSE b := b - a END
25       END;
26       Out.Int(a, 10); Out.Ln
27     END ggT2;
28
29     BEGIN
30     END MODggT.
31
32     Beispielaufruf durch:
33     MODggT.ggT1 693 147~
34     Ausgabe:
35     21
36
37     Beispielaufruf durch:
38     MODggT.ggT2 693 147~
39     Ausgabe:
40     21
41
42

```

Binäres Suchen

vgl. [Eng88, S. 85]

Schnelles Verfahren zur Suche in einem z.B. aufsteigend sortierten Array (etwa Suche

nach Telefonnummer im Telefonbuch)

Acht-Damenproblem

vgl. [Eng88, S. 132]

Problem: gesucht eine Stellung für 8 Damen auf einem Schachbrett, so daß diese sich entsprechend den Schachregeln nicht gegenseitig schlagen können, d.h., jede Zeile, jede Spalte und jede Diagonale darf höchstens eine Dame enthalten. (Es gibt genau 92 Lösungen.)

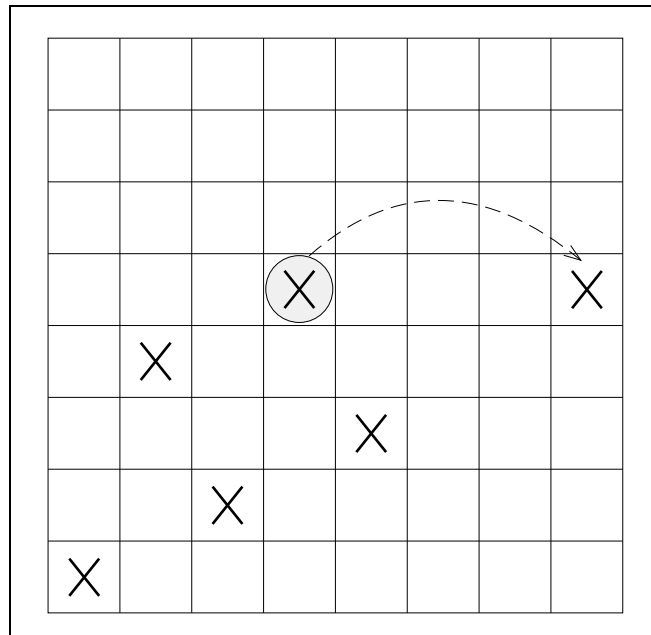


Abbildung 2.5: Acht-Damen-Problem

s. Abbildung 2.5

Algorithmus ist Beispiel für sog. Backtracking-Verfahren

Hinweis: Macht die Beschäftigung mit „Spielen“ einen Sinn?

Spieltheorie: Nobelpreis 1994 Wirtschaftswissenschaften, Reinhard Selten, Uni Bonn (vgl. SPIEGEL 42/94, S. 134)

Mechanik: Bewegung einer Rakete

s. Abbildung 2.6

Physikalisches Prinzip: Impulserhaltung (bei eingeschaltetem Schub):

$$\underbrace{mv}_{\text{Ausgangsimpuls}} = \underbrace{(m - (\Delta m)')(v + \Delta v)}_{\text{neuer Impuls Raketenkörper}} + \underbrace{(\Delta m)'(v - u)}_{\text{Impuls abgestoßenes Gas}} + \underbrace{mg\Delta t}_{\text{Impulsabgabe durch Gravitation}}$$

daraus durch $\Delta t \rightarrow 0$: grundlegende Gleichungen (mit Anfangsbedingungen $x_1 = x(t_1)$, $v_1 = v(t_1)$):

$$\begin{aligned} m \frac{dv}{dt} &= m\dot{v} = \mu u - mg \\ \frac{dm}{dt} &= \dot{m} = -\mu = \text{const} \end{aligned}$$

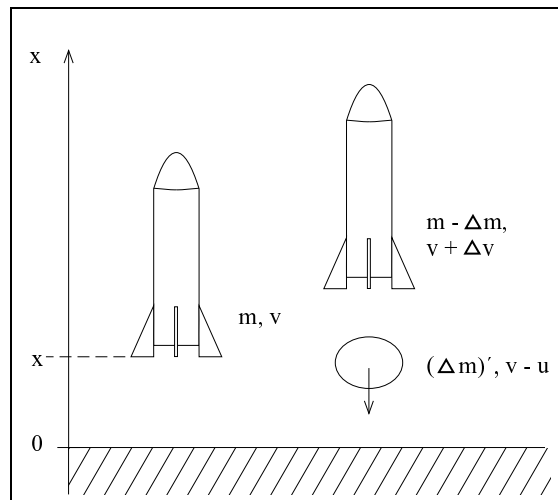


Abbildung 2.6: Simulation von Raketenstart und -landung

$(\Delta m)'$ = Masse des im Intervall Δt ausgestoßenen Gases

$$\mu = \lim_{\Delta t \rightarrow 0} \frac{(\Delta m)'}{\Delta t} = -\dot{m}$$

daraus: Lösung der (Differential-) Gleichungen:

$$\begin{aligned} x(t) &= \begin{cases} H_0(x_1, v_1) + H_1(\tau_1) & \text{mit Schub} \\ H_0(x_1, v_1) & \text{ohne Schub} \end{cases} \\ v(t) &= \begin{cases} V_0(v_1) + V_1(\tau_1) & \text{mit Schub} \\ V_0(v_1) & \text{ohne Schub} \end{cases} \end{aligned}$$

wobei

$$\begin{aligned} H_0(x_1, v_1) &= x_1 + v_1 \Delta t - \frac{1}{2} g (\Delta t)^2 \\ H_1(\tau_1) &= u \left(\Delta t + \tau_1 \left(1 - \frac{\Delta t}{\tau_1} \right) \ln \left(1 - \frac{\Delta t}{\tau_1} \right) \right) \\ V_0(v_1) &= v_1 - g \Delta t \\ V_1(\tau_1) &= -u \ln \left(1 - \frac{\Delta t}{\tau_1} \right) \end{aligned}$$

und $\Delta t = t - t_1$

Dabei bedeuten:

x_1 Ort zum Zeitpunkt t_1

v_1 Geschwindigkeit

u Ausströmgeschwindigkeit der Verbrennungsgase relativ zum Raketenkörper

m_1 Masse

μ ausgestoßene Masse pro Zeiteinheit

$\tau = m/\mu$

Der **Algorithmus** (vereinfacht, in sog. *Pseudo-Code*):

```

 $x_1 :=$  Anfangsort;  $v_1 :=$  Anfangsgeschwindigkeit;  $t_1 :=$  Anfangszeitpunkt;
WHILE (Endetaste nicht gedrückt) DO
  IF (Schub ein) & (noch Treibstoff vorhanden) THEN
     $x_2 := H_0(x_1, v_1) + H_1(...)$ ; (* Neuer Ort *)
     $v_2 := V_0(v_1) + V_1(...)$ ; (* neue Geschwindigkeit *)
     $\tau_2 := \tau_1 - \Delta t$  (* neue Masse, reduzierte Einheit  $\tau := m/\mu$  *)
  ELSE (* ohne Schub, d.h. freier Fall *)
     $x_2 := H_0(x_1, v_1)$ ;  $v_2 := V_0(v_1)$ ;
  END;
  WarteBis( $t_2$ ); (* für zeitlich realistische Darstellung *)
  NeuePositionZeichnen;
  (* neue Startbedingungen für nächsten Zeitschritt: *)
   $x_1 := x_2$ ;  $v_1 := v_2$ ;  $t_1 := t_2$ ;  $t_2 := t_2 + \Delta t$ ;  $m_1 := m_2$ ;
END

```

Problem von Ulam

Die Terminierung eines Algorithmus: kann ein Problem sein. Von dem folgenden Algorithmus ist bis heute unbekannt, ob er für alle n mit $1 < n \in \mathbb{N}$ terminiert (*Problem von Ulam* oder *Syrakusproblem*)

2.4.2 PROGRAMMBEISPIEL (PROBLEM VON ULAM)

```

1
2  MODULE UlamWerte;
3  (* Gibt Wertetabelle aller Werte aus, die durch den Algorithmus
4     durchlaufen werden, beginnend mit dem eingelesenen Wert. *)
5  IMPORT In, Out;
6
7  PROCEDURE Start*;
8    VAR n: LONGINT; k: INTEGER; (* k für übersichtliche Ausgabe *)
9  BEGIN
10    In.Open; In.LongInt(n); k := 1; Out.Ln;
11    WHILE n > 1 DO
12      Out.Int(n, 10); IF (k MOD 6) = 0 THEN Out.Ln END;
13      IF ODD(n) THEN n := 3*n + 1 ELSE n := n DIV 2 END;
14      INC(k)
15    END;
16  END Start;
17
18  END UlamWerte.
19
20  Beispielaufruf durch:
21    UlamWerte.Start 33 ~
22  Ausgabe:
23    33      100      50      25      76      38
24    19      58      29      88      44      22
25    11      34      17      52      26      13

```

```

26   40      20      10      5      16      8
27   4       2
28
29

```

Türme von Hanoi

2.4.3 PROGRAMMBEISPIEL (TÜRME VON HANOI)

```

1
2  MODULE TuermeVonHanoi;
3  (* vgl. Gumm/Sommer: Einführung in die Informatik, S. 120 *)
4  IMPORT In, Out;
5  VAR N: INTEGER;
6
7  PROCEDURE ZieheScheibe(Ausgang, Ziel: INTEGER);
8  BEGIN
9    Out.String("Bewege Scheibe von "); Out.Int(Ausgang, 4);
10   Out.String(" nach "); Out.Int(Ziel, 4); Out.Ln
11  END ZieheScheibe;
12
13  PROCEDURE Hanoi(N, A, B, C: INTEGER);
14  (* Die Prozedur Hanoi führt das Spiel Türme von Hanoi
15   durch, indem der Scheibenstapel von Stab A nach Stab C
16   unter Verwendung von Stab B zur Zwischenablage verwendet wird. *)
17  BEGIN
18    IF N = 1 THEN ZieheScheibe(A, C)
19    ELSE          (* N > 1*)
20      Hanoi(N-1, A, C, B); (* Stapel der Höhe N-1 von A nach B mit Zwischenablage C *)
21      ZieheScheibe(A, C);
22      Hanoi(N-1, B, A, C) (* Stapel der Höhe N-1 von B nach C mit Zwischenablage A *)
23    END
24  END Hanoi;
25
26  PROCEDURE Start*;
27  BEGIN
28    In.Open; In.Int(N); Out.Ln; Hanoi(N, 1, 2, 3); Out.Ln
29  END Start;
30
31  BEGIN
32  END TuermeVonHanoi.
33
34  Beispielaufruf durch:
35  TuermeVonHanoi.Start 4 ~
36  Ausgabe:
37  Bewege Scheibe von 1 nach 2
38  Bewege Scheibe von 1 nach 3
39  Bewege Scheibe von 2 nach 3
40  Bewege Scheibe von 1 nach 2

```

```

41  Bewege Scheibe von 3 nach 1
42  Bewege Scheibe von 3 nach 2
43  Bewege Scheibe von 1 nach 2
44  Bewege Scheibe von 1 nach 3
45  Bewege Scheibe von 2 nach 3
46  Bewege Scheibe von 2 nach 1
47  Bewege Scheibe von 3 nach 1
48  Bewege Scheibe von 2 nach 3
49  Bewege Scheibe von 1 nach 2
50  Bewege Scheibe von 1 nach 3
51  Bewege Scheibe von 2 nach 3
52
53

```

Fibonacci-Zahlen

Fibonacci-Zahlen: spielen eine Rolle z.B. in der Kombinatorik

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

einige Werte:

```

f(0) = 0
f(1) = 1
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
...

```

s. Abbildung 2.7

Busy Beaver

„Der eifrige Biber“

vgl. [Eng88, S. 100] und Brauer, Inf. Spektr. **13**, Heft 2, 61(1990)

Spiel mit dem Ziel, möglichst viele Striche auf ein ursprünglich leeres Band zu schreiben (Jeder Strich ist ein gefällter Baum).

Regel: Jeder Spieler denkt sich eine Folge von n Anweisungen ($n \geq 1$) der folgenden Struktur aus:

z.B.:

$$1 \quad \left\{ \begin{array}{c|c|c} 0 & R & 2 \\ | & L & 3 \end{array} \right.$$

Bedeutung: Anweisung Nr. 1 besagt:

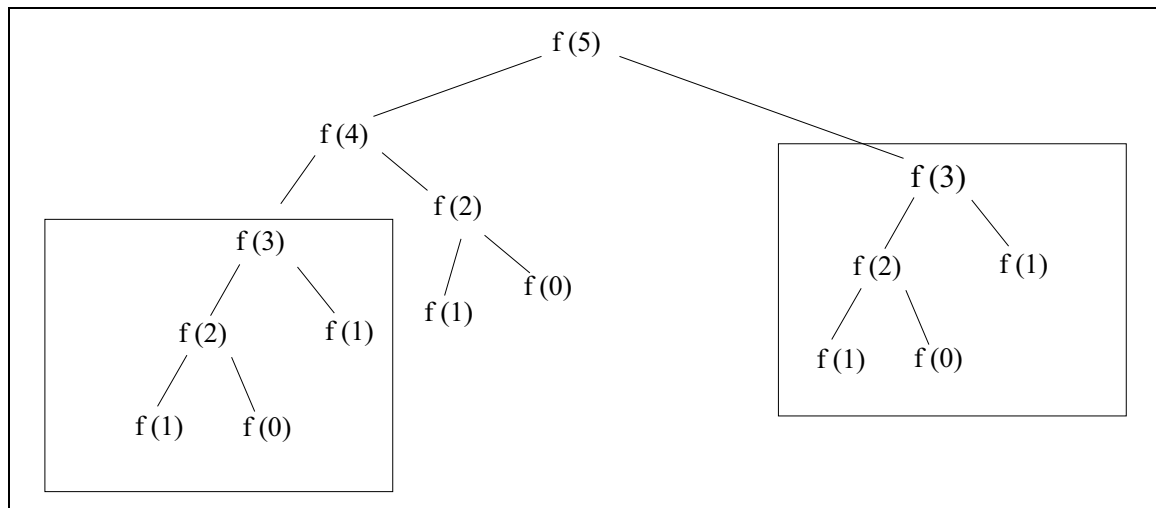


Abbildung 2.7: rekursive Berechnung der Fibonacci-Zahlen

- Wenn auf dem Band eine 0 (d.h. ein Blank) vorgefunden wird, so ist folgendes zu unternehmen:
 1. einen | schreiben, d.h. 0 durch | ersetzen
 2. eine Bewegung nach rechts (R) vornehmen und
 3. fortfahren mit Anweisung Nr. 2
- wenn auf dem Band ein | vorgefunden wird, dann:
 1. einen | schreiben, d.h. in diesem Fall: den | stehen lassen
 2. eine Bewegung nach links (L) vornehmen und
 3. fortfahren mit Anweisung Nr. 3

Gewonnen hat der Spieler, dessen Programm

1. innerhalb einer vorher vereinbarten Frist anhält und
2. die meisten Striche geschrieben hat.

Beispiel für ein vollständiges nicht-triviales busy-beaver- (bb-)Programm:

$$\begin{array}{l}
 1 \quad \left\{ \begin{array}{lll} 0 & | & R \quad 2 \\ | & | & L \quad 3 \end{array} \right. \\
 2 \quad \left\{ \begin{array}{lll} 0 & | & R \quad 3 \\ | & | & R \quad \text{Halt} \end{array} \right. \\
 3 \quad \left\{ \begin{array}{lll} 0 & | & L \quad 1 \\ | & 0 & L \quad 2 \end{array} \right.
 \end{array}$$

Ablauf für obiges Beispiel:
(s. Abbildung 2.8)

• • •								• • •
1				2		3		
2			3		1			
1		3		1		3		
2		Halt						

Abbildung 2.8: Busy Beaver

in diesem Fall also: 6 Striche mit 11 Zügen, dann Halt

Es sei $\Sigma(n)$ die Maximalzahl von Strichen, die ein haltendes, aus n Anweisungen bestehendes bb-Programm auf das leere Band schreiben kann

bekannt ist:

Anzahl Anweisungen n	$\Sigma(n)$	min. erforderliche Schrittzahl
1	1	1
2	4	6
3	6	11
4	13	96
5	?	?
6	?	?
7	?	?

Frage: Kann man $\Sigma(n)$ durch ein Programm berechnen lassen? Annahme: es steht beliebig (aber endlich viel) Zeit zur Verfügung.

Frage wird beantwortet durch den folgenden (streng beweisbaren)

2.4.4 SATZ

Es gibt kein Verfahren (d.h. kein Programm, keinen Algorithmus), mit dem man den Wert von $\Sigma(n)$ für jedes $n \in \mathbb{N}$ berechnen kann.

Der **Beweis** wird indirekt geführt: Angenommen, es gäbe ein derartiges Verfahren. Dann läßt sich ein Widerspruch herleiten.

Einzelheiten: aus Theorie der Turing-Maschinen

Ergebnis ist insbesondere deswegen überraschend, weil man sich vorstellen kann, daß man (mit entsprechendem Zeitaufwand) die Programme explizit aufschreibt:

Programme mit $n = 1, 2, 3, \dots$ Anweisungen:

- 1: $P_1^{(1)}$ $P_2^{(1)}$ $P_3^{(1)}$... $P_{64}^{(1)}$
- 2: $P_1^{(2)}$ $P_2^{(2)}$ $P_3^{(2)}$... $P_{20736}^{(2)}$
- 3: $P_1^{(3)}$ $P_2^{(3)}$ $P_3^{(3)}$... $P_{16777216}^{(3)}$
- 4: ...

Bemerkung: Gesamtzahl der Programme mit n Anweisungen:

$$(4(n+1))^{2n}$$

Begründung:

$$\underbrace{(2 \times 2 \times (n+1))^2 \cdot \dots \cdot (2 \times 2 \times (n+1))^2}_{n \text{ mal}}$$

(Einfache) Übungsaufgabe: Spielzüge eines bb-Programms automatisch durchführen lassen

Teil II

Strukturierte Programmierung. Prozedurale Abstraktion

Kapitel 3

Elemente der Programmierung

3.1 Symbole und Grundtypen

Jedes in einer Programmiersprache geschriebene Programm besteht aus einer Folge von Zeichen, z.B. A, B, C, ..., a, b, c, ..., 0, 1, 2, ... und weiteren.

Zwei wesentliche Komponenten eines Compilers sind der *Scanner* und der *Parser*.

Der Scanner liest den Eingabetext (das Quellprogramm) zeichenweise und identifiziert dabei die *Token* (oder *Symbole*) (sog. *lexikalische Analyse*). Der Parser liest die entstandene Tokenfolge und führt die sog. *syntaktische Analyse* durch (u.a.: Prüfung der syntaktischen Korrektheit des Quellprogramms).

Das Vokabular von Oberon

Symbole (oder *Token*) in Oberon sind

1. Namen (identifiers)
2. Zahlen
3. Zeichenkonstanten
4. Zeichenketten (strings)
5. Operatoren und Begrenzer

Namen

```
ident = letter {letter | digit}.
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
        "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
        "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
        "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
        "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
        "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

s. Abbildung 3.1

(*digit* bedeutet *Ziffer*, nicht *Zahl*!)

integer:

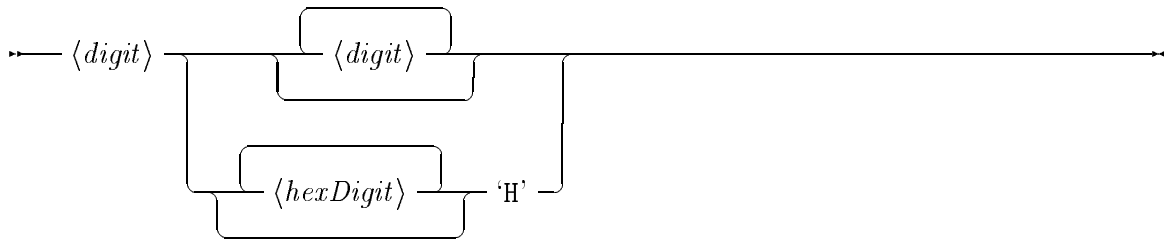


Abbildung 3.4: Syntax von `integer`

ScaleFactor:

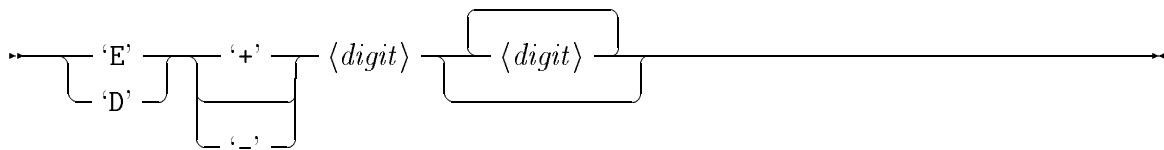


Abbildung 3.5: Syntax von `ScaleFactor`

Ergänzung: Diese Beschreibung des Aufbaus von Zahlen bezieht sich auf die **Sprache** Oberon, **nicht** auf die Ein- und Ausgabe. Die Ein- und Ausgabe wird durch Bibliotheksmodule geregelt, für die `In` und `Out` Primitivbeispiele sind. Für wirkliche Anwendungen dienen u.a. die Module `Texts`, `Oberon`, `Fonts`, ...

Beispiel für die Ausgabe von Zahlen ohne Verwendung von Modul `Out`:

3.1.1 PROGRAMMBEISPIEL (AUSGABE VON GLEITPUNKTZAHLN)

```

1
2  MODULE FloatingPointAusgabe;
3  IMPORT Texts, Oberon, Fonts;
4  VAR
5    W: Texts.Writer;
6    fnt: Fonts.Font;
7
8  PROCEDURE Test*;
9    VAR x: REAL; n, f, E, y: LONGINT;
10  BEGIN
11    x := 12.34;
12    y := 1024; (* Werte, die ausgegeben werden sollen *)
13
14    n := 20; (* Felddbreite, in die der Wert geschrieben wird *)
15    f := 10; (* Anzahl der Nachkommastellen *)
16    E := 0; (* gewünschter Exponent *)
17    fnt := Fonts.This("Syntax14.Scfn.Fnt"); (* Font setzen *)
18
19    Texts.OpenWriter(W);
20    Texts.SetColor(W, 3); Texts.SetFont(W, fnt);

```

```

21
22   E := 3; (* gewünschter Exponent *)
23   Texts.WriteRealFix(W, x, n, f, E); Texts.WriteLn(W);
24
25   E := 0;
26   Texts.WriteRealFix(W, x, n, f, E); Texts.WriteLn(W);
27
28   Texts.WriteHex(W, y); Texts.WriteLn(W);
29
30   Texts.Append(Oberon.Log, W.buf)
31   END Test;
32
33   BEGIN
34   END FloatingPointAusgabe.
35
36   Beispielaufruf durch:
37   FloatingPointAusgabe.Test ~
38
39   Ausgabe:
40       0.0123400000E+03
41       12.3400000000
42       00000400
43       123456789012345678901
44

```

Zeichenkonstanten

CharConst:

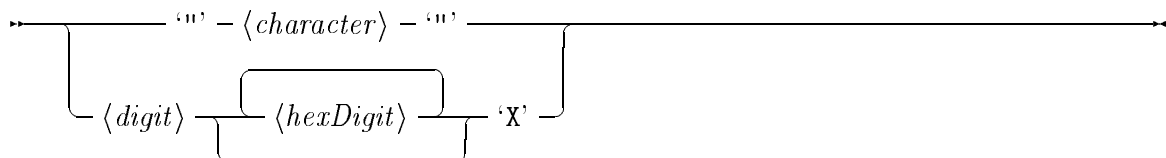


Abbildung 3.6: Syntax von Zeichenkonstanten

s. Abbildung 3.6

character:



Abbildung 3.7: Syntax von Zeichen

s. Abbildung 3.7

Sonderzeichen (`specialChar`) im ASCII-Zeichensatz (ASCII: **A**merican **S**tandard **C**ode of **I**nformation **I**nterchange):

Beispiele für Zeichenkonstanten: `"a"`, `"1"`, `61X`

Zeichenketten (Strings)

```
string = "" {character} "" | "'" {character} "'.
```

string:

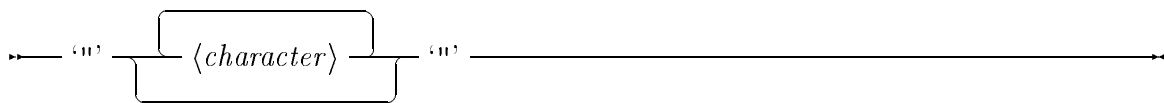


Abbildung 3.8: Syntax von Zeichenketten

s. Abbildung 3.8

Beispiele: `"Eine Zeichenkette"`
`"Noch 'ne Zeichenkette"`
`'DOS ist ein "Betriebssystem".'`

Operatoren und Begrenzer

Operatoren und Begrenzer sind

- einzelne Sonderzeichen
- Paare gewisser Sonderzeichen
- reservierte Wörter (Schlüsselwörter)

vollständige Liste:

- einzelne Sonderzeichen:

`+` `-` `*` `/` `~` `&` `.` `,` `;` `|` `(` `)`
`[` `]` `{` `}` `^` `=` `#` `<` `>` `:`

- Paare gewisser Sonderzeichen:

`:=` `<=` `>=` `..`

- reservierte Wörter (in Oberon-2)

 Schlüsselwörter

ARRAY	IMPORT	RETURN
BEGIN	IN	THEN
BY	IS	TO
CASE	LOOP	TYPE
CONST	MOD	UNTIL
DIV	MODULE	VAR
DO	NIL	WHILE
ELSE	OF	WITH
ELSIF	OR	
END	POINTER	
EXIT	PROCEDURE	
FOR	RECORD	
IF	REPEAT	

Reservierte Wörter können nicht als Namen verwendet werden.

Operatoren und Begrenzer sind *Terminalsymbole* in der Oberon-Syntax.

Vordefinierte Namen

vollständige Liste (Bedeutung wird – soweit noch nicht bekannt – im Verlaufe der beiden Vorlesungen Informatik I und II erklärt):

Vordeclarierte		
Datentypen	Funktionsprozeduren	gewöhnliche Prozeduren
SHORTINT	ABS(x)	COPY(x,v)
INTEGER	ASH(x,n)	DEC(v)
LONGINT	CAP(x)	DEC(v,n)
REAL	CHR(x)	EXCL(v,x)
LONGREAL	ENTIER(x)	HALT(x)
BOOLEAN	LEN(v,n)	INC(v)
SET	LEN(v)	INC(v,n)
CHAR	LONG(x)	INCL(v,x)
	MAX(T)	NEW(v)
	MIN(T)	NEW(v,x0,...,xn)
	ODD(x)	
	ORD(x)	
	SHORT(x)	
	SIZE(T)	

Grundtypen

Hier werden zunächst nur die Typen als solche besprochen, (noch) nicht die Zugriffsoperationen. Es gibt 8 Grundtypen:

1. SHORTINT
2. INTEGER

3. LONGINT
4. REAL
5. LONGREAL
6. BOOLEAN
7. SET
8. CHAR

minimaler Wert vom Typ T: durch Aufruf MIN(T)

maximaler Wert vom Typ T: durch Aufruf MAX(T)

Anwendung von MIN(T) und MAX(T):

3.1.2 PROGRAMMBEISPIEL (BASISTYPEN IN OBERON)

```
1  MODULE SizeOfBasicTypes;
2  IMPORT Texts, Oberon, Out;
3
4  CONST f = 30;
5  VAR W: Texts.Writer;
6
7  PROCEDURE Show*;
8  BEGIN Out.Ln;
9
10     Out.String("SHORTINT: ");
11     Out.Int(MIN(SHORTINT), f); Out.String(" ... ");
12     Out.Int(MAX(SHORTINT), f); Out.Ln;
13
14     Out.String("INTEGER: ");
15     Out.Int(MIN(INTEGER), f); Out.String(" ... ");
16     Out.Int(MAX(INTEGER), f); Out.Ln;
17
18     Out.String("LONGINT: ");
19     Out.Int(MIN(LONGINT), f); Out.String(" ... ");
20     Out.Int(MAX(LONGINT), f); Out.Ln;
21
22
23     Out.String("REAL: ");
24     Out.Real(MIN-REAL), f); Out.String(" ... ");
25     Out.Real(MAX-REAL), f); Out.Ln;
26
27
28     Out.String("LONGREAL: ");
29     Texts.WriteLongReal(W, MIN(LONGREAL), f);
30     Texts.WriteString(W, " ... ");
31     Texts.WriteLongReal(W, MAX(LONGREAL), f); Texts.WriteLine(W);
32     Texts.Append(Oberon.Log, W.buf);
33
```

```

34
35   Out.String("CHAR: ");
36   Out.Int(ORD(MIN(CHAR)), f); Out.String("  ... ");
37   Out.Int(ORD(MAX(CHAR)), f); Out.Ln;
38   (* Die beiden auftretenden Werte vom Typ CHAR sind nicht abdruckbar:
39     deswegen diese Art der Ausgabe. *)
40
41   Out.String("SET: ");
42   Out.Int(MIN(SET), f); Out.String("  ... ");
43   Out.Int(MAX(SET), f); Out.Ln; Out.Ln
44
45 END Show;
46
47 BEGIN Texts.OpenWriter(W)
48 END SizeOfBasicTypes.
49
50 Beispielaufruf durch:
51 SizeOfBasicTypes.Show ~
52
53 Ausgabe:
54   SHORTINT:           -128  ...           127
55   INTEGER:            -32768 ...          32767
56   LONGINT: -2147483648 ...          2147483647
57   REAL:              -3.4028235E+38 ...    3.4028235E+38
58   LONGREAL: -1.797693134862316D+308 ...    1.797693134862316D+308
59   CHAR:              0  ...           255
60   SET:               0  ...           31
61
62   SHORTINT:           -128  ...           127
63   INTEGER:            -32768 ...          32767
64   LONGINT: -2147483648 ...          2147483647
65   REAL:              -3.4028235E+38 ...    3.4028235E+38
66   LONGREAL: -1.797693134862316D+308 ...    1.797693134862316D+308
67   CHAR:              0  ...           255
68   SET:               0  ...           31

```

Die Typen SHORTINT, INTEGER und LONGINT

ganzzahlige Typen

Möglicher Überlauf muß im Programm abgefangen werden!

Die Typen REAL und LONGREAL

Gleitkommazahlen: Annäherung der reellen Zahlen (im Sinne der Mathematik)

Die Hierarchie der numerischen Typen

Es gilt:

$$\text{LONGREAL} \supseteq \text{REAL} \supseteq \text{LONGINT} \supseteq \text{INTEGER} \supseteq \text{SHORTINT}$$

Diese Hierarchie ist so zu verstehen, daß die kleineren Bereiche *kompatibel* zu den größeren Bereichen sind.

Z.B.: `x := 102345;` (mit `VAR x: REAL;`). Genauigkeitsverlust ist möglich!

Der Typ BOOLEAN

logische Wahrheitswerte: `FALSE`, `TRUE`

Einzelheiten im folgenden Abschnitt

Der Typ SET

Darstellung von *Mengen* (im Sinne der Mathematik)

Beispiele:

Oberon-Schreibweise	mathematische Schreibweise
<code>{}</code>	$\emptyset$
<code>{1,2,5}</code>	$\{1, 2, 5\}$
<code>{0, 2 .. 4, 8}</code>	$\{0, 2, 3, 4, 8\}$
<code>{0 .. 10}</code>	$\{0, \dots, 10\}$

Die zusammenstehenden Punkte `..` bilden **ein** Symbol. Zu beachten ist:

1. Die Oberon-Syntax muß präzise eingehalten werden, die mathematische Schreibweise muß nur verständlich sein!
2. Unterschied zu Pascal: geschweifte Klammern sind in Oberon **keine** Alternative zu `( * *)`!

Einzelheiten zum Datentyp `SET` im folgenden Abschnitt

Der Typ CHAR

Basis: ASCII-Code

Die Menge der Werte vom Typ `CHAR` ist vollständig geordnet, und jedes Zeichen hat eine feste Position oder *Ordnungszahl*. Die Ordnungszahl ist erhältlich durch

```
...
VAR i: INTEGER;
    c: CHAR;
...
i := ORD(c);
...
```

Umkehroperation `CHR()`, z.B.: `c := CHR(i);`

3.1.3 PROGRAMMBEISPIEL (AUSGABE VON ASCII-ZEICHEN)

1

```

2  MODULE ASCIIAusgabe;
3  IMPORT Out;
4
5  PROCEDURE Start*;
6  VAR i, k: INTEGER;
7  BEGIN
8  FOR k := 0 TO 15 DO
9  Out.Int(16*k, 4); Out.String(": ");
10  FOR i := 0 TO 15 DO Out.Char(CHR(i + 16*k)); Out.String(" ") END;
11  Out.Ln
12  END;
13  Out.Ln
14  END Start;
15
16 END ASCIIAusgabe.
17
18 ASCIIAusgabe.Start ~
19
20

```

3.2 Rechnerinterne Darstellung ganzzahliger Werte

Rechner (genauer: der Prozessor) kennt nur 0, 1: z.B. 001111100001...

Die Interpretation ist sehr verschieden, je nach Zusammenhang; z.B. kann Bitfolge Instruktion bedeuten: die darauffolgenden Bits werden dann als Daten für diese Instruktion gedeutet; Deutung kann je nach Instruktion verschieden sein (fiktives) Beispiel:

```

Add R1,15      * Addiere 15 zum Inhalt von Register 1
                * 15: waeren in diesem Beispiel die Daten

```

Detail-Kenntnisse der internen Zahlendarstellung:

- für höhere Programmiersprachen in der Regel entbehrlich,
- wichtig dagegen z.B. für C: Es gibt in C Operationen, die bitweise wirken, in Oberon nur in speziellen Modulen.

zur Vereinfachung: nur 4 bits betrachtet:

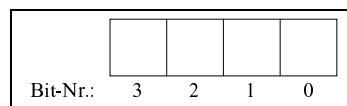


Abbildung 3.9: zur binären Zahlendarstellung

Abbildung 3.9

damit sind 2^4 verschiedene Werte darstellbar, z.B.:

```

0000 --> 0
0001 --> 1
0010 --> 2
0011 --> 3
0100 --> 4
....
1111 --> 15

```

Frage: Darstellung *negativer* Werte?

eine Möglichkeit wäre:

$$\text{Bit Nr. 3} = \begin{cases} 0 & : \text{ wenn Zahl} \geq 0 \\ 1 & : \text{ wenn Zahl} < 0 \end{cases}$$

sonst gleiche Darstellung?

wird in Wirklichkeit (bei INTEGER-Werten) so **nicht** gemacht

Grund: andere Art der Darstellung macht elektronische Schaltungen im Addierwerk einfacher

Prinzip: $16 = 2^4$ mögliche Bitmuster auf Kreis darstellen:

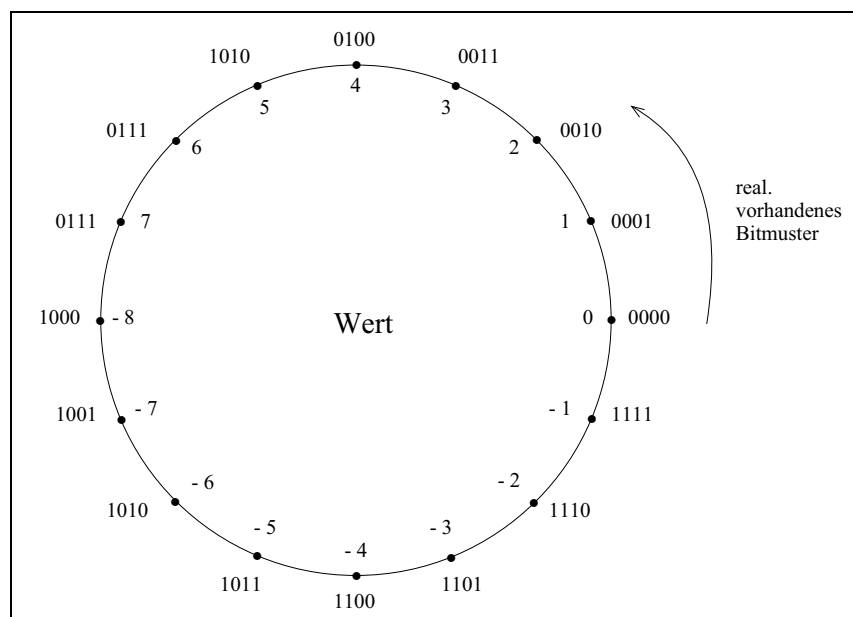


Abbildung 3.10: Binäre Zahlendarstellung: Zweierkomplementdarstellung

s. Abbildung 3.10

3.3 Deklarationen, Ausdrücke und Wertzuweisungen

Deklarationen

Jedes Objekt in Oberon (z.B. Konstante, Variable, Prozedur, ...) muß vor seiner Benutzung deklariert werden.

Konstantendeklarationen

z.B.:

```
CONST a = 256; m = 1024; q = m DIV a; r = m MOD a; pi = 3.1415;
```

Auswertung von Ausdrücken auf der rechten Seite: muß *statisch* möglich sein, d.h. nur Konstanten können auftreten

nicht möglich ist z.B. (sinngemäß):

```
In.Int(M); CONST N = 10*M; VAR A: ARRAY N OF INTEGER;
```

Konstruktion wäre auch syntaktisch falsch!

s. Abbildung 3.11

Vorteile der Einführung von benannten Konstanten:

- Programm mit benannten Konstanten ist leichter lesbar
- einfache Änderung von Konstanten bei mehrmaligem Auftreten der Konstanten oder Abhängigkeiten im Programm

Variablendeklarationen

Eigenschaften von Variablen:

- Variablen besitzen Typ; der Typ legt die Menge der möglichen Werte fest, z.B.:
`VAR x: INTEGER;` bedeutet: $x = 0, \pm 1, \pm 2 \dots$ und $-32768 \leq x \leq +32767$

- allgemeiner: $-2^{n-1} \leq x \leq +2^{n-1} - 1$, wobei

$n = 8$	für Typ	SHORTINT	(1 Byte)	-128	$\leq x \leq$	$+127$
$n = 16$	für Typ	INTEGER	(2 Bytes)	$-32\,768$	$\leq x \leq$	$+32\,767$
$n = 32$	für Typ	LONGINT	(4 Bytes)	$-2\,147\,483\,648$	$\leq x \leq$	$+2\,147\,483\,647$

- Variablen besitzen aktuellen Wert

Sprechweisen:

- Eine Variablendeklaration *definiert eine Variable* und *bindet einen Namen und einen Typ* an sie.
- Eine Variable ist ein *Instanz (Instanzierung, Inkarnation)* ihres Typs.
- Der Variablenname *bezeichnet (repräsentiert)* die Variable.

mögliche Typen: INTEGER, REAL, ...; aber auch z.B. ARRAY N OF INTEGER

s. Abbildung 3.11

Ausdrücke

Syntax und allgemeine Regeln

Beispiel für Ausdrücke:

`a + b`

`a*a + b*b`

`(2*sin(x) + 3*a - (x+y)*y)/(x + y)`

DeclSeq:

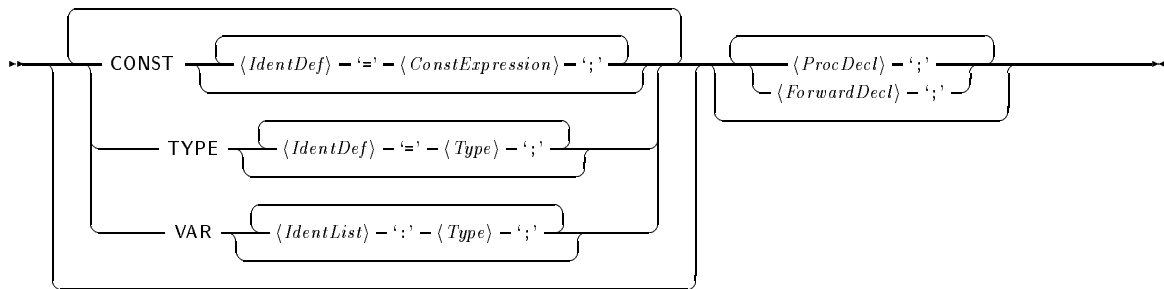


Abbildung 3.11: Syntax von Deklarationsfolge

Ausdrücke sind aufgebaut aus Operatoren und Operanden; Ausdrücke werden ausgewertet, in der Regel von links nach rechts

Frage: Was macht die allgemeine Definition von Ausdruck schwierig? (die beliebige Einsetzbarkeit von Ausdrücken in andere Ausdrücke)

s. Abbildung 3.12

Typregeln

Welches sind zulässige Kombinationen in einem Ausdruck?

z.B. sind

"a" + 3.1415

(a=b) - 12

syntaktisch korrekt(!) aber kaum sinnvoll.

Regel in Oberon: Typen der Operanden müssen **identisch** sein mit Ausnahme einiger wohldefinierter und leicht verständlicher Fälle.

Prozeduren zur Typkonversion:

ORD(x)

CHR(x)

SHORT(x)

LONG(x)

ENTIER(x)

s. Abbildung 3.13

Anwendungsbeispiel:

3.3.1 PROGRAMMBEISPIEL (TYPKONVERTIERUNGEN)

```

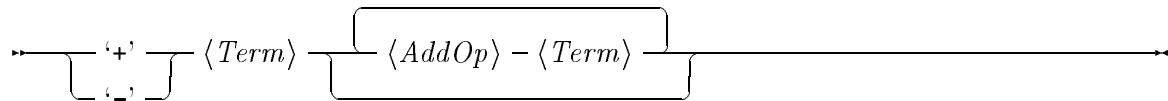
1
2  MODULE TypeConversion;
3    IMPORT Out;
4
5    PROCEDURE Test*;
6      VAR x: REAL;
7    BEGIN

```

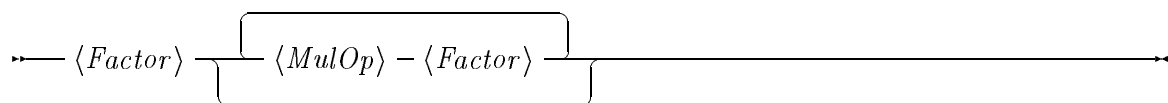
Expr:



SimplExpr:



Term:



Factor:

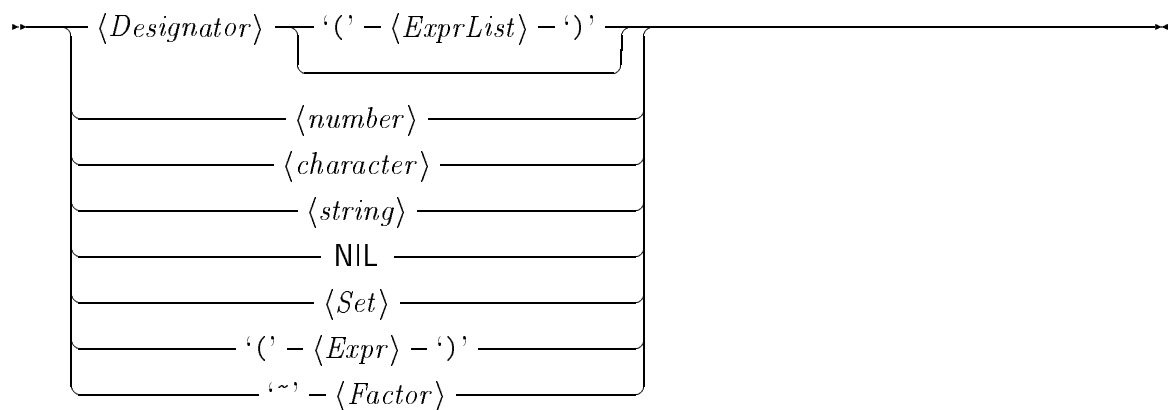


Abbildung 3.12: Syntax von Ausdrücken

```

8      x := 5.234;   Out.Int(ENTIER(x), 7); Out.Ln;
9      x := -5.234;  Out.Int(ENTIER(x), 7); Out.Ln;
10     END Test;
11
12     BEGIN
13     END TypeConversion.
14
15     Beispielaufufr durch:
16     TypeConversion.Test ~
17
18     Ausgabe:
19         5
20        -6
21

```

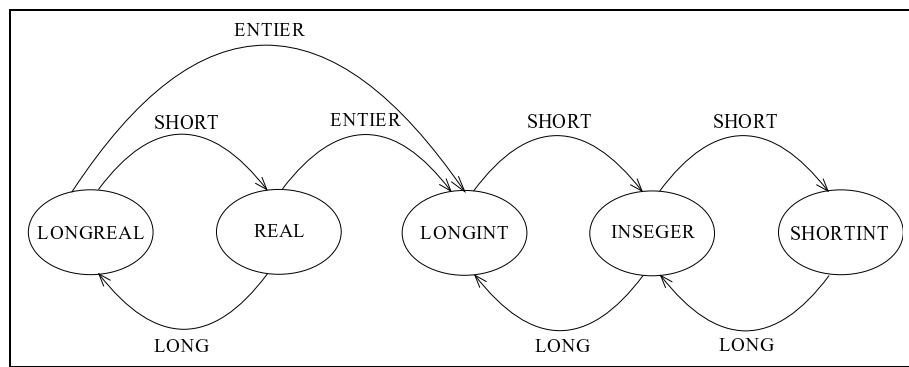


Abbildung 3.13: Datentypen von Zahlen

22

23

Boolesche Ausdrücke

zwei mögliche Werte: FALSE und TRUE, z. B.

```

VAR i: INTEGER;
...
IF i >= 0 THEN ... END;
...

```

Boolescher Ausdruck `i >= 0` wird ausgewertet:

- Aussage ist bei gegebenem Wert von `i` **richtig** (Wert TRUE)
- oder **nicht richtig** (Wert FALSE)

in Oberon direkt vorgesehene logische Operatoren zur Verknüpfung Boolescher Operanden:

- `&` (Und, sowohl-als-auch)
- `OR` (Oder, entweder-oder – oder beide)
- `~` (Nicht)

Diese Operatoren reichen aus, um **jede** Boolesche Operation darzustellen. (auch `&`, `~` allein würden reichen.)

Bsp.: Exklusives Oder (XOR): kann nachgebildet werden:

`(a & ~b) OR (b & ~a)`

(Einzelheiten: Theoretische Informatik)

genauere Beschreibung durch sog. **Wahrheitstabellen** für logische Operatoren:

`a` und `b` seien Variablen (oder Ausdrücke) vom Typ `BOOLEAN`. Dann ist:

<code>a</code>	<code>~a</code>
FALSE	TRUE
TRUE	FALSE

a	b	a & b	a OR b	a = b
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	TRUE

Anwendungsbeispiel: logischer Schalter:

```
VAR a: BOOLEAN;
...
a := a = FALSE; (* Alternative: a := ~a; *)
```

Wirkung:

vorher	nachher
FALSE	TRUE
TRUE	FALSE

vordeclarierte Funktionsprozedur, die Boolesche Werte liefert:

$$\text{ODD}(i) = \begin{cases} \text{TRUE} & : \text{ wenn Integer-Variable } i \text{ ungerade} \\ \text{FALSE} & : \text{ wenn Integer-Variable } i \text{ gerade} \end{cases}$$

3.3.2 PROGRAMMBEISPIEL (EIGENE IMPLEMENTIERUNG VON ODD)

```
1
2  MODULE ImplOdd;
3  (* eigene Implementierung von ODD(i) *)
4  IMPORT In, Out;
5
6  PROCEDURE Odd(i: LONGINT): BOOLEAN;
7  BEGIN RETURN i MOD 2 = 1
8  END Odd;
9
10 PROCEDURE Test*;
11   VAR i: LONGINT;
12   BEGIN In.Open; In.LongInt(i);
13
14   IF Odd(i) THEN Out.String(" i ungerade "); Out.Ln
15   ELSE Out.String(" i gerade "); Out.Ln
16   END;
17
18 END Test;
19
20 END ImplOdd.
21
22 ImplOdd.Test ~
23
24
```

Syntaktisch richtige boolesche Ausdrücke und Zuweisungen

```
VAR b: BOOLEAN;
```

```
b := TRUE; b := FALSE;
b := b = TRUE; (* richtig, aber keine Wirkung, wie b := b; *)
b := b = FALSE; (* Wirkung eines logischen Schalters: b := ~b; *)
b := 17 < 3;
b := i # j; (* i, j: INTEGER-Werte *)
b := (12 > 2) & ( 7 < 2);
b := "abc" < "bc"; (* Resultat: TRUE *)
b := 'A' < "AB"; (* richtig in Oberon-2 *)
```

Auch die Vergleiche $1 \# 2.0$ und $1 = 1.0$ sind syntaktisch zulässig.

Vergleich der Form IF $x = y$ THEN ... END;

mit VAR x, y : REAL;

ist *syntaktisch* richtig aber ein *semantischer* Fehler, weil der Vergleich auf präzise Gleichheit der Bitmuster, die x und y zugrundeliegen, von Zufälligkeiten einer vorherigen Rundung abhängt.

Syntaktisch falsche Zuweisungen

```
VAR a, b: BOOLEAN;
a := 0; a := 1;
b := 12 + 3;
b := a + 1;
```

In der Mathematik wird in der Booleschen Algebra 0, 1, + und $\cdot$ nur dann geschrieben, wenn nicht gleichzeitig die natürliche Zahlen 0 und 1 mit den entsprechenden Rechenoperationen auftreten.

In der Mathematik gilt für boolesche Variable stets:

$a \& b = b \& a$ (Kommutativgesetz).

In Programmiersprachen, z.B. Oberon, gilt Entsprechendes nicht ohne Einschränkung.

Beispiel: Suche in Array das erste auftretende Element > 0 , falls ein solches existiert:

3.3.3 PROGRAMMBEISPIEL (BOOLESCHER AUSDRÜCKE)

```
1
2  MODULE BooleanExpression;
3    IMPORT Out;
4    CONST N = 5;
5    VAR A: ARRAY N OF INTEGER;
6
7    PROCEDURE Test*;
8      VAR i: INTEGER;
9    BEGIN
10     A[0] := -10; A[1] := -1; A[2] := 12; A[3] := -4;
11     A[4] := -3;
12     i := 0;
13     WHILE (i <= N-1) & (A[i] <= 0) DO INC(i) END;
```

```

14
15     IF i <= N-1 THEN
16         Out.Int(i, 9); Out.String(": "); Out.Int(A[i], 9); Out.Ln
17     ELSE Out.String(" kein Element "); Out.Ln
18     END;
19
20 (* zweite Form führt i.a. zu Laufzeitfehler:
21    i := 0;
22    WHILE (A[i] <= 0) & (i <= N-1) DO INC(i) END;
23    ...
24 *)
25 END Test;
26 END BooleanExpression.
27
28 Beispielaufruf durch:
29 BooleanExpression.Test
30
31 Ausgabe:
32 2:      12
33
34

```

Die Ausgabe Boolescher Werte ist nicht vorgesehen in `Out`. Einfache Alternative:

3.3.4 PROGRAMMBEISPIEL (BOOLESCHE WERTE: AUSGABE)

```

1
2  MODULE BooleanIO;
3  IMPORT In, Out;
4
5  PROCEDURE WriteBoolean(b: BOOLEAN);
6  BEGIN
7      IF b THEN Out.String("TRUE") ELSE Out.String("FALSE") END
8      (* Alternative: b = TRUE : wäre überflüssig und umständlich! *)
9  END WriteBoolean;
10
11  PROCEDURE Test*;
12  VAR s: BOOLEAN; i: INTEGER;
13  BEGIN In.Open; In.Int(i);
14      s := i > 2; WriteBoolean(s); Out.Ln;
15      WriteBoolean("ab"<"b"); Out.Ln
16  END Test;
17
18  END BooleanIO.
19
20 Beispielaufruf durch:
21 BooleanIO.Test -8 ~
22

```

```

23  Ausgabe:
24  FALSE
25  TRUE
26
27

```

Anwendungsbeispiel zum Datentyp BOOLEAN

Beispiel aus Graphik Zeichne Pixel innerhalb eines gegebenen Rahmens (z.B. Rechteck oder Kreis)

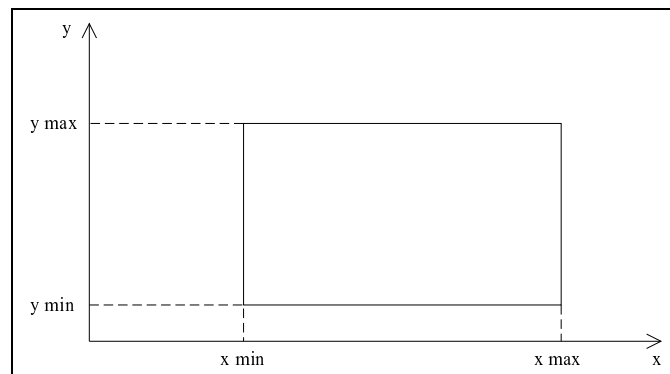


Abbildung 3.14: zu Booleschen Variablen

(s. Abbildung 3.14)

```

IF (x >= xmin) & (x <= xmax) & (y >= ymin) & (y <= ymax)
THEN XYplane.Dot(x, y, 1)
ELSE Out.String(" ... Fehlermeldung ... ")
END;

```

vollständiges **Programmbeispiel**:

Wähle durch einen Zufallszahlengenerator einen zufälligen Punkt innerhalb eines gegebenen Rechtecks aus, und zeichne ihn, falls er innerhalb eines gegebenen Kreises liegt. Bedingung dafür, daß Punkt in Kreis liegt:

$$(x - x_0)^2 + (y - y_0)^2 \leq R^2$$

Formel für Zufallszahlen (mit $m = 2^{31} - 1$ und $a = 7^5$):

$$Z_{n+1} = (a * Z_n) \text{ MOD } m$$

3.3.5 PROGRAMMBEISPIEL (BOOLESCHE WERTE)

```

1
2  MODULE RandomNumbers;
3  (* Random Number Generator, RW, page 12 *)
4  IMPORT Math;
5
6  VAR Z: LONGINT;
7

```

```

8  PROCEDURE Uniform*(): REAL;
9  CONST
10   a = 16807; m = 2147483647; q = m DIV a; r = m MOD a;
11  VAR
12   g: LONGINT;
13  BEGIN
14   g := a*(Z MOD q) - r*(Z DIV q);
15   IF g > 0 THEN Z := g ELSE Z := g + m END;
16   RETURN Z*(1.0/m) (* Normierung auf 0.0 ... 1.0 *)
17 END Uniform;
18
19 PROCEDURE Exp*(mu: REAL): REAL;
20 BEGIN RETURN -Math.In(Uniform())/mu
21 END Exp;
22
23 PROCEDURE InitSeed*(seed: LONGINT);
24 BEGIN Z := seed
25 END InitSeed;
26
27 BEGIN Z := 1;
28 END RandomNumbers. (* Copyright M. Reiser, 1992 *)
29
30
31 MODULE PixelTest;
32 IMPORT Out, XYplane, RandomNumbers, Math;
33 CONST
34   lmax = 100000; (* Anzahl Durchläufe *)
35   Xmin = 50; Xmax = 450; Ymin = 30; Ymax = 530;
36   N = Xmax - Xmin; M = Ymax - Ymin;
37
38 PROCEDURE DotInCircle(x, y, x0, y0, R: LONGINT): BOOLEAN;
39 (* Liegt der Punkt (x,y) im Kreis um (x0,y0) mit Radius R ? *)
40 BEGIN
41   RETURN Math.sqrt((x-x0)*(x-x0) + (y-y0)*(y-y0)) <= R;
42 END DotInCircle;
43
44 PROCEDURE Test*;
45   VAR x, y, x0, y0, R: INTEGER; i: LONGINT;
46 BEGIN XYplane.Open; RandomNumbers.InitSeed(2145);
47   x0 := 250; y0 := 150; R := 180; (* Def. des Kreises *)
48
49   i := 0;
50   WHILE i <= lmax DO
51     x := Xmin+ SHORT(ENTIER(N*RandomNumbers.Uniform()));
52     y := Ymin+ SHORT(ENTIER(M*RandomNumbers.Uniform()));
53     IF DotInCircle(x, y, x0, y0, R) THEN
54       XYplane.Dot(x, y, XYplane.draw)

```

```

55     END;
56     INC(i);
57     END;
58     Out.String(" --- Ende --- "); Out.Ln
59 END Test;
60
61 BEGIN
62 END PixelTest.
63
64 Beispielaufruf durch:
65     PixelTest.Test ~
66
67

```

Der mathematische Mengenbegriff. Zusammenfassung

Grundbegriffe *Menge* ist Zusammenfassung von „wohlunterschiedenen“ Objekten zu einem Ganzen; Objekte heißen *Elemente* charakteristische Eigenschaften

- Die Reihenfolge der Elemente spielt bei der Festlegung einer Menge keine Rolle.
- Kein Element tritt mehrfach auf.

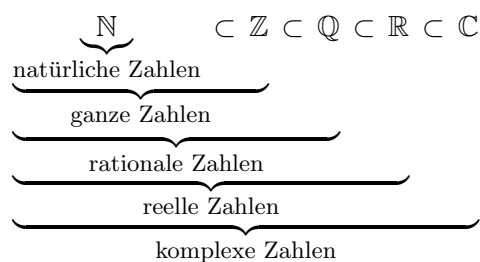
Beispiele:

Menge der natürlichen Zahlen: Schreibweisen: $\mathbb{N} = \{0, 1, 2, \dots\}$ oder $\mathbb{N} = \{ i \mid i = 0, 1, 2, \dots \}$

Menge der Großbuchstaben:

$$\mathcal{B} = \{A, B, C, \dots\}$$

Standardzahlenmengen in der Mathematik:



Grundbegriffe:

1. x ist *Element* von A : $x \in A$ z.B. $1 \in \mathbb{N}$ und $-3 \notin \mathbb{N}$
s. Abbildung 3.15
2. Es seien A, B Mengen: *Durchschnitt* von A und B : $A \cap B$: Menge der Elemente, die sowohl zu A als auch zu B gehören. also:

$$x \in A \cap B \Leftrightarrow x \in A \text{ und } x \in B$$

(s. Abbildung 3.16)

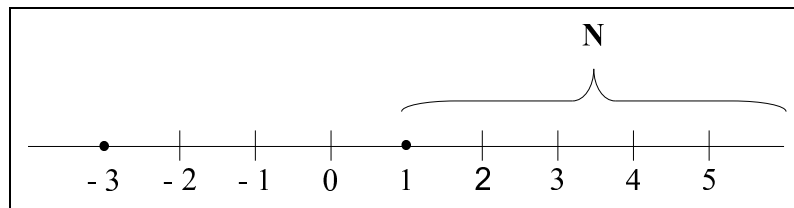
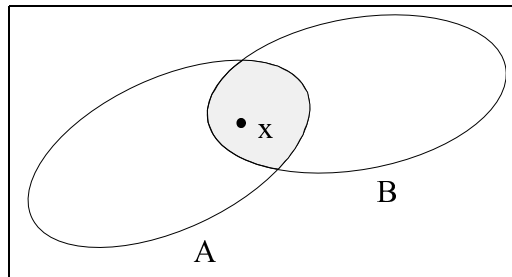
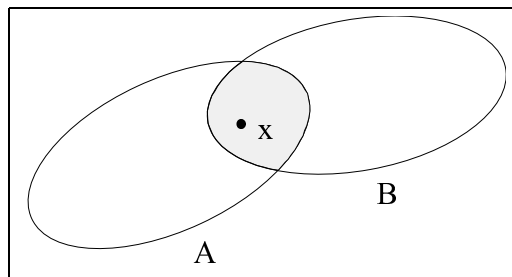


Abbildung 3.15: Natürliche Zahlen

Abbildung 3.16: Schnitt von Mengen: $x \in A, x \in B$

3. *Vereinigung*: $A \cup B$: Menge der Elemente, die zu A oder zu B gehören:

$$x \in A \cup B \Leftrightarrow x \in A \text{ oder } x \in B$$

Abbildung 3.17: Vereinigung von Mengen: $A \cup B$

s. Abbildung 3.17

4. *Differenz*: $A \setminus B$: Menge der Elemente, die zu A , jedoch nicht zu B gehören:

$$x \in A \setminus B \Leftrightarrow x \in A \text{ und } x \notin B$$

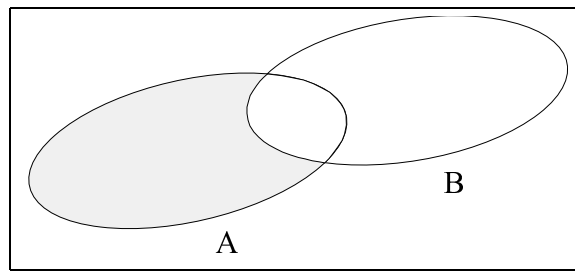
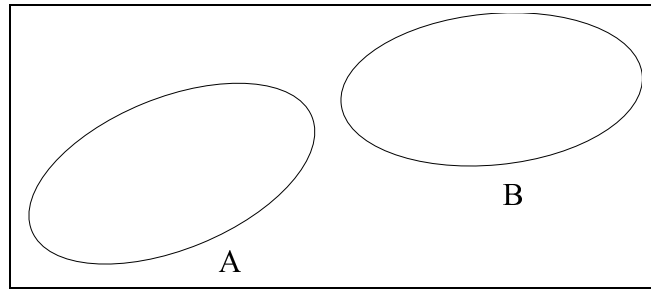
s. Abbildung 3.18

5. *Leere Menge*: $\emptyset$: Menge, die kein Element enthält, z.B. $A \cap B = \emptyset$ bedeutet: A und B haben keine gemeinsamen Elemente

s. Abbildung 3.19

6. *Gleichheit* von Mengen: $A = B$: Jedes Element von A gehört zu B und umgekehrt:

$$x \in A \Rightarrow x \in B \text{ und } x \in B \Rightarrow x \in A$$

Abbildung 3.18: Differenz von Mengen: $A \setminus B$ Abbildung 3.19: Leerer Durchschnitt von Mengen: $A \cap B = \emptyset$

oder

$$A \subseteq B \text{ und } B \subseteq A$$

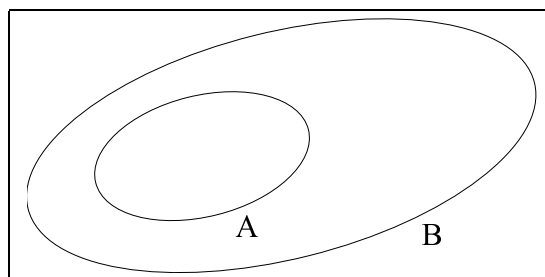
7. A ist *Teilmenge* von B : $A \subseteq B$ d.h.:

$$x \in A \Rightarrow x \in B$$

Schreibweise darf auch umgekehrt werden:

$$A \subseteq B \Leftrightarrow B \supset A$$

B heißt auch *Obermenge* von A

Abbildung 3.20: Enthaltensein von Mengen: $A \subset B$

s. Abbildung 3.20

(Alle Begriffe gibt es entsprechend auch in Oberon; aber: Symbole sind andere, da $\in$, $\notin$, ... auf Tastaturen nicht vorhanden sind.)

Beispiele

$$M = \{0, 1, 2, 3\} = \{ i \mid i \in \mathbf{N}, 0 \leq i \leq 3 \}$$

ist Menge, bestehend aus den Zahlen 0, 1, 2 und 3.

Beispiel für *wahre* Aussagen (später: Boolescher Wert: TRUE):

$$1 \in M$$

$$4 \notin M$$

$$\{1\} \subset M$$

$$\{0, 3\} \subset M$$

$$\{0, 4, 5\} \not\subset M$$

Beispiel für *falsche* Aussagen (später: Boolescher Wert: FALSE):

$$5 \in M$$

$$\{0, 1, 2, 3, 4\} \subset M$$

Alle Teilmengen einer festen gegebenen Menge M bilden die sog. Potenzmenge über M :

Definition:

Sei M eine (fest vorgegebene) Menge. Dann heißt die Menge

$$\mathcal{P}(M) = \{A \mid A \subseteq M\}$$

die Potenzmenge über M .

Satz:

Es gilt

$$|\mathcal{P}(M)| = 2^{|M|}$$

Beweis: z.B.: durch Induktion über die Anzahl der Elemente in M

oder:

Elemente von M seien $\omega_0, \omega_1, \dots, \omega_{n-1}$

Charakterisierung einer Teilmenge durch explizite Angabe, ob ein Element dazugehört

(1) oder nicht (0): z.B. für $n = 4$:

ω_3	ω_2	ω_1	ω_0	
0	0	0	0	$= \emptyset$
0	0	0	1	$= \{\omega_0\}$
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	$= \{\omega_2, \omega_1\} = \{\omega_1, \omega_2\}$
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	$= M$

explizite Aufzählung sämtlicher *Teilmengen* von M : (insgesamt $2^4 = 16$)

Leere Menge:

$$\emptyset$$

Einelementige Teilmengen von M :

$$\{0\}$$

$$\{1\}$$

$$\{2\}$$

$$\{3\}$$

Zweielementige Teilmengen von M :

$$\{0, 1\}$$

$$\{0, 2\}$$

$$\{0, 3\}$$

$$\{1, 2\}$$

$$\{1, 3\}$$

$$\{2, 3\}$$

Dreielementige Teilmengen von M :

$$\{0, 1, 2\}$$

$$\{0, 1, 3\}$$

$$\{0, 2, 3\}$$

$$\{1, 2, 3\}$$

Vollständiges M :

$$\{0, 1, 2, 3\}$$

Beispiele für Operationen mit *zwei* Mengen:

$$\text{wenn } M = \{0, 1, 2, 3\}$$

$$\text{und } N = \{2, 3, 4, 5\}$$

$$M \cap N = \{2, 3\}$$

$$M \cup N = \{0, 1, 2, 3, 4, 5\}$$

Die Potenzmenge $\mathcal{P}(\{0, 1, 2, 3\})$ von $\{0, 1, 2, 3\}$ lautet dann explizit:

$$\underbrace{\{\emptyset, \{0\}, \{1\}, \{2\}, \{3\}, \{0, 1\}, \{0, 2\}, \{0, 3\}, \dots, \{0, 1, 2\}, \{0, 1, 3\}, \dots, \{0, 1, 2, 3\}\}}_{\text{insgesamt 16 Elemente}}$$

Mengenausdrücke in Oberon

Tabelle der Mengenoperationen A und B seien Mengen. Dann gilt folgende Schreibweise:

mathematische Schreibweise	Oberon-Schreibweise
$A \cup B$	$A + B$
$A \cap B$	$A * B$
$A \setminus B$	$A - B$
$e \in A$	$e \text{ IN } A$
z.B.: $e \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$	$e \text{ IN } \{0 \dots 9\}$
$A \cup B \setminus (A \cap B)$ (<i>symmetrische Differenz</i>)	$A \text{ / } B$
$A = B$	$A = B$
$A \neq B$	$A \# B$
A^c (Komplement)	$\neg A$

Hinweis: In Oberon ist für Variablen A, B des Typs SET **keiner** der folgenden Ausdrücke zulässig:

- $A < B$
- $A \leq B$
- $A > B$
- $A \geq B$

Programmbeispiel

3.3.6 PROGRAMMBEISPIEL (DEMO ZUM TYP SET)

```

1
2  MODULE SetDemo2;
3    IMPORT In, Out;
4    CONST c = 5;
5
6    VAR s1, s2, s3: SET; (* 3 Variablen vom Typ SET *)
7
8    PROCEDURE ShowElements*(s: SET);
9      (* alle Elemente aus s anzeigen *)
10     VAR i: INTEGER;
11     BEGIN
12       FOR i := MIN(SET) TO MAX(SET) DO
13         IF i IN s THEN Out.Int(i, c) END
14       END; Out.Ln
15     END ShowElements;
16
17     PROCEDURE IsSubset*(s1, s2: SET): BOOLEAN;
18       (* im Sinne von s1 <= s2 *)
19       VAR b: BOOLEAN; i: INTEGER;
20       BEGIN

```

```
21   i := MIN(SET); b := TRUE;
22   WHILE b & (i <= MAX(SET)) DO
23     IF i IN s1 THEN b := i IN s2 END; INC(i)
24   END;
25   RETURN b;
26 END IsSubset;
27
28 (* 2. Variante: *)
29 PROCEDURE IsSubset2*(s1, s2: SET): BOOLEAN;
30 BEGIN RETURN s1*s2 = s1
31 END IsSubset2;
32
33 PROCEDURE Start*;
34   VAR i, j: INTEGER;
35
36 BEGIN Out.Int(MIN(SET), c); Out.Int(MAX(SET), c); Out.Ln;
37
38   s1 := {1,2,3 .. 7, 9}; ShowElements(s1);
39   s2 := {12};           ShowElements(s2+{1,2,3});
40   s3 := s1 + s2;        ShowElements(s3);
41   i := 2; j := 4;       ShowElements({i .. i*j});
42
43   s1 := {0,1,2}; s2 := {0,1,2};
44   IF s1 = s2 THEN Out.String(" Mengen sind gleich. "); Out.Ln END;
45
46   s2 := {0,1,2,3};
47   IF s1 # s2 THEN Out.String(" Mengen sind ungleich. "); Out.Ln END;
48
49   ShowElements(-{0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30});
50
51   s1 := {0 .. 30}; s2 := {0 .. 10, 12.. 31};
52   IF ~IsSubset2(s1, s2) THEN
53     Out.String(" ... ist nicht enthalten "); Out.Ln
54   END;
55
56   ShowElements(s2/s1);
57
58   Out.Ln; Out.Ln
59 END Start;
60
61 BEGIN
62 END SetDemo2.
63
64
65 MODULE SetTest;
66   IMPORT Out, S := SetDemo2;
67   VAR A, B, C: SET;
```

```

68
69  PROCEDURE Start*;
70  BEGIN
71    Out.Int(MIN(SET), 8); Out.Int(MAX(SET), 8); Out.Ln;
72    A := {2, 4, 6}; S.ShowElements(A);
73    (*
74      INCL(A, 7); S.ShowElements(A);
75      EXCL(A, 2); S.ShowElements(A);
76    *)
77    A := A + {7}; S.ShowElements(A);
78    A := A - {2}; S.ShowElements(A);
79
80    B := {2 .. 30};
81    S.ShowElements(-B);
82    S.ShowElements({0 .. 31}-B);
83    Out.Ln;
84
85    A := {0, 1, 2}; B := {0, 1, 2};
86
87    IF S.IsSubset2(A, B) & (B - A # {}) THEN
88      Out.String("ist echte Teilmenge");
89    END;
90    Out.Ln;
91
92    IF S.IsSubset(A, B) & (B - A # {}) THEN
93      Out.String("ist echte Teilmenge");
94    END
95
96  END Start;
97
98  BEGIN
99  END SetTest.
100
101  Beispielaufruf durch:
102  SetDemo.Start ~
103
104  Ausgabe:
105    0 31
106    1 2 3 4 5 6 7 9
107    1 2 3 12
108    1 2 3 4 5 6 7 9 12
109    2 3 4 5 6 7 8
110    Mengen sind gleich.
111    Mengen sind ungleich.
112    1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
113    ... ist nicht enthalten
114    11 31

```

```

115
116 Beispielaufruf durch:
117   SetTest.Start
118
119 Ausgabe:
120     0    31
121     2    4    6
122     2    4    6    7
123     4    6    7
124     0    1    31
125     0    1    31
126
127 System.Free SetTest SetDemo
128
129

```

Beispiel:

	bedeutet:	mathematisch:
$i \text{ IN } (-A)$	$(i \text{ IN } \{0 \dots \text{MAX}(\text{SET})\}) \ \& \ \sim (i \text{ IN } A)$	$i \in \bar{A}$
$i \text{ IN } (A + B)$	$(i \text{ IN } A) \text{ OR } (i \text{ IN } B)$	$i \in A \cup B$
$i \text{ IN } (A - B)$	$(i \text{ IN } A) \ \& \ \sim (i \text{ IN } B)$	$i \in A \setminus B$
$i \text{ IN } (A * B)$	$(i \text{ IN } A) \ \& \ (i \text{ IN } B)$	$i \in A \cap B$
$i \text{ IN } (A / B)$	$(i \text{ IN } A) \ \# \ (i \text{ IN } B)$	$i \in (A \cup B) \setminus (A \cap B)$

3.3.7 PROGRAMMBEISPIEL (INTERNE DARSTELLUNG VON SET)

```

1
2  MODULE SETDemo2;
3    IMPORT SYSTEM, Out;
4    CONST c = 5;
5
6    PROCEDURE ShowElements*(s: SET);
7      (* alle Elemente aus s anzeigen *)
8      VAR i: INTEGER;
9      BEGIN
10       FOR i := MIN(SET) TO MAX(SET) DO
11         IF i IN s THEN Out.Int(i, c) END
12       END
13     END ShowElements;
14
15     PROCEDURE Start*;
16       VAR k: INTEGER; i: LONGINT; s: SET;
17       BEGIN
18         s := {};
19         FOR k := 0 TO 15 DO
20           INCL(s, k); i := SYSTEM.VAL(LONGINT, s);
21           Out.String("{}"); ShowElements(s); Out.String("}, ");
22           Out.String("interne Darst., als LONGINT interpr.: ");
23           Out.Int(i, c); Out.Ln;

```

```

24     EXCL(s, k)
25     END;
26     Out.Ln; Out.Ln
27     END Start;
28
29     BEGIN
30     END SETDemo2.
31
32     Beispielaufruf durch:
33     SETDemo2.Start ~
34
35     Ausgabe:
36     { 0}, interne Darst., als LONGINT interpr.: 1
37     { 1}, interne Darst., als LONGINT interpr.: 2
38     { 2}, interne Darst., als LONGINT interpr.: 4
39     { 3}, interne Darst., als LONGINT interpr.: 8
40     { 4}, interne Darst., als LONGINT interpr.: 16
41     { 5}, interne Darst., als LONGINT interpr.: 32
42     { 6}, interne Darst., als LONGINT interpr.: 64
43     { 7}, interne Darst., als LONGINT interpr.: 128
44     { 8}, interne Darst., als LONGINT interpr.: 256
45     { 9}, interne Darst., als LONGINT interpr.: 512
46     {10}, interne Darst., als LONGINT interpr.: 1024
47     {11}, interne Darst., als LONGINT interpr.: 2048
48     {12}, interne Darst., als LONGINT interpr.: 4096
49     {13}, interne Darst., als LONGINT interpr.: 8192
50     {14}, interne Darst., als LONGINT interpr.: 16384
51     {15}, interne Darst., als LONGINT interpr.: 32768
52
53

```

Frage: Wie kann man Mengen definieren (und damit umgehen), die z.B. Werte vom Typ `Real` oder kompliziertere Werte enthalten?

Vordeclarierte Funktionen

Tabelle aller vordeclarierten Funktionen (Funktionsprozeduren):

Vordeclarierte Funktionsprozeduren	
ABS(x)	LONG(x)
ASH(x,n)	MAX(T)
CAP(x)	MIN(T)
CHR(x)	ODD(x)
ENTIER(x)	ORD(x)
LEN(v,n)	SHORT(x)
LEN(v)	SIZE(T)

aufgerufene Funktion: ist Bestandteil innerhalb eines Ausdrucks im Unterschied zu einer aufgerufenen (gewöhnlichen) Prozedur: diese bildet eine Anweisung.

3.4 Kontrollstrukturen

Bedingte Anweisungen

Bedingte Anweisungen sind

- If-Anweisung
- Case-Anweisung

Struktur der Case-Anweisung im einfachsten Fall (beispielhaft):

```
CASE i OF
  0: ...
| 1: ...
| 2: ...
| 3: ...
  ...
ELSE ...
END;
```

Beispiel zur Case-Anweisung:

3.4.1 PROGRAMMBEISPIEL (ZUR CASE-ANWEISUNG)

```
1
2  MODULE CaseDemo;
3  (* Das Modul CaseDemo zählt in einem Eingabestring die Anzahl
4     kleiner Vokale, die Anzahl von Großbuchstaben sowie die
5     Anzahl sonstiger Zeichen aus. *)
6
7  IMPORT In, Out;
8  CONST N = 40;
9
10 PROCEDURE Start*;
11   VAR
12     i: INTEGER; S: ARRAY N OF CHAR;
13     VokalKlein, Grossbuchstabe, SonstWas: INTEGER;
14
15 BEGIN In.Open; i := 0;
16   WHILE In.Done & (i <= N - 1) DO In.Char(S[i]); INC(i) END;
17
18   (* ... zum Testen ... *)
19   i := 0;
20   WHILE i <= N - 1 DO
21     Out.Int(i, 4); Out.String(": "); Out.Char(S[i]); Out.Ln; INC(i)
22   END;
23   (* ... zum Testen ... *)
24
25   i := 0; VokalKlein := 0; Grossbuchstabe := 0; SonstWas := 0;
26   WHILE (i <= N - 1) & (S[i] # "") DO
```

```

27     CASE S[i] OF
28         "a", "e", "i", "o", "u": INC(VokalKlein) |
29         "A" .. "Z": INC(Grossbuchstabe)
30     ELSE INC(SonstWas)
31     END;
32     INC(i);
33 END; (* von WHILE *)
34
35     Out.Int(VokalKlein, 5); Out.Int(Grossbuchstabe, 5);
36     Out.Int(SonstWas, 5); Out.Ln;
37 END Start;
38
39 BEGIN
40 END CaseDemo.
41
42 Beispielaufruf durch:
43 CaseDemo.Start 123 Dieses ist ein Text ...
44
45

```

Hinweise zu `In.Done`:

1. `TRUE` (nach `Open`)
2. bleibt `TRUE`, solange Leseoperationen korrekt terminieren

Wiederholungsanweisungen

Wiederholungsanweisungen sind

- While-Anweisung
- Repeat-Anweisung
- Loop-Anweisung

Beispiel zur Loop-Anweisung:

3.4.2 PROGRAMMBEISPIEL (ZUR LOOP-ANWEISUNG)

```

1
2  MODULE LoopDemo;
3  (* Die Prozedur Start im Modul LoopDemo läuft in einer LOOP-
4     Schleife solange, bis entweder die Laufzeit Tmax ms
5     überschritten ist oder der Wert des Zählindex i den Maximalwert
6     Imax erreicht hat. *)
7
8  IMPORT Out, Oberon;
9
10  PROCEDURE Start*;
11      VAR Tmax, Imax, t, t0, i: LONGINT;
12  BEGIN

```

```
13   Tmax := 200; lmax := 800; i := 0; t0 := Oberon.Time();
14   LOOP
15     (*
16       ... irgendwelche Aktionen der Prozedur ...
17     *)
18     t := Oberon.Time() - t0; Out.Int(t, 8); Out.Int(i, 8); Out.Ln;
19     IF (t >= Tmax) OR (i >= lmax) THEN
20       Out.String(" Ende von LOOP "); Out.Ln; EXIT
21     END;
22     INC(i)
23   END
24   END Start;
25
26   END LoopDemo.
27
28   Beispielaufruf durch:
29   LoopDemo.Start ~
30
31   Ausgabe:
32       0      0
33      55      1
34      55      2
35      55      3
36     110      4
37     110      5
38     110      6
39     165      7
40     165      8
41     165      9
42     220     10
43   Ende von LOOP
44
45
```


Kapitel 4

Prozedurale Abstraktion. Konkrete Datenstrukturen

4.1 Allgemeines zur Rolle der Abstraktion

Diskussion:

- Was bedeutet *Abstraktion*?
- Abstraktionsstufenmodell

4.2 Prozeduren

Parameter einer Prozedur

Beispiel für Prozedur ohne Parameter (als benannte Anweisungsfolge): Bruch kürzen:
Algorithmus: größeren der Werte a und b durch Differenz ersetzen: z.B.:

a	b
49	35
14	35
14	21
14	7
7	7

ggT von a , b mit $a > b$ ist auch ggT von $a - b$, b

4.2.1 PROGRAMMBEISPIEL (BRÜCHE (1))

```
1
2  MODULE BruchKuerzen; (* Variante 1 *)
3    IMPORT Out;
4
5    VAR Z, N, X, Y: INTEGER;
6
7    PROCEDURE Kuerze;
8      VAR a, b: INTEGER;
9    BEGIN
10     (* ggT von Z und N feststellen und dann kuerzen *)
```

```

11    a := Z; b := N;
12    WHILE a # b DO
13      IF a > b THEN a := a - b ELSE b := b - a END
14    END;
15    Z := Z DIV a; N := N DIV a;
16    Out.Int(Z, 6); Out.String("/"); Out.Int(N, 6); Out.Ln
17  END Kuerze;
18
19  PROCEDURE Start*;
20  BEGIN
21    Z := 42; N := 120; Kuerze;
22
23    (* Folgender Aufruf funktioniert natürlich nicht: *)
24    X := 24; Y := 16; Kuerze;
25    Out.Ln
26  END Start;
27
28  BEGIN
29  END BruchKuerzen.
30
31  Beispielaufruf durch:
32  BruchKuerzen.Start~
33
34  Ausgabe:
35      7/   20
36      7/   20
37
38
39

```

Problem: Prozedur kann in der Form nur verwendet werden, wenn der Zähler Z und der Nenner N heißt. (Z und N sind sog. globale Variablen.) daher:

4.2.2 PROGRAMMBEISPIEL (BRÜCHE (2))

```

1
2  MODULE BruchKuerzen; (* Variante 2 *)
3  IMPORT Out;
4
5  VAR Z, N, X, Y: INTEGER;
6
7  PROCEDURE Kuerze(Z, N: INTEGER);
8    VAR a, b: INTEGER;
9  BEGIN
10    (* ggT von Z und N feststellen und dann kuerzen *)
11    a := Z; b := N;
12    WHILE a # b DO
13      IF a > b THEN a := a - b ELSE b := b - a END
14    END;

```

```

15   Z := Z DIV a; N := N DIV a;
16   Out.Int(Z, 6); Out.String("/"); Out.Int(N, 6); Out.Ln
17   END Kuerze;
18
19   PROCEDURE Start*;
20   BEGIN
21     Z := 42; N := 120; Kuerze(Z, N);
22     X := 24; Y := 16; Kuerze(X, Y);
23     Out.Ln
24   END Start;
25
26   BEGIN
27   END BruchKuerzen.
28
29   Beispielaufruf durch:
30   BruchKuerzen.Start ~
31
32   Ausgabe:
33       7/   20
34       3/    2
35
36

```

allgemein:

Prozeduren (auch Unterprogramme genannt) sind vorab definierte Programmteile

Vorteile der Verwendung von Prozeduren:

- lösen Teilprobleme einmal; können beliebig häufig aufgerufen werden
- erhöhen Übersichtlichkeit durch Untergliederung des Programms, daher: leichtere Fehlersuche
- Vorteil nur, wenn folgende (provisorische) generelle Regeln beachtet werden: Prozeduren sollten immer so geschrieben werden, daß sie
 - eine genau definierte Aufgabe autonom lösen (z.B. Bruch kürzen)
 - und diese Aufgabe auch dann noch lösen, wenn sie aus ihrer (Programm-) Umgebung herausgelöst und in eine andere Umgebung versetzt werden

Nachteil

- geringfügiger Geschwindigkeitsverlust (in praktisch allen Fällen vollkommen bedeutungslos)

4.2.3 PROGRAMMBEISPIEL (VERTAUSCHEN ZWEIER WERTE)

```

1
2   MODULE WerteVertauschen;
3   IMPORT Out;
4

```

```

5  PROCEDURE Swap(VAR x, y: INTEGER);
6    VAR h: INTEGER;
7  BEGIN
8    h := x; x := y; y := h
9  END Swap;
10
11 PROCEDURE Start*;
12   VAR x, y: INTEGER;
13 BEGIN x := 3; y := 7;
14   Out.String(" vor Aufruf: "); Out.Int(x, 5); Out.Int(y, 5); Out.Ln;
15   Swap(x, y);
16   Out.String(" nach Aufruf: "); Out.Int(x, 5); Out.Int(y, 5); Out.Ln;
17   Out.Ln
18 END Start;
19
20 BEGIN
21   WerteVertauschen.
22
23   Beispielaufruf durch:
24   WerteVertauschen.Start ~
25
26   Ausgabe:
27   vor Aufruf:    3    7
28   nach Aufruf:   7    3
29
30

```

insgesamt: 2 Arten von Parametern in Prozeduren:

1. Wertparameter
2. Variablenparameter (VAR-Parameter)

Erklärung der Unterschiede:

Wertparameter: kann man sich als in der Prozedur deklarierte Variablen vorstellen, die durch das aufrufende Programm initialisiert (gesetzt) werden. Das aufrufende Programm kann deren Werte ändern; aber: keine Wertänderung der übergebenen Variablen nach außen. Wertübergabe erfolgt durch Ausdruck, insbesondere also auch durch Variable oder Konstante

Variablenparameter: Nur Variablen können übergeben werden (keine Ausdrücke, keine Konstanten). Änderungen an diesen Variablen werden nach außen hin sichtbar. Interpretation: rufendes Programm stellt den *Speicherplatz*, der mit den Variablenparametern verbunden ist, direkt zur Bearbeitung zur Verfügung

Zusammenstellung:

	Wertparameter	Variablenparameter
	Übergabe von Wert	Übergabe von Variablen
Aktualparameter	Ausdruck, insbesondere also Konstante oder Variable	nur Variable, auch strukturiert

Hinweise auf Klausuraufgaben:

Fehler der Art entdecken:

```
...

PROCEDURE P(VAR x: INTEGER);
BEGIN
  ...
END P;

PROCEDURE Start*;
  VAR i: INTEGER;
BEGIN
  ...
  P(3*i + 7);      (* Fehler! *)
  ...
END Start;

...
```

Sichtbarkeitsbereich

Frage: Wo sind deklarierte Namen sichtbar, d.h., wo können Sie verwendet werden?

4.2.4 PROGRAMMBEISPIEL (SICHTBARKEITSBEREICH)

```
1
2  MODULE M;
3    VAR x: REAL;
4
5    PROCEDURE P;
6      VAR i: INTEGER;
7        PROCEDURE A;
8          PROCEDURE B;
9            BEGIN
10             END B;
11
12        BEGIN
13        END A;
14
15    BEGIN
16    END P;
17
18
19    PROCEDURE Q;
20      PROCEDURE R;
21        BEGIN
22        END R;
```

```

23
24     PROCEDURE S;
25     BEGIN
26     END S;
27
28     BEGIN END Q;
29
30
31 BEGIN
32 END M.
33
34

```

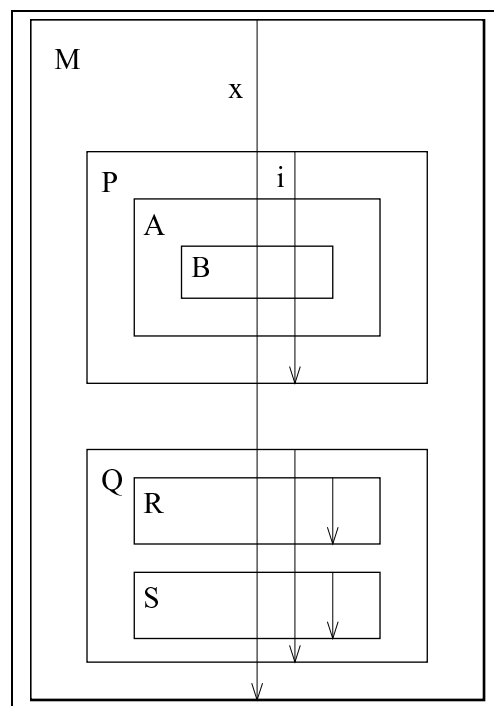


Abbildung 4.1: Sichtbarkeitsbereiche für Namen

(s. Abbildung 4.1) Erläuterungen

Problem bei unzulässiger Verwendung globaler Parameter:

4.2.5 PROGRAMMBEISPIEL (GLOBALE PARAMETER)

```

1
2  MODULE ProblemGlobaleParam;
3    IMPORT Out;
4    CONST N = 10;
5    VAR i: INTEGER;
6
7    PROCEDURE P;
8    BEGIN

```

```

9      (* ... *)
10     (* irgendwelche Aktionen einer laengeren Prozedur *)
11     (* ... *)
12
13     INC(i);
14
15     (* ... *)
16     (* weitere Aktionen ... *)
17     (* ... *)
18     END P;
19
20     PROCEDURE Test*;
21     VAR k: INTEGER;
22     BEGIN i := 5;                (* nur zur Sicherheit: *) k := 0;
23     WHILE (i >= 0) & (k <= N-1) DO
24         Out.Int(i, 6); Out.Ln; P; DEC(i); (* nur zur Sicherheit: *) INC(k)
25     END;
26     Out.Ln; Out.Ln
27     END Test;
28
29     BEGIN
30     END ProblemGlobaleParam.
31
32     Beispielaufruf durch:
33     ProblemGlobaleParam.Test ~
34
35

```

Regel: Sichtbarkeitsbereich von Variablen stets so klein wie möglich machen

Gewöhnliche Prozeduren und Funktionsprozeduren

Schablone für Deklarationen:

gewöhnliche Prozedur:

```

PROCEDURE P(x: INTEGER);
BEGIN
    ...
    ...
END P;

```

Funktionsprozedur:

```

PROCEDURE f(x: INTEGER): INTEGER;
BEGIN
    ...
    RETURN Ausdruck;
    ...
END f;

```

1. **Funktionsprozedur:** Rückgabewert einer Funktionsprozedur wird festgelegt in

RETURN Ausdruck;

Aufruf einer Funktionsprozedur erfolgt innerhalb von Ausdruck; Typ des Rückgabewertes muß dem Typ im Prozedurkopf „entsprechen“

2. **gewöhnliche Prozeduren** haben (in diesem Sinne) *keine* Rückgabewerte

Aufruf einer gewöhnlichen Prozedur: in Prozeduranweisung

Beispiel zum Vergleich einer gewöhnlichen mit einer Funktionsprozedur:

4.2.6 PROGRAMMBEISPIEL (FUNKTIONSPROZEDUR / GEWÖHNLICHE PROZEDUR)

```

1
2  MODULE Vergleich;
3    (* von gewoehnlichen Prozeduren und Funktionsprozeduren*)
4    IMPORT Out, Math;
5
6    PROCEDURE S(VAR x: REAL);
7    BEGIN
8      x:= 0.5*(Math.exp(x) - Math.exp(-x))
9    END S;
10
11   PROCEDURE sinh(x: REAL): REAL;
12   BEGIN
13     RETURN 0.5*(Math.exp(x) - Math.exp(-x))
14   END sinh;
15
16   PROCEDURE Start*;
17     VAR y: REAL;
18   BEGIN
19     (* 1. Moeglichkeit: Verwendung von S(x): *)
20     y := 0.7;
21     S(y); Out.Real(2*y + 4, 12); Out.Ln;
22
23     (* 2. Moeglichkeit: Verwendung von sinh(x): *)
24     Out.Real(2*sinh(0.7) + 4, 12); Out.Ln;
25
26     Out.Ln
27   END Start;
28
29   BEGIN
30   END Vergleich.
31
32   Beispielaufruf durch:
33   Vergleich.Start ~
34
35   Ausgabe:

```

```

36  5.5172E+00
37  5.5172E+00
38
39

```

Prozedurale Variablen

allgemeine Bemerkung zum Begriff Funktion:

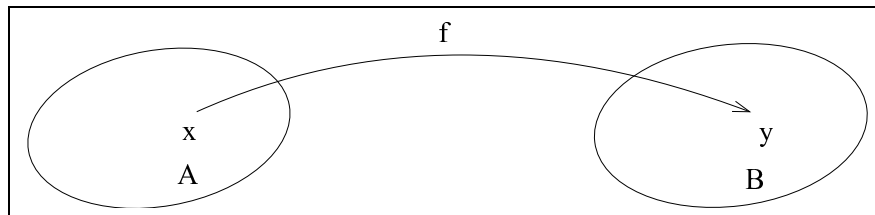


Abbildung 4.2: mathematischer Begriff *Funktion*

(s. Abbildung 4.2)

4.2.7 PROGRAMMBEISPIEL (FUNKTIONSTABELLIERUNG)

```

1  MODULE FunktionsTabelle;
2  IMPORT Out, Math;
3
4  PROCEDURE f1(x: REAL): REAL;
5  BEGIN RETURN 1 + x*x
6  END f1;
7
8  PROCEDURE f2(x: REAL): REAL;
9  BEGIN RETURN 2 + x*x*x
10 END f2;
11
12 PROCEDURE f3(x: REAL): REAL;
13 BEGIN RETURN x*Math.sin(x)
14 END f3;
15
16 PROCEDURE Tabelliere(f: PROCEDURE(x: REAL): REAL; a,b: REAL; n:INTEGER);
17 (* Die Prozedur Tabelliere erstellt eine Tabelle von n+1
18 Funktionswerten  $f(x_i)$  der Funktionsprozedur f mit
19  $x_i := a + ((b - a)/n)i$ ,  $0 \leq i \leq n$ .
20 *)
21 CONST fmt = 15;
22 VAR i: INTEGER; epsilon: REAL;
23 BEGIN
24   epsilon := (b-a)/n;
25   FOR i := 0 TO n DO Out.Real(f(a + epsilon*i), fmt) END
26 END Tabelliere;
27
28 PROCEDURE Start*;

```

```

29  VAR i: INTEGER;
30  BEGIN
31    Tabelliere(f1, 0.0, 1.0, 4); Out.Ln;
32    Tabelliere(f2, 0.0, 1.0, 4); Out.Ln;
33    Tabelliere(f3, 0.0, 1.0, 4); Out.Ln; Out.Ln
34  END Start;
35
36  BEGIN
37  END FunktionsTabelle.
38
39  Beispielaufruf durch:
40  FunktionsTabelle.Start ~
41
42  Ausgabe:
43  1.0000000E+00  1.0625000E+00  1.2500000E+00  1.5625000E+00  2.0000000E+00
44  2.0000000E+00  2.0156250E+00  2.1250000E+00  2.4218750E+00  3.0000000E+00
45  0.0000000E+00  6.1850991E-02  2.3971277E-01  5.1122910E-01  8.4147096E-01

```

Numerische Integration nach der Simpsonschen Regel:

$$\int_a^b f(x) dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \dots + 2f_{2m-2} + 4f_{2m-1} + f_{2m})$$

mit $h = (b - a)/2m$ und $f_i = f(a + ih)$, wobei $0 \leq i \leq 2m$.

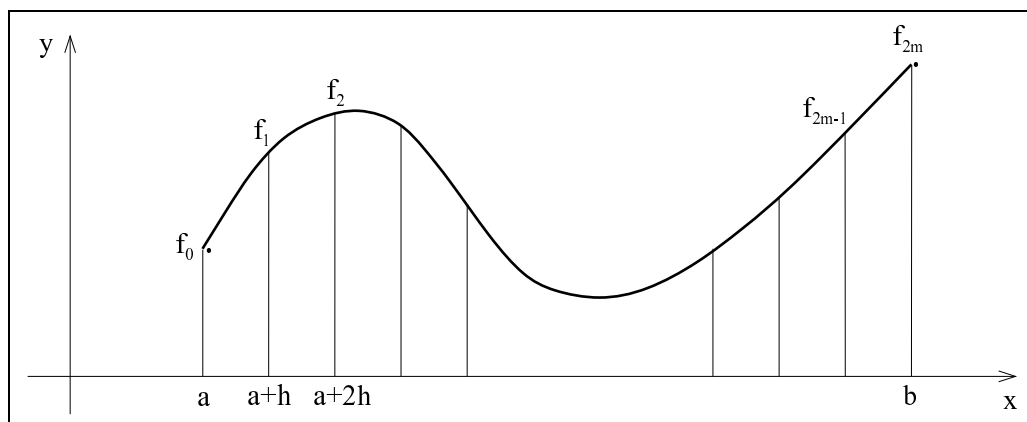


Abbildung 4.3: Simpsonsche Regel

(s. Abbildung 4.3)

Im folgenden Programm: diverse Beispiele und Möglichkeiten.

Zur Kontrolle: Es gilt

$$\frac{1}{\sqrt{\pi}} \int_0^\infty e^{-x^2} dx = \frac{1}{2}$$

4.2.8 PROGRAMMBEISPIEL (SIMPSONSCHE REGEL)

```

1
2  MODULE SimpsonRegel;

```

```
3  IMPORT Out, Math;
4
5  (* Alternative ueber TYPE-Deklaration: später! *)
6
7  PROCEDURE f1(x: REAL): REAL;
8  BEGIN RETURN 3*x*x
9  END f1;
10
11 PROCEDURE f2(x: REAL): REAL;
12   CONST pi = 3.1415;
13 BEGIN RETURN (1/Math.sqrt(pi))*Math.exp(-x*x)
14 END f2;
15
16 PROCEDURE f3(x: REAL): REAL;
17 BEGIN RETURN Math.sin(x)
18 END f3;
19
20 PROCEDURE S(x: REAL): REAL;
21 (* Beispiel Übungsaufgabe *)
22 BEGIN
23   RETURN Math.sin(x)*Math.sin(x)
24 END S;
25
26 PROCEDURE Integrate(f: PROCEDURE(x: REAL): REAL; a, b: REAL): REAL;
27   CONST m = 500;
28   VAR h, s: REAL; i: INTEGER;
29 BEGIN h := (b-a)/(2*m); s := f(a) + f(b);
30   FOR i := 1 TO 2*m - 1 DO
31     IF ODD(i) THEN s := s + 4*f(a+i*h)
32     ELSE s := s + 2*f(a+i*h)
33     END
34   END;
35   RETURN (h/3)*s
36 END Integrate;
37
38 PROCEDURE Start*;
39 BEGIN
40   Out.Real(Integrate(f2,0,20), 15); Out.Ln;
41   Out.Real(Integrate(S, 0, 3141.592654), 15); Out.Ln;
42   Out.String('richtig ist: '); Out.Real(1.0E3*Math.pi/2, 15); Out.Ln
43 END Start;
44
45 BEGIN
46 END SimpsonRegel.
47
48 Beispielaufruf durch:
49 SimpsonRegel.Start ~
```

```
50
51  Ausgabe:
52    5.0000685E-01
53    1.4537686E-05
54  richtig ist:  1.5707964E+03
55
56
```

Diese Formel ist exakt beweisbar.

Rekursion

frühere Beispiele:

- Fakultät
- Fibonacci-Folge
- Türme von Hanoi

Demobeispiel: Zählen von Wert

4.2.9 PROGRAMMBEISPIEL (REKURSION (1))

```
1
2  MODULE RekursionsDemo;
3    IMPORT Out;
4
5    VAR i: INTEGER;
6
7    PROCEDURE Rekursion;
8    BEGIN
9      DEC(i);
10     IF i >= 0 THEN
11       Out.Int(i, 4); Out.Ln;
12       Rekursion
13     END
14   END Rekursion;
15
16   PROCEDURE Start*;
17   BEGIN
18     i := 3; Rekursion; Out.Ln
19   END Start;
20
21 BEGIN
22 END RekursionsDemo.
23
24 Beispielaufruf durch:
25 RekursionsDemo.Start ~
26
27 Ausgabe:
```

```
28  2
29  1
30  0
31
32
```

4.2.10 PROGRAMMBEISPIEL (REKURSION (2))

```
1
2  MODULE RekursionsDemo;
3  IMPORT Out;
4  VAR i: INTEGER;
5  PROCEDURE Rekursion;
6  BEGIN
7    DEC(i);
8    IF i >= 0 THEN
9      Out.Int(i, 4); Out.Ln;
10     Rekursion;
11     Out.Int(i, 4); Out.Ln
12   END
13 END Rekursion;
14
15 PROCEDURE Start*;
16 BEGIN
17   i := 4; Rekursion; Out.Ln
18 END Start;
19
20 BEGIN
21 END RekursionsDemo.
22
23 Beispielaufruf durch:
24 RekursionsDemo.Start ~
25
26 Ausgabe:
27  3
28  2
29  1
30  0
31  -1
32  -1
33  -1
34  -1
35
36
```

Beispiel: Umkehrung einer Zeichenkette (String)

(s. Abbildung 4.4)

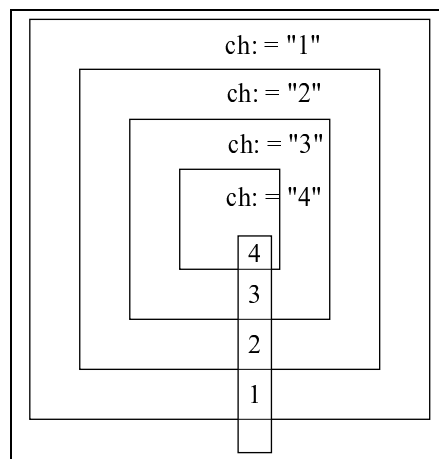


Abbildung 4.4: Umkehrung einer Zeichenkette

Die vordefinierten Prozeduren

Tabellen:

<u>Vordeclarierte Funktionsprozeduren</u>	<u>Vordeclarierte gewöhnliche Prozeduren</u>
ABS(x)	COPY(x,v)
ASH(x,n)	DEC(v)
CAP(x)	DEC(v,n)
CHR(x)	EXCL(v,x)
ENTIER(x)	HALT(x)
LEN(v,n)	INC(v)
LEN(v)	INC(v,n)
LONG(x)	INCL(v,x)
MAX(T)	NEW(v)
MIN(T)	NEW(v,x0,...,xn)
ODD(x)	
ORD(x)	
SHORT(x)	
SIZE(T)	

Hinweise zur Verwendung von Prozeduren

Nebeneffekte (Seiteneffekte, side-effects): wann sinnvoll, wann nicht? VAR-Parameter in Funktionen?

sinnvolles Beispiel für Verwendung von Nebeneffekt:

```
PROCEDURE Uniform(): REAL;
```

weiteres Beispiel:

4.2.11 PROGRAMMBEISPIEL (SEITENEFFEKTE (1))

```
1
2  MODULE Nebeneffekt; (* 2. Variante *)
```

```
3  IMPORT Out;
4
5  VAR z: INTEGER;
6
7  PROCEDURE Quadrat(a: INTEGER): INTEGER;
8  BEGIN INC(z); RETURN a*a
9  END Quadrat;
10
11 PROCEDURE Show*;
12   VAR x, y1, y2, y3: INTEGER;
13 BEGIN
14   z := 1; x := 2;
15   y1 := Quadrat(z) + Quadrat(x); Out.Int(y1, 5);
16   z := 1; x := 2;
17   y2 := Quadrat(x) + Quadrat(z); Out.Int(y2, 5);
18
19   z := 2;
20   y3 := z + Quadrat(z + Quadrat(z)); Out.Int(y3, 5);
21
22   Out.Ln
23 END Show;
24
25 BEGIN
26 END Nebeneffekt.
27
28 Beispielaufruf durch:
29   Nebeneffekt.Show ~
30
31 Ausgabe: (mit VAR-Parameter)
32   8      13
33
34 Ausgabe: (mit Wert-Parameter)
35   5      8      53
36
37
```

4.3 Strukturierte Datentypen

Strukturierte Datentypen sind analog zu strukturierten Anweisungen: sie werden gebildet durch Zusammensetzen anderer Datentypen

Strukturierte Datentypen in Oberon:

- Array-Typ (Array = Reihung)
- Record-Typ (Record = Satz)

Hinweis für Pascal-Programmierer: Teilbereichstypen und Aufzählungstypen gibt es in Oberon nicht.

Typdeklarationen

Datentyp: bezeichnet die Menge, aus der Werte einer Variablen dieses Typs entnommen sind, z.B.:

```
VAR i: INTEGER
```

bedeutet: $i \in \mathbb{N}$; genauer: $i \in [-32768 \dots + 32767]$

Datentypen können mit eigenen Namen bezeichnet werden, z.B.:

```
TYPE
  Zeit = REAL;
  Geburtsjahr = INTEGER;
  Vektor = ARRAY 3 OF REAL;
```

ein Vorteil: Abkürzung für lange und komplexe Namen; Exportmarke ist zulässig:

```
TYPE Zeit* = REAL;
```

Deklaration von Variablen dieses Typs:

```
VAR
  t1, t2, t3: Zeit;
  v1, v2 : Vektor;
```

Prozedurale Variable, mit TYPE formuliert (Umformulierung von früherem Beispiel):

```
TYPE RealFunction = PROCEDURE (x: REAL): REAL;
...
VAR f, g, h: RealFunction;
...
PROCEDURE f1(x: REAL): REAL;
BEGIN RETURN ...
END f1;
...
f := f1;
Tabelliere(f, 0.0, 1.0, 4);
```

Typ-Definitionen haben in Wirklichkeit eine wesentlich weitergehende Bedeutung (später). [RW92, S. 117]: *Namensäquivalenz* und *Strukturäquivalenz*:

Ein Ausdruck des Typs T_2 ist (zuweisungs-) kompatibel mit einem Bezeichner vom Typ T_1 , wenn eines der beiden folgenden gilt:

- entweder ist T_2 von T_1 eingeschlossen:
 $T_1 \supseteq T_2$
`VAR t:  $T_1$ ; expr:  $T_2$ ; (* mit  $T_1 \supseteq T_2$  *)`
`...`
`t := expr;`
- oder T_2 ist als gleich zu T_1 deklariert.
`TYPE  $T_2 = T_1$ ; (Namensäquivalenz)`

Array-Typ

Array: besteht aus einer Reihe von Größen (den *Elementen*) eines fest vorgegebenen Typs (des *Elementtyps*); das Array wird daher auch *Reihung* genannt.

jedes Element ist direkt zugänglich über einen Index, d.h. das Array ist eine sog. *random-access* Struktur.

Beispiele:

- `VAR a, b, c, d: ARRAY N OF REAL;`

Hier ist `REAL` der Elementtyp.

- mit Typ-Namen:

```
TYPE Vektor = ARRAY N OF REAL;
```

...

```
VAR a, b, c, d: Vektor;
```

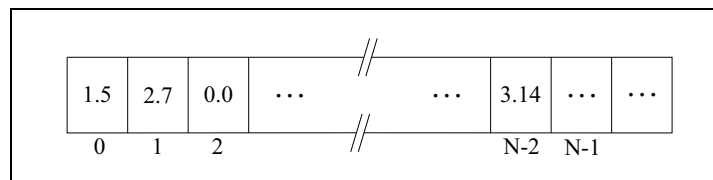


Abbildung 4.5: Darstellung der Struktur `ARRAY`

(s. Abbildung 4.5)

`N` ist Konstante, d.h., `N` ist durch statischen Programmtext festgelegt und somit zur Compilezeit bekannt

Zugriff auf die einzelnen Elemente:

```
a[0] := 10.0;
```

```
b[5] := 20.0;
```

```
c[i+j] := 30.0;
```

`a[0]`, `b[5]`, ... wird jeweils genauso wie jede andere Variable vom Typ `REAL` behandelt, z.B.:

```
FOR i := 0 TO N-1 DO a[i] := i*i; b[i] := 0.5*a[i] END;
```

Die Elemente eines Arrays können ebenfalls wieder Arrays sein: der Array-Typ heißt dann viel- (oder multi-)dimensional.

Beispiel:

```
VAR M: ARRAY 3 OF ARRAY 4 OF REAL;
```

Elementauswahl geschieht dann z.B. durch `y := M[2][3];`

Abkürzungen dafür und für Elementauswahl:

```
VAR M: ARRAY 3, 4 OF REAL;
```

...

...

...

```
y := M[2,3];
```

Die Variable M bildet eine Matrix mit den Komponenten $M[i, j]$:

Schema:

$M[0]$	$M[0,0]$	$M[0,1]$	$M[0,2]$	$M[0,3]$
$M[1]$	$M[1,0]$	$M[1,1]$	$M[1,2]$	$M[1,3]$
$M[2]$	$M[2,0]$	$M[2,1]$	$M[2,2]$	$M[2,3]$

Jede Komponente $M[i]$ ist selbst wieder ein Array.

Vordefinierte Funktionsprozedur $LEN(A, n)$:

A ist ein ARRAY, n die Dimension, beginnend mit 0. Resultat ist die Länge (Typ LONGINT) von A in der Dimension n .

z.B.:

$LEN(M, 0)$ ergibt 3, $LEN(M, 1)$ ergibt 4

Programmbeispiel: Gesucht ist eine Prozedur, durch welche die Matrixmultiplikation $\vec{v} = M\vec{u}$ implementiert wird (mit $\vec{u}, \vec{v}$ = Vektoren, M = Matrix).

In Komponentenschreibweise:

$$v_i = \sum_{j=0}^{n-1} M_{ij} u_j \quad 0 \leq i \leq n-1$$

Matrizen spielen eine Rolle in vielen Anwendungsbereichen, z.B. bei linearen Gleichungssystemen, in der Geometrie, der graphischen Datenverarbeitung, ...

4.3.1 PROGRAMMBEISPIEL (MATRIZENMULTIPLIKATION)

```

1
2  MODULE MatrixMalVektor;
3    IMPORT In, Out;
4    CONST n = 4;
5    TYPE
6      Vektor = ARRAY n OF REAL;
7      Matrix = ARRAY n OF Vektor;
8
9    PROCEDURE Print(VAR u: Vektor);
10     VAR i: INTEGER;
11   BEGIN
12     FOR i := 0 TO n-1 DO Out.Real(u[i], 12) END
13   END Print;
14
15   PROCEDURE Multiply(VAR M: Matrix; VAR u, v: Vektor);
16   (* Die Prozedur Multiply bildet das Matrixprodukt  $\mathbf{v} := M\mathbf{u}$ 
17      und schreibt das Resultat nach v. *)
18     VAR i, j: INTEGER; s: REAL;
19   BEGIN
20     FOR i := 0 TO n-1 DO
21       s := 0;      (* Frage: warum nicht s[i] ? *)
22       FOR j := 0 TO n-1 DO s := s + M[i, j] * u[j] END;
23       v[i] := s
24     END
25   END Multiply;
```

```

26
27  PROCEDURE Start*;
28    VAR i, j: INTEGER; u, v: Vektor; M: Matrix;
29  BEGIN
30    FOR i := 0 TO n-1 DO
31      u[i] := i + 1;
32      FOR j := 0 TO n-1 DO M[i, j] := 2*i - j + 1 END
33    END;
34    Multiply(M, u, v); Print(v); Out.Ln
35  END Start;
36
37  BEGIN
38  END MatrixMalVektor.
39
40  Beispielaufruf durch:
41  MatrixMalVektor.Start ~
42
43  Ausgabe:
44  -1.0000E+01  1.0000E+01  3.0000E+01  5.0000E+01
45
46

```

Beispielwerte aus Programm:

$$\begin{pmatrix} 1 & 0 & -1 & -2 \\ 3 & 2 & 1 & 0 \\ 5 & 4 & 3 & 2 \\ 7 & 6 & 5 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} -10 \\ 10 \\ 30 \\ 50 \end{pmatrix}$$

Erläuterungen:

- Nur v muß ein Var-Parameter sein.
- Wertparameter empfehlen sich für u, M, falls Sicherheit wichtig (keine unbeabsichtigte Änderung von u, M)
- Var-Parameter empfehlen sich aus Effizienzgründen (kein Kopieren von Werten). In aller Regel wird diese letztere Möglichkeit bevorzugt.

Problem bei Parametern vom Array-Typ: Größe ist fest vorgegeben, daher: Prozedur, die z.B. mit 3×3 -Matrizen richtig arbeitet, könnte nicht auch mit 4×4 -Matrizen arbeiten; Lösung des Problems sind sog. *Offene Array-Parameter*:
z.B.: Berechnung der Norm (Länge) eines Vektors $\mathbf{v} = (v_0, v_1, v_2, \dots, v_{n-1})$:

$$||v|| = \sqrt{\sum_{i=0}^{n-1} v_i^2}$$

Prozedurkopf mit offenem Array-Parameter:

```
PROCEDURE Norm(VAR v: ARRAY OF REAL);
```

Offene Array-Parameter sind formale Parameter, die mit jedem aktuellen Array-Parameter gleicher Dimensionalität und mit demselben Elementtyp kompatibel sind.

Beispiel im Detail:

4.3.2 PROGRAMMBEISPIEL (NORM VON VEKTOREN)

```

1
2  MODULE VektorNorm;
3    IMPORT Out, Math;
4
5    PROCEDURE Norm(VAR v: ARRAY OF REAL): REAL;
6      VAR i: INTEGER; s: REAL;
7      BEGIN s := 0;
8        FOR i := 0 TO SHORT(LEN(v)) - 1 DO s := s + v[i]*v[i] END;
9        RETURN Math.sqrt(s)
10   END Norm;
11   (* SHORT() nur wegen der Dekl. von i als INTEGER *)
12
13   PROCEDURE Norm2(VAR v: ARRAY OF REAL): REAL;
14     VAR i: LONGINT; s: REAL;
15     BEGIN s := 0;
16       FOR i := 0 TO LEN(v) - 1 DO s := s + v[i]*v[i] END;
17       RETURN Math.sqrt(s)
18   END Norm2;
19
20   PROCEDURE Berechne*;
21     CONST N = 4;
22     VAR v: ARRAY N OF REAL; i: INTEGER;
23     BEGIN
24       FOR i := 0 TO N-1 DO v[i] := i END;  (* als Beispiel *)
25       Out.Real(Norm(v), 14); Out.Ln
26   END Berechne;
27
28   END VektorNorm.
29
30   Beispielaufruf durch:
31   VektorNorm.Berechne ~
32
33   Ausgabe:
34   3.741657E+00
35
36
```

Beispiel zu offenen Array-Parametern: Matrix mal Vektor

4.3.3 PROGRAMMBEISPIEL (MATRIZENMULTIPLIKATION: OFFENE ARRAY-PARAMETER)

```

1
2  MODULE MatrixMalVektor;
3    (* 2. Variante mit offenen Array-Parametern *)

```

```

4  IMPORT In, Out;
5  CONST n = 4;
6  TYPE
7    Vektor = ARRAY n OF REAL;
8    Matrix = ARRAY n OF Vektor;
9
10 PROCEDURE Print(VAR u: ARRAY OF REAL);
11   VAR i: INTEGER;
12 BEGIN
13   FOR i := 0 TO SHORT(LEN(u)) - 1 DO Out.Real(u[i], 12) END
14 END Print;
15
16 (* Folgende Prozedur nur sinnvoll, wenn man von vornherein weiß,
17    daß Matrix und Vektor zueinander „passen“.*)
18 PROCEDURE Multiply(VAR M: ARRAY OF ARRAY OF REAL; VAR u, v: ARRAY OF REAL);
19 (* Die Prozedur Multiply bildet das Matrixprodukt  $v := Mu$ 
20    und schreibt das Resultat nach  $v$ . *)
21   VAR i, j: INTEGER; s: REAL;
22 BEGIN
23   FOR i := 0 TO SHORT(LEN(M, 0)) - 1 DO
24     s := 0;
25     FOR j := 0 TO SHORT(LEN(M, 1)) - 1 DO s := s + M[i, j] * u[j] END;
26     v[i] := s
27   END
28 END Multiply;
29
30 PROCEDURE Start*;
31   VAR i, j: INTEGER; u, v: Vektor; M: Matrix;
32 BEGIN
33   FOR i := 0 TO n - 1 DO
34     u[i] := i + 1;
35     FOR j := 0 TO n - 1 DO M[i, j] := 2 * i - j + 1 END
36   END;
37   Multiply(M, u, v); Print(v); Out.Ln
38 END Start;
39
40 END MatrixMalVektor.
41
42 Beispielaufruf durch:
43 MatrixMalVektor.Start ~
44
45 Ausgabe:
46 -1.0000E+01  1.0000E+01  3.0000E+01  5.0000E+01
47
48

```

Beispiel allgemeine Matrixmultiplikation:

$$C = AB$$

Wenn nicht von vornherein klar ist, daß die Matrizen zusammenpassen, dann sollte folgende Formulierung gewählt werden:

```
PROCEDURE Multiply(VAR A, B, C: ARRAY OF ARRAY OF REAL; l, m, n: INTEGER);
```

Offene Arrays nur entweder

1. als formale Array-Parameter oder
2. als sog. Zeigerbasistyp (in Oberon-2):

Bedeutung von Zeigern (Pointer): im Detail später

In Oberon-2 gibt es auch die Möglichkeit der Deklaration von dynamischen Arrays. Verwendet werden hier sog. Zeiger (Pointer); dieses ist aber noch nicht die typische Verwendung von Zeigern zur Erzeugung dynamischer Datenstrukturen.

4.3.4 PROGRAMMBEISPIEL (OFFENE ARRAYS)

```

1
2  MODULE OpenArrays;
3    IMPORT In, Out;
4    TYPE
5      IntArray = ARRAY OF INTEGER;
6      IntArray2 = ARRAY OF IntArray;
7      (* technisch genau: Offenes Array als Zeigerbasistyp *)
8
9    PROCEDURE Test*;
10     VAR
11       p: POINTER TO IntArray; q: POINTER TO IntArray2;
12       i, n: INTEGER;
13     BEGIN
14       (* Umgang mit eindimensionalem dyn. Array: *)
15       (* Größe des Arrays wird erst zur Laufzeit festgelegt:*)
16       In.Open; In.Int(n);
17       (* vordef. Prozedur zur Erzeugung (Allokation) von Speicherplatz *)
18       NEW(p, n);
19       FOR i := 0 TO n-1 DO p↑[i] := i*i END;
20       FOR i := 0 TO n-1 DO Out.Int(p↑[i], 9); Out.Ln END;
21       Out.Ln;
22
23       (* Umgang mit zweidimensionalem Array: *)
24       NEW(q, 5, 7); q↑[3, 6] := 123; Out.Int(q↑[3, 6], 9); Out.Ln;
25       Out.Ln; Out.Ln
26     END Test;
27
28   END OpenArrays.
29
```

```

30  Beispielaufruf durch:
31  OpenArrays.Test 5 ~
32
33  Ausgabe:
34
35      0
36      1
37      4
38      9
39     16
40
41     123
42
43

```

Zeichenketten-Typen

Zeichenkette (String) ist eine Aneinanderreihung von Elementen des Typs `CHAR`
z.B.

```

CONST
    m = 20; n = 100;
TYPE
    Text = ARRAY m OF CHAR;
    NameList = ARRAY n OF Text;
VAR
    txt1, txt2: Text;
    nl1, nl2: NameList;

```

alle üblichen `ARRAY`-Operationen sind zulässig; darüber hinaus gilt:
Compiler behandelt `ARRAY n OF CHAR` in spezieller Weise
Besonderheiten (die z.B. bei `ARRAY n OF INTEGER` nicht vorhanden sind):

- Zuweisung der Art `txt1 := "Ein kurzer Text"` ist erlaubt.
- Vergleiche sind zulässig: z.B.:

```

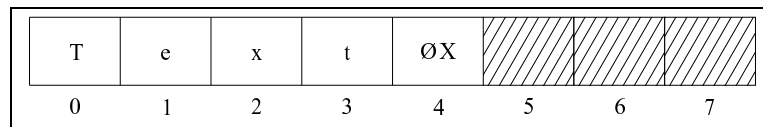
– txt1 < txt2
– txt1 > "Text"

```

Vergleichskriterium ist die normale lexikographische Anordnung und die Anordnung im ASCII-Code, z.B. haben folgende Boolesche Ausdrücke den Wert `TRUE`:
`"a" < "b"`, `"A" < "a"`, `"1" < "2"`, `"1" < "a"`

- Zeichenkette wird abgeschlossen durch das spezielle Zeichen `0X`, welches kleiner als alle anderen Zeichen ist.

interne Darstellung einer Zeichenkette:

Abbildung 4.6: Zeichenkette in `ARRAY n OF CHAR`

```

VAR s: ARRAY 8 OF CHAR;
...
s := "Text";

```

(s. Abbildung 4.6)

Länge einer Zeichenkette und Länge des `ARRAY`s: sind genau zu unterscheiden!
 Modul mit einer Reihe von Prozeduren zur Stringbearbeitung, z.B.:

- Suche nach einer Zeichenkette in einer Tabelle
- Suche nach einem Textmuster
- Einfügen in einen Text

4.3.5 PROGRAMMBEISPIEL (STRINGS)

```

1  MODULE Str;
2  (* MODULE Str: Modul zur Manipulation von Strings:
3     dazu: vgl. TopSpeed Modula, User's Manual, p. 347 und
4     Reiser/Wirth: Oberon, p. 122 *)
5
6  PROCEDURE Len*(x: ARRAY OF CHAR): INTEGER;
7     VAR j: INTEGER;
8  BEGIN
9     j := 0;
10    WHILE x[j] > 0X DO INC(j) END;
11    RETURN j
12  END Len;
13
14  PROCEDURE Locate*(VAR txt: ARRAY OF CHAR; x: ARRAY OF CHAR;
15                    VAR pos: INTEGER);
16    VAR j, Lx, Lt: INTEGER;
17  BEGIN
18    Lx := Len(x); Lt := Len(txt); pos := -1;
19    REPEAT
20      j := 0;
21      INC(pos);
22      WHILE (x[j] = txt[pos + j]) & (j < Lx) DO INC(j) END
23    UNTIL (j=Lx) OR ((pos + Lx) > Lt);
24    IF j < Lx THEN pos := -1 (* pattern not found *) END
25  END Locate;
26
27  (* Predeclared Function COPY: *)
28  PROCEDURE COPY*(s: ARRAY OF CHAR; VAR x: ARRAY OF CHAR);

```

```

29   VAR j: INTEGER;
30   BEGIN
31     j := 0;
32     WHILE (s[j] # 0X) & (j < LEN(x) - 1) DO x[j] := s[j]; INC(j) END;
33     x[j] := 0X
34   END COPY;
35
36 END Str.
37
38
39 MODULE StringTest;
40   IMPORT Str, Out;
41
42   PROCEDURE Start*;
43     CONST N = 10;
44     VAR s1, s2, s3: ARRAY N OF CHAR; pos: INTEGER;
45   BEGIN
46     s1 := "Ein Test"; Out.Int(Str.Len(s1), 4); Out.Ln;
47     s2 := "n T"; Str.Locate(s1, s2, pos); Out.Int(pos, 3); Out.Ln;
48     Str.COPY("abcdefghijklmno", s2); Out.String(s2); Out.Ln; Out.Ln;
49   END Start;
50
51 BEGIN END StringTest.
52
53 System.Free StringTest Str ~
54
55 Beispielaufruf durch:
56 StringTest.Start ~
57
58 Ausgabe:
59     8
60     2
61 abcdefghi

```

4.3.6 PROGRAMMBEISPIEL (STRINGS)

```

1
2  MODULE StringTest;
3    IMPORT Out;
4    CONST N = 6;
5
6    TYPE
7      PString = POINTER TO ARRAY OF CHAR;
8      String = ARRAY N OF CHAR;
9
10   PROCEDURE Test*;
11     VAR
12       p: PString; i, n: INTEGER; s: String;
13   BEGIN n := N; (* eigentlich: einlesen, hier nur für Vergleich *)

```

```

14    NEW(p, n);
15    FOR i := 0 TO n-2 DO p↑[i] := CHR(ORD('A')+i) END; p↑[n-1] := 0X;
16    Out.String(p↑); Out.Ln;
17    s := "12345"; (* 5 noch erlaubt, 6 nicht mehr, s.o.: Deklaration *)
18    Out.String(s); Out.Ln;
19    (* p↑ := "sdfgh"; *) (* Zuweisung wird nicht erlaubt *)
20
21    (* p↑ := s;
22       s := p↑; beide Zuweisungen nicht erlaubt: incompatible assignment *)
23
24    Out.Ln; Out.Ln
25    END Test;
26
27    BEGIN
28    END StringTest.
29
30    Beispielaufruf durch:
31    StringTest.Test ~
32
33    Ausgabe:
34    ABCDE
35    12345
36
37

```

Unterschiede $x := s$; und $COPY(s, x)$; diskutieren!

Diskussion: Nachteile von ARRAYS ...

Arten der Kompatibilität

(aus Mössenböck, (D) S. 254, (E) p. 13:)

Wichtige Begriffe:

- zuweisungskompatibel
- arraykompatibel
- ausdruckskompatibel

Ein Ausdruck e vom Typ T_e ist **zuweisungskompatibel** mit einer Variablen v vom Typ T_v , wenn eine der folgenden Bedingungen gilt:

1. T_e und T_v sind *dieselben* Typen.
2. T_e und T_v sind numerische Typen, und T_v *schließt* T_e *ein*.
3. T_e und T_v sind Recordtypen, T_e ist eine Erweiterung von T_v und der dynamische Typ von v ist T_v .
4. T_e und T_v sind Zeigertypen, und T_e ist eine Erweiterung von T_v .
5. T_v ist ein Zeigertyp oder ein Prozedurtyp und e ist NIL.

6. T_v ist `ARRAY  $n$  OF CHAR`, e ist eine Zeichenkettenkonstante der Länge m und $m < n$.
7. T_v ist ein Prozedurtyp und e ist der Name einer Prozedur, deren formale Parameterliste mit der von T_v *übereinstimmt*.

Ein aktueller Parameter vom Typ T_a ist mit einem formalen Parameter f vom Typ T_f **arraykompatibel**, wenn eine der folgenden Bedingungen gilt:

1. T_f und T_a sind *dieselben* Typen
2. T_f ist ein offenes Array, T_a ist ein beliebiges Array, und ihre Elementtypen sind *arraykompatibel*.
3. T_f ist `ARRAY OF CHAR` und a ist eine Zeichenkettenkonstante

ausdruckskompatibel (Tabelle)

zu erklären sind also die Begriffe:

- derselbe Typ, gleiche Typen
- Einschließen von Typen
- Formale Parameterlisten stimmen überein

Zwei Variablen a und b mit den Typen T_a und T_b haben *denselben Typ*, wenn eine der folgenden Aussagen gilt:

1. T_a und T_b werden durch den gleichen Typpnamen bezeichnet, z.B.
`VAR  $a$ : INTEGER;  $b$ : INTEGER;`
2. T_a ist deklariert als $T_a = T_b$, z.B.
`TYPE Zeit = REAL;`
3. a und b treten in der gleichen Namenliste einer Variablen-, Parameter- oder Rekordfeld-Deklaration auf und sind keine offenen Arrays, z.B.
`VAR  $a$ ,  $b$ : LONGINT;`

Zwei Typen T_a und T_b sind gleich, wenn eine der folgenden Aussagen gilt:

1. T_a und T_b sind derselbe Typ.
2. T_a und T_b sind offene Arraytypen, deren Elementtypen gleich sind;
3. T_a und T_b sind Prozedurtypen, deren formale Parameterlisten übereinstimmen

Numerische Typen schließen die Werte kleinerer numerischer Typen gemäß folgender Hierarchie ein: `LONGREAL` $\supseteq$ `REAL` $\supseteq$ `LONGINT` $\supseteq$ `INTEGER` $\supseteq$ `SHORTINT`

zu Typerweiterung und Basistyp:

In einer Typdeklaration

$T_b = \text{RECORD } (T_a)$

...
END

ist T_b eine direkte Erweiterung von T_a , und T_a ist der direkte Basistyp von T_b . Allgemein ist ein Typ T_b eine Erweiterung eines Typs T_a (T_a ist ein Basistyp von T_b), wenn eine der folgenden Aussagen gilt:

1. T_a und T_b sind dieselben Typen.
2. T_b ist eine direkte Erweiterung einer Erweiterung von T_a .

Wenn $P_a = \text{POINTER TO } T_a$ und $P_b = \text{POINTER TO } T_b$ gilt, wobei T_b eine Erweiterung von T_a ist, so ist auch P_b eine Erweiterung von P_a , und P_a ist der Basistyp von P_b .

Zwei formale Parameterlisten stimmen überein (match), falls sämtliche der folgenden Bedingungen zutreffen:

1. Sie enthalten dieselbe Anzahl Parameter.
2. Sie haben denselben Funktionsergebnistyp oder keinen.
3. Parameter an sich entsprechenden Positionen haben gleiche Typen.
4. Parameter an sich entsprechenden Positionen sind entweder beide Wert- oder beide Variablenparameter.

Beispiel zu Kompatibilität:

4.3.7 PROGRAMMBEISPIEL (KOMPATIBILITÄT)

```

1
2  MODULE Test4;
3    IMPORT Out;
4    (* Test von Zuweisungskompatibilität (Punkt 7 der Liste) *)
5
6    TYPE Tv = PROCEDURE (x: REAL): REAL;
7
8    (*
9      * Tv = PROCEDURE (x: REAL): INTEGER;
10     * führt zur Fehlermeldung: result type doesn't match
11     *)
12
13    VAR v: Tv;
14
15    PROCEDURE e(x: REAL): REAL;
16    BEGIN RETURN 7.0 END e;
17
18    (*
19     * PROCEDURE e(x: INTEGER): REAL;
20     * BEGIN RETURN 7.0 END e;
21     * führt zur Fehlermeldung: parameter doesn't match
22     *)
23
24    (* und entsprechend umgekehrt: hier REAL und oben INTEGER:
25     parameter doesn't match *)
```

```

26  (*
27      PROCEDURE e(x: REAL): REAL;
28      BEGIN RETURN 7.0 END e; *)
29  (* führt zur Fehlermeldung: parameter doesn't match *)
30
31  PROCEDURE Start*;
32  BEGIN v := e; Out.Real(v(2), 12); Out.Ln
33  END Start;
34
35  BEGIN
36  END Test4.
37
38  Test4.Start ~
39
40

```

Recordtyp

Das Record als heterogene Datenstruktur

- bisher: **Array**: besteht aus Elementen eines *gemeinsamen Typs*; die Anzahl ist fest vorgegeben. Das Array ist eine *homogene Datenstruktur*
- jetzt: **Record** (Verbund): besteht aus Elementen, die i.a. *verschiedenen Typ* haben (gleicher Typ ist aber nicht ausgeschlossen); Anzahl ebenfalls fest vorgegeben. Das Record ist eine *heterogene Struktur*.

Vergleichende Übersicht über die Strukturarten Array und Record:

	Array	Record
Typ der Elemente:	untereinander gleich	i.a. voneinander verschieden
Anzahl der Elemente:	fest	fest
Auswahl von Element:	durch (berechneten) Index	durch Feldnamen
typische Anwendung:	Vektor, Matrix, Tabelle	Datensatz Kontrollblock (in Betriebssystem)

Beispiel (Elemente verschiedenen Typs):

```

CONST n = 30;
TYPE
    Student = RECORD
        Name, Vorname: ARRAY n OF CHAR;
        MatNr: LONGINT
    END;
...
VAR s1, s2, s3: Student;

```

alternativ:

```

TYPE
    String30 = ARRAY 30 OF CHAR;
    Student = RECORD

```

```

    Name, Vorname: String30;
    MatNr: LONGINT
END;
...
VAR s1, s2, s3: Student;

```

(s. Abbildung 4.7)

Zugriff auf Record-Felder, z.B.: `s1.Name := "Meier"`; oder auch: `s2.MatNr := 8512399`;

Beispiel (Elemente gleichen Typs):

komplexe Zahlen als neuer (selbstdefinierter) Typ:

```

TYPE
    COMPLEX = RECORD
        Re, Im: REAL
    END;
...
VAR u, v, w: COMPLEX;

```

Zugriff auf Record-Elemente: `w.Re := 5`; `w.Im := 3`;

entspricht in Mathematik: $w = 5 + 3i$

Komplexe Zahlen als Records (und nicht als 2-elementige Arrays) hat deswegen u. U. einen Sinn, weil man damit näher an der mathematischen Schreibweise $\Re w$, $\Im w$ bleibt. Begriffsdefinitionen:

- `Student`, `COMPLEX`: Typ-Bezeichner (type identifier)
- `Name`, `Vorname`, `MatNr`, `Re`, `Im`: Feld-Bezeichner (field identifier);
Anmerkung: Daher sollte das Array nicht *Feld* genannt werden!
- `s1`, `s2`, `s3`, `u`, `v`, `w`: Variable des jeweiligen Typs (`Student`, `COMPLEX`)

Anmerkung: In Oberon ist für Record-Variablen (z.B. komplexe Zahlen) keine der Schreibweisen `x := x1 + x2` oder `x := Add(x1, x2)` zulässig.

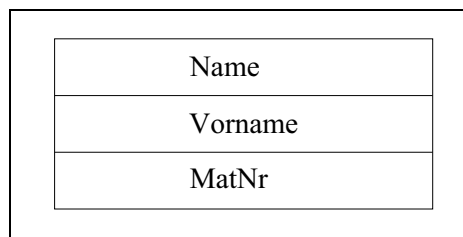


Abbildung 4.7: Darstellung von Records

Programmbeispiele zum Umgang mit Records

4.3.8 PROGRAMMBEISPIEL (KOMPLEXE ZAHLEN)

```

1
2  MODULE KomplexeZahlen;
3  IMPORT Out, Math;

```

```
4  TYPE
5    COMPLEX = RECORD
6      Re, Im: REAL
7    END;
8
9  PROCEDURE Assign(VAR u: COMPLEX; r, i: REAL);
10 BEGIN
11   u.Re := r; u.Im := i
12 END Assign;
13
14 PROCEDURE Norm(u: COMPLEX): REAL;
15 BEGIN
16   RETURN Math.sqrt(u.Re*u.Re + u.Im*u.Im)
17 END Norm;
18
19 PROCEDURE Add(u, v: COMPLEX; VAR w: COMPLEX);
20 BEGIN
21   w.Re := u.Re + v.Re;
22   w.Im := u.Im + v.Im
23 END Add;
24
25 PROCEDURE Multiply(u, v: COMPLEX; VAR w: COMPLEX);
26 BEGIN
27   w.Re := u.Re*v.Re - u.Im*v.Im;
28   w.Im := u.Im*v.Re + u.Re*v.Im
29 END Multiply;
30
31 PROCEDURE PowerMinOne(u: COMPLEX; VAR w: COMPLEX);
32   VAR n2: REAL;
33 BEGIN
34   n2 := Norm(u); n2 := n2*n2;
35   w.Re := u.Re/n2;
36   w.Im := -u.Im/n2;
37 END PowerMinOne;
38
39 PROCEDURE Divide(u, v: COMPLEX; VAR w: COMPLEX);
40 BEGIN
41   PowerMinOne(v, w); Multiply(u, w, w)
42 END Divide;
43
44 PROCEDURE Print(u: COMPLEX; f: INTEGER);
45   (* Sollte weiter ausgearbeitet werden *)
46 BEGIN
47   Out.Real(u.Re, f); Out.String(" + i * "); Out.Real(u.Im, f)
48 END Print;
49
50 PROCEDURE Start*;
```

```

51     CONST f = 12;
52     VAR u, v, w: COMPLEX;
53     BEGIN
54         Assign(u, 1.0, 3.0); Assign(v, 2.0, 5.0); Print(u, f); Out.Ln; Print(v, f); Out.Ln;
55         Out.Real(Norm(u), f); Out.Real(Norm(v), f); Out.Ln;
56         Add(u, v, w); Print(w, f); Out.Ln;
57         Multiply(u, v, w); Print(w, f); Out.Ln;
58         Divide(u, v, w); Print(w, f);
59         Out.Ln; Out.Ln;
60     END Start;
61
62     BEGIN
63     END ComplexeZahlen.
64
65     Beispielaufruf durch:
66     ComplexeZahlen.Start ~
67
68     Ausgabe:
69     1.0000E+00 + i * 3.0000E+00
70     2.0000E+00 + i * 5.0000E+00
71     3.1623E+00 5.3852E+00
72     3.0000E+00 + i * 8.0000E+00
73     -1.3000E+01 + i * 1.1000E+01
74     5.8621E-01 + i * 3.4483E-02
75
76     (*
77     Komplexe Zahlen als Records: sinnvoll nur wegen der zur
78     Mathematik ähnlichen Schreibweise Re z, Im z.
79     Sonst wäre Array ebenso möglich und sinnvoll!
80     *)
81
82
83

```

Weiteres zu Records

Vorteile der Verwendung von Variablen vom Recordtyp (ebenso wie vom Arraytyp):
 ganze Variablen werden angesprochen (komplexe Zahl, Person, Matrix, Vektor, ...)
 z.B.:

```

TYPE
    COMPLEX = RECORD
        R, I: REAL
    END;
...
VAR z1, z2, z3;
...
z1 := z2; z3 := z1; Print(z3);

```

...

Zuweisung ist *Kopiervorgang*: Real- und Imaginärteile werden jeweils einzeln kopiert.

Sichtbarkeit von Feldern: nur innerhalb von Record, außerhalb: z.B.: `z1.Re`, `z1.Im`

Array und Record dürfen gemischt und geschachtelt werden:

Beispiele:

Student enthält `ARRAY n OF CHAR`; als Typ eines Feldes.

entsprechend:

ein Array kann Elemente enthalten, die selbst vom Recordtyp sind.

4.3.9 PROGRAMMBEISPIEL (RECORDS)

```

1
2  MODULE PersonenDB;
3  (* Simple Rudiment einer Personaldatenbank, allerdings ohne
4     systematische Zugriffsprozeduren *)
5  IMPORT In, Out;
6  CONST I = 30; N = 100;
7  TYPE
8    String = ARRAY I OF CHAR;
9    Person = RECORD
10      Name, Vorname: String;
11    END;
12  DB = ARRAY N OF Person; (* Datenbank, database *)
13
14  VAR d: DB; (* Modul-globale Variable:
15      Modul stellt sozusagen die Datenbank dar *)
16      n: INTEGER; (* aktuelle Anzahl Einträge *)
17
18  PROCEDURE Init*;
19  (* alle Einträge löschen *)
20    VAR i: INTEGER;
21  BEGIN
22    FOR i := 0 TO N-1 DO d[i].Name := ""; d[i].Vorname := "" END
23  END Init;
24
25  PROCEDURE ShowDB*;
26    VAR i: INTEGER;
27  BEGIN
28    FOR i := 0 TO n-1 DO
29      Out.String(d[i].Name); Out.String(", ");
30      Out.String(d[i].Vorname); Out.Ln
31    END;
32    Out.String("Anzahl Eintraege: "); Out.Int(n, 5);
33    Out.Ln; Out.Ln
34  END ShowDB;
35
36  PROCEDURE Test*;
37    VAR Name, Vorname: String; i: INTEGER;
```

```

38   BEGIN In.Open; i := 0;
39   In.String(Name); In.String(Vorname);
40   WHILE In.Done DO
41     d[i].Name := Name; d[i].Vorname := Vorname;
42     In.String(Name); In.String(Vorname);
43     INC(i);
44   END;
45   n := i (* evtl. auch n direkt verwenden *)
46   END Test;
47
48   END PersonenDB.
49
50   Beispielaufruf durch:
51   PersonenDB.Init ~
52
53   PersonenDB.Test
54     "Meyer" "Hugo"
55     "Schmitt" "Peter"
56     "Mayr" "Kuno" ~
57
58   PersonenDB.ShowDB ~
59
60   Ausgabe:
61     Meyer, Hugo
62     Schmitt, Peter
63     Mayr, Kuno
64     Anzahl Eintraege:    3
65
66

```

auch Records dürfen (in beliebiger Tiefe) geschachtelt sein:
Beispiel (Datum):

4.3.10 PROGRAMMBEISPIEL (RECORDS)

```

1
2   MODULE RecordBeispiel3;
3   IMPORT Out;
4
5   CONST n = 20;
6   TYPE
7     Datum = RECORD
8       Tag, Monat, Jahr: INTEGER
9     END;
10    Person = RECORD
11      Name, Vorname: ARRAY n OF CHAR;
12      GebDatum: Datum
13    END;
14    (* in Oberon gibt es weder einen Aufzählungstyp noch einen Teilbereichstyp *)

```

```

15   Firma = RECORD
16     Name: ARRAY n OF CHAR;
17     Chef: Person;
18     Umsatz: INTEGER
19   END;
20
21   VAR F1, F2, F3: Firma; p1, p2: Person; d: Datum;
22
23   PROCEDURE Start*;
24   BEGIN
25     (* zulässige Zuweisungen sind dann: *)
26     d.Tag := 20;
27     d.Monat := 12 ;   (* aber auch: *) d.Monat := 13;
28     d.Jahr := 1970;
29
30     p1.Vorname := "Hugo";
31     p1.GebDatum := d;
32
33     p2.GebDatum.Jahr := 1980;
34     p2.GebDatum.Tag := 32;
35
36     F1.Chef := p1;
37     F1.Chef.GebDatum.Jahr := 1899;
38
39     Out.Int(p2.GebDatum.Jahr, 12); Out.Ln
40   END Start;
41
42   BEGIN
43   END RecordBeispiel3.
44   Beispielaufruf durch:
45   RecordBeispiel3.Start ~
46
47   Ausgabe:
48       1980
49
50

```

eine direkte Rekursion ist nicht zulässig:
z.B. ist *nicht* möglich:

4.3.11 PROGRAMMBEISPIEL (RECORDS)

```

1   MODULE RecordRekursion;
2   (* nicht möglich ist: *)
3
4   TYPE
5     R = RECORD
6       i: INTEGER;

```

```

8  (* r: R *)
9  END;
10
11 (* Fehlermeldung: recursive type definition *)
12
13 BEGIN
14 END RecordRekursion.
15

```

Records werden zum Aufbau rekursiver Datenstrukturen verwendet. Das geschieht jedoch mit anderen technischen Hilfsmitteln.

Recordvariablen als Rückgabewerte (über **RETURN**) von Funktionsprozeduren? geht nicht! (bereits behandelt)

erwähnen: Variablen vom Prozedurtyp als Recordfelder ...

4.4 Konkrete Datenstrukturen

Datenstrukturen in bisherigen Beispielen: werden als *konkrete Datenstrukturen* bezeichnet, der beschriebene Zugriff durch Prozeduren als *prozedurale Abstraktion*.

Beides gehört der Ideenwelt von Pascal an.

Merkmale konkreter Datenstrukturen

- innerhalb eines Moduls: Datenstruktur ist offen, d.h., Details sind für den Programmierer (d.h. für andere Programmteile) sichtbar

z.B. ist Zugriff auf Felder in **Student** möglich durch **S[3].Name[5] := "d";**

Ein solcher Zugriff ist für die meisten Anwendungen sicherlich sinnlos, ebenso die Addition von **MatNr** oder z.B.

Student.MatNr := Student.MatNr DIV 2;

(oder sonstige entsprechende Rechenoperationen)

Es gibt keine technische Sicherung gegen solche versehentliche Datenzugriffe und -manipulationen.

- Zugriff auf die Datenstrukturen erfolgt entweder direkt oder über Prozeduren (prozedurale Abstraktion)

Was geschieht bei einer Änderung der Implementierung?

z.B.: **COMPLEX** als Array:

```

TYPE
  COMPLEX = ARRAY 2 OF REAL;
....
VAR z1, z2, z3;
....
z1 := z2; ...

```

Zugriff auf Real- und Imaginärteil lautet dann (z.B.):

```
Out.Real(z1[0], 12); Out.Real(z1[1], 12);
```

Programmcodes, der direkte Zugriffe dieser Art enthält, muß nach einer Änderung der Datenstruktur angepaßt werden, ebenso Prozeduren;

Ein Zugriff durch Prozeduren schafft eine Abstraktion, die eine Anpassung *außerhalb* dieser Prozeduren nicht erforderlich macht.

Beispiele

komplexe Zahlen: sehr einfach und durchsichtig; weniger durchsichtiges Beispiel: Prioritätswarteschlange

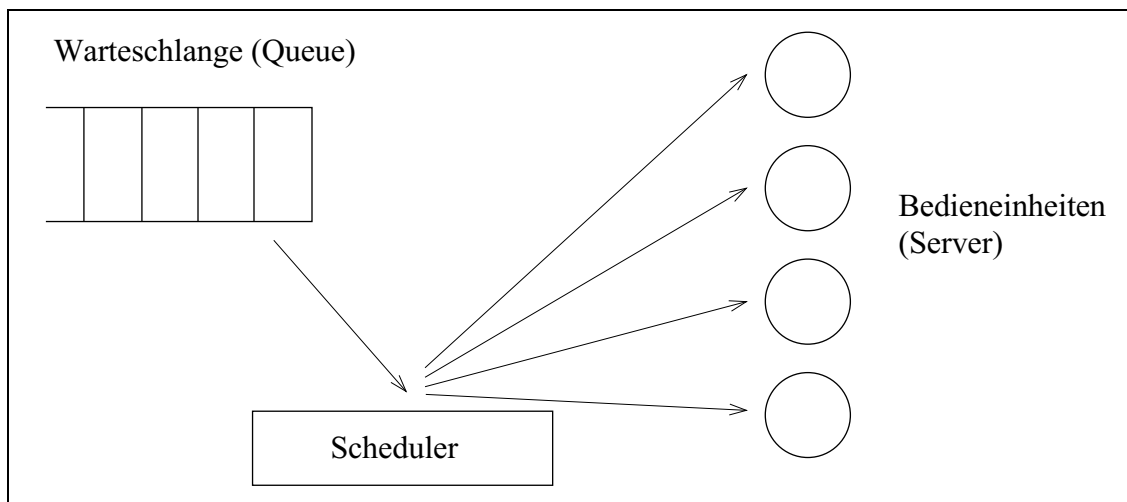


Abbildung 4.8: Prioritätswarteschlange

(s. Abbildung 4.8)

Eine erste Modul-Variante zur Prioritätswarteschlange (naive Form, nicht effizient):

4.4.1 PROGRAMMBEISPIEL (PRIORITÄTSWARTESCHLANGE – NAIVE FORM)

```

1  MODULE PriorityQueue1;
2  (* naive Lösung: *)
3  IMPORT Out;
4  CONST length = 10;
5  VAR
6    n: INTEGER;      (* aktuelle Anzahl Kunden *)
7    a: ARRAY length OF INTEGER;
8
9  PROCEDURE Clear*;
10   (* System in Grundzustand zurücksetzen *)
11   BEGIN n := 0;
12   (* a[0] wird in diesem Beispiel nicht verwendet. *)
13   END Clear;
14
15  PROCEDURE Insert*(x: INTEGER);
```

```
16  (* Füge Kunden der Priorität x entsprechend seiner Priorität in
17     Warteschlange ein. Keine Aktion, wenn das Array voll ist. *)
18  VAR i, j: INTEGER;
19  BEGIN
20    IF n < length - 1 THEN (* keine Aktion, wenn n = length - 1 *)
21      i := 1;
22      IF n > 0 THEN
23        WHILE (i <= n) & (x < a[i]) DO INC(i) END;
24        FOR j := n TO i BY -1 DO a[j+1] := a[j] END;
25      END;
26      a[i] := x; INC(n)
27    END
28  END Insert;
29
30  PROCEDURE Remove*(VAR x: INTEGER);
31  (* Entferne den Kunden höchster Priorität aus der
32     Warteschlange und schreibe seine Priorität nach x *)
33  VAR i: INTEGER;
34  BEGIN
35    x := a[n];
36    DEC(n)
37  END Remove;
38
39  PROCEDURE ShowAll*;
40  VAR i: INTEGER;
41  BEGIN i := 1;
42    WHILE i <= n DO
43      Out.Int(a[i], 4); INC(i)
44    END;
45    Out.Ln; Out.Ln
46  END ShowAll;
47
48  BEGIN Clear
49  END PriorityQueue1.
50
51
52  MODULE PrioTest;
53  IMPORT P := PriorityQueue1, In, Out;
54
55  PROCEDURE Insert*;
56  VAR x: INTEGER;
57  BEGIN In.Open; In.Int(x);
58    WHILE In.Done DO
59      P.Insert(x); In.Int(x)
60    END;
61  END Insert;
62
```

```

63  PROCEDURE Remove*;
64    CONST fmt = 6;
65    VAR x: INTEGER;
66  BEGIN
67    IF P.n > 0 THEN
68      P.Remove(x); Out.Int(x, fmt);
69      Out.String(", noch "); Out.Int(P.n, fmt); Out.String(" Elemente: ");
70      Out.Ln; P.ShowAll
71    END
72  END Remove;
73
74  END PrioTest.
75
76  Beispielaufruf durch:
77  PrioTest.Insert  19~
78  PrioTest.Insert  20 15 14 20 10 6 ~
79  PrioTest.Remove ~
80  PriorityQueue1.ShowAll ~
81  PriorityQueue1.Clear ~
82  Out.Ln ~
83
84  System.Free PrioTest  PriorityQueue1 ~

```

Es gibt einen wesentlich eleganteren Weg, nämlich die Realisierung einer Prioritätswarteschlange als ein sog. *Heap* („Haufen“, „Halde“).

zunächst: Ergänzungen zu Graphen, Bäumen:

Definition (informell) von Graphen: Ein Graph ist eine Struktur, bestehend aus Knoten und Verbindungen dieser Knoten, genannt Kanten. (mathematisch präzise Definition: u.a. in Theoretischer Informatik)

Die Kanten können auch mit einem Richtungssinn versehen sein. Der Graph heißt dann ein gerichteter Graph (oder Digraph).

Beim Durchlaufen eines gerichteten Graphen werden Kanten und damit Knoten in einer bestimmten Reihenfolge durchlaufen: aus der Sicht eines Knotens gibt es daher im allgemeinen Vorgänger(knoten) und Nachfolger(knoten).

Ein Baum ist ein gerichteter Graph ohne Kreise mit einem ausgezeichneten Knoten ohne Vorgänger, genannt Wurzel (root). Alle anderen Knoten haben jeweils genau einen Vorgänger. Knoten ohne Nachfolger heißen Blätter. Vorgänger heißen auch Eltern(knoten), Nachfolger heißen auch Kind(knoten).

Ein Binärbaum ist ein Baum, dessen Knoten höchstens zwei Nachfolger haben.

Heapstruktur: Ein Heap ist ein binärer Baum, dessen Knoten Werte haben und der die zwei folgenden Bedingungen erfüllt:

1. Der Baum ist fast horizontal abgeschnitten (ausgewogen, balanziert).
2. Knotenwerte von Kindknoten sind stets $\geq$ Knotenwerten von Elternknoten.

Kunden werden eingeordnet in sog. binären Baum; Struktur ist konsistent, wenn Vaterknoten $\leq$ Kindknoten; Kinder untereinander nicht geordnet

sentinel: Wache, Wachposten (Anmerkung: im heutigen Englisch üblicher Ausdruck: *sentry*)

Kindknoten von $a[i]: a[2*i], a[2*i+1]$

Vaterknoten von $a[j]: a[j \text{ DIV } 2]$ (s. Abbildung 4.9)

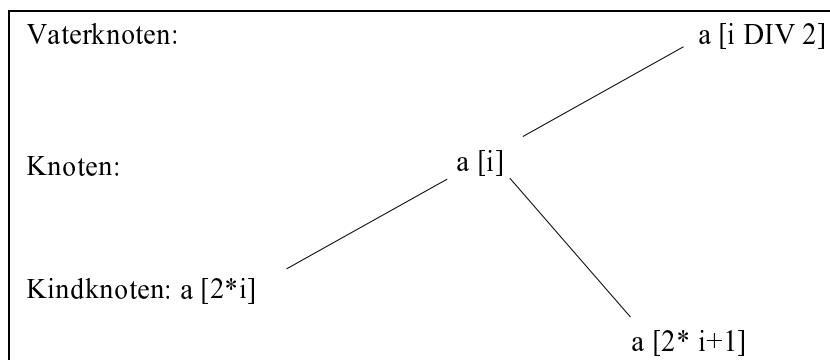


Abbildung 4.9: Heapstruktur

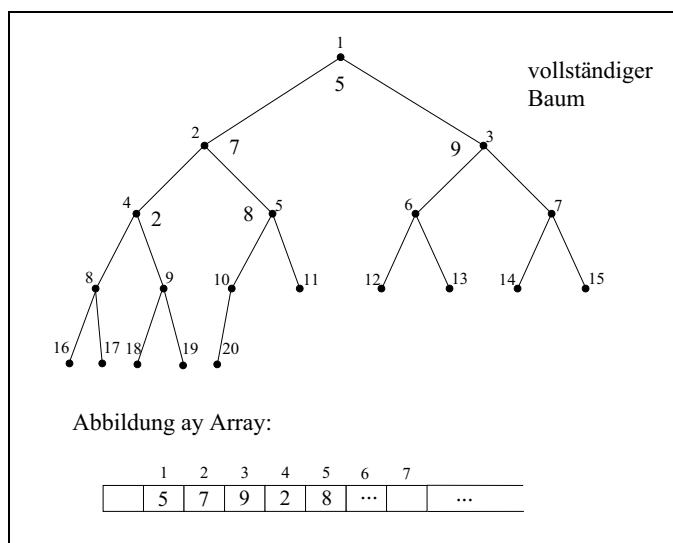


Abbildung 4.10: Vollständiger Baum: Abbildung auf Array

(s. Abbildung 4.10, zu den Operationen Insert und Delete: s. Abbildungen 4.11 und 4.12)

Modul-Variante 2, die auf einer Heap-Struktur beruht, zunächst als konkrete Datenstruktur:

4.4.2 PROGRAMMBEISPIEL (PRIORITÄTSWARTESCHLANGE ALS KONKRETE DATENSTRUKTUR)

```
1 MODULE PriorityQueue2;
```

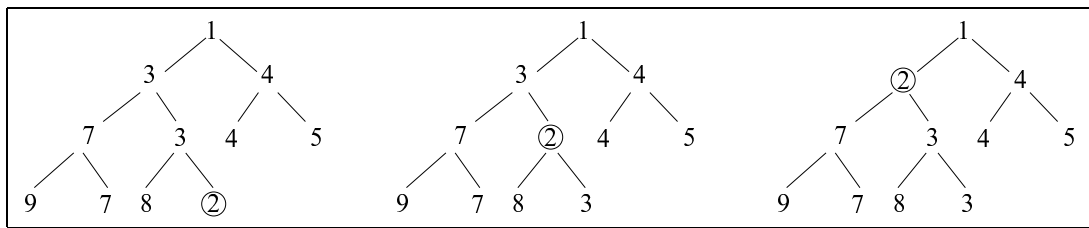


Abbildung 4.11: Prioritätswarteschlange: Knoten einfügen

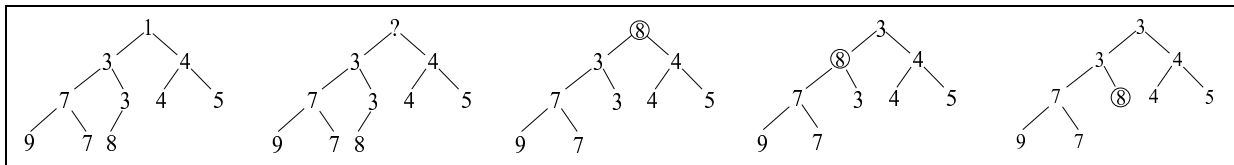


Abbildung 4.12: Prioritätswarteschlange: Knoten löschen

```

2  (* als konkrete Datenstruktur *)
3  IMPORT In, Out;
4  CONST length = 20000;
5  VAR
6    n: INTEGER; (* Anzahl Elemente *)
7    a: ARRAY length OF INTEGER;
8
9  PROCEDURE Clear;
10 BEGIN n := 0; a[0] := MIN(INTEGER) (* als Sentinel *)
11 END Clear;
12
13 PROCEDURE Insert(x: INTEGER);
14   VAR i: LONGINT;
15 BEGIN
16   IF n < length - 1 THEN
17     INC(n); i := n;
18     WHILE x < a[i DIV 2] DO a[i] := a[i DIV 2]; i := i DIV 2 END;
19     a[i] := x
20   END;
21 END Insert;
22
23 PROCEDURE Remove(VAR x: INTEGER);
24   VAR y, i, j: INTEGER; ready: BOOLEAN;
25 BEGIN (* Voraussetzung: n > 0, sonst keine Aktion *)
26   IF n > 0 THEN
27     x := a[1]; y := a[n];
28     DEC(n); i := 1; ready := FALSE;
29     WHILE (i <= n DIV 2) & ~ready DO
30       j := 2*i;
31       IF (j < n) & (a[j] > a[j+1]) THEN INC(j) END;
32       IF y > a[j] THEN a[i] := a[j]; i := j ELSE ready := TRUE END

```

```
33     END;
34     a[i] := y
35     END
36 END Remove;
37
38 PROCEDURE ShowAll;
39     VAR i: INTEGER;
40 BEGIN i := 0;
41     WHILE (i <= length-1) & (i <= n) DO
42         Out.Int(a[i], 4); INC(i)
43     END;
44     Out.Ln; Out.Ln
45 END ShowAll;
46
47 PROCEDURE Start*; (* eine kleine Beispielanwendung *)
48     CONST fmt = 3;
49     VAR x: INTEGER;
50 BEGIN Clear; In.Open;
51     In.Int(x);
52     WHILE In.Done DO
53         Insert(x); In.Int(x)
54     END;
55     ShowAll;
56
57     IF n > 0 THEN
58         Remove(x); Out.Int(x, fmt);
59         Out.String(", noch "); Out.Int(n, fmt); Out.String(" Elemente: ");
60     END;
61     ShowAll;
62
63     IF n > 0 THEN
64         Remove(x); Out.Int(x, fmt);
65         Out.String(", noch "); Out.Int(n, fmt); Out.String(" Elemente: ");
66     END;
67     ShowAll;
68
69     (* Bis hier ist die Anwendung korrekt. Die folgende Zuweisung
70        ist auch möglich; sie zerstört die Struktur irreversibel. Rein
71        technisch läßt sich eine solche Aktion mit den Mitteln,
72        die konkrete Datenstrukturen bieten, nicht unterbinden.*)
73
74     a[3] := 1; ShowAll;
75
76     IF n > 0 THEN (* ab hier fehlerhaft *)
77         Remove(x); Out.Int(x, fmt);
78         Out.String(", noch "); Out.Int(n, fmt); Out.String(" Elemente: ");
79     END;
```

```

80     ShowAll;
81
82     END Start;
83
84     BEGIN Clear
85     END PriorityQueue2.
86
87     Beispielaufruf durch:
88     PriorityQueue2.Start  2 9 5 3 7 1 ~
89
90     Ausgabe:
91
92     -32768  1  3  2  9  7  5
93
94     1, noch  5 Elemente: -32768  2  3  5  9  7
95
96     2, noch  4 Elemente: -32768  3  7  5  9
97
98     -32768  3  7  1  9
99
100    3, noch  3 Elemente: -32768  1  7  9

```

Vergleich der beiden Implementierungen:

Gemeinsamkeiten beider Beispiele:

- Bei Zugriff über die Prozeduren **Insert** und **Remove** zeigen beide Implementierungen gleiches funktionales Verhalten
- Bei direktem Zugriff auf die Array-Elemente (absichtlich oder unabsichtlich) ohne genaue Kenntnis des Verwaltungsalgorithmus geschieht folgendes:
 - lesender Zugriff: willkürliches, nicht-interpretierbares Ergebnis
 - schreibender Zugriff: i.a. irreversible Zerstörung der Struktur
- Auch Details der Art „Wird ein Sentinel verwendet?“ müssen bekannt sein (z.B.: der erste Kunde ist in `a[0]` oder `a[1]`?).

Unterschiede beider Beispiele:

- Die Implementierungen sind im Detail völlig verschieden voneinander.
- Die naive Variante wird langsam bei großer Kundenzahl.

Vorteile der Verwendung konkreter Datenstrukturen:

- weniger Overhead
- nicht nur spezielle Zugriffsprozeduren stehen zur Verfügung: sofern man die Struktur genau kennt, kann man auch direkt zugreifen.

Ein beliebtes Beispiel aus dem Bereich der „Hobbyprogrammierung“: direkter Zugriff auf den Bildschirmspeicher

Nachteile:

- Einzelheiten der Deklaration der Datenstruktur muß bekannt sein
- ebenso die Zugriffsalgorithmen
- Duplizierung von Code ist erforderlich
- Konsistenz der Datenstruktur kann leicht zerstört werden

Teil III

Modularisierung und Abstrakte Datentypen

Kapitel 5

Modularisierung. Einfache Datenabstraktion

5.1 Die Rolle von Modulen in der Softwaretechnik

Das Phasenmodell

Das *Phasenmodell* (oder *Wasserfallmodell*) ist eines der sog. *Vorgehensmodelle* zur Entwicklung eines großen Softwaresystems (andere Vorgehensmodelle: *Prototyp-Entwicklung*, *Spiralmodell*) (s. Abbildung 5.1)

Die Entwurfsphase

in Problemanalysephase: noch keine Hinweise auf konkrete Realisierung des Systems
Ziele in Entwurfsphase:

- Entwicklung von Modell des Gesamtsystems
- Abstand zwischen sog. Benutzermaschine und dem vorhandenen (oder zu beschaffenden) Computersystem überbrücken (s. Abbildung 5.2)
- dazu: Unterteilung des Systems in Einzelbausteine (Module): *Modularisierung* und Beschreibung der Funktionen dieser Bausteine und ihrer Beziehungen
- Vorteile der Modularisierung:
 - Verteilung von Aufgaben auf mehrere Programmierer: Module können unabhängig voneinander entwickelt werden
 - Wiederverwendung von Bausteinen (reuse of software components)
 - Austausch eines Programmteils (bei Wartung)
 - Beschränkung von Details, die Programmierer über andere Programmteile wissen muß
 - jedes Modul kann durch Modul gleicher Funktion ersetzt werden (Funktion unabhängig von interner Implementierung)
- Konzept dazu: hierarchische Modularisierung; Strategien: **Top-down-Methode** und **Bottom-up-Methode**

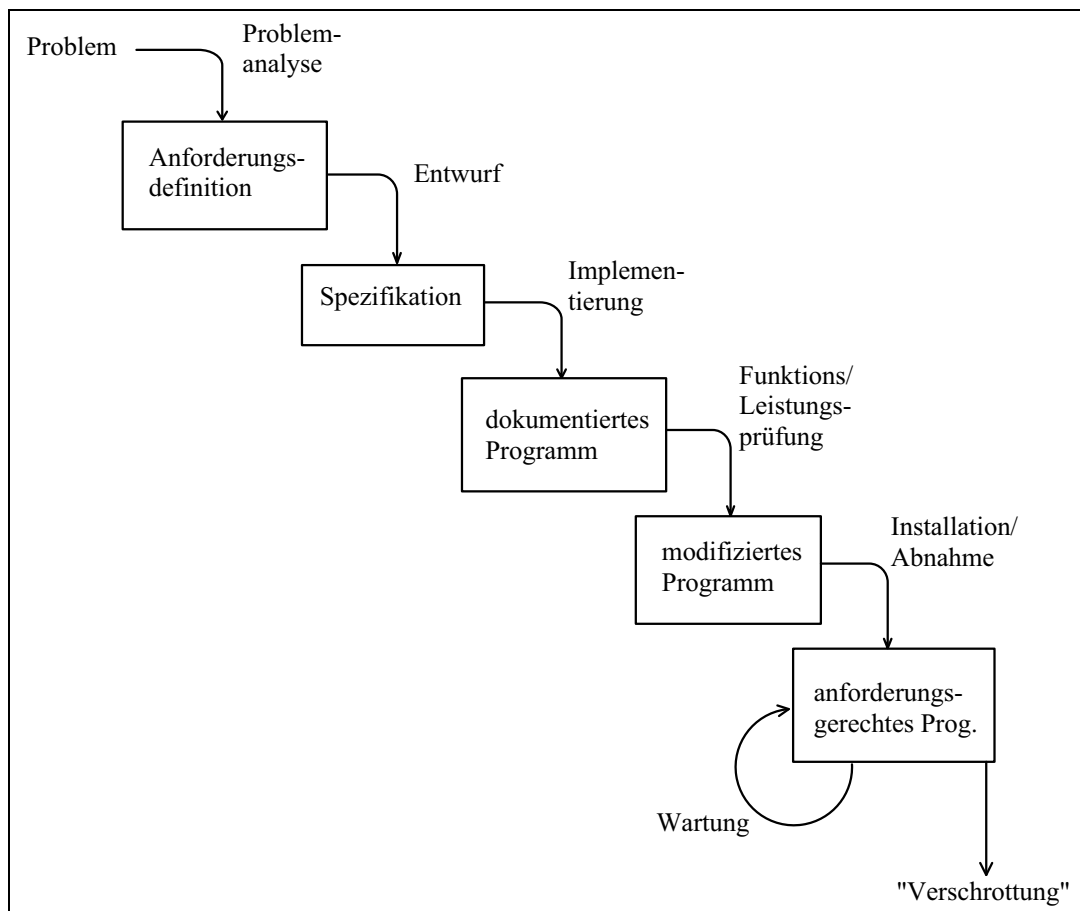


Abbildung 5.1: Das Phasenmodell, vgl. [Eng88, S. 546]

- **Top-down-Methode** (schrittweise Verfeinerung): beginnt bei Benutzermaschine; Funktionen der Module jeder Ebene werden *nur* durch die Funktionen der Module der unmittelbar darunter liegenden Ebene realisiert; es entsteht Hierarchie von Verfeinerungsstufen
- Abstand zwischen Benutzermaschine und Computersystem (Basismaschine) ständig verkleinert (s. Abbildung 5.3)
- **Bottom-up-Methode**: beginnt bei vorhandenem Computersystem; in jedem Entwurfsschritt: Funktionen der jeweils obersten Entwurfsebene werden zu komfortableren Funktionen zusammengesetzt; dadurch: höhere Entwurfsebene
- in Praxis: beide Methoden gemischt
- für übersichtliches Zusammenwirken von Modulen:
 - jedes Modul sollte Funktionen möglichst weniger anderer Module verwenden
 - Zahl der notwendigen Annahmen über ein Modul möglichst gering
 - innere Struktur eines Moduls nach außen möglichst weitgehend verborgen (Geheimnisprinzip); Beispiel dazu: s. Abbildung 5.4

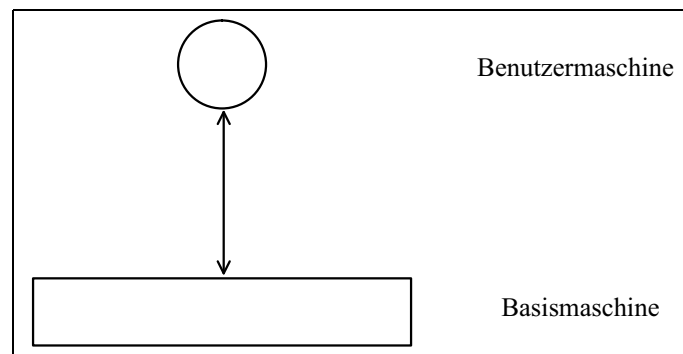


Abbildung 5.2: Computersystem und Benutzermaschine

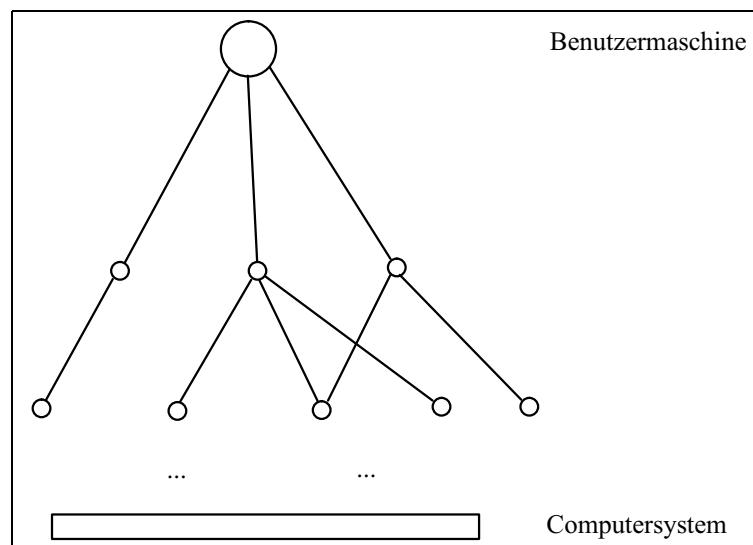


Abbildung 5.3: zur hierarchischen Modularisierung

5.2 Aufbau eines Moduls. Modulzustand

Zur Syntax eines Moduls (als Syntaxdiagramm): s. Abbildung 1.7 auf Seite 22
Modulschablone:

```
MODULE ident;
  IMPORT IMod1, IMod2, ... ;
```

(* Folge von Deklarationen: *)

```
CONST ... ; TYPE ... ; VAR ... ;
(* von Syntax her: beliebig wiederholt, beliebige Reihenfolge
* aber: Jeder im Modul verwendete Name muss vor seinem ersten
* Auftreten deklariert sein. (Dazu gibt es hier genau eine
* Ausnahme, im folgenden Teil genau eine weitere, naemlich die
* Vorwaertsdeklaration von Prozeduren.)
*)
```

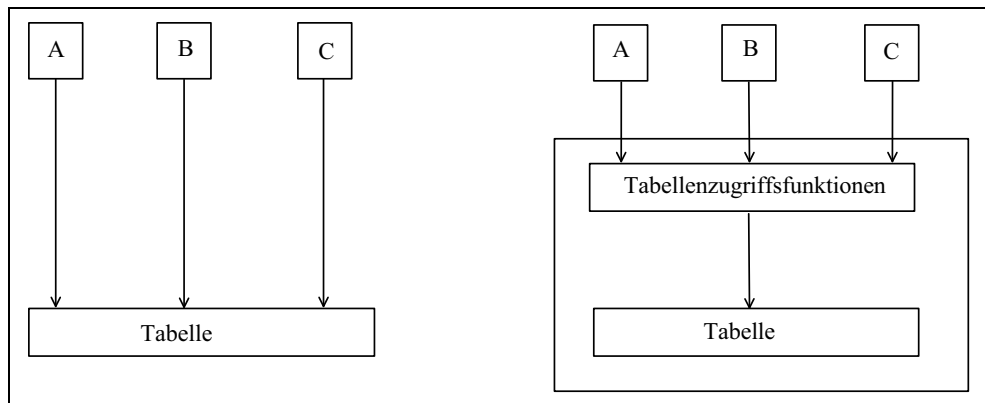


Abbildung 5.4: Tabellenzugriff. Anwendung des Geheimnisprinzips

```

ProcDecl;      (* Prozedurdeklaration *)
ForwardDecl;  (* evtl. Forwarddeklaration *)
(* auch hier: beliebig wiederholt, beliebige Reihenfolge *)

BEGIN
  StatSeq      (* Anweisungsfolge *)
END ident.

```

Deklarationen im Modul enthalten u.a. Prozedurdeklarationen, die ihrerseits Deklarationen (gleicher formaler Struktur) enthalten:

Das führt zu einer Blockstruktur: s. Abbildung 5.5

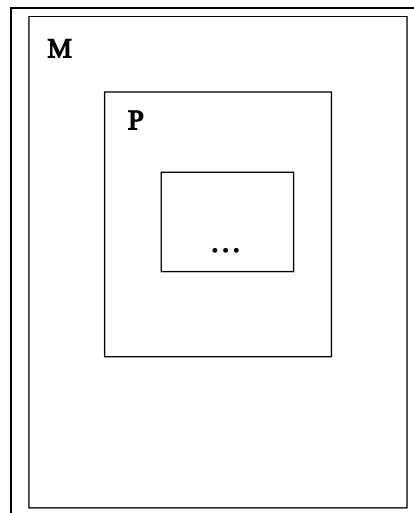


Abbildung 5.5: Blockstruktur in Modul

Regeln für Sichtbarkeitsbereiche gelten wie für geschachtelte Prozeduren. In Oberon gibt es keine Schachtelung von Modulen (keine lokalen Module). Man kann eine solche Struktur jedoch sinngemäß nachbilden, wenn auch ohne durch Sprachmittel unterstützte Kapselung (s. Abbildung 5.6).

Lebensdauer von Variablen, die in den verschiedenen Blöcken deklariert sind:

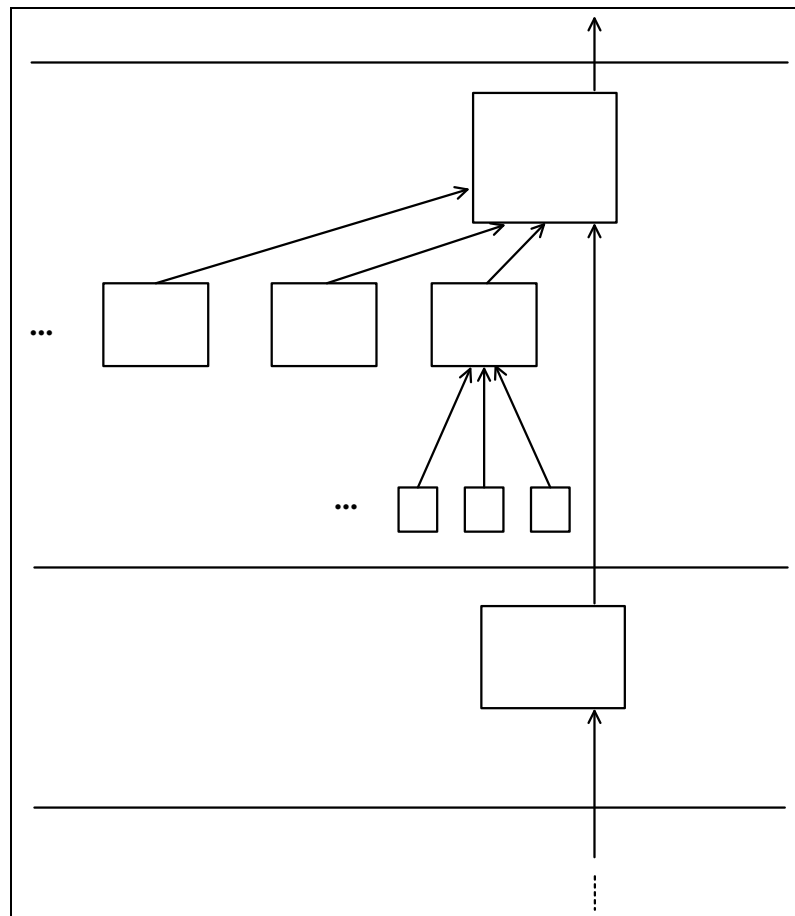


Abbildung 5.6: Geschichtete Struktur zur Nachbildung lokaler Module

- Speicherplatz für (lokale) Variablen in Prozeduren ist belegt, solange die Prozedur aktiv ist
- Speicherplatz für (modul-)globale Variablen wird belegt, wenn das Modul geladen wird und bleibt belegt, solange das Modul geladen bleibt.

Ein Modul wird dann geladen, wenn es benötigt wird; es bleibt geladen, bis es explizit entladen oder neu geladen wird.

Entladen geschieht durch

- Kommando `System.Free Modname1 ModName2 ...` oder durch
- Kommando `System.Free Modname*` (* bedeutet: auch importierte Module werden entladen, sofern sie nicht obendrein anderweitig importiert sind) oder durch
- explizites Entladen mit anschließendem Neuladen durch Maus-Interclick (ML)

Rolle des Anweisungsteils in Modul:

Anweisungen im Modulanweisungsteil werden beim Laden des Moduls einmalig ausgeführt. typische Anwendung: *Initialisierung* der globalen Variablen des Moduls, d.h., des Modulzustands

eingführende Beispiele hätten auch so formuliert werden können:

5.2.1 PROGRAMMBEISPIEL ()

```

1
2  MODULE HelloVariante2;
3    IMPORT Out;
4
5    PROCEDURE Start*;
6    BEGIN
7    END Start;
8
9  BEGIN
10   Out.String(" Hello, world! "); Out.Ln
11 END HelloVariante2.
12
13 Beispielaufruf durch:
14 HelloVariante2.Start ~
15 Ausgabe:
16 Hello, world!
17
18
```

Man hätte generell so die einführenden Programme in Oberon schreiben können. Der Vorteil (?) wäre eine größere Ähnlichkeit zu Pascal gewesen. Für Oberon ist das jedoch eine unnatürliche Art der Einführung, denn

- die Notwendigkeit der leeren Prozedur Start bleibt unklar und
- zum Start muß das Modul jedesmal neu geladen werden (durch Interclick mit ML).

Beispiel für modulglobale Variablen und deren Initialisierung:

5.2.2 PROGRAMMBEISPIEL ()

```

1
2  MODULE Counter;
3  (* Beispiel fuer abstrakte Sicht auf einen Zähler:
4     der Zähler wird beim ersten Starten initialisiert (Wert 100)
5     der interne Wert kann dann um einen (gegebenen)
6     einzulesenden Wert erhöht und ausgegeben werden. *)
7
8    IMPORT In, Out;
9
10   VAR i: INTEGER; (* interner Zählerzustand *)
11
12   PROCEDURE Init*;
13   BEGIN i := 100
14     (* willkürlicher Wert; 100 wegen der besseren
15        Erkennbarkeit gegenueber 0 *)
16   END Init;
17
18   PROCEDURE Write*;

```

```

19  BEGIN Out.Int(i, 5); Out.Ln
20  END Write;
21
22  PROCEDURE ReadAndAdd*;
23    VAR k: INTEGER;
24    BEGIN In.Open; In.Int(k); INC(i, k)
25    END ReadAndAdd;
26
27  BEGIN
28    Out.String(" im Initialisierungsteil von MODULE Counter "); Out.Ln;
29    Init
30  END Counter.
31  Beispielaufruf durch:
32    Counter.Init ~
33    Counter.ReadAndAdd 2 ~
34    Counter.Write ~
35
36

```

5.3 Intermodulare Beziehungen

Ein Modul kann jedes andere Modul importieren und Namen (insbesondere von Variablen und Prozeduren) aus diesem Modul verwenden, sofern diese mit einer *Exportmarke* versehen sind.

diskutieren: bei wem bzw. wo liegen die Verantwortlichkeiten ...

Das gesamte System von Modulen bildet einen *zyklenfreien Graphen*, s. Abbildung 5.7

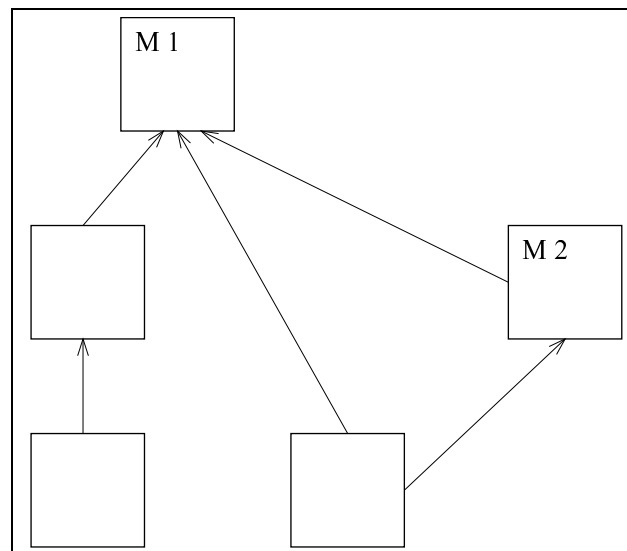


Abbildung 5.7: Modulgraph

Warum ist ein zyklischer Import von Modulen nicht möglich, vgl. Abbildung 5.8? Ein zyklischer Import wie in dem Beispiel wird bei der Compilation akzeptiert, führt jedoch

zu einem Laufzeitfehler.

5.3.1 PROGRAMMBEISPIEL (ZUM VERBOT DES ZYKLISCHEN IMPORTS)

```
1
2  MODULE Mod1;
3    IMPORT Out,
4
5    Mod3  ;
6  VAR i*: INTEGER;
7    PROCEDURE Start*;
8    BEGIN Out.String(" ... Start in Mod1 ... "); Out.Ln
9    END Start;
10
11 BEGIN Out.String("Init. von Mod 1: "); Start;
12   i := Mod3.i
13 END Mod1.
14
15
16 MODULE Mod2;
17   IMPORT Out, Mod1;
18   VAR i*: INTEGER;
19
20   PROCEDURE Start*;
21   BEGIN Out.String(" ... Start in Mod2 ... "); Out.Ln
22   END Start;
23
24 BEGIN Out.String("Init. von Mod 2: "); Start;
25   i := Mod1.i;
26 END Mod2.
27
28
29 MODULE Mod3;
30   IMPORT Out, Mod2;
31   VAR i*: INTEGER;
32   PROCEDURE Start*;
33   BEGIN Out.String(" ... Start in Mod3 ... "); Out.Ln
34   END Start;
35
36 BEGIN Out.String("Init. von Mod 3: "); Start;
37   i := Mod2.i;
38 END Mod3.
39
40
41 Beispielaufruf durch:
42 Out.Ln ~
43 Mod1.Start ~
44 Mod2.Start ~
45 Mod3.Start ~
```

```

46
47 System.Free Mod3 Mod2 Mod1 ~
48
49

```

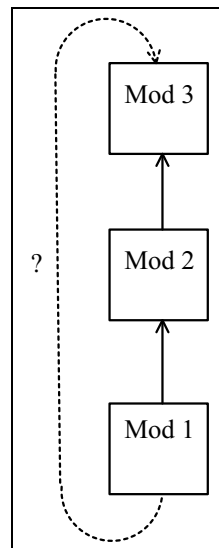


Abbildung 5.8: Import-Beziehungen von Modulen

Arten der Kommunikation von Modulen,
 Modulschnittstelle (browser)
 Erweiterung von Modulen,
 Warum kann in `IMPORT M := Math; M` nicht exportiert werden?

5.3.2 PROGRAMMBEISPIEL ()

```

1
2  MODULE AliasName;
3    IMPORT M := Math, Out; (* M* := Math, ... ist nicht zulässig *)
4
5    PROCEDURE Start*;
6      BEGIN Out.Real(M.sin(M.pi), 12); Out.Ln
7    END Start;
8
9  END AliasName.
10
11  Beispielaufruf durch:
12  AliasName.Start ~
13
14

```

5.4 Abstrakte Datenstrukturen

Was ist daran typisch?

- Der Zugriff auf die Datenstruktur ist *abstrakt*, d.h., die Struktur mit allen ihren Eigenschaften ist definiert ausschließlich über die Zugriffsprozeduren. Die Implementierung ist nicht (direkt) sichtbar; sie ist allenfalls im Zeitverhalten der Zugriffsoperationen erkennbar.
- Eine abstrakte Datenstruktur hat einen Zustand (internes Gedächtnis) im Unterschied zu einer Prozedur.
- Es ist nur ein Exemplar (eine Instanz, Inkarnation) erzeugbar (anders als beim *Abstrakten Datentyp* (ADT)).

Technische Realisierung in Oberon (wie in Modula): durch *Module*
nochmals Beispiel Priority Queue:

(als Beispiel für abstrakte Datenstruktur nach Mössenböck (E), p. 32)

5.4.1 PROGRAMMBEISPIEL (PRIORITÄTSWARTESCHLANGE ALS ABSTRAKTE DATENSTRUKTUR)

```

1
2  MODULE PriorityQueue3;
3    IMPORT Out;
4    (* als abstrakte Datenstruktur, Mössenboeck, p. 33 *)
5
6    CONST length = 1024;
7    VAR
8      n: INTEGER;  (* aktuelle Anzahl Kunden *)
9      a: ARRAY length OF INTEGER;
10
11   PROCEDURE Clear*;
12   BEGIN n := 0; a[0] := MIN(INTEGER)  (* als sog. Sentinel *)
13   END Clear;
14
15   PROCEDURE Insert*(x: INTEGER);
16     VAR i: INTEGER;
17   BEGIN
18     IF n < length-1 THEN
19       INC(n); i := n;
20       WHILE x < a[i DIV 2] DO a[i] := a[i DIV 2]; i := i DIV 2 END;
21       a[i] := x
22     END;
23   END Insert;
24
25   PROCEDURE Remove*(VAR x: INTEGER);
26     VAR y, i, j: INTEGER; ready: BOOLEAN;
27   BEGIN
28     IF n > 0 THEN
29       x := a[1]; y := a[n];
30       DEC(n); i := 1; ready := FALSE;
31       WHILE (i <= n DIV 2) & ~ready DO
32         j := i + i; IF (j < n) & (a[j] > a[j+1]) THEN INC(j) END;

```

```

33     IF y > a[j] THEN a[i] := a[j]; i := j ELSE ready := TRUE END;
34     END;
35     a[i] := y;
36     END;
37 END Remove;
38
39 PROCEDURE ShowAll*;
40   VAR i: INTEGER;
41   BEGIN FOR i := 1 TO n DO Out.Int(a[i], 3) END
42   END ShowAll;
43
44 BEGIN Clear
45 END PriorityQueue3.
46
47
48 MODULE PrioTest;
49   IMPORT P := PriorityQueue3, Out;
50
51   PROCEDURE Exec*;
52     VAR x: INTEGER;
53   BEGIN
54     P.Insert(3); P.Insert(2); P.Insert(7); P.Insert(-1); P.ShowAll; Out.Ln;
55     P.Remove(x); P.Remove(x);
56     P.ShowAll; Out.Ln; Out.Int(P.n, 4); Out.Ln; Out.Ln;
57   END Exec;
58
59 END PrioTest.
60
61
62 System.Free PrioTest* ~
63 System.Free PrioTest PriorityQueue ~
64
65 Beispielaufruf durch:
66 PrioTest.Exec ~
67 Ausgabe nach "System.Free PrioTest* "
68 und dann nach 1., 2., 3., ... Aufruf:
69
70 -1 2 7 3
71 3 7
72 2
73
74 -1 3 2 7 7 3
75 3 7 3 7
76 4
77
78 -1 2 3 3 7 3 7 7
79 3 7 3 7 7 3

```

```
80      6
81
82    -1 2 3 7 3 3 3 7 7 7
83    3 7 3 7 7 3 3 7
84      8
85
86    -1 3 2 7 3 3 3 7 7 7 7 3
87    3 3 3 7 7 3 3 7 7 7
88    10
89
90
```

Kapitel 6

Abstrakte Datentypen und Module

Allgemeine Diskussion über Bedeutung der Abstrakten Datentypen (ADTs).

Gegenüberstellung:	
Prinzip	Merkmal
konkrete Datenstruktur	Implementierungsdetails sind sichtbar
abstrakte Datenstruktur	Implementierung: • Details sind verborgen • nur ein Exemplar / eine Instanz erzeugbar
abstrakter Datentyp	auch mehr als eine Instanz erzeugbar, technisch: • Typdefinition ohne Zeiger oder • mit Zeiger (für dynamische Datenstrukturen)

6.1 Prioritätswarteschlange als ADT

wesentliche Änderung: es wird ein Typname eingeführt:

6.1.1 PROGRAMMBEISPIEL (PRIORITÄTSWARTESCHLANGE ALS ABSTRAKTER DATENTYP)

```
1
2  MODULE PriorityQueue;
3    IMPORT Out;
4    (* als abstrakter Datentyp, Mössenboeck, p. 36 *)
5
6    CONST length = 1024;
7    TYPE
8      Queue* = RECORD
9        n—: INTEGER;  (* aktuelle Anzahl Kunden *)
10       a: ARRAY length OF INTEGER
11     END;
12
```

```

13  PROCEDURE Clear*(VAR q: Queue);
14  BEGIN q.n := 0; q.a[0] := MIN(INTEGER)  (* als sog. Sentinel *)
15  END Clear;
16
17  PROCEDURE Insert*(VAR q: Queue; x: INTEGER);
18  VAR i: INTEGER;
19  BEGIN
20  IF q.n < length-1 THEN
21  INC(q.n); i := q.n;
22  WHILE x < q.a[i DIV 2] DO q.a[i] := q.a[i DIV 2]; i := i DIV 2 END;
23  q.a[i] := x
24  END;
25  END Insert;
26
27  PROCEDURE Remove*(VAR q: Queue; VAR x: INTEGER);
28  VAR y, i, j: INTEGER; ready: BOOLEAN;
29  BEGIN
30  IF q.n > 0 THEN
31  x := q.a[1]; y := q.a[q.n];
32  DEC(q.n); i := 1; ready := FALSE;
33  WHILE (i <= q.n DIV 2) & ~ready DO
34  j := i + i; IF (j < q.n) & (q.a[j] > q.a[j+1]) THEN INC(j) END;
35  IF y > q.a[j] THEN q.a[i] := q.a[j]; i := j ELSE ready := TRUE END;
36  END;
37  q.a[i] := y;
38  END;
39  END Remove;
40
41  PROCEDURE ShowAll*(VAR q: Queue);
42  VAR i: INTEGER;
43  BEGIN FOR i := 1 TO q.n DO Out.Int(q.a[i], 3) END
44  END ShowAll;
45
46  END PriorityQueue.
47
48
49  MODULE PrioTest;
50  IMPORT P := PriorityQueue, Out;
51  VAR q1, q2, q3: P.Queue;
52
53  PROCEDURE Exec*;
54  VAR x: INTEGER;
55  BEGIN P.Clear(q1);
56  P.Insert(q1, 3); P.Insert(q1, 2); P.Insert(q1, 7); P.Insert(q1, -1);
57  P.ShowAll(q1); Out.Ln;
58  P.Remove(q1, x); P.Remove(q1, x);
59  P.ShowAll(q1); Out.Ln; Out.Int(q1.n, 4); Out.Ln; Out.Ln;

```

```

60
61  (* analog fuer q2, q3 *)
62  END Exec;
63
64  END PrioTest.
65
66  Beispielaufruf durch:
67  PrioTest.Exec ~
68  Ausgabe:
69  -1 2 7 3
70  3 7
71  2
72
73  System.Free PrioTest* ~
74  System.Free PrioTest PriorityQueue ~
75
76

```

Mit dieser Version können jetzt auch mehrere Exemplare (Instanzen) erzeugt werden, z.B.: q1, q2, q3.

ADTs unterscheiden sich von abstrakten Datenstrukturen dadurch, daß mehr als ein Objekt des betreffenden Typs „instanziert“ werden kann.

Nachteil: etwas geringere Effizienz

abstrakte Datenstrukturen sollten dann benutzt werden, wenn nur ein Exemplar benötigt wird (Bsp.: s. Mössenböck ([Mö93]): Maus, Terminal)

Vergleich und Zusammenfassung: Es wurden vier verschiedene Varianten einer Prioritätswarteschlange eingeführt. Ohne viel nachzudenken wird man vielleicht auf die erste – naive – Programmvariante 4.4.1 (S. 127) kommen. Diese Variante ist ansonsten nicht weiter von Interesse.

Bei den anderen drei Varianten ist recht auffällig, dass man die textuellen Unterschiede schon fast suchen muß. Diese außerordentliche Ähnlichkeit in der Implementierung täuscht aber darüber hinweg, dass zwischen den drei Varianten große konzeptionelle Unterschiede bestehen. Man mache sich klar, dass in einer Sprache wie Pascal in der ursprünglichen Form, also ohne die Units aus Turbo-Pascal, nur die konkrete Datenstruktur 4.4.2 (S. 130) realisierbar ist. Dieses Beispiel zeigt allerdings auch, dass eine Demonstration konkreter Datentypen mit den Mitteln von Oberon eigentlich etwas gekünstelt ist; es ist konzeptionell ein Schritt zurück, der nicht mehr ganz passend präsentiert werden kann. Daher wurde auch – etwas inkonsistent – das Beispiel 4.4.1 bereits als abstrakte Datenstruktur formuliert. Was diese Lösung naiv macht ist die Verwendung des Algorithmus, nicht die Struktur. Erst das Konzept der Module erlaubt also den Schritt zur abstrakten Datenstruktur 5.4.1 (S. 146). Es ist heute ein offener Diskussionspunkt zwischen Sprachdesignern, ob man nicht alles von vornherein gleich in der Art eines abstrakten Datentyps wie im Beispiel 6.1.1 realisieren sollte. Die Designer von Java vertreten einen derartigen Standpunkt, offenbar mit erheblichem Erfolg, wie die Verbreitung von Java zeigt. Nicht zufällig gilt aber Java auch als recht langsam, z. B. beim Fensteraufbau mit den in Java realisierten sog. Swing-Klassen.

6.2 Wert- und Referenzsemantik

vgl. dazu: Switzer, Eiffel ([Swi93, p. 8])

Variablen (Objekte) können angesprochen werden entweder durch

- einen Variablennamen, der den Wert bezeichnet (sog. *Wertsemantik*, *value semantics*, *copy semantics*) oder durch
- die Adresse, z.B. die physikalische Adresse im Hauptspeicher oder eine geeignete Abstraktion davon (sog. *Referenzsemantik*, *reference semantics*)

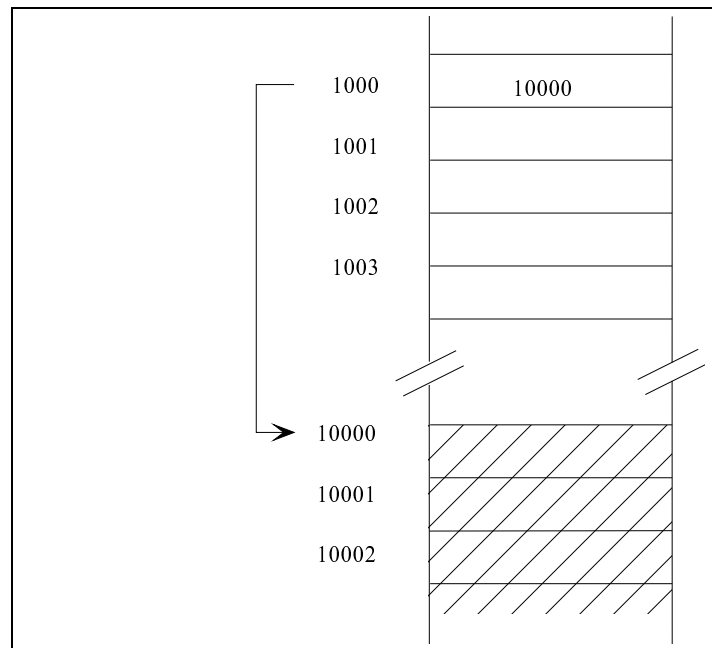


Abbildung 6.1: Ansprechen von Objekten

s. Abbildung 6.1

diskutieren: Vorteile/Nachteile?

z.B.: Sortieren: kein Bewegen (d.h. Kopieren) von (evtl. großen) Objekten bei Referenzsemantik

Aufbau von dynamischen Datenstrukturen nur durch Zeiger (Referenzsemantik)

Eine direkte Verwendung der Adressen im Hauptspeicher ist sehr fehleranfällig, deshalb werden *Zeiger* als „Abstraktion einer Adresse“ im Hauptspeicher verwendet.

technisches Hilfsmittel zur Formulierung von ADTs: Zeiger

ADT werden praktisch nur so formuliert, also nicht so wie in dem vorhergehendem Beispiel.

Das gesamte Oberon-System ist eine Ansammlung von ADTs, die geeignet konstruiert sind

Nachteil: auch Zeiger sind fehleranfällig: „Die Zeiger sind die *GOTOs* der Datenstrukturen.“ (Hoare, zitiert bei B. Meyer ([Mey90, p. 197]))

Es gibt heute aber keine allgemein akzeptierte Alternative (anders als bei *GOTOs*)

6.3 Zeigertypen und dynamische Variablen

Zeiger repräsentieren bzw. sind Abstraktionen der Adressen von Variablen eines gegebenen Typs; Abstraktion ist anzusehen als eine „low-level“-Abstraktion

in Oberon: Adresse ist (normalerweise) nicht direkt (d.h. zahlenmäßig) zugänglich; es sind nur wenige spezielle Operationen möglich;

In Oberon können nur solche Zeiger definiert werden, die auf Variablen des Record und des Array-Typs zeigen (anders als in Pascal, Modula, C, ...)

Beispiel für Deklaration:

TYPE

```
CharZeiger = POINTER TO CharDesc; (* Desc = Descriptor *)
                                (* konventionelle Standardbezeichnung in Oberon *)
CharDesc = RECORD
  c: CHAR
END;
...
```

VAR p, q: CharZeiger;

Ausschnitt aus Arbeitsspeicher: s. Abbildung 6.2

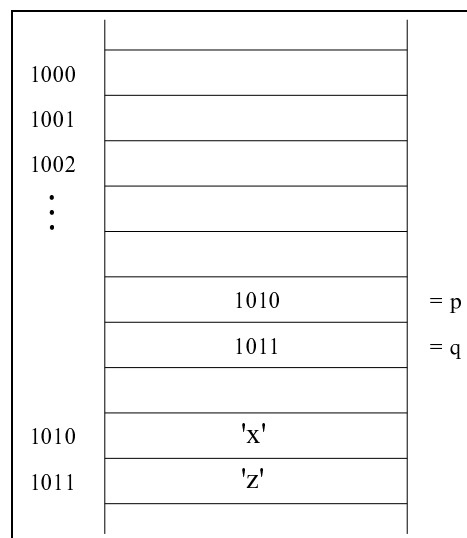


Abbildung 6.2: Ausschnitt aus Arbeitsspeicher

zusätzliche Information in p, q: die Zeiger p, q kennen den Datentyp, auf den sie zeigen; für definierten Zustand von p, q:

p := NIL; q := NIL;

anschaulich:

s. Abbildung 6.3

ohne die sog. „Erdung“ stehen irgendwelche zufälligen Werte in p und q. (Allerdings macht der Compiler eine automatische Initialisierung mit NIL (Unterdrückung mit \p).)

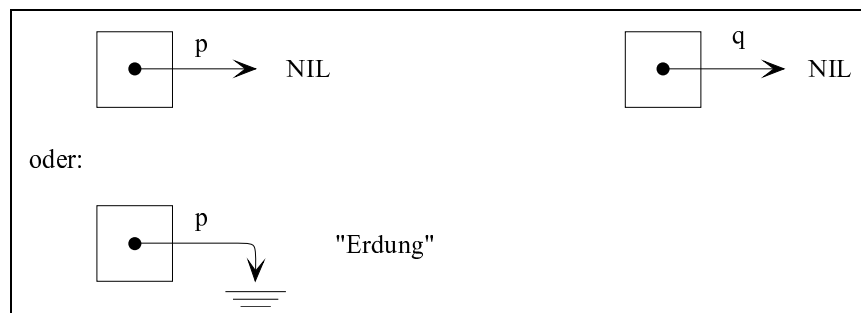


Abbildung 6.3: „geerdete“ Zeiger

Erzeugung von dynamischen Variablen, auf die p und q zeigen: durch vordefinierte Prozedur `NEW`

Aufruf:

`NEW(p); NEW(q);`

Wirkung: dynamische Bereitstellung von (ausreichend viel) Speicherplatz: wo?

erläutern: Speicherorganisation im Oberon-System

- Stack
- Heap
- ...

dynamisch bedeutet: Speicherplatz wird zur Laufzeit bereitgestellt, entsprechend:

statisch bedeutet: Speicherplatz wird bereits bei Programmstart bereitgestellt:

alle bisherigen Variablen waren statisch, z.B.

`VAR i, j: INTEGER; A: ARRAY 10000 OF LONGREAL;`

Die dynamischen Variablen sind zugänglich als $p^{\wedge}$, $q^{\wedge}$:

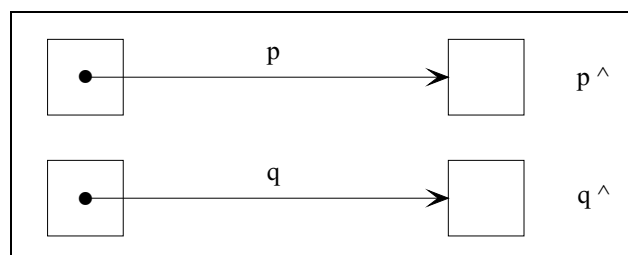


Abbildung 6.4: Dynamische Variablen

s. Abbildung 6.4

Zuweisung von Werten: $p^{\wedge}.c := 'x'$; $q^{\wedge}.c := 'z'$; (s. Abbildung 6.5)

In Oberon gibt es dafür die folgende Abkürzung:

$p.c := 'x'$;

Praktisch wird **nur** diese Abkürzung verwendet; die obige Schreibweise $p^{\wedge}.c := 'x'$; kommt faktisch nicht vor (*implizite Dereferenzierung*).

entsprechend bei Arrays:

```

...
TYPE
  Vector = POINTER TO VectorDesc;
  VectorDesc = ARRAY n OF REAL;
...
VAR
  a, b, c: Vector;
...
(* Zugriff durch: *)
a^[i] := b^[j]; (* mit 0 <= i, j <= n-1 *)
(* zulaessige Abkuerzung:*)
a[i] := b[j]; (* implizite Dereferenzierung *)

```

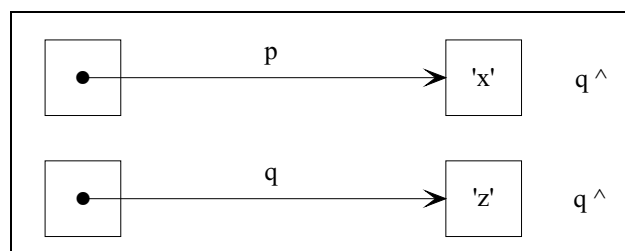


Abbildung 6.5: mit zugewiesenen Werten

Zwei Arten von Zuweisungen sind genauestens zu unterscheiden:

1. Zuweisung $p^{\wedge} := q^{\wedge}$; Die in dem durch q referenzierten Objekt enthaltenen Werte werden kopiert in das durch p referenzierte Objekt (s. Abbildung 6.6)

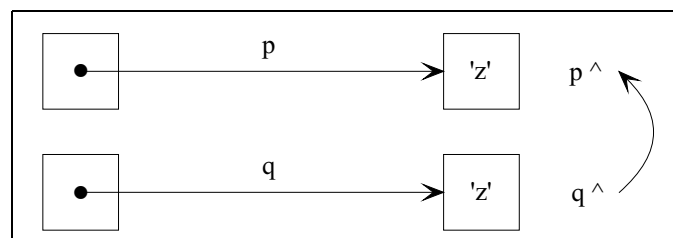


Abbildung 6.6: Zuweisung von dynamischen Variablen

2. Zuweisung $p := q$; Die in p enthaltene Referenz (Adresse) wird ersetzt durch die in q enthaltene Referenz, konkret: nach der Zuweisung $p := q$; zeigt p auf *dasselbe* Objekt wie (vorher bereits) q (s. Abbildung 6.7)

Die Objekte, auf die p , q zeigen, müssen (vorläufig) vom gleichen Typ sein, damit die obigen Zuweisungen zulässig sind.

Beispielprogramm mit syntaktisch korrekter Verwendung von Zeigern und der Erzeugung sog. „anonymer“ Variablen:

6.3.1 PROGRAMMBEISPIEL ()

```
1 MODULE IntObjs;
```

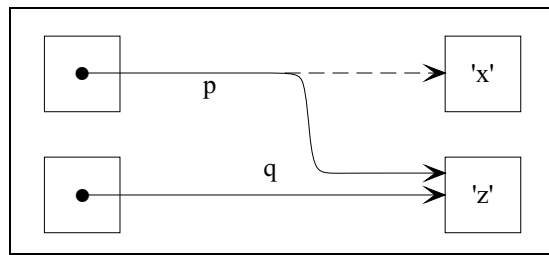


Abbildung 6.7: Zuweisung von Zeigern

```

2  IMPORT Out;
3  TYPE
4    IntObj* = POINTER TO IntObjDesc;
5    IntObjDesc = RECORD
6      I: INTEGER
7    END;
8
9  PROCEDURE Copy*(p: IntObj): IntObj;
10 (* Alternativname: Assign (Zuweisung)
11    Erzeuge neue Instanz und kopiere Inhalt von p dorthin. *)
12    VAR q: IntObj;
13    BEGIN NEW(q); q↑ := p↑; RETURN q
14  END Copy;
15
16  PROCEDURE AssignVal*(p: IntObj; i: INTEGER);
17 (* Kopiere i in das bestehende Objekt p *)
18  BEGIN p.I := i
19  END AssignVal;
20
21  PROCEDURE Create*(i: INTEGER): IntObj;
22 (* Erzeuge neue Instanz und initialisiere sie mit i *)
23  VAR p: IntObj;
24  BEGIN NEW(p); p.I := i; RETURN p
25  END Create;
26
27 (* folgendes ist auch eine Möglichkeit: *)
28  PROCEDURE Create1*(VAR p: IntObj; i: INTEGER);
29  BEGIN NEW(p); p.I := i
30  END Create1;
31
32 (* folgendes führt zu Fehler! diskutieren, warum!
33  PROCEDURE Create2*(p: IntObj; i: INTEGER);
34  BEGIN NEW(p); p↑.I := i
35  END Create2;
36  *)
37
38  PROCEDURE Add*(p, q: IntObj): IntObj;

```

```
39   VAR r: IntObj;
40   BEGIN NEW(r); r.l := p.l + q.l; RETURN r
41   END Add;
42
43   PROCEDURE Write*(p: IntObj; f: INTEGER);
44   BEGIN Out.Int(p.l, f)
45   END Write;
46
47   END IntObjs.
48
49
50   MODULE IntObjsTest;
51   IMPORT Ints := IntObjs, Out;
52
53   PROCEDURE Start*;
54   VAR p, q: Ints.IntObj;
55   BEGIN
56   p := Ints.Create(22); q := Ints.Create(7);
57   Ints.Write(p, 12); Ints.Write(q, 12);
58   Ints.Write(Ints.Add(p, q), 12); Out.Ln; (* Summe *)
59   (* 22      7      29 *)
60
61   (* oder auch: *)
62   Ints.Create1(p, 22); Ints.Create1(q, 7);
63   Ints.Write(p, 12); Ints.Write(q, 12);
64   Ints.Write(Ints.Add(p, q), 12); Out.Ln; (* Summe *)
65   (* 22      7      29 *)
66
67
68   Ints.Write(p, 12); Ints.Write(q, 12);
69   p := q;
70   Ints.Write(p, 12); Ints.Write(q, 12); Out.Ln;
71   (* 22      7      7      7 *)
72
73   Ints.AssignVal(q, 8);
74   Ints.Write(p, 12); Ints.Write(q, 12); Out.Ln;
75   (* beide sind 8, obwohl nur q↑ explizit geändert *)
76   (* 8      8 *)
77
78   p := Ints.Create(15); q := Ints.Create(3);
79   Ints.Write(p, 12); Ints.Write(q, 12);
80   p := Ints.Copy(q);
81   Ints.Write(p, 12); Ints.Write(q, 12); Out.Ln;
82   (* 15 3 3 3 *)
83
84   Ints.AssignVal(q, 8);
85   Ints.Write(p, 12); Ints.Write(q, 12); Out.Ln;
```

```

86      (* 3 8 *)
87
88      Out.Ln;Out.Ln;
89      END Start;
90  END IntObjsTest.
91
92  Beispielaufruf durch:
93  IntObjsTest.Start ~
94  Ausgabe:
95      22      7      29
96      22      7      29
97      22      7      7      7
98      8      8
99      15      3      3      3
100     3      8
101
102  System.Free IntObjsTest IntObjs ~

```

6.4 Dynamische Datenstrukturen

Komplexe Zahlen als ADT

6.4.1 PROGRAMMBEISPIEL ()

```

1
2  MODULE ComplexNmbs;
3  IMPORT Out, Math;
4  CONST eps = 0.001;
5
6  TYPE
7  Complex* = POINTER TO ComplexDesc;
8  ComplexDesc = RECORD
9  R, I: REAL
10 END;
11
12 PROCEDURE SetVal*(z: Complex; x, y: REAL);
13 BEGIN z.R := x; z.I := y
14 END SetVal;
15
16 PROCEDURE Create*(x, y: REAL): Complex;
17 VAR z: Complex;
18 BEGIN NEW(z); SetVal(z, x, y); RETURN z
19 END Create;
20
21 PROCEDURE Add*(u, v: Complex): Complex;
22 BEGIN RETURN Create(u.R + v.R, u.I + v.I);
23 END Add;
24

```

```

25  PROCEDURE UnMin*(z: Complex): Complex;
26  BEGIN RETURN Create(-z.R, -z.I)
27  END UnMin;
28
29  PROCEDURE Norm*(z: Complex): REAL;
30  BEGIN RETURN Math.sqrt(z.R*z.R + z.I*z.I)
31  END Norm;
32
33  PROCEDURE SameObjs*(u, v: Complex): BOOLEAN;
34  BEGIN RETURN u = v
35  END SameObjs;
36
37  PROCEDURE EqualObjs*(u, v: Complex): BOOLEAN;
38  BEGIN RETURN Norm(Add(u, UnMin(v))) <= eps
39  (* naive Variante waere: RETURN u↑ = v↑; *)
40  END EqualObjs;
41
42  PROCEDURE Write*(z: Complex; f: INTEGER);
43  BEGIN Out.Real(z.R, f); Out.String(" + i* "); Out.Real(z.I, f)
44  END Write;
45
46  END ComplexNmbs.
47
48
49  Schnittstelle:
50  DEFINITION ComplexNmbs;
51
52  TYPE
53    Complex = POINTER TO ComplexDesc;
54
55  PROCEDURE Add (u, v: Complex): Complex;
56  PROCEDURE Create (x, y: REAL): Complex;
57  PROCEDURE EqualObjs (u, v: Complex): BOOLEAN;
58  PROCEDURE Norm (z: Complex): REAL;
59  PROCEDURE SameObjs (u, v: Complex): BOOLEAN;
60  PROCEDURE SetVal (z: Complex; x, y: REAL);
61  PROCEDURE UnMin (z: Complex): Complex;
62  PROCEDURE Write (z: Complex; f: INTEGER);
63
64  END ComplexNmbs.
65
66
67  MODULE CmplTest;
68  IMPORT Out, C := ComplexNmbs;
69  CONST f = 10;
70  PROCEDURE Start*;
71  VAR u, v, w: C.Complex;

```

```

72 BEGIN
73   u := C.Create(1, 2); v := C.Create(3, 5);
74   C.Write(u, f); C.Write(v, f); Out.Ln;
75   C.Write(C.Add(u, v), f); Out.Ln;
76
77   Out.Real(C.Norm(C.Create(3, 4)), f); Out.Ln;
78   (* Alternative: *)
79   C.SetVal(u, 3, 4);
80   Out.Real(C.Norm(u), f); Out.Ln;
81
82   u := C.Create(15, 20); v := C.Create(15, 20);
83   IF C.SameObjs(u, v) THEN
84     Out.String("dieselben Objekte "); Out.Ln
85   ELSE
86     Out.String(" nicht dieselben Objekte "); Out.Ln
87   END;
88
89   IF C.EqualObjs(u, v) THEN
90     Out.String("Objekte gleichen Wertes"); Out.Ln
91   ELSE
92     Out.String(" Objekte verschiedenen Wertes "); Out.Ln
93   END;
94
95   Out.Ln; Out.Ln;
96 END Start;
97
98 END CmplTest.
99
100 Beispielaufruf durch:
101 CmplTest.Start ~
102 Ausgabe:
103   1.00E+00 + i*  2.00E+00  3.00E+00 + i*  5.00E+00
104   4.00E+00 + i*  7.00E+00
105   5.00E+00
106   5.00E+00
107   nicht dieselben Objekte
108   Objekte gleichen Wertes
109
110
111 System.Free CmplTest ComplexNmbs ~
112
113

```

weiterer Vorteil hier: jetzt können auch komplexe Zahlen Rückgabewerte einer Funktionsprozedur sein (genauer: Referenzen auf komplexe Zahlen)

Lineare Listen

Entstehen soll eine Struktur der folgenden Art: s. Abbildung 6.8

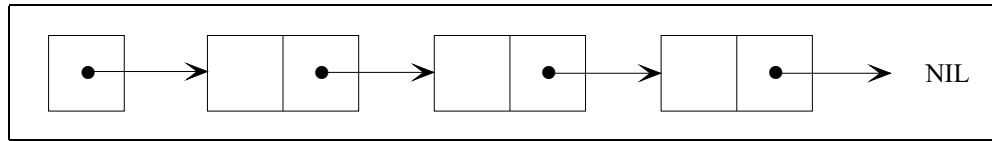


Abbildung 6.8: Lineare Liste

Vorwärtsreferenzen bei Typvereinbarungen: sind zwingend erforderlich, wenn ein Feld eines Records selbst wieder ein Zeiger ist, der auf ein Record gleichen Typs zeigen soll. Allgemeines zu Vorwärtsreferenzen:

6.4.2 PROGRAMMBEISPIEL (VORWÄRTSREFERENZEN)

```

1  MODULE Ptr3Test;
2
3  TYPE (* Variante mit Vorwärtsreferenz: ist erlaubt *)
4      Obj = POINTER TO ObjDesc;
5      ObjDesc = RECORD
6          C: CHAR
7      END;
8
9  TYPE (* Variante ohne Vorwaertsreferenz: ist (natürlich) auch erlaubt *)
10     ObjDesc1 = RECORD
11         C: CHAR
12     END;
13     Obj1 = POINTER TO ObjDesc1;
14
15     TYPE (* Variante mit Vorwaertsreferenz: ist erlaubt und auch nur so *)
16         Obj2 = POINTER TO ObjDesc2;
17         ObjDesc2 = RECORD
18             C: CHAR;
19             p: Obj2
20         END;
21
22     (*
23         TYPE Variante mit Vorwärtsreferenz: ist so nicht erlaubt
24         ObjDesc3 = RECORD
25             C: CHAR;
26             p: Obj3
27         END;
28         Obj3 = POINTER TO ObjDesc3;
29     *)
30
31     (* auch die folgenden beiden Varianten sind erlaubt: *)
32     TYPE
33         Obj4 = POINTER TO ObjDesc4;

```

```

34   ObjDesc4 = ARRAY 10 OF REAL;
35
36   TYPE
37     ObjDesc5 = ARRAY 10 OF REAL;
38     Obj5 = POINTER TO ObjDesc5;
39
40
41   PROCEDURE Start*;
42   END Start;
43
44   END Ptr3Test.
45
46   Beispielaufruf durch:
47   Ptr3Test.Start ~

```

Datenstruktur einer linearen Liste:

```

...
TYPE
  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD
    key: INTEGER; (* als Beispiel fuer "Nutzdaten" *)
    next: Node
  END;
...
VAR p, q: Node;

```

Darstellungsweisen, die im folgenden verwendet werden: s. Abbildung 6.9

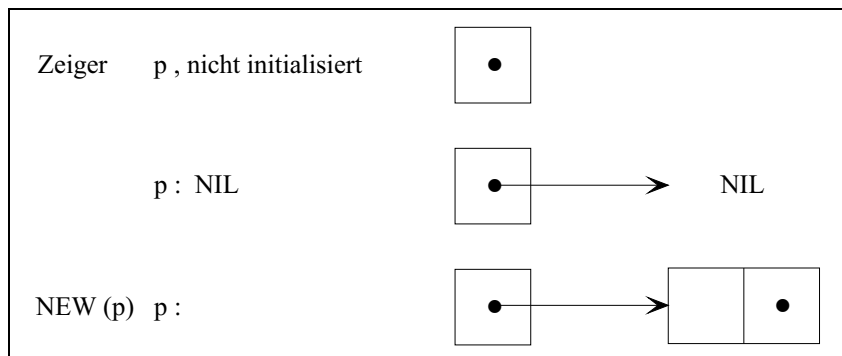


Abbildung 6.9: Darstellungen von Zeigern

Einzelne Schritte zum Aufbau einer linearen Liste, wenn diese an der Wurzel wächst (Prinzip): Abbildungen 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.17, 6.18 usw.

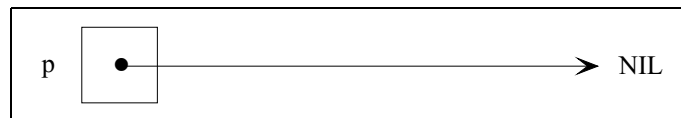
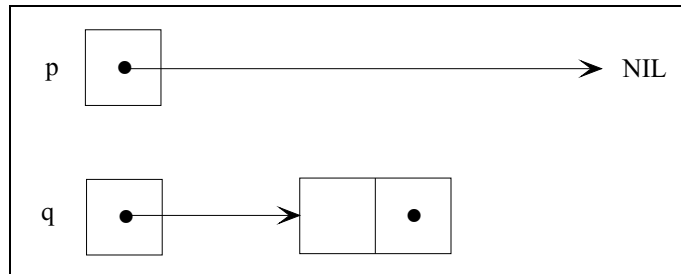
Implementierung einer linearen Liste im Detail:

6.4.3 PROGRAMMBEISPIEL (LINEARE LISTE)

```

1
2   MODULE LinLists; (* nach Reiser/Wirth, S. 147 *)

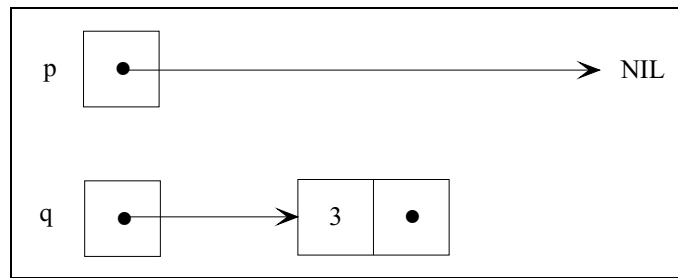
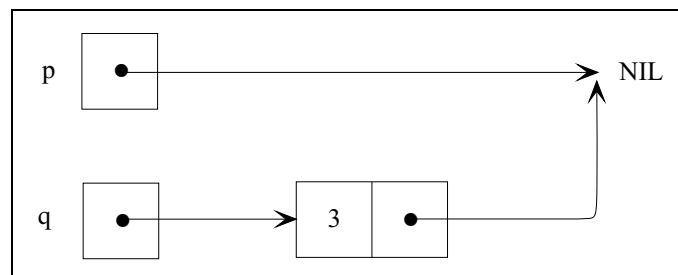
```

Abbildung 6.10: Initialisierung von p: $p := \text{NIL}$;Abbildung 6.11: Schritt 1.1: $\text{NEW}(q)$;

```

3  IMPORT Out;
4  TYPE
5      Key = INTEGER;
6      Node* = POINTER TO NodeDesc;
7      NodeDesc = RECORD
8          key: Key;
9          next: Node
10     END;
11
12  PROCEDURE CreateNode*(VAR new: Node; k: Key);
13      (* Neuen Knoten erzeugen, auf den 'new' zeigt und mit
14         Wert k initialisieren *)
15      BEGIN NEW(new); new.key := k; new.next := NIL
16      END CreateNode;
17
18  PROCEDURE P(p: Node); (* dazu: vgl. Reiser/Wirth *)
19      (* Aktion mit Knoten p ausführen; hier: Wert ausgeben *)
20      (* abhängig vom Typ von p.key *)
21      BEGIN Out.Int(p.key, 4); Out.Ln
22      END P;
23
24  PROCEDURE Insert*(VAR first: Node; new: Node);
25      (* Einfügen von Knoten 'new' an Anfang der Liste mit Anker
26         'first' *)
27      BEGIN new.next := first; first := new
28      END Insert;
29
30  PROCEDURE FirstNode*(VAR first: Node): Node;
31      (* Ersten Knoten aus Liste mit Anker 'first' löschen
32         und diesen als Rückgabewert bereitstellen *)
33      VAR n: Node;

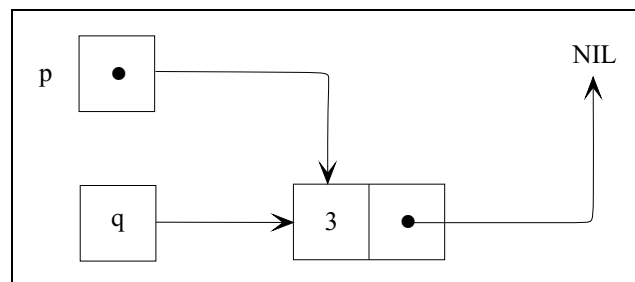
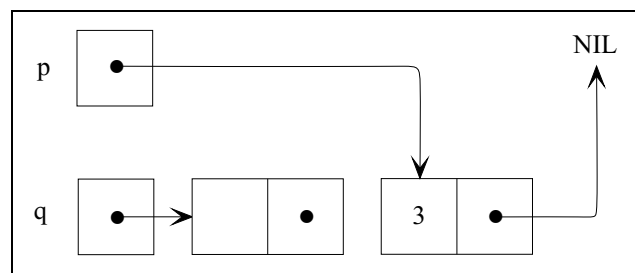
```

Abbildung 6.12: Schritt 1.2: $q.key := 3$;Abbildung 6.13: Schritt 1.3: $q.next := p$;

```

34 BEGIN
35   n := first; IF n # NIL THEN first := n.next;
36   (* evtl. ergänzen: *) n.next := NIL END;
37   RETURN n
38 END FirstNode;
39
40 PROCEDURE Enumerate*(first: Node);
41   (* Liste traversieren und mit jedem Knoten P() ausführen *)
42   BEGIN WHILE first # NIL DO P(first); first := first.next END
43   END Enumerate;
44
45 PROCEDURE InsertLast*(VAR first: Node; new: Node);
46   (* Knoten 'new' an das Ende der Liste mit Anker 'first' anhängen *)
47   VAR n: Node;
48   BEGIN
49     IF first = NIL THEN new.next := first; first := new
50     ELSE
51       n := first;
52       WHILE n.next # NIL DO n := n.next END;
53       new.next := n.next; n.next := new
54     END
55   END InsertLast;
56
57 PROCEDURE InsertRanked*(VAR first: Node; new: Node);
58   VAR n: Node;
59   BEGIN

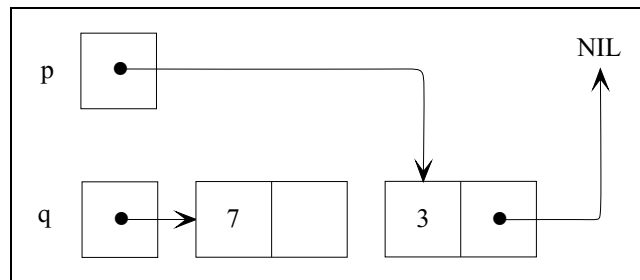
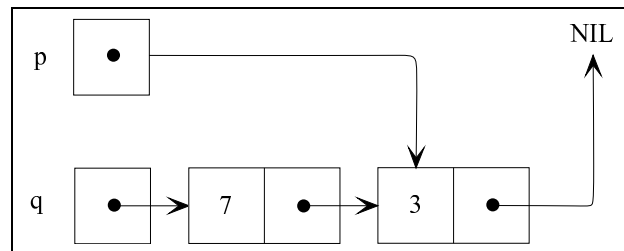
```

Abbildung 6.14: Schritt 1.4: $p := q$;Abbildung 6.15: Schritt 2.1: $NEW(q)$;

```

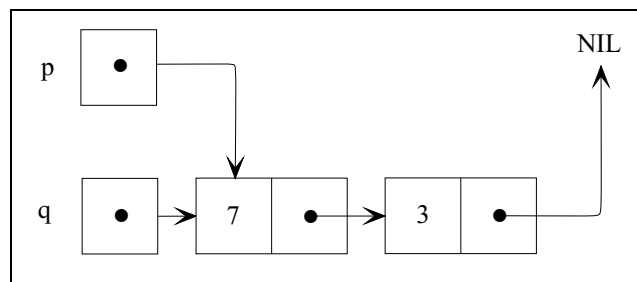
60  IF (first = NIL) OR (new.key < first.key) THEN
61      new.next := first; first := new;
62  ELSE
63      n := first;
64      WHILE (n.next # NIL) & (new.key >= n.next.key) DO
65          n := n.next
66      END;
67      new.next := n.next; n.next := new;
68  END;
69  END InsertRanked;
70
71  PROCEDURE Search*(first: Node; k: Key): Node;
72  (* Knoten suchen und als Rückgabewert bereitstellen, der
73     den Wert k enthält *)
74  BEGIN
75      WHILE (first # NIL) & (first.key # k) DO first := first.next END;
76      RETURN first
77  END Search;
78
79  PROCEDURE Delete*(VAR first: Node; node: Node);
80  (* Knoten 'node' aus Liste mit Anker 'first' löschen *)
81  VAR n: Node;
82  BEGIN
83      IF first # NIL THEN
84          IF first = node THEN first := node.next
85          ELSE

```

Abbildung 6.16: Schritt 2.2: `q.key := 7;`Abbildung 6.17: Schritt 2.3: `q.next := p;`

```

86         n := first;
87         WHILE (n.next # NIL) & (n.next # node) DO n := n.next END;
88         IF n.next # NIL THEN n.next := n.next.next END
89     END
90 END
91 END Delete;
92
93 END LinLists.
94
95
96 MODULE TestLists;
97     IMPORT LL := LinLists, Out;
98
99     PROCEDURE Start*;
100         CONST N = 10;
101         VAR a1, a2, new, p: LL.Node; i: INTEGER;
102     BEGIN
103         a1 := NIL; a2 := NIL; new := NIL; p := NIL;
104         i := 1;
105         WHILE i <= N DO
106             LL.CreateNode(new, i*i); LL.Insert(a1, new); INC(i)
107         END;
108         LL.Enumerate(a1); Out.Ln; Out.Ln;
109
110         p := LL.Search(a1, 9);
111         IF p=NIL THEN Out.String(" nicht gefunden "); Out.Ln END;
112         LL.Delete(a1, p);
  
```

Abbildung 6.18: Schritt 2.4: $p := q$;

```

113
114     LL.CreateNode(p, 20);
115     Out.String("--->"); LL.Enumerate(p); Out.Ln; Out.Ln;
116
117     LL.Insert(a1, p);
118     LL.Enumerate(a1); Out.Ln; Out.Ln;
119
120     Out.String("--->"); LL.Enumerate(p); Out.Ln; Out.Ln;
121     p := LL.FirstNode(a1);
122     LL.Enumerate(p); Out.Ln; Out.Ln
123   END Start;
124
125   END TestLists.
126
127
128   Beispielaufruf durch:
129   TestLists.Start ~
130   Ausgabe:
131
132   100
133   81
134   64
135   49
136   36
137   25
138   16
139   9
140   4
141   1
142
143
144   ---> 20
145
146
147   20
148   100

```

```
149      81
150      64
151      49
152      36
153      25
154      16
155      4
156      1
157
158
159      ---> 20
160      100
161      81
162      64
163      49
164      36
165      25
166      16
167      4
168      1
169
170
171      20
172
173
174      Schnittstelle:
175
176      DEFINITION LinLists;
177
178          TYPE
179              Node = POINTER TO NodeDesc;
180
181          PROCEDURE CreateNode (VAR new: Node; k: INTEGER);
182          PROCEDURE Delete (VAR first: Node; node: Node);
183          PROCEDURE Enumerate (first: Node);
184          PROCEDURE FirstNode (VAR first: Node): Node;
185          PROCEDURE Insert (VAR first: Node; new: Node);
186          PROCEDURE InsertLast (VAR first: Node; new: Node);
187          PROCEDURE InsertRanked (VAR first: Node; new: Node);
188          PROCEDURE Search (first: Node; k: INTEGER): Node;
189
190      END LinLists.
191
192
193      System.Free TestLists LinLists ~
194
195
```

Binärbäume

geordneter binärer Baum: s. Abbildung 6.19

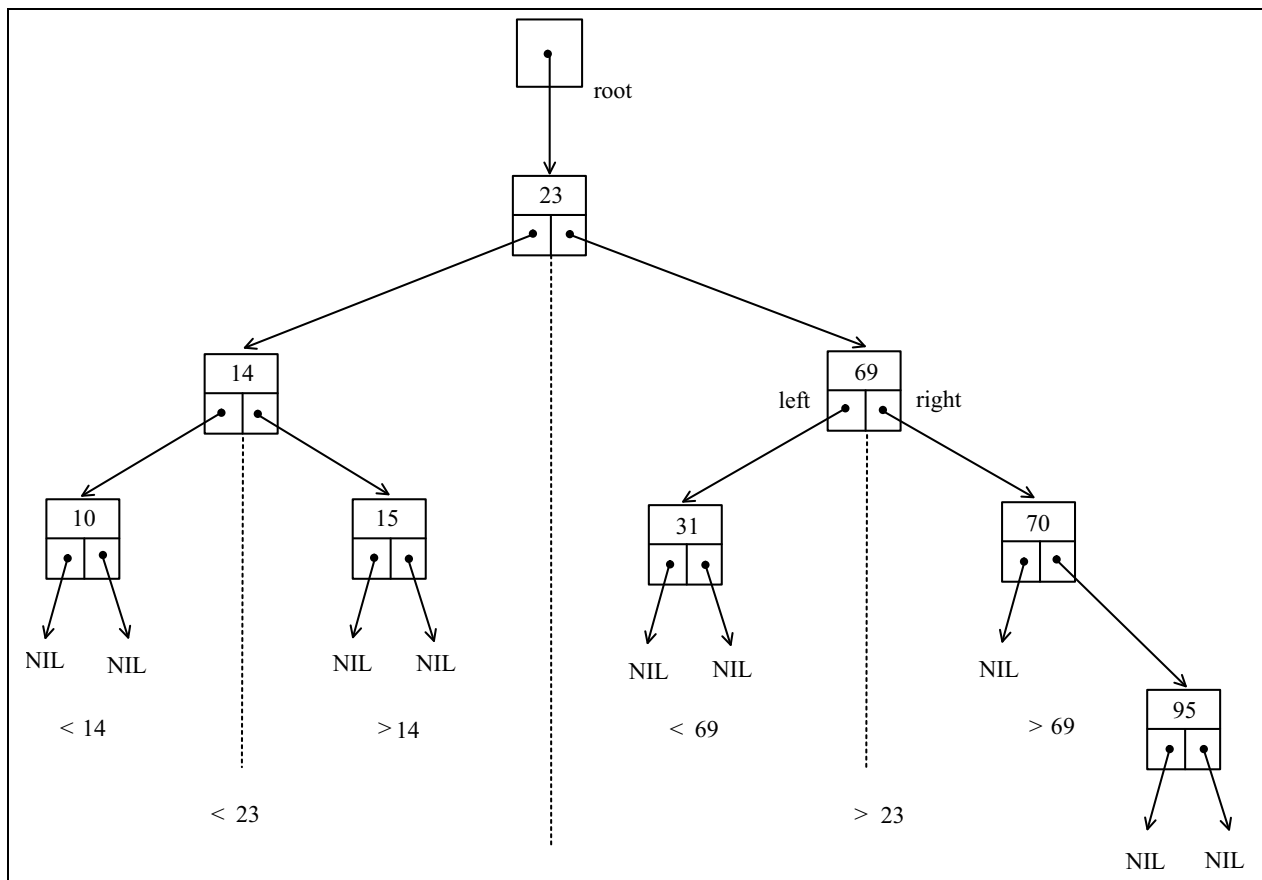


Abbildung 6.19: Geordneter binärer Baum (Binärer Suchbaum)

6.4.4 PROGRAMMBEISPIEL (BINÄRE BÄUME)

```

1  MODULE Trees1;  (* aus Reiser/Wirth, S. 152 *)
2  IMPORT Out;
3  TYPE
4    Key = INTEGER;
5    Node* = POINTER TO NodeDesc;
6    NodeDesc = RECORD
7      key: Key;
8      left, right: Node
9    END;
10
11  PROCEDURE SetVal*(n: Node; k: Key);
12  BEGIN n.key := k
13  END SetVal;
14
15  PROCEDURE P*(root: Node);
16  (* Ausgabe: nur speziell für Key = INTEGER *)

```

```
17 BEGIN Out.Int(root.key, 12); Out.Ln
18 END P;
19
20 PROCEDURE Enumerate*(root: Node);
21 BEGIN
22   IF root # NIL THEN
23     Enumerate(root.left); P(root); Enumerate(root.right)
24   END
25 END Enumerate;
26
27 PROCEDURE Insert*(VAR root: Node; new: Node);
28 BEGIN
29   IF root = NIL THEN
30     root := new; root.left := NIL; root.right := NIL
31   ELSIF new.key < root.key THEN Insert(root.left, new)
32   ELSIF new.key > root.key THEN Insert(root.right, new)
33   ELSE (* doppelter Schluessel *)
34     END;
35 END Insert;
36
37 PROCEDURE Delete*(VAR root: Node; key: Key);
38   PROCEDURE RemoveMax(VAR root, max: Node);
39   BEGIN
40     IF root.right # NIL THEN RemoveMax(root.right, max)
41     (* RETURN (deutsche Ausgabe) zu streichen *)
42     ELSE max := root; root := max.left
43     END
44   END RemoveMax;
45
46   VAR node: Node;
47
48 BEGIN
49   IF root # NIL THEN
50     IF key < root.key THEN Delete(root.left, key)
51     ELSIF key > root.key THEN Delete(root.right, key)
52     ELSE (* lösche root *)
53       IF root.left = NIL THEN root := root.right
54       ELSIF root.right = NIL THEN root := root.left
55       ELSE (* root hat zwei Nachfolger *)
56         RemoveMax(root.left, node);
57         node.left := root.left; node.right := root.right;
58         root := node
59       END
60     END
61   END;
62 END Delete;
63
```

```
64  PROCEDURE Search*(root: Node; key: Key): Node;
65  BEGIN
66    WHILE (root # NIL) & (root.key # key) DO
67      IF key < root.key THEN root := root.left
68      ELSE root := root.right
69      END;
70    END;
71    RETURN root
72  END Search;
73
74  END Trees1.
75
76
77  MODULE TestTree;
78  IMPORT Out, T := Trees1;
79  VAR
80    r, n1, n2, n3, n4, max: T.Node;
81    k: INTEGER;
82
83  PROCEDURE Start*;
84  BEGIN
85    NEW(r); NEW(n1); NEW(n2); NEW(n3); NEW(n4);
86    T.SetVal(r, 10);
87    T.SetVal(n1, 11); T.SetVal(n2, 4); T.SetVal(n3, 3); T.SetVal(n4, 7);
88    T.Insert(r, n1); T.Insert(r, n2); T.Insert(r, n3); T.Insert(r, n4);
89    T.Enumerate(r);
90
91    T.Delete(r, 4);
92    T.Enumerate(r);
93
94    k := 10;
95    IF T.Search(r, k) = NIL THEN
96      Out.String(" Knoten nicht gefunden "); Out.Ln
97    ELSE
98      T.SetVal(T.Search(r, k), 15);
99    END;
100    T.Enumerate(r);
101    Out.Ln; Out.Ln
102  END Start;
103
104  END TestTree.
105
106  Beispielaufruf durch:
107  TestTree.Start ~
108  Ausgabe:
109      3
110      4
```

```
111         7
112         10
113         11
114         3
115         7
116         10
117         11
118         3
119         7
120         15
121         11
122
123 System.Free TestTree Trees1 ~
124
125 Schnittstelle:
126
127 DEFINITION Trees1;
128
129     TYPE
130         Node = POINTER TO NodeDesc;
131
132     PROCEDURE Delete (VAR root: Node; key: INTEGER);
133     PROCEDURE Enumerate (root: Node);
134     PROCEDURE Insert (VAR root: Node; new: Node);
135     PROCEDURE P (root: Node);
136     PROCEDURE Search (root: Node; key: INTEGER): Node;
137     PROCEDURE SetVal (n: Node; k: INTEGER);
138
139 END Trees1.
```

Teil IV

Jenseits von Pascal und Modula

Kapitel 7

Objektorientierung

7.1 Informelle Einführung

Schlüsselkonzepte der objektorientierten Programmierung (OOP)

Was soll/kann mit OOP erreicht werden?

Erweiterbarkeit von („wiederverwendbaren“) Softwarekomponenten ohne Recompilation, also ohne Änderung des Quelltextes

Probleme:

- Numerische Bibliotheken: nur komplexe Algorithmen (Prozeduren), keine komplexen Datenstrukturen
- bei ADT: Datenfelder sind an Record-Typ statisch gebunden
- ebenso die Zugriffsprozeduren

Bsp.: Lineare Liste: Wenn eine lineare Liste als Datenstruktur mit Zugriffsprozeduren vorliegt, kann diese nachträglich weder um Datenfelder erweitert werden noch können weitere Prozeduren hinzugefügt werden, ohne daß das Modul recompiliert wird. OOP löst bzw. mindert diese Probleme.

Was macht Objekt-Orientierung aus? nach Reiser und Wirth ([RW92, S. 221]) (dort *Schlüsselkonzepte* der OOP genannt):

generische Module : nur Strukturinformationen sind vorhanden; anwendungsspezifische Datenfelder werden (nachträglich) vom Anwender hinzugefügt: ein solches Modul heißt *generisch*

heterogene Datenstruktur : Struktur besteht aus verschiedenen, aber kompatiblen Knotentypen

Objekt und seine Repräsentierung : Instanz einer Datenstruktur mit Zugriffsprozeduren

dynamische Bindung von Prozeduren : Prozeduren werden erst zur Laufzeit ausgewählt

Modulstruktur eines objektorientierten Programms : invertierte Modulstruktur

Informelle Beispiele zur OOP

Spezialisierung/Generalisierung, Unterklasse/Oberklasse: s. Abbildungen 7.1, 7.2

ACHTUNG! Die übliche Darstellung von Vererbungsbeziehungen kollidiert mit der Darstellung der Import-Beziehungen von Modulen. Hier geht es im Moment um Vererbungsbeziehungen!

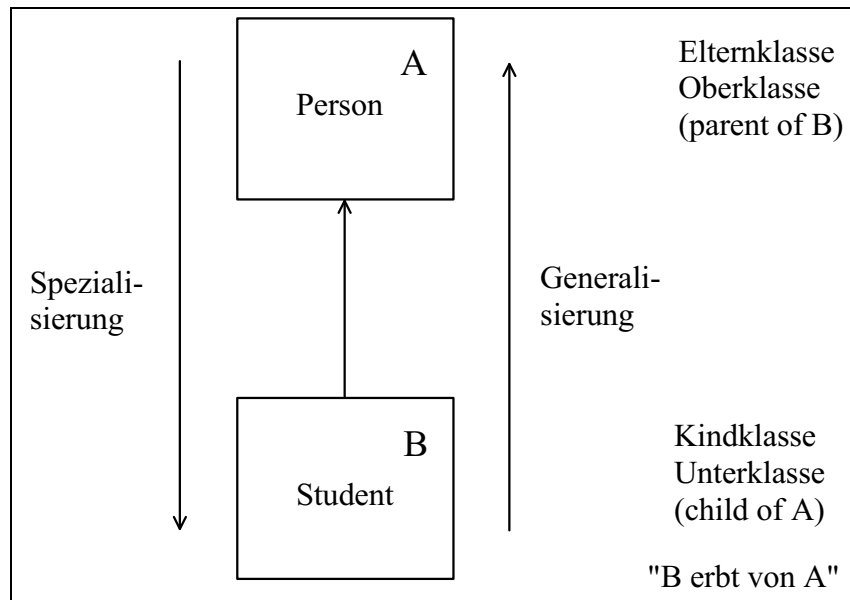


Abbildung 7.1: Spezialisierung und Generalisierung

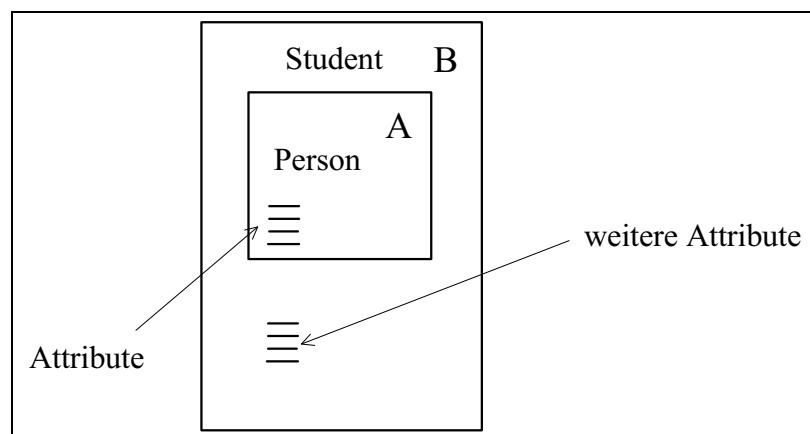


Abbildung 7.2: Spezialisierung und Generalisierung, andere Darstellung

Beispiel Graphikeditor: Was ist das, was wird damit bewirkt? Vererbungsbeziehungen für einen graphischen Editor, s. Abbildungen 7.3 und 7.4

Mögliche Realisierung der Datenstruktur für einen Graphikeditor: s. Abbildung 7.5

Begriffe der OOP

Vergleich der Nomenklatur von Oberon und anderen OOP-Sprachen:

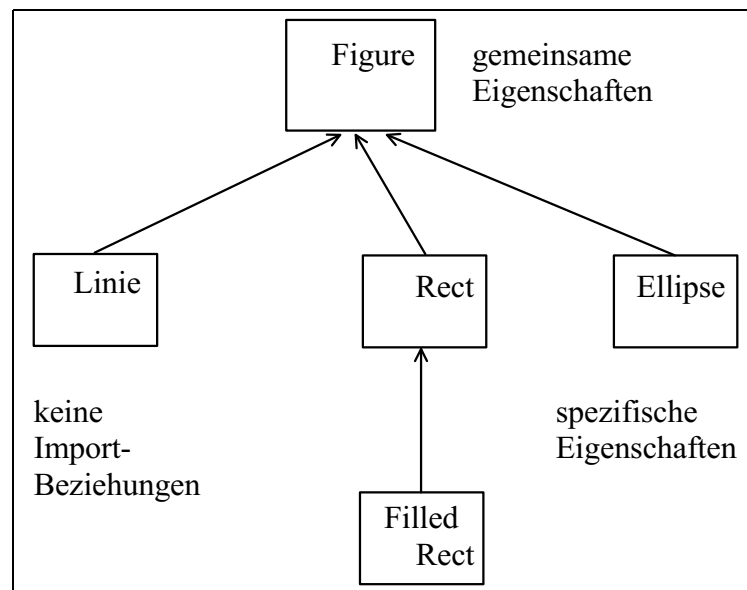


Abbildung 7.3: Vererbungsbeziehungen in einem Graphikeditor

traditionelle Bezeichnung	Oberon-Terminologie
Klasse	Record-Typ mit typgebundenen Prozeduren
Objekt	Variable dieses Typs
Methode	typgebundene Prozedur
Dynamisches Binden	Aufruf einer typgebundenen Prozedur
Meldung senden	Aufruf einer dynamisch gebundenen Prozedur
Unterklasse (abgeleitete Klasse, subclass, Kindklasse)	Erweiterter Record-Typ
Oberklasse (superclass, Elternklasse)	Basistyp einer Erweiterung
Vererbung	Typerweiterung
Super-Call	Aufruf einer Prozedur, die an den Basistyp einer Erweiterung gebunden ist

Ein fundamentaler Unterschied zwischen Oberon und anderen OOP-Sprachen, z.B. Eiffel:

Kapselungseigenschaft liegt in Oberon beim Modul, nicht bei der Klasse. Beides sind verschiedene Dinge. In Eiffel dagegen z.B. gibt es nur Klassen, die in gewissem Sinne die Rolle von Modulen übernehmen.

7.2 Technische Umsetzung in Oberon-2

Typerweiterung von Record-Typen

Person/Student als Beispiel, s. Abbildungen 7.6 und 7.7

TYPE

```

Person = POINTER TO PersonDesc;
PersonDesc = RECORD

```

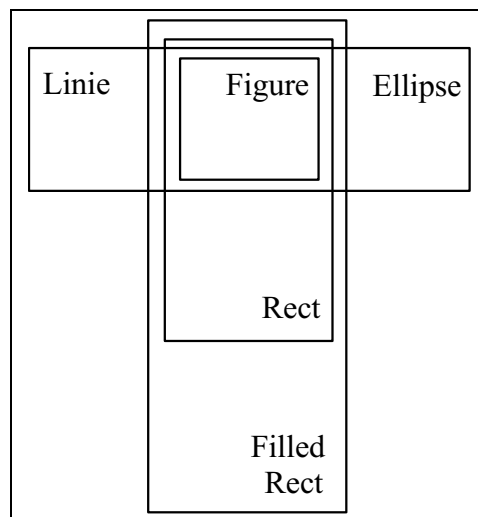


Abbildung 7.4: Vererbungsbeziehungen, anders dargestellt

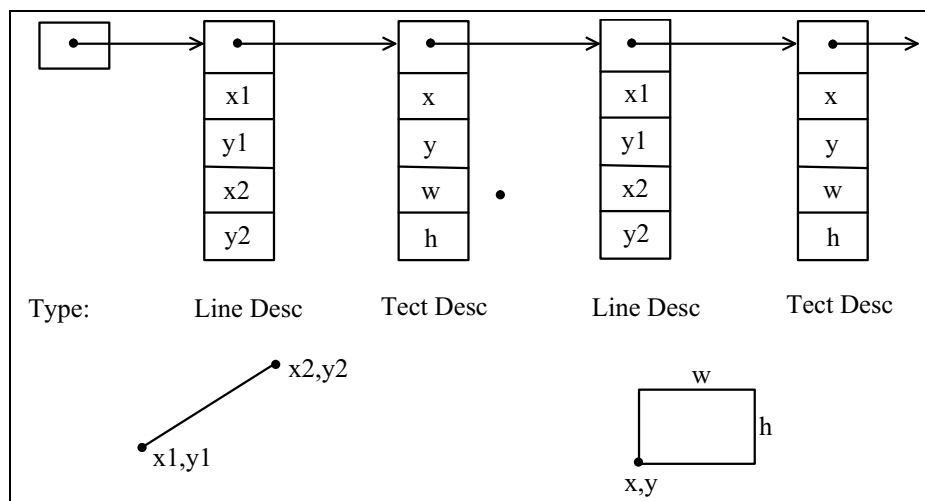


Abbildung 7.5: angestrebte Datenstruktur für den Graphikeditor

```
Name, Vorname: ARRAY N OF CHAR
END;
```

```
Student = POINTER TO StudentDesc;
StudentDesc = RECORD(PersonDesc)
  MatNr: LONGINT
END;
```

PersonDesc heißt *Basistyp*

StudentDesc heißt (direkte) *Erweiterung* von PersonDesc

Felder Name, Vorname: werden von StudentDesc *geerbt*. Sie sind damit Bestandteil der 2. Record-Definition

Typenerweiterung kann entsprechend fortgesetzt werden:

s. Abbildung 7.8

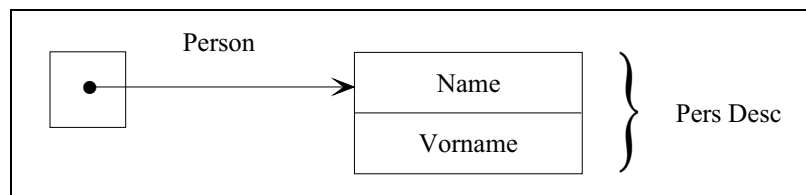


Abbildung 7.6: Basistyp Person

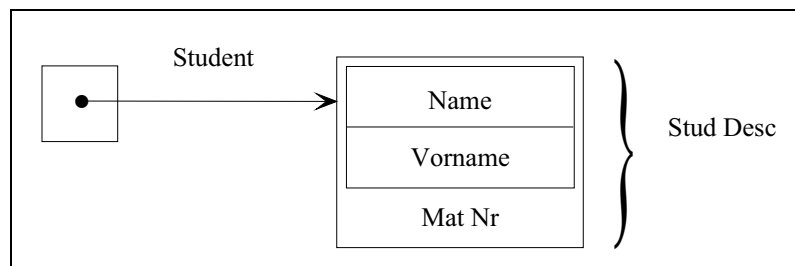


Abbildung 7.7: Erweiterter Typ Student

```

TYPE
  P0 = POINTER TO T0;
  T0 = RECORD
    next: P0
  END;
  T1 = RECORD(T0)
    a: Data1
  END;
  T2 = RECORD(T1)
    b: Data2
  END;

```

Erweiterter Typ: ist *Spezialisierung* seines Basistyps

aber: nicht jede Art von Spezialisierung ist in dieser Weise vernünftig implementierbar

Beispiele:

- Z.B. ist eine reelle Zahl eine spezielle komplexe Zahl, nämlich eine solche mit $\Im(z) = 0$ oder
- Ein Quadrat ist ein spezielles Rechteck, nämlich eines mit $a - b = 0$.

Spezialisierung bedeutet hier: Aufstellen von einer (oder mehreren) Nebenbedingungen.

Durch Typerweiterung ist diese Art Spezialisierung *nicht darstellbar*.

ansprechen von Feldnamen:

```

VAR
  p1, p2: PersonDesc;
  s1, s2: StudentDesc;

```

zulässig:

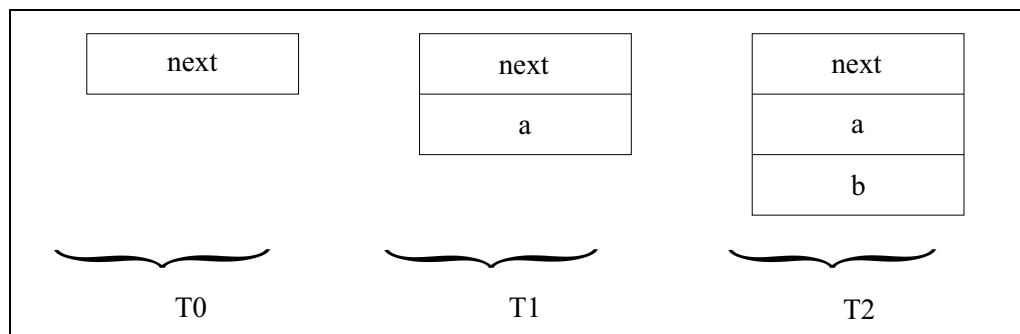


Abbildung 7.8: Mehrfache Typerweiterung

`p1.Name, s1.Vorname`

unzulässig:

`p1.MatNr`

(zu beachten: hier ist `p1` Recordvariable, keine Pointer-Variable!)

erlaubte Zuweisungen: `p1 := p2;`

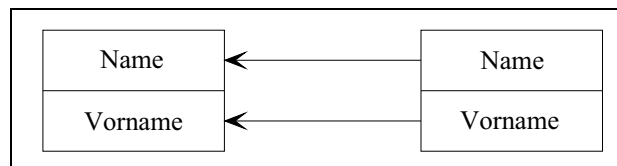


Abbildung 7.9: Typerweiterung und Zuweisung (1)

s. Abbildung 7.9

`p1 := s2;`

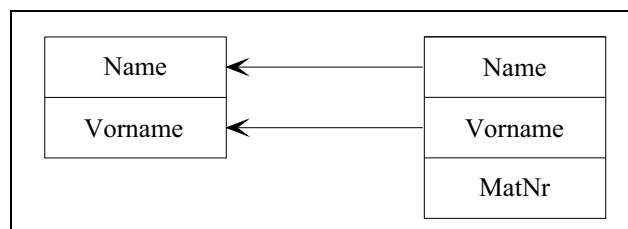


Abbildung 7.10: Typerweiterung und Zuweisung (2)

s. Abbildung 7.10

unzulässig: `s2 := p1;`

s. Abbildung 7.11

Typerweiterung und Zeigertypen

bisherige Beschreibung: wäre ausreichend bei ausschließlicher Verwendung von *Wertsemanantik*;

jetzt: *Referenzsemanantik* (wichtig u.a. in dynamischen Datenstrukturen):

Was geschieht mit Zeigern, die auf erweiterte Records zeigen?

Effekt von `NEW()` bei obiger Definition, also:

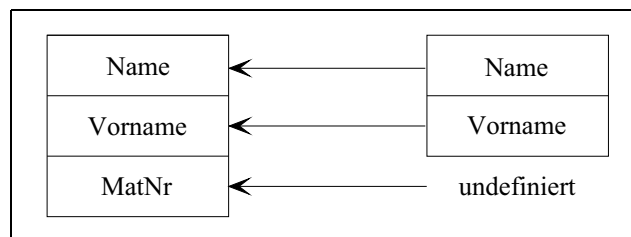


Abbildung 7.11: Typerweiterung und Zuweisung (3)

```

TYPE
  P0 = POINTER TO T0;
  T0 = RECORD
    next: P0
  END;
  P1 = POINTER TO T1;
  T1 = RECORD(T0)
    a: Data1
  END;
  P2 = POINTER TO T2;
  T2 = RECORD(T1)
    b: Data2
  END;
...
VAR
  p0: P0; p1: P1; p2: P2;
  ...
  NEW(p0); NEW(p1); NEW(p2);

```

Wirkung:

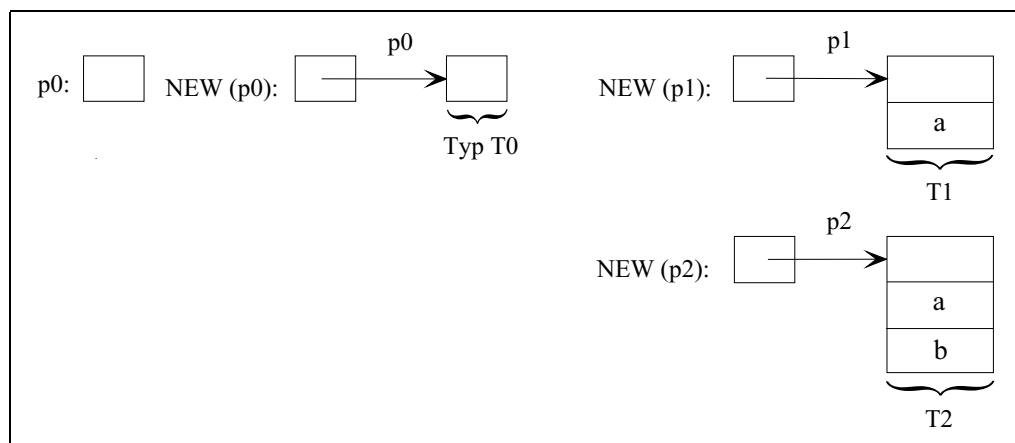


Abbildung 7.12: Instanz eines erweiterten Record-Typs

s. Abbildung 7.12

Der Begriff *Erweiterung* wird auf die Zeigertypen sinngemäß übertragen.
Zuweisungen von Zeigern?

```
...
p1 := p2;
```

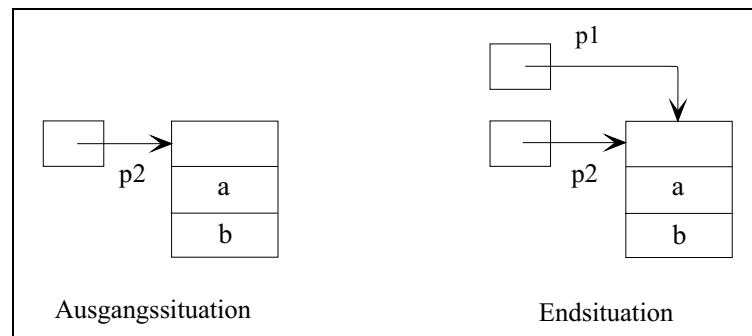


Abbildung 7.13: Zuweisung bei Zeigern verschiedener Typen

s. Abbildung 7.13

$p1$ zeigt auf dasselbe Objekt wie $p2$;

Frage: kann man auch über $p1$ auf die erweiterten Felder zugreifen?

dynamischer (oder tatsächlicher) Typ (im Unterschied zum deklarierten oder statischen Typ):

Record- und Zeigervariablen können neben ihrem *statischen* Typ (dem deklarierten Typ, vom Compiler zur Typüberprüfung verwendet) auch noch einen *dynamischen* Typ haben, der gleich dem statischen Typ oder einer Erweiterung davon ist.

dynamischer Typ: Typ eines Objektes zur Laufzeit: wird verwendet zur Auswahl von Prozeduren, die zu einem Record gehören (den sog. typgebundenen Prozeduren) (Einzelheiten später)

Zugriff auf die Felder des erweiterten Typs:

durch *Typzusicherung*: $p1 := p2$; $d2 := p1(P2).b$;

z.B.:

```
s := p(Student); k := p(Student).MatNr;
```

sofern sicher ist, daß p vom dynamischen Typ **Student** ist

Bsp.:

Person, **Student** = Erweiterung von **Person**:

```
VAR p1, p2: Person; s1, s2: Student;
```

Erzeugung von $s1^{\wedge}$ durch $NEW(s1)$: dyn. Typ: **StudentDesc**, $s1$: dyn. Typ **Student**

Zuweisung: $p1 := s1$; dyn. Typ von $p1$ ist **Student**, stat. Typ bleibt **Person**

Zuweisung $p1^{\wedge} := s1^{\wedge}$; nur Felder **Name**, **Vorname** werden kopiert, nicht **MatNr**

dyn. Typ von $p1^{\wedge}$ bleibt **PersonDesc**.

dyn. Typ kann zur Laufzeit geprüft werden:

```
IF p1 IS Student THEN ...
ELSE ...
END;
```

Boolescher Ausdruck `p1 IS Student` ergibt TRUE, wenn `p1` vom dynamischen Typ `Student` ist oder eine Erweiterung davon.

Wenn `p1`, `p2` tatsächlich (d.h. zur Laufzeit) vom dynamischen Typ `Student` sind, dann kann auch auf die Felder der Erweiterung zugegriffen werden:

z.B.:

```
s1 := p1(Student);
i  := p2(Student).MatNr;
p2(Student).MatNr := 9466123;
```

Laufzeitfehler vermeiden durch die Kombination:

```
IF p1 IS Student THEN Out.Int(p2(Student).MatNr, 12)
ELSE ...
END;
```

7.2.1 PROGRAMMBEISPIEL (TYPERWEITERUNG)

```
1  MODULE TypeExtension;
2  IMPORT Out;
3  CONST N = 20;
4  TYPE
5    Person = POINTER TO PersonDesc;
6    PersonDesc = RECORD
7      Name, Vorname: ARRAY N OF CHAR
8    END;
9    Student = POINTER TO StudentDesc;
10   StudentDesc = RECORD (PersonDesc)
11     MatNr: LONGINT
12   END;
13
14  PROCEDURE WrPerson(p: Person);
15    (* Daten von Person schreiben *)
16  BEGIN Out.String(p.Vorname); Out.String(" "); Out.String(p.Name)
17  END WrPerson;
18
19  PROCEDURE WrStudent(s: Student);
20    (* Daten von Student schreiben *)
21  BEGIN WrPerson(s); Out.Int(s.MatNr, 12)
22  END WrStudent;
23  (* zu beachten: der Wertparamter in WrPerson ist vom Typ
24     Person, der Aufruf mit einem Aktualparameter vom Typ
25     Student bedeutet in der Prozedur eine Zuweisung der Gestalt
26     p := s;
27     diese ist zulässig. Ein Aufruf der Gestalt
28     WrStudent(p);
29     wobei p vom Typ Person ist, wäre unzulässig. *)
30
31  PROCEDURE Test*;
32  VAR p1, p2, p3: Person; s1, s2, s3: Student;
```

```

33 BEGIN NEW(p1); NEW(p2); NEW(p3); NEW(s1); NEW(s2); NEW(s3);
34 (* zulässige Zuweisungen: *)
35   s1.Name := "Mair";
36   s1.Vorname := "Hugo";
37   s1.MatNr := 7865123;
38   s2 := s1; (* normale Zuweisung, gleicher Typ *)
39   p1 := s2; (* p1 hat jetzt den dynamischen Typ Student *)
40
41   (* Out.Int(p1.MatNr, 12); *)
42   (* unzulässig: Fehlermeldung: undefined record field *)
43
44   (* s1 := p1; *)
45   (* Fehler: incompatible assignment *)
46
47   WrPerson(p1); Out.Ln; WrStudent(s2); Out.Ln;
48
49   (* zulässig dagegen ist: *)
50   s3 := p1(Student); (* setzt aber voraus, daß p1 tatsächlich vom
51     dynamischen Typ Student ist *)
52
53   WrStudent(s3); Out.Ln;
54
55   (* ebenso ist zulässig: *)
56   Out.Int(p1(Student).MatNr, 12); Out.Ln; Out.Ln;
57
58   p3.Name := "Schmied"; p3.Vorname := "Helmut";
59
60   (* zu einem Laufzeitfehler dagegen führt:
61     s3 := p3(Student);
62     Trap 19 (implicit type guard check in record assignment) *)
63   (* denn: p3 hat nicht den dynamischen Typ Student *)
64
65   (* Typ kann getestet werden: *)
66   IF p1 IS Student THEN
67     Out.String(" ist Student, MatNr: ");
68     Out.Int(p1(Student).MatNr, 12); Out.Ln;
69   ELSE
70     Out.String(" kein Student "); Out.Ln;
71   END;
72
73   IF p3 IS Student THEN
74     Out.String(" ist Student, MatNr: ");
75     Out.Int(p3(Student).MatNr, 12); Out.Ln;
76   ELSE
77     Out.String(" kein Student "); Out.Ln;
78   END;
79

```

```

80   Out.Ln; Out.Ln
81   END Test;
82   END TypeExtension.
83   (* Beispielaufruf durch: *)
84   TypeExtension.Test ~
85   (* Ausgabe: *)
86   Hugo Mair
87   Hugo Mair    7865123
88   Hugo Mair    7865123
89   7865123
90
91   ist Student, MatNr:    7865123
92   kein Student

```

Typgebundene Prozeduren

Beispiel: Person/Student;

Prozeduren: WrPerson, WrStudent; erforderlich ist Fallunterscheidung, z.B.:

```

IF p1 IS IStudent THEN ...
  WrIStudent
ELSIF p1 IS EStudent THEN ...
  WrEStudent
ELSE ...
  WrPerson
END;

```

Zusammenziehen in eine Prozedur: geht nicht (d.h. nicht ohne weiteres), wenn IStudent, EStudent, ... in verschiedenen Modulen stehen.

Vereinfachung durch *typgebundene Prozeduren (Methoden)*:

Zur Laufzeit wird aufgrund des dynamischen Typs eines Objektes entschieden, welche Prozedur zu aktivieren ist. Der Aufruf selbst braucht auch nach einer späteren Erweiterung nicht geändert zu werden.

(*dynamische* oder *späte* Bindung im Gegensatz zur *statischen* oder *frühen* Bindung)

Frage: Wie wird im Programmtext eine Prozedur mit einem Record verknüpft?

durch neue Art von Parameter in Prozedur (dem *Empfänger, Receiver*):

Definition von Person, Student, wie oben:

Print einmal für Person und einmal für Student:

```

PROCEDURE (p: Person) Print;
BEGIN ...
END Print;

PROCEDURE (s: Student) Print;
BEGIN ...
END Print;

```

Es gilt:

1. Prozedur muß Level-0-Prozedur sein, also global

2. Empfänger-Parameter muß entweder vom

- Zeigertyp sein oder
- VAR-Parameter vom Record-Typ

z.B.:

```
PROCEDURE (VAR sd: StudentDesc) Print;
BEGIN ...
END Print;
```

3. Die Namen der (in diesem Fall: zwei) Prozedur-Varianten müssen gleich sein, ebenso eventuelle Parameterlisten.

4. Record und zugehörige Prozeduren müssen im gleichen Modul stehen.

5. Der Aufruf der Prozeduren lautet: `p.Print`; es wird dann das „richtige“ `Print` aufgrund des dynamischen Typs von `p` ausgewählt.

Anmerkung: `p` in `p.Print` wird behandelt wie ein Recordfeld: `p↑.Print`

7.2.2 PROGRAMMBEISPIEL ()

```
1  MODULE Persons;
2  IMPORT In, Out;
3  CONST L = 30;
4  TYPE
5    Person* = POINTER TO PersonDesc;
6    PersonDesc* = RECORD
7      Name*, FirstName*: ARRAY L OF CHAR;
8    END;
9
10  PROCEDURE Create*(VAR p: Person);
11  BEGIN NEW(p)
12  END Create;
13
14  PROCEDURE SetValues* (p: Person);
15  BEGIN In.String(p.Name); In.String(p.FirstName)
16  END SetValues;
17
18  PROCEDURE (p: Person) Print*;
19
20  (* damit Print außen sichtbar wird muß Person
21     und PersonDesc exportiert werden
22     Das gilt für Record-Felder generell: ein Record-Feld ist
23     sichtbar nur dann, wenn
24     1. das Feld selbst exportiert ist und
25     2. auch der Name des Record-Typs
26
27     Wenn das Record durch einen Zeiger referenziert wird,
```

```
28      dann muß auch der Name des Zeiger-Typs exportiert sein. *)
29
30      BEGIN Out.String(p.Name); Out.String(p.FirstName)
31      END Print;
32
33  END Persons.
34
35
36  MODULE Students;
37      IMPORT In, Out, P := Persons;
38      CONST f = 15;
39      TYPE
40          Student* = POINTER TO StudentDesc;
41          StudentDesc* = RECORD (P.PersonDesc)
42              RegNo: INTEGER
43          END;
44
45      PROCEDURE Create* (VAR s: Student);
46      BEGIN NEW(s)
47      END Create;
48
49      PROCEDURE SetValues* (s: Student);
50      BEGIN P.SetValues(s); In.Int(s.RegNo)
51      END SetValues;
52
53      PROCEDURE (s: Student) Print*;
54      BEGIN s.Print↑; Out.Int(s.RegNo, f);
55      END Print;
56
57  END Students.
58
59
60  MODULE Test;
61      IMPORT P := Persons, S := Students, In, Out;
62
63      PROCEDURE Start*;
64          VAR p1, p2: P.Person; s1, s2: S.Student;
65      BEGIN In.Open;
66          P.Create(p1); P.Create(p2);
67          P.SetValues(p1); P.SetValues(p2);
68          p1.Print; Out.Ln; p2.Print; Out.Ln;
69
70          S.Create(s1); S.Create(s2);
71          S.SetValues(s1); S.SetValues(s2);
72          s1.Print; Out.Ln; s2.Print; Out.Ln;
73
74      (* ein s1.Print↑ ist an dieser Stelle unzulässig: *)
```

```

75      (* „invalid call of base procedure“*)
76
77      p1 := s1; p2 := s2;
78      p1.Print; Out.Ln; p2.Print; Out.Ln;
79
80      (* syntaktisch erlaubt wäre hier auch: p1(S.Student).Print;
81         weiter oben würde der Compiler das auch erlauben, dort führt das aber
82         zum Laufzeitfehler! *)
83
84      (* Warum ist das hier nicht erforderlich, bzw.: es macht auch keinen Sinn!?
85         Für den Compiler ist p1.Print definiert, denn der statische Typ von p1 ist Person,
86         und zu diesem Typ gibt es eine typgebundene Prozedur Print. Damit ist der
87         Compiler zufrieden.
88
89         Bei p1.MatNr ist das anders: dieses Record-Feld ist für den Typ Person nicht
90         definiert. Die Typzusicherung bedeutet für den Compiler, daß der statische Typ
91         vorübergehend ausgeweitet wird, hier zu Student.
92
93         Das Laufzeitsystem wählt nun die richtige Prozedur entsprechend dem
94         dynamischen Typ aus.
95      *)
96      Out.Ln; Out.Ln
97  END Start;
98
99  END Test.
100
101  Beispielaufruf durch:
102  Test.Start
103  " Meier " " Hugo " " Schmidt " " Peter "
104  " Meier " " Hugo " 123 " Schmidt " " Peter " 456
105  Ausgabe:
106  Meier Hugo
107  Schmidt Peter
108  Meier Hugo      123
109  Schmidt Peter   456
110  Meier Hugo      123
111  Schmidt Peter   456
112
113  System.Free Test Students Persons ~

```

Weiteres Beispiel zum Überschreiben von Methoden

7.2.3 PROGRAMMBEISPIEL ()

```

1  MODULE Comparables;
2  TYPE
3    Comparable* = POINTER TO ComparDesc;
4    ComparDesc* = RECORD END;    (* leeres Record *)
5

```

```

6  PROCEDURE (x: Comparable) LessThan*(y: Comparable): BOOLEAN;
7  BEGIN HALT(99)
8  END LessThan;  (* abstrakte Prozedur *)
9  (* HALT() entspricht der Empfehlung von Mössenböck *)
10
11  PROCEDURE (x: Comparable) GreaterOrEqual* (y: Comparable): BOOLEAN;
12  BEGIN RETURN ~x.LessThan(y)
13  END GreaterOrEqual;
14
15  (* ... und weitere Prozeduren, die <=, =, > nachbilden. *)
16
17  END Comparables.
18
19
20  MODULE IntObjs;
21  IMPORT Comparables,      (* ... und nur zum Testen: *) Out;
22  TYPE
23    IntObj* = POINTER TO IntObjDesc;
24    IntObjDesc* = RECORD(Comparables.ComparDesc)
25      i: INTEGER
26    END;
27
28  (* Addition ueber reine Referenzsemantik: *)
29  PROCEDURE (x: IntObj) Add*(y: IntObj): IntObj;
30    VAR w: IntObj;
31  BEGIN NEW(w); w.i := x.i + y.i; RETURN w
32  END Add;
33
34  (* Überschreiben der "Methode" LessThan:
35   zu beachten: Parameterliste der überschreibenden Prozedur
36   muß gleich sein zu der überschriebenen Prozedur *)
37
38  PROCEDURE (x: IntObj) LessThan*(y: Comparables.Comparable): BOOLEAN;
39  BEGIN RETURN x.i < y(IntObj).i
40  END LessThan;
41
42  PROCEDURE Test*;
43    VAR x, y, z: IntObj;
44  BEGIN NEW(x); NEW(y);
45    x.i := 4; y.i := 5;
46    z := x.Add(y);      (* so wird addiert per Referenzsemantik *)
47    Out.Int(z.i, 7); Out.Ln;
48
49  (* ... und Aufruf der vererbten Methode GreaterThan: *)
50  IF (x IS IntObj) & y.GreaterOrEqual(x) THEN
51    Out.String(" ... ist groesser oder gleich ..."); Out.Ln
52  END

```

```
53   END Test;
54
55   END IntObjs.
56   Beispielaufruf durch:
57   IntObjs.Test ~
58   Ausgabe:
59       9
60   ... ist groesser oder gleich ...
61
62   System.Free IntObjs Comparables ~
63
64
65   Bemerkung:
66   Typgebundene Prozeduren sind i.a. ungeeignet zur Initialisierung
67   von Objekten, denn: bei Initialisierung sind Parameterlisten in aller
68   Regel nicht gleich.
69
70
71   Schnittstellen:
72   DEFINITION Comparables;
73
74       TYPE
75           Comparable = POINTER TO ComparDesc;
76           ComparDesc = RECORD
77               PROCEDURE (x: Comparable) GreaterOrEqual (y: Comparable): BOOLEAN;
78               PROCEDURE (x: Comparable) LessThan (y: Comparable): BOOLEAN;
79           END;
80
81   END Comparables.
82
83   DEFINITION IntObjs;
84
85       IMPORT Comparables;
86
87       TYPE
88           IntObj = POINTER TO IntObjDesc;
89           IntObjDesc = RECORD (Comparables.ComparDesc)
90               PROCEDURE (x: IntObj) Add (y: IntObj): IntObj;
91               PROCEDURE (x: IntObj) LessThan (y: Comparables.Comparable): BOOLEAN;
92           END;
93
94       PROCEDURE Test;
95
96   END IntObjs.
```

Demonstration von Supercall

7.2.4 PROGRAMMBEISPIEL ()

```
1  MODULE TestSuperCall;
2  IMPORT Out;
3
4  TYPE
5    Object0* = POINTER TO ObjectDesc0;
6    ObjectDesc0* = RECORD
7      END;
8
9    Object1* = POINTER TO ObjectDesc1;
10   ObjectDesc1* = RECORD (ObjectDesc0)
11     x: INTEGER
12   END;
13
14   Object2* = POINTER TO ObjectDesc2;
15   ObjectDesc2* = RECORD (ObjectDesc1)
16     y: INTEGER
17   END;
18
19   Object3* = POINTER TO ObjectDesc3;
20   ObjectDesc3* = RECORD (ObjectDesc2)
21     z: INTEGER
22   END;
23
24   PROCEDURE (p: Object0) Print*;
25   BEGIN Out.String(" Object is: ");
26   END Print;
27
28   PROCEDURE (p: Object1) Print*;
29   BEGIN p.Print↑; Out.Int(p.x, 10)
30   END Print;
31
32   PROCEDURE (p: Object2) Print*;
33   BEGIN p.Print↑; Out.Int(p.y, 10)
34   END Print;
35
36   PROCEDURE (p: Object3) Print*;
37   BEGIN p.Print↑; Out.Int(p.z, 10)
38   END Print;
39
40   PROCEDURE Start*;
41   VAR p0: Object0; p1: Object1; p2: Object2; p3: Object3;
42   BEGIN NEW(p0); NEW(p1); NEW(p2); NEW(p3);
43
44   p0.Print; Out.Ln;
45
46   p1.x := 123;
```

```

47     p1.Print; Out.Ln;
48
49     p2.x := 234; p2.y := 345;
50     p2.Print; Out.Ln;
51
52     p3.x := 456; p3.y := 567; p3.z := 678;
53     p3.Print; Out.Ln;
54
55     Out.Ln;
56     END Start;
57
58 END TestSuperCall.
59
60 Beispielaufruf durch:
61 TestSuperCall.Start ~
62 Ausgabe:
63 Object is:
64 Object is:      123
65 Object is:      234      345
66 Object is:      456      567      678
67
68 Schnittstelle:
69 DEFINITION TestSuperCall;
70
71     TYPE
72         Object0 = POINTER TO ObjectDesc0;
73         ObjectDesc0 = RECORD
74             PROCEDURE (p: Object0) Print;
75         END;
76         Object1 = POINTER TO ObjectDesc1;
77         ObjectDesc1 = RECORD (ObjectDesc0)
78             PROCEDURE (p: Object1) Print;
79         END;
80         Object2 = POINTER TO ObjectDesc2;
81         ObjectDesc2 = RECORD (ObjectDesc1)
82             PROCEDURE (p: Object2) Print;
83         END;
84         Object3 = POINTER TO ObjectDesc3;
85         ObjectDesc3 = RECORD (ObjectDesc2)
86             PROCEDURE (p: Object3) Print;
87         END;
88
89     PROCEDURE Start;
90
91 END TestSuperCall.

```

7.3 Zusammenspiel der OO-Konzepte: Beispiele

Heterogenes Array

7.3.1 PROGRAMMBEISPIEL ()

```
1  MODULE GenericArrays;
2  (* Demonstration der Funktionsweise eines heterogenen Arrays:
3     heterogen bedeutet, daß das Array (in einem gewissen Sinne)
4     Objekte verschiedenen Typs aufnehmen kann. *)
5
6  IMPORT Out;
7  CONST N = 10;
8  TYPE
9     Object* = POINTER TO ObjectDesc;
10    ObjectDesc* = RECORD
11        END;
12    Array* = ARRAY N OF Object;
13
14  PROCEDURE (p: Object) Print*;
15  BEGIN HALT(99)
16  END Print;
17
18  PROCEDURE Put*(VAR A: Array; i: INTEGER; p: Object);
19  (* Füge in A an der Stelle mit Index i das Objekt p ein. *)
20  BEGIN A[i] := p
21  END Put;
22
23  PROCEDURE Get*(A: Array; i: INTEGER): Object;
24  (* Liefere aus A das Objekt an der Stelle mit Index i *)
25  BEGIN RETURN A[i]
26  END Get;
27
28  PROCEDURE NumberOfObjects*(A: Array): INTEGER;
29  (* Liefere die Gesamtzahl an Objekten aus A *)
30  VAR i, k: INTEGER;
31  BEGIN i := 0; k := 0;
32  WHILE i <= N-1 DO
33      IF A[i] # NIL THEN INC(k) END; INC(i)
34  END;
35  RETURN k
36  END NumberOfObjects;
37
38  PROCEDURE ShowArray*(A: Array);
39  (* Drucke Array A vollständig; bei dieser Realisierung ist zu
40     berücksichtigen, daß Arraykomponenten auch leer sein können. *)
41  CONST fmt = 5;
42  VAR i: INTEGER;
43  BEGIN
```

```

44   Out.String("Gesamtzahl Objekte: "); Out.Int(NumberOfObjects(A), 12); Out.Ln;
45   Out.Ln; Out.Ln;
46   i := 0;
47   WHILE i <= N-1 DO (* For-Schleife wäre hier ebenso möglich *)
48     Out.String("Index Nr. "); Out.Int(i, fmt);
49     IF A[i] # NIL THEN Out.String(": "); A[i].Print; Out.Ln
50     ELSE Out.String(": kein Objekt"); Out.Ln
51     END;
52     INC(i)
53   END
54   END ShowArray;
55
56   END GenericArrays.
57
58
59   MODULE ConcreteObjects;
60     IMPORT Out, G := GenericArrays;
61
62     TYPE
63       IntObject* = POINTER TO IntObjectDesc;
64       IntObjectDesc* = RECORD (G.ObjectDesc)
65         i*: INTEGER
66       END;
67
68       CharObject* = POINTER TO CharObjectDesc;
69       CharObjectDesc* = RECORD (G.ObjectDesc)
70         c*: CHAR
71       END;
72
73     PROCEDURE (I: IntObject) Print*; (* Überschreiben setzt gleiche Namen voraus *)
74     BEGIN Out.Int(I.i, 12)
75     END Print;
76
77     PROCEDURE (C: CharObject) Print*;
78     BEGIN Out.Char(C.c)
79     END Print;
80
81   END ConcreteObjects.
82
83
84   MODULE Test;
85     IMPORT G := GenericArrays, C := ConcreteObjects, Out;
86     VAR A: G.Array;
87
88     PROCEDURE Start*;
89     (* Im folgenden wird nun demonstriert, daß man in ein und dasselbe
90      Array Objekte verschiedenen Typs einfügen kann, genauer:

```

```

91      Das Array enthält Referenzen, die auf die Objekte gesetzt werden;
92      es werden keine Daten umkopiert.*)
93
94      VAR p: C.IntObject; q: C.CharObject; o: G.Object;
95      BEGIN
96          NEW(p); NEW(q); p.i := 123; q.c := 'A';
97          G.Put(A, 3, p); G.Put(A, 5, q);
98
99          (* Die folgende Zuweisung würde auch die bestehenden Objekte
100             im Array ändern!! *)
101          (* p.i := 126; q.c := 'C'; *)
102
103          (* um neue (weitere) Objekte einzufügen: *)
104          NEW(p); NEW(q); p.i := 126; q.c := 'C';
105          G.Put(A, 7, p); G.Put(A, 1, q);
106
107          (* In der Prozedur G.ShowArray wird Print aufgerufen.
108             Das Laufzeitsystem wählt aufgrund des aktuellen
109             dynamischen Typs der jeweiligen Arraykomponente
110             die „richtige“ Prozedur Print aus. *)
111
112          G.ShowArray(A); Out.Ln; Out.Ln
113      END Start;
114
115  END Test.
116
117  Beispielaufruf durch:
118  Test.Start ~
119  Ausgabe:
120
121  Index Nr.    0: kein Objekt
122  Index Nr.    1: C
123  Index Nr.    2: kein Objekt
124  Index Nr.    3:      123
125  Index Nr.    4: kein Objekt
126  Index Nr.    5: A
127  Index Nr.    6: kein Objekt
128  Index Nr.    7:      126
129  Index Nr.    8: kein Objekt
130  Index Nr.    9: kein Objekt
131
132  System.Free Test ConcreteObjects GenericArrays ~
133
134  Die Schnittstellen:
135  DEFINITION GenericArrays;
136
137  TYPE

```

```

138     Array = ARRAY 10 OF Object;
139     Object = POINTER TO ObjectDesc;
140     ObjectDesc = RECORD
141         PROCEDURE (p: Object) Print;
142     END;
143
144     PROCEDURE Get (A: Array; i: INTEGER): Object;
145     PROCEDURE NumberOfObjects (A: Array): INTEGER;
146     PROCEDURE Put (VAR A: Array; i: INTEGER; p: Object);
147     PROCEDURE ShowArray (A: Array);
148
149 END GenericArrays.
150
151 DEFINITION ConcreteObjects;
152
153     IMPORT GenericArrays;
154
155     TYPE
156         CharObject = POINTER TO CharObjectDesc;
157         CharObjectDesc = RECORD (GenericArrays.ObjectDesc)
158             c: CHAR;
159             PROCEDURE (C: CharObject) Print;
160         END;
161         IntObject = POINTER TO IntObjectDesc;
162         IntObjectDesc = RECORD (GenericArrays.ObjectDesc)
163             i: INTEGER;
164             PROCEDURE (I: IntObject) Print;
165         END;
166
167 END ConcreteObjects.
168
169 DEFINITION Test;
170
171     PROCEDURE Start;
172
173 END Test.

```

Generische lineare Liste

(nach Reiser, Wirth)

7.3.2 PROGRAMMBEISPIEL ()

```

1  MODULE Queues;
2  (* adapted from [RW92, p. 247] *)
3  IMPORT Out;
4  TYPE
5      Item* = POINTER TO ItemDesc;
6      ItemDesc* = RECORD

```

```
7      next: Item
8      END;
9
10     Queue* = POINTER TO QueueDesc;
11     QueueDesc* = RECORD
12         first: Item
13     END;
14
15     FIFO* = POINTER TO FIFODesc;
16     FIFODesc* = RECORD(QueueDesc)
17         last: Item
18     END;
19
20     LIFO* = POINTER TO LIFODesc;
21     LIFODesc* = RECORD(QueueDesc)
22     END;
23
24     Ranked* = POINTER TO RankedDesc;
25     RankedDesc* = RECORD(QueueDesc)
26     END;
27
28     PROCEDURE (i: Item) EqualTo*(j: Item): BOOLEAN;
29     BEGIN HALT(21)
30     END EqualTo;
31
32     PROCEDURE (i: Item) LessThan*(j: Item): BOOLEAN;
33     BEGIN HALT(22)
34     END LessThan;
35
36     PROCEDURE (q: Queue) Enqueue*(i: Item);
37     BEGIN HALT(23)
38     END Enqueue;
39
40     PROCEDURE (q: Queue) DequeuedItem*(): Item;
41     VAR x: Item;
42     BEGIN
43         x := q.first;
44         IF x # NIL THEN q.first := x.next END;
45         RETURN x
46     END DequeuedItem;
47
48     PROCEDURE (q: FIFO) Enqueue*(i: Item);
49     BEGIN
50         i.next := NIL;
51         IF q.first # NIL THEN q.last.next := i
52         ELSE q.first := i
53         END;
```

```
54     q.last := i
55   END Enqueue;
56
57   PROCEDURE (q: LIFO) Enqueue*(i:Item);
58   BEGIN i.next := q.first; q.first := i
59   END Enqueue;
60
61   PROCEDURE (q: Ranked) Enqueue*(i: Item);
62     VAR x: Item;
63   BEGIN
64     IF (q.first = NIL) OR i.LessThan(q.first) THEN
65       i.next := q.first; q.first := i
66     ELSE
67       x := q.first;
68       WHILE (x.next # NIL) & ~i.LessThan(x.next) DO
69         x := x.next
70       END;
71       i.next := x.next; x.next := i
72     END
73   END Enqueue;
74
75   PROCEDURE (q: Queue) ForAll*(P: PROCEDURE(i: Item));
76     VAR x: Item;
77   BEGIN x := q.first;
78     WHILE x # NIL DO P(x); x := x.next END
79   END ForAll;
80
81   PROCEDURE (q: Queue) Empty*(): BOOLEAN;
82   BEGIN RETURN q.first = NIL
83   END Empty;
84
85   PROCEDURE (q: Queue) Open*;
86   BEGIN q.first := NIL
87   END Open;
88
89   END Queues.
90
91
92   MODULE TestQueues;
93   (* some typical operations on queues *)
94   IMPORT Out, Q := Queues;
95   CONST N = 15;
96   TYPE
97     Item = POINTER TO ItemDesc;
98     ItemDesc = RECORD (Q.ItemDesc)
99       key: INTEGER
100   END;
```

```
101
102  PROCEDURE Show*(i: Q.Item);
103  BEGIN Out.Int(i(Item).key, 9); Out.Ln
104  END Show;
105
106  PROCEDURE (i: Item) EqualTo*(j: Q.Item): BOOLEAN;
107  BEGIN RETURN i.key = j(Item).key
108  END EqualTo;
109
110  PROCEDURE (i: Item) LessThan*(j: Q.Item): BOOLEAN;
111  BEGIN RETURN i.key < j(Item).key
112  END LessThan;
113
114  PROCEDURE g(k: INTEGER): INTEGER;
115  (* create key values for testing various queue types *)
116  BEGIN RETURN k*k * (N - k - 2)
117  END g;
118
119  PROCEDURE Start*;
120  VAR k: INTEGER;
121  f: Q.FIFO; l: Q.LIFO; r: Q.Ranked; i: Item; x: Q.Item;
122  BEGIN
123  NEW(f); NEW(l); NEW(r); f.Open; l.Open; r.Open;
124
125  FOR k := 3 TO N-1 DO (* willkürlicher Beispielwert 3 *)
126  NEW(i); i.key := g(k); f.Enqueue(i)
127  END;
128  f.ForAll(Show); Out.Ln;
129
130  FOR k := 3 TO N-1 DO
131  NEW(i); i.key := g(k); l.Enqueue(i)
132  END;
133  l.ForAll(Show); Out.Ln;
134
135  FOR k := 3 TO N-1 DO
136  NEW(i); i.key := g(k); r.Enqueue(i)
137  END;
138  r.ForAll(Show); Out.Ln;
139
140  x := f.DequeuedItem(); Show(x);
141  x := l.DequeuedItem(); Show(x);
142  x := r.DequeuedItem(); Show(x);
143
144  Out.String("FIFO: "); Out.Ln; f.ForAll(Show); Out.Ln;
145  Out.String("LIFO: "); Out.Ln; l.ForAll(Show); Out.Ln;
146  Out.String("Ranked: "); Out.Ln; r.ForAll(Show); Out.Ln;
147
```

```
148   END Start;
149
150   END TestQueues.
151
```

Generischer binärer Baum

(nach Mössenböck)

7.3.3 PROGRAMMBEISPIEL ()

```
1
2  MODULE Trees;
3  (* vgl. Mössenböck, p. 77 *)
4
5  TYPE
6    Node* = POINTER TO NodeDesc;
7    NodeDesc* = RECORD
8      left, right: Node
9    END;
10   Tree* = RECORD
11     root: Node
12   END;
13
14   PROCEDURE (x: Node) LessThan*(y: Node): BOOLEAN;
15   BEGIN HALT(97)
16   END LessThan;
17
18   PROCEDURE (x: Node) EqualTo*(y: Node): BOOLEAN;
19   BEGIN HALT(98)
20   END EqualTo;
21
22   PROCEDURE (x: Node) Print*;
23   BEGIN HALT(99)
24   END Print;
25
26   PROCEDURE (x: Node) GreaterEqual*(y: Node): BOOLEAN;
27   BEGIN RETURN ~x.LessThan(y)
28   END GreaterEqual;
29
30   PROCEDURE (x: Node) GreaterThan*(y: Node): BOOLEAN;
31   BEGIN RETURN x.GreaterEqual(y) & ~x.EqualTo(y)
32   END GreaterThan;
33
34   PROCEDURE (VAR t: Tree) Insert* (x: Node);
35   VAR this, parent: Node;
36   BEGIN
37     this := t.root; x.left := NIL; x.right := NIL;
38     WHILE this # NIL DO
```

```
39     parent := this;
40     IF x.EqualTo(this) THEN RETURN END; (* keine Duplikate *)
41     IF x.LessThan(this) THEN this := this.left
42     ELSE this := this.right
43     END
44 END;
45 IF t.root = NIL THEN t.root := x
46 ELSIF x.LessThan(parent) THEN parent.left := x
47 ELSE parent.right := x END
48 END Insert;
49
50 PROCEDURE (VAR t: Tree) Search* (x: Node);
51 BEGIN
52 (* ... *)
53 END Search;
54
55 PROCEDURE (VAR t: Tree) Delete* (x: Node);
56 BEGIN
57 (* ... *)
58 END Delete;
59
60 PROCEDURE (VAR t: Tree) ShowTree*;
61   VAR this: Node; s: Tree; (* lokale Hilfsvariable: t ist VAR-Parameter! *)
62 BEGIN s := t; this := s.root;
63   IF this # NIL THEN
64     s.root := this.left; s.ShowTree;
65     this.Print;
66     s.root := this.right; s.ShowTree;
67   END
68 END ShowTree;
69
70 END Trees.
71
72
73 MODULE Test;
74   IMPORT Out, T := Trees;
75
76   TYPE
77     IntObject = POINTER TO IntObjectDesc;
78     IntObjectDesc = RECORD (T.NodeDesc)
79       i: INTEGER
80     END;
81
82   PROCEDURE (x: IntObject) LessThan*(y: T.Node): BOOLEAN;
83   BEGIN RETURN x.i < y(IntObject).i
84   END LessThan;
85
```

```

86  PROCEDURE (x: IntObject) EqualTo*(y: T.Node): BOOLEAN;
87  BEGIN RETURN x.i = y(IntObject).i
88  END EqualTo;
89
90  PROCEDURE (x: IntObject) Print*;
91  BEGIN Out.Int(x.i, 12); Out.Ln
92  END Print;
93
94  PROCEDURE Start*;
95  VAR I, J: IntObject; t: T.Tree; x: T.Node;
96  BEGIN
97    NEW(I); I.i := 10; t.Insert(I);
98    NEW(I); I.i := 9; t.Insert(I);
99    NEW(I); I.i := 11; t.Insert(I);
100   NEW(I); I.i := 5; t.Insert(I);
101   NEW(I); I.i := 13; t.Insert(I);
102   NEW(I); I.i := 12; t.Insert(I);
103   NEW(I); I.i := 15; t.Insert(I);
104   NEW(I); I.i := 23; t.Insert(I);
105   NEW(I); I.i := 23; t.Insert(I);
106
107   t.ShowTree; Out.Ln; Out.Ln;
108
109   NEW(I); I.i := 119;
110   NEW(J); J.i := 125; x := J;
111
112   (* Testen der Prozedur GreaterEqual: Vor.: x und I # NIL *)
113   IF J IS IntObject THEN
114     IF I.GreaterEqual(x) THEN
115       Out.String(" ... ist groesser oder gleich. ")
116     ELSE
117       Out.String(" ... ist kleiner. ")
118     END
119   END;
120
121   Out.Ln; Out.Ln
122  END Start;
123
124  END Test.
125
126  Beispielaufruf durch:
127  Test.Start ~
128  Ausgabe:
129      5
130      9
131     10
132     11

```

133 12
134 13
135 15
136 23
137
138
139 ... ist kleiner.
140
141 System.Free Test Trees ~
142
143

Kapitel 8

Erweiterbare Systeme

Diskussion: Vergleich mit anderen Sprachen: was muß man machen, um ein bestehendes Softwaresystem zu erweitern? Geht das dynamisch? Wo könnten entsprechende Anwendungen liegen?

8.1 Up-Calls

Eines der technischen Hilfsmittel in Oberon, um erweiterbare Systeme zu realisieren, ist der Up-Call.

Wenn eine Prozedur D, die einem Modul M0 definiert ist, aus einem Modul M1 heraus aufgerufen werden soll, dann muß M0 nach M1 importiert werden; der Aufruf von M1 aus lautet dann M0.D. Eine solche Aktivierung einer Prozedur könnte man sinnvoll auch als *Down-Call* bezeichnen.

Ein Up-Call realisiert nun eine Prozeduraktivierung in der umgekehrten Richtung, was auf den ersten Blick unmöglich erscheint.

Beispiel für Up-Call (s. Abbildung 8.1)

8.1.1 PROGRAMMBEISPIEL (UP-CALL)

```
1  MODULE M1;
2    IMPORT M0, Out;
3
4    PROCEDURE U*(i: INTEGER): INTEGER;
5      (* to be called from M0 *)
6    BEGIN RETURN i*i;
7    END U;
8
9    PROCEDURE Set*;
10   BEGIN M0.P := U
11   END Set;
12
13   PROCEDURE Start*;
14     VAR k: INTEGER;
15   BEGIN
16     k := M0.D(5); Out.Int(k, 9); Out.Ln;
17   END Start;
```

```

18
19  END M1.
20  MODULE M0;
21    IMPORT Out;
22
23    VAR P*: PROCEDURE(i: INTEGER): INTEGER;
24
25    PROCEDURE D*(i: INTEGER): INTEGER;
26    BEGIN RETURN i*i
27    END D;
28
29    PROCEDURE Start*;
30      VAR k: INTEGER;
31    BEGIN
32      k := P(3); Out.Int(k, 9); Out.Ln
33    END Start;
34
35  END M0.
36
37  M1.Set ~
38  M0.Start ~
39  M1.Start ~
40

```

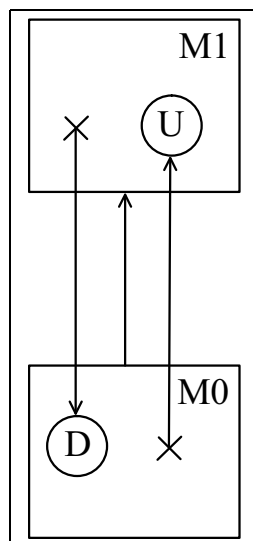


Abbildung 8.1: Up-Call und Down-Call

8.2 Simulationspaket nach Reiser/Wirth ([RW92])

Importbeziehungen der Module des Simulationspaketes, s. Abbildung 8.2

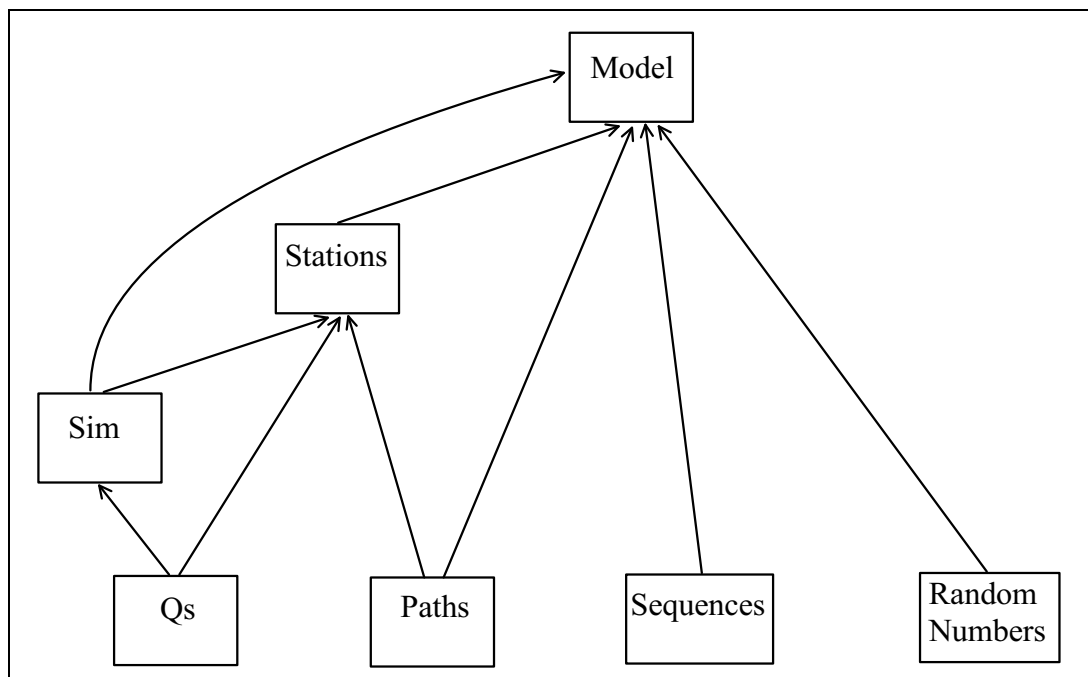


Abbildung 8.2: Importbeziehungen der Module des Simulationspaketes

8.2.1 PROGRAMMBEISPIEL ()

```

1  MODULE Model;
2  (* s. RW, S. 260 *)
3  IMPORT Sim, Stations, Paths, Sequences, RandomNumbers, In, Out;
4  CONST fmt = 18;
5
6  TYPE
7    Customer = POINTER TO CustomerDesc;
8    CustomerDesc = RECORD (Sim.ActorDesc)
9      ts: REAL          (* Time stamp of arrival epoch *)
10   END;
11
12  VAR (* Global state variables *)
13    lambda, mu: REAL;
14    s: Stations.Station;
15    srce: Sim.Actor;
16    w: Sequences.Sequence;
17
18  PROCEDURE HandleCust(cust: Sim.Actor; VAR msg: Sim.Message);
19  BEGIN
20    WITH cust: Customer DO
21      IF msg IS Stations.BeginMsg THEN
22        Sim.Schedule(cust, Sim.time + RandomNumbers.Exp(mu))
23      ELSIF msg IS Sim.CalendarMsg THEN
24        Stations.Free(s); Sequences.Add(w, Sim.time - cust.ts)

```

```

25     END
26     END
27 END HandleCust;
28
29 PROCEDURE HandleSrce(srce: Sim.Actor; VAR msg: Sim.Message);
30     VAR c: Customer;
31 BEGIN
32     NEW(c); c.handle := HandleCust; c.ts := Sim.time;
33     Stations.Request(s, c);
34     Sim.Schedule(srce, Sim.time + RandomNumbers.Exp(lambda))
35 END HandleSrce;
36
37 PROCEDURE Setup*;
38     VAR c: Customer;
39 BEGIN
40
41     In.Open; Sim.Reset; Sequences.Open(w); Out.Open;
42
43     NEW(srce); srce.handle := HandleSrce;
44     In.Real(lambda); Out.String("lambda ="); Out.Real(lambda, fmt);
45     Sim.Schedule(srce, 0);
46     NEW(s); Stations.Open(s);
47     In.Real(mu); Out.String("    mu ="); Out.Real(mu, fmt); Out.Ln
48 END Setup;
49
50 PROCEDURE Run*;
51     VAR dt: REAL;
52 BEGIN
53     In.Open; In.Real(dt);
54     Sim.Simulate(dt);
55     Out.String(" Sim.time ="); Out.Real(Sim.time, fmt);
56     Out.String("    E[W] ="); Out.Real(Sequences.Mean(w), fmt);
57     Out.String("    var[W] ="); Out.Real(Sequences.Var(w), fmt);
58     Out.Ln
59 END Run;
60
61 END Model.
62
63
64 Model.Setup      5.0 10.0 ~ (* Eingabewerte:  $\lambda$   $\mu$  *)
65 Model.Run       1000.0 ~ (* (Eingabewert: dt) *)
66
67 1 MODULE Paths;
68 2 (* s. RW, S. 169 *)
69 3
70 4 TYPE Path* = RECORD
71 5     n*: INTEGER;
72 6     W, t: REAL;

```

```

7   END;
8
9   PROCEDURE Init*(VAR p: Path);
10  BEGIN p.W := 0; p.n := 0; p.t := 0
11  END Init;
12
13  PROCEDURE Up*(VAR p: Path; t: REAL);
14  BEGIN p.W := p.W + p.n*(t - p.t); INC(p.n); p.t := t
15  END Up;
16
17  PROCEDURE Down*(VAR p: Path; t: REAL);
18  BEGIN p.W := p.W + p.n*(t - p.t); DEC(p.n); p.t := t
19  END Down;
20
21  PROCEDURE Mean*(p: Path; tEnd: REAL): REAL;
22  BEGIN RETURN (p.W + p.n*(tEnd - p.t))/tEnd
23  END Mean;
24
25  BEGIN END Paths.

1  MODULE Qs;
2  (* s. Reiser/Wirth, S. 247 *)
3  TYPE
4    Key* = REAL;
5    Item* = POINTER TO ItemDesc;
6    ItemDesc* = RECORD
7      key*: Key;
8      next: Item
9    END;
10   Queue* = POINTER TO QueueDesc;
11   QueueDesc* = RECORD
12     first: Item
13   END;
14
15   FIFO* = POINTER TO FIFODesc;
16   FIFODesc* = RECORD(QueueDesc)
17     last: Item
18   END;
19
20   LIFO* = POINTER TO LIFODesc;
21   LIFODesc* = RECORD(QueueDesc) END;
22
23   Ranked* = POINTER TO RankedDesc;
24   RankedDesc* = RECORD(QueueDesc) END;
25
26   PROCEDURE InsertFIFO(q: FIFO; i: Item);
27   BEGIN
28     i.next := NIL;

```

```
29   IF q.first # NIL THEN q.last.next := i
30   ELSE q.first := i
31   END;
32   q.last := i;
33   END InsertFIFO;
34
35   PROCEDURE InsertLIFO(q: LIFO; i:Item);
36   BEGIN i.next := q.first; q.first := i END InsertLIFO;
37
38   PROCEDURE InsertRanked(q: Ranked; i: Item);
39   VAR x: Item;
40   BEGIN
41   IF (q.first = NIL) OR (i.key < q.first.key) THEN
42     i.next := q.first; q.first := i
43   ELSE
44     x := q.first;
45     WHILE (x.next # NIL) & (i.key >= x.next.key) DO
46       x := x.next
47     END;
48     i.next := x.next; x.next := i
49   END
50   END InsertRanked;
51
52   PROCEDURE Enqueue*(q: Queue; i: Item);
53   VAR x: Item;
54   BEGIN
55   IF q IS FIFO THEN InsertFIFO(q(FIFO), i)
56   ELSIF q IS LIFO THEN InsertLIFO(q(LIFO), i)
57   ELSIF q IS Ranked THEN InsertRanked(q(Ranked), i)
58   END
59   END Enqueue;
60
61   PROCEDURE DequeuedItem*(q: Queue): Item;
62   VAR x: Item;
63   BEGIN
64   x := q.first;
65   IF x # NIL THEN q.first := x.next END;
66   RETURN x
67   END DequeuedItem;
68
69   PROCEDURE Enumerate*(q: Queue; P: PROCEDURE(i: Item));
70   VAR x: Item;
71   BEGIN x := q.first; WHILE x # NIL DO P(x); x := x.next END
72   END Enumerate;
73
74   PROCEDURE Empty*(q: Queue): BOOLEAN;
75   BEGIN RETURN q.first = NIL
```

```

76   END Empty;
77
78   PROCEDURE Open*(q: Queue);
79   BEGIN q.first := NIL
80   END Open;
81
82   END Qs.

1   MODULE Sequences;
2   (* computes mean and variance of a sequence of sample values
3     Exercise 10.2, page 182 *)
4
5   TYPE
6     Sequence* = RECORD
7       X: REAL;
8       S2: REAL;
9       n: LONGINT
10    END;
11
12   PROCEDURE Open*(VAR s: Sequence);
13   BEGIN
14     s.X := 0; s.S2 := 0; s.n := 0
15   END Open;
16
17   PROCEDURE Add*(VAR s: Sequence; x: REAL);
18   BEGIN
19     INC(s.n);
20     IF s.n > 1 THEN
21       s.S2 := s.S2*(s.n - 2)/(s.n - 1) + (x - s.X)*(x - s.X)/s.n
22     END;
23     s.X := s.X + (x - s.X)/s.n
24   END Add;
25
26   PROCEDURE Mean*(s: Sequence): REAL;
27   BEGIN
28     RETURN s.X
29   END Mean;
30
31   PROCEDURE Var*(s: Sequence): REAL;
32   BEGIN
33     RETURN s.S2
34   END Var;
35
36   END Sequences. (* Copyright M. Reiser, 1992 *)

1   MODULE Sim;
2   (* s. RW; S. 252 *)
3   IMPORT Qs, Out;

```

```

4
5  TYPE
6    Actor* = POINTER TO ActorDesc;
7    Message* = RECORD END;
8    CalendarMsg* = RECORD(Message) END;
9    Handler* = PROCEDURE(a: Actor; VAR msg: Message);
10   ActorDesc* = RECORD(Qs.ItemDesc)
11     handle*: Handler
12   END;
13  VAR
14    time*: REAL;          (* Global simulation time *)
15    clndr: Qs.Ranked;    (* Abstract data structure *)
16
17  PROCEDURE Schedule*(a: Actor; t: REAL);
18  BEGIN a.key := t; Qs.Enqueue(clndr, a)
19  END Schedule;
20
21  PROCEDURE Simulate*(dt: REAL);
22  VAR
23    cur: Qs.Item;
24    msg: CalendarMsg;
25    tEnd: REAL;
26  BEGIN
27    tEnd := time + dt;
28    LOOP
29      IF ~Qs.Empty(clndr) THEN
30        cur := Qs.DequeuedItem(clndr);
31        WITH cur: Actor DO
32          time := cur.key; cur.handle(cur, msg)
33        END
34      ELSE Out.String("empty calendar"); Out.Ln; EXIT
35      END;
36      IF time > tEnd THEN EXIT END
37    END
38  END Simulate;
39
40  PROCEDURE Reset*;
41  BEGIN Qs.Open(clndr); time := 0
42  END Reset;
43
44  BEGIN NEW(clndr); Reset
45  END Sim.
46
47  MODULE Stations;
48  (* s. RW, S. 258 *)
49  IMPORT Qs, Sim, Paths;
50
51  TYPE

```

```

6   Station* = POINTER TO StationDesc;
7   StationDesc* = RECORD
8     wl: Qs.FIFO;          (* Waiting Line *)
9     path*: Paths.Path
10  END;
11  BeginMsg* = RECORD(Sim.Message) s*: Station END;
12
13  PROCEDURE Request*(s: Station; customer: Sim.Actor);
14    VAR msg: BeginMsg;
15  BEGIN
16    IF s.path.n = 0 THEN      (* server empty *)
17      msg.s := s; customer.handle(customer, msg)
18    ELSE Qs.Enqueue(s.wl, customer)
19    END;
20    Paths.Up(s.path, Sim.time)
21  END Request;
22
23  PROCEDURE Free* (s: Station);
24    VAR
25      customer: Qs.Item;
26      msg: BeginMsg;
27  BEGIN
28    Paths.Down(s.path, Sim.time);
29    IF s.path.n > 0 THEN      (* Customers waiting *)
30      customer := Qs.DequeuedItem(s.wl);
31      WITH customer: Sim.Actor DO
32        msg.s := s; customer.handle(customer, msg)
33      END
34    END
35  END Free;
36
37  PROCEDURE Open* (s: Station);
38  BEGIN
39    Paths.Init(s.path); NEW(s.wl); Qs.Open(s.wl)
40  END Open;
41
42  END Stations.

```

8.3 Strukturmodelle erweiterbarer Systeme. Graphik-Editor

Details und Beispiele: s. Technischer Bericht *Programmierung erweiterbarer Systeme. Eine Kurzeinführung in Oberon*