Einführung in Ada

von
Ulrich Grude

Version 2.0, 1998. Kritik, Verbesserungsvorschläge und Hinweise auf Fehler sind jederzeit sehr willkommen. E-mail: grude@tfh-berlin.de. Fon: (030) 4504-2265. Post-Adresse: Ulrich Grude, TFH Berlin, Luxemburger Str. 10, D-13353 Berlin.

Inhaltsverzeichnis

Zentraldokument: Nach Filialdokument A95-Inhaltsverz

1. Einleitung

Dieses Skript gibt eine Einführung in die Programmierung mit der Sprache **Ada95**. Es entstand 1998 als Grundlage für Lehrveranstaltungen am Fachbereich Informatik der Technischen Fachhochschule Berlin. Die vorliegende Version 2.0 des Skripts löst die frühere Version 1.0 (vom März 1995) ab, in der Ada83 behandelt wurde.

Welche Programmiersprache man einem ersten Kurs über Programmierung zugrunde legen sollte, ist eine wichtige Frage, auf die es offensichtlich mehrere praktikable und gute Antworten gibt. Für Ada sprechen unter anderem die folgenden Gründe:

1. Ada kombiniert auf eine besonders gut begründete und konsequente Weise praktisch alle heute wichtigen **Konzepte** von Programmiersprachen (Objektorientierung, ein starkes Typensystem, Nebenläufigkeit, verteilte Systeme, Modularisierung, Kontrolle der Genauigkeit von numerischen Berechnungen, Generizität etc.).

2. Ada ist **international standardisiert** (ISO/IEC 8652:1995(E)). Ein gründliches und wirksames Validierungsverfahren sorgt dafür, daß Ada95-Compiler auch wirklich dem Standard entsprechen und weder mehr noch weniger können.

3. Man kann davon ausgehen, daß der gegenwärtige (zweite) Ada-Standard mindestens bis zum Jahr 2005 **stabil und unverändert** bleibt. Der erste Ada-Standard, ANSI/MIL-STD-1815A-1983, ISO 8652:1987, "hielt" zwölf Jahre lang von 1983 bis 1995. Was Studenten im ersten Semester heute über Ada lernen ist wahrscheinlich auch dann noch gültig, wenn sie ihr Studium nach acht Semestern abschließen.

4. Ada ist auf praktisch allen Plattformen (Rechnerarchitekturen und Betriebssystemen) **verfügbar**. Für Ada83 gab (und gibt es noch) ca. 400 verschiedene Compiler. Für Ada95 gibt es heute (Anfang 98) "nur" etwa 25 validierte Compiler, aber die Zahl nimmt noch zu. Der von der Free Software Foundation entwickelte **Gnat-Compiler** (Gnu Ada Translator) ist von hoher Qualität, steht kostenlos zur Verfügung und wurde bereits auf viele verbreitete Plattformen portiert, z.B. auf PCs unter Windows 95 oder NT, auf PCs unter Linux, auf PCs unter OS/2, auf Sun Sparc-Workstations unter Sun Solaris etc.. Die Firma **Aonix** stellt Studenten einen weiteren sehr guten Ada-Compiler und eine komfortable Entwicklungsumgebung für PCs unter Windows 95 und NT kostenlos zur Verfügung.

5. Vor allem für **technische Informatiker** interessant ist die enge Verwandschaft zwischen Ada und **VHDL**, einer Sprache, in der man Schaltungsentwürfe durch Programme modellieren und testen kann, ehe man sie z.B. "in Silicon gießt".

Hier die wichtigsten **Quellen für Informationen über Ada**:

1. Das **Ada95 Referenz Manual** (ARM). Es enthält die vollständige und verbindliche Definition der Sprache Ada95 und ist in verschiedenen Formaten (HTML, Postscript, ASCII etc.) an vielen Stellen im Internet und auf CDs erhältlich (s.u.). Die gedruckte Ausgabe bei Springer kostet 78,- DM. Das vorliegende Skript enthält zahlreiche Verweise auf das ARM, die z.B. so aussehen: (ARM 3.7) oder (ARM A.10) etc.. Das ARM ist nicht besonders leicht zu lesen. Trotzdem lohnt es sich, ab und zu hineinzusehen.

"**Ada95 Reference Manual**"
S. Tucker Taft, Robert A. Duff (eds.)
Lecture Notes in Computer Science Vol. 1246, Springer Verlag 1995

Das **Annotated Ada Reference Manual** (AARM) enthält zusätzlich zum Text des ARM zahlreiche Anmerkungen, Erläuterungen und Hinweise (annotations) der Sprachentwickler. Das AARM ist nur im Internet erhältlich.

2. Im sogenannten **Ada95 Rationale** beschreiben die Entwickler, wie sie Ada83 zu Ada95 erweitert haben. Dabei erklären und begründen sie ihre Entwurfsentscheidungen sehr verständlich und einsichtig. Auch dieses Dokument ist im Internet erhältlich. Die gedruckte Ausgabe bei Springer kostet 68,- DM:

"**Ada95 Rationale**"
John Barnes (ed.)
Lecture Note in Computer Science Vol. 1247, Springer Verlag 1995

3. Im **Ada83 Rationale** erklären und begründen die Entwickler der ursprünglichen Sprache Ada83 ihre Entwursentscheidungen, ebenfalls sehr verständlich und einsichtig. Dieses Dokument ist wohl nur noch im Internet zugänglich. Seine letzte Version stammt aus dem Jahr 1986 und der vollständige Titel lautet:

"Rationale for the Design of the Ada Programming Language"
Jean D. Ichbiah, John G.P. Barnes, Robert J. Firth, Mike Woodger
Honeywell Systems and Research Center, Minneapolis USA and Alsys, France

4. Die folgenden Internetadressen sind gute Startpunkte, um Dokumente und Informationen über Ada, kostenlose Ada-Compiler und Entwicklungsumgebungen, eine Liste aller verfügbaren Ada-Compiler, einen Lebenslauf von Augusta Ada Byron (1815-1852), Countess of Lovelace etc. etc. zu finden:

http://www.adahome.com	-- Home of the Brave Ada Programmer
http://sw-eng.falls-church.va.us/adaic	-- Ada Information Clearinghouse
http://info.acm.prg/sigada	-- ACM Special Interest Group Ada

5. Das folgende Buch ist umfangreich (ca. 800 Seiten), verständlich geschrieben und enthält viele Beispiele. Allerdings werden einige Ada-Konstrukte nicht behandelt (z.B. selbst definierte Ganzzahltypen). Das Buch kostet ca. 126,- DM.

"**Ada95, Problem Solving and Program Design**", Second Edition
Michael B. Feldman, Elliot B. Koffman
Addison-Wesley Publishing Company, 1996

6. Hier noch ein etwas kürzeres Buch ("nur" ca. 480 Seiten) mit guten Erklärungen und Beispielen:

"**Ada95, The Craft of Object-Oriented Programming**"
John English
Prentice Hall Europe, 1997

7. Die **Ada-CD** der Firma Walnut Creek (besteht eigentlich aus 2 CDs und) enthält den Gnat-Compiler (Gnu Ada Translator) für verschiedene Plattformen, alle wichtigen Dokumente über Ada (ARM, AARM, Ada95 Rationale, Ada83 Rationale) und unüberschaubar viele weitere Dokumente, Programme, Tools etc. etc.. Alle paar Monate kommt eine neue Version dieser Doppel-CD heraus. Sie ist in größeren wissenschaftlichen Buchhandlungen (Lehmanns, Kiepert etc.) erhältlich und kostet ca. 50,- DM.

Das **vorliegende Skript** enthält ein paar Besonderheiten, auf die schon hier in der Einleitung hingewiesen werden soll:

1. Grundlage der meisten Erläuterungen in diesem Skript ist eine **Metapher** (ein Vergleich, ein gedankliches Modell), derzufolge die Tätigkeit des Programmierens ein **Rollenspiel** ist, an dem vor allem 3 Hauptrollen beteiligt sind: der **Programmierer**, der **Ausführer** und der **Benutzer**.

In der Rolle des **Ausführers** ist alles zusammengefaßt, was für die Ausführung eines (Ada-) Programms nötig ist, üblicherweise also ein Editor (zum Eingeben des Programmtextes), ein Compiler (zum Prüfen und Übersetzen des Programms), ein Binder (zum Erstellen eines Maschinenprogramms) und last but not least eine Plattform (Rechner plus Betriebssystem) zum Ausführen des Maschinenprogramms.

Der Begriff des **Ausführers** soll vor allem **abstrahieren**, und zwar:

- von der Tatsache, daß heute Ada-Programme meistens **compiliert** ("in die Maschinensprache eines bestimmten Computers übersetzt") werden. In Zukunft werden möglicherweise andere Ausführungsmodelle an Bedeutung gewinnen (z.B. Interpretation des unveränderten Programmtextes oder neuartige Kombinationen von Compilation und Interpretation wie etwa bei der Programmiersprache Java),
- von den **Unterschieden** zwischen den zahllosen Editoren, Compilern, integrierten Entwicklungsumgebungen, Betriebssystemen, Rechnerarchitekturen etc., die es heute gibt,
- von der Tatsache, daß Programme meistens von **Computersystemen** ausgeführt werden. Dem Leser wird geraten, die Rolle des Ausführers möglichst oft selbst zu übernehmen und seine Programme "mit Papier und Bleistift" auszuführen.

Die Rollenspiel-Metapher (und vor allem die Rolle des **Ausführers**) soll die Erläuterungen in diesem Skript möglichst unabhängig machen von den sich atemlos-schnell ändernden Eigenschaften heutiger Computersysteme.

2. Einzelne Ada-Befehle und kleine Programme werden in diesem Skript unter anderem dadurch erklärt, daß sie **ins Deutsche übersetzt** werden. In einigen Übungsaufgaben wird der Leser aufgefordert, solche Übersetzungen selbst durchzuführen. Grundlage für diese Übersetzungen ist außer der Rollenspiel-Metapher die Vorstellung, daß ein Programm aus **Befehlen** des **Programmierers** an den **Ausführer** besteht. Falls der Ausführer ein Mensch ist, der Deutsch versteht, kann man die Befehle auch auf Deutsch formulieren statt "auf Ada". Die Programmiersprache Ada wird alos in diesem Skript stellenweise so ähnlich behandelt wie eine **natürliche Fremdsprache** (z.B. Englisch, Französisch, Türkisch, etc.).

3. Am Beispiel von Ada sollen vor allem **abstrakte Konzepte** von Programmiersprachen verständlich gemacht und nur nebenbei die **konkreten Eigenheiten** einer bestimmten Sprache dargestellt werden. Allerdings ist die Unterscheidung zwischen abstrakten Konzepten und konkreten Eigenheiten nicht immer so einfach und natürlich, wie man zuerst vermuten mag.

4. Die vorgestellten Konzepte werden bewertet oder gewichtet, z.B. als **wichtig** oder **besonders wichtig**. Ein Konzept ist wichtig, wenn es in vielen (verbreiteten) Programmiersprachen vorkommt oder wenn es in Ada an vielen verschiedenen Stellen Einfluß auf die Sprache hat. Z.B. kommt das Konzept eines **Typs** in vielen Sprachen vor und spielt bei fast allen Ada-Befehlen eine Rolle. Die Unterscheidung von verschiedenen Parameter-Modi (**in**, **out** und **in out**) kommt dagegen (fast) nur in Ada vor und spielt auch in Ada nur an wenigen Stellen eine Rolle. Das Konzept eines Typs ist also besonders wichtig, das Konzept der Parameter-Modi dagegen deutlich weniger wichtig.

Die Gewichtung der Konzepte ist natürlich nicht völlig objektiv, sondern spiegelt auch bestimmte Ansichten und Vorurteile des Autors wieder. Der Leser sollte sie möglichst oft in Frage stellen und eine eigene Bewertung entwickeln.

5. Ziel dieses Skripts ist es, dem Leser ein "tieferes Verständnis" für bestimmte Konzepte von Programmiersprachen zu vermitteln und nicht, ihm möglichst schnell eine (eventuell nur oberflächliche) Beherrschung vieler Befehle zu ermöglichen. Praktisch bedeutet das, daß anhand der ersten kleinen Beispielprogramme ziemlich viele und zum Teil komplizierte Grundbegriffe eingeführt werden und die Lernkurve vor allem am Anfang sehr steil ist. Aber wenn der Leser diese starke Steigung mitmacht, winken ihm als Belohnung ein paar wirklich allgemeine und nützliche Einsichten.

So viel schon hier in der Einleitung zu den Besonderheiten dieses Skripts.

Im folgenden **Abschnitt 2** wird die Metapher "Programmieren ist ein Rollenspiel" genauer beschrieben. Die 3 wichtigsten Grundkonzepte von Programmiersprachen werden im **Abschnitt 3** eingeführt noch bevor das erste Ada-Programm vorgestellt wurde. Daß es im Grunde genommen nur drei Arten von Befehlen gibt, die ein Programmierer einem Ausführer geben kann, wird im **Abschnitt 4** erläutert. Im **Abschnitt 5** findet man wichtige Erklärungen zur Organisation und Darstellung der Beispielprogramme. Im **Abschnitt 6** geht es dann richtig los mit dem ersten Beispielprogramm HALLO_01.

2. Programmieren als ein Rollenspiel

Wenn sie das Wort **Programm** oder **Programmierung** hören, denken viele Menschen gleich sehr konkret an heute übliche Computersysteme (Prozessoren, Betriebssysteme, Compiler etc.) und ihre Eigenarten. Das ist ganz natürlich und realistisch, hat aber auch einen Nachteil: Computersysteme verändern sich sehr rasch, nicht nur **quantitativ** (z.B. bezüglich ihrer Speichergröße oder Rechengeschwindigkeit), sondern auch **qualitativ** (z.B. werden isolierte Computer durch vernetzte Systeme und Rechner mit **einem** Prozessor durch solche mit **mehreren** Prozessoren abgelöst). Wenn unser Wissen über Programme und Programmierung zu sehr von heute üblichen Computersystemen abhängt, dann steht es dauernd in Gefahr, zu veralten und ungültig zu werden.

Hier soll deshalb versucht werden, den Begriff der **Programmierung** etwas abstrakter zu fassen und ihn (soweit das möglich ist) von den unwichtigen Eigenschaften heutiger Computer unabhängig zu machen. Der abstrakte Begriff soll **die** Aspekte des Schreibens und Ausführens von Programmen hervorheben, die wirklich wichtig sind und vermutlich auch noch eine Weile unverändert und wichtig bleiben werden.

Die Tätigkeit des Programmierens kann man als eine Art **Rollenspiel** auffassen, bei dem vor allem drei Hauptrollen beteiligt sind: der **Programmierer**, der **Ausführer** und der **Benutzer**. Jeder Rolle sind bestimmte charakteristische Tätigkeiten zugeordnet:

1. Der Programmierer **schreibt** Programme und **übergibt sie** dem Ausführer.

2. Der Ausführer **prüft** die Programme daraufhin, ob sie formale Fehler ("Rechtschreibungs- und Grammatikfehler") enthalten. Wenn ja, **lehnt** er sie **ab** und informiert den Programmierer über die gefundenen Fehler. Wenn ein Programm keine formalen Fehler enthält, **akzeptiert** der Ausführer es und legt es in seiner Bibliothek ab.

Das Prüfen und Ablehnen bzw. Akzeptieren von Programmen ist aber nur eine Art Nebentätigkeit des Ausführers. Seine Haupttätigkeit besteht darin, Programme **auszuführen**. Er tut das allerdings nur, wenn der Benutzer ihn dazu auffordert.

3. Der Benutzer ist derjenige, der den Ausführer zum Ausführen eines Programms **auffordert**. Der Ausführer kann einer solchen Aufforderung nur dann nachkommen, wenn er das betreffende Programm bereits akzeptiert hat und es sich in seiner Bibliothek befindet.

Wenn der Ausführer ein Programm ausführt, benötigt er dazu in aller Regel gewisse **Eingabedaten** und produziert daraus (entsprechend den Befehlen in dem Programm) bestimmte **Ausgabedaten**. Es gehört zu den charakteristischen Tätigkeiten des Benutzers, daß er für alle Ein- und Ausgabedaten **zuständig ist**. D.h. er muß dem Ausführer alle benötigten Eingabedaten zugänglich machen und bekommt dafür alle produzierten Ausgabedaten.

Außer den drei Hauptrollen (Programmierer, Ausführer, Benutzer) gibt es noch ein paar Nebenrollen. Hier sollen nur zwei davon erwähnt werden: die **Kollegen** des Programmierers, genauer: der **Kollege-1** und der **Kollege-2**.

Ähnlich wie der Programmierer schreibt auch der **Kollege-1** Programme. Seine charakteristische Tätigkeit besteht aber darin, Programmteile, die der Programmierer schon geschrieben hat, **wiederzuverwenden**, statt sie selbst noch mal zu schreiben. Mit dieser Wiederverwendung von

Programmteilen will er sich Mühe und seiner Firma Kosten sparen. Der Programmierer sollte seine Programme so schreiben, daß der Kollege-1 möglichst viele (Teile) davon möglichst oft wiederverwenden kann.

Die Rolle des Kollegen-1 könnte man auch "die Rolle des Wiederverwenders von Programmteilen" nennen, aber "Kollege-1" ist kürzer und klingt kollegialer.

Der **Kollege-2** wird typischerweise erst viele Monate nach Fertigstellung eines Programms tätig. In diesen Monaten hat sich herausgestellt, daß das Programm ein paar kleine inhaltliche Fehler enthält ("es tut nicht immer, was es soll") . Außerdem hat der Benutzer des Programms mehrere Änderungs- und Erweiterungswünsche entwickelt. Aufgabe des Kollegen-2 ist es, das Programm des Programmierers zu **lesen** und zu **verstehen** und es dann zu **verbessern**, zu **verändern** und zu **erweitern**. Diese Tätigkeiten faßt man auch unter dem Begriff der **Wartung** zusammen. Der Kollege-2 **wartet** die Programme des Programmierers.

Ebenfalls typisch für den Kollegen-2 ist, daß er keinen persönlichen Kontakt zum Programmierer hat (weil dieser inzwischen in eine andere Abteilung versetzt wurde, die Firma gewechselt oder einen längeren Urlaub angetreten hat). Der Programmierer sollte seine Programme deshalb so schreiben, daß der Kollege-2 sie ohne Rückfragen verstehen und verbessern kann.

Die Rollen **Kollege-1** und **Kollege-2** werden hier als **Nebenrollen** bezeichnet, weil man programmieren kann, ohne sie zu besetzen und weil sie in diesem Skript sehr viel seltener auftreten als die drei Hauptrollen. Das soll aber nicht bedeuten, daß der Programmierer seine Kollegen nicht sehr ernst nehmen sollte. Das Gegenteil ist der Fall.

Die einzelnen Rollen des Rollenspiels **Programmieren** kann man auf sehr unterschiedliche Weisen **besetzen**.

Z.B. kann man die Rolle des Programmierers einer einzelnen Frau übertragen oder einem ganzen Team von ProgrammiererInnen. Entsprechendes gilt auch für die Rolle des Benutzers und für die Rollen der beiden Kollegen des Programmierers.

Die Rolle des **Ausführers** wird in der Praxis typischerweise mit einem **Computer** (hardware), einem **Betriebssystem**, einem **Editor**, einem **Compiler** und einem **Binder** besetzt. In diesem Fall **übergibt** der Programmierer eine Programm an den Ausführer, indem er den Programmtext mit dem Editor **eingibt** und vom Compiler **prüfen** und **übersetzen** läßt. Der Benutzer **fordert** den Ausführer **auf**, ein Programm auszuführen (der Benutzer "startet ein Programm"), indem er den Namen des Programms über eine Tastatur **eingibt** oder ihn (im Rahmen eines entsprechenden Dialogs) in einer Liste mit der Maus **anklickt**.

Das **Binden** eines Programms (bei dem alle übersetzten Teile eines Programms zu **einem** Maschinenprogramm zusammengebunden werden) gehörte früher zu den Tätigkeiten des Programmierers. Heute ist es konzeptuell häufig günstiger, es der Rolle des Benutzers zuzuordnen (wichtige Teile des Bindevorgangs finden heute häufig erst dann statt, wenn der Benutzer ein Programm startet). Solche Einzelheiten sind hier aber nicht weiter wichtig.

Manchmal wird anstelle eines Compilers und eines Binders ein **Interpreter** eingesetzt oder der Editor, der Compiler und der Binder sind zu einer **integrierten Entwicklungsumgebung** zusammengefaßt oder es werden anstelle **eines** Binders **zwei** verschiedene Binder eingesetzt etc. etc.. Es gibt schon heute viele verschiedene Arten von maschinellen Ausführern, und ihre Zahl nimmt

offenbar noch zu. Die **Rolle des Ausführers** soll von den zahllosen Unterschieden zwischen verschiedenen Ausführern abstrahieren und **die** Tätigkeiten benennen und hervorheben, die jeder Ausführer beherrschen muß: Programme **prüfen**, **ablehnen**, **akzeptieren**, **ablegen** und **ausführen**.

Hier noch eine besonders wichtiger **Besetzungsvorschlag** für das Rollenspiel: Um eine Programmiersprache zu lernen, sollte man möglichst oft alle drei Hauptrollen **selbst übernehmen** und Programme nicht nur **schreiben** (als Programmierer), sondern sie auch **ausführen** (als Ausführer) und sich nebenbei auch noch um die Ein- und Ausgabedaten kümmern (als Benutzer). Denn ein Programm ist eine Folge von Befehlen des Programmierers an den Ausführer und die genaue Bedeutung dieser Befehle ("ihre Wirkung") kann man sich besonders intensiv einprägen, wenn man sie selbst ("mit Papier und Bleistift") ausführt. Falls man einen Befehl nocht nicht genau kennt, merkt man das beim Ausführen in aller Regel sehr schnell und kann die Wissenslücke gezielt schließen.

Aber nicht nur beim Lernen einer neuen Programmiersprache ist es vorteilhaft, die Rolle des Ausführers bisweilen selbst zu übernehmen. Auch erfahrene Programmierer führen häufig besonders komplizierte und wichtige Befehlsfolgen Schritt für Schritt selbst aus (in einfachen Fällen im Kopf und in schwierigeren Fällen auch mit Papier und Bleistift), um ihre Bedeutung genau zu verstehen.

Ein **maschineller** Ausführer ist in den meisten Fällen deutlich schneller als ein **menschlicher** Ausführer. Wenn das auszuführende Programm noch inhaltliche Fehler enthält ("nicht immer das tut, was es tun soll"), dann bedeutet das aber nur: Der maschinelle Ausführer macht pro Sekunde deutlich mehr Fehler als der Mensch. Außerdem kann der menschliche Ausführer (wenn er das fehlerhafte Programm Schritt für Schritt ausführt) oft herausfinden, **wo** genau der Fehler liegt und **warum** er auftritt.

Wenn ein Mensch ein Programm ausführt, dann lernt er es damit auch besser kennen und verstehen, und wenn man ihn nach der Ausführung z.B. fragt: "**Warum** ist als Ergebnis 42 herausgekommen und nicht 37?", dann kann er einem häufig eine vernünftige und verständliche Antwort geben. Wenn bei der maschinellen Ausführung eines Programms auf dem Bildschirm als Ergebnis die Zahl 42 erscheint, und man den Computer fragt, warum nicht 37 herausgekommen ist, erhält man in aller Regel keine befriedigende Antwort.

Die Rolle des Ausführers selbst zu übernehmen, ist zwar mit einiger Mühe verbunden, hat aber sonst große Vorteile und wird dem Leser **nachdrücklich empfohlen**.

Die Welt der Computer verändert sich sehr schnell und in kaum vorhersehbarer Weise. Die Rollenspielmetapher ist eine Art **Schnittstelle** zu dieser Welt. Von dieser Schnittstelle kann man hoffen, daß sie deutlich länger stabil und unverändert bleibt, als die vielen Einzelheiten, die sich hinter ihr verbergen und von denen sie abstrahiert.

Um ehrlich zu sein: Einige Eigenschaften der Sprache Ada kann man nur dann richtig erklären und verstehen, wenn man bestimmte Eigenarten (Verschrobenheiten?) von heute üblichen Computern in Betracht zieht. An einigen Stellen des Skripts müssen wir deshalb die abstrakte Ebene der Rollenspiel-Metapher verlassen und uns dazu hinablassen, auf tiefer gelegenen Ebenen konkrete Einzelheiten zu diskutieren. Das soll aber nur geschehen, wenn es nicht zu vermeiden ist. **o**

Zusammenfassung 2.:

- Der Programmierer **schreibt** Programmteile und **übergibt** sie dem Ausführer.

- Der Ausführer **prüft** diese Programmteile, **lehnt** sie **ab** oder **akzeptiert** sie und **führt** sie **aus**.
- Der Benutzer **fordert** den Ausführer dazu **auf**, ein Programm auszuführen und **ist** für alle Ein- und Ausgabedaten **zuständig**.
- Der Kollege-1 möchte Programme und Programmteile, die der Programmierer geschrieben hat, **wiederverwenden**.
- Der Kollege-2 muß die Programme des Programmierers **lesen**, **verstehen**, **verbessern**, **verändern** und **erweitern** (kurz: **warten**).

3. Die drei wichtigsten Grundkonzepte von Programmiersprachen

Die folgende **Analogie** verbindet **Autotypen** und **Programmiersprachen**: Es gibt unüberschaubar viele Autotypen, aber nur relativ wenige Grundkonzepte, auf denen all diese Typen beruhen (Sportwagen oder Limousine, Diesel oder Benziner, 2-Rad-Antrieb oder Allradantrieb etc.). Ganz ähnlich gibt es unüberschaubar viele verschiedene Programmiersprachen, aber nur relativ wenige Grundkonzepte, auf denen alle diese Sprachen beruhen. In diesem Abschnitt sollen die drei wichtigsten Grundkonzepte von Programmiersprachen kurz beschrieben werden. Das geschieht schon hier, bevor das erste Ada-Programm vorgestellt wird, und zwar aus drei Gründen:

- Um die **Wichtigkeit** der drei Konzepte zu betonen
- Um die **Unabhängigkeit** der drei Konzepte von einer konkreten Sprache (z.B. Ada) hervorzuheben
- Um einige Begriffe und Vorstellungen einzuführen, die die Erläuterung des ersten Ada-Programms (im Abschnitt 6) **vereinfachen** werden

3.1. Das Konzept einer beliebig oft veränderbaren Variablen

Mathematiker verwenden schon seit langem **Variablen**, z.B. in Formeln. Die folgende Formel beschreibt, wie man die Summe aller Ganzzahlen von 1 bis n berechnen kann:

(n + 1) * n / 2 (*)

Setzt man für die Variable **n** etwa die Zahl 4 ein, so erhält man als Wert der Formel die Zahl 10. Tatsächlich gilt 1 + 2 + 3 + 4 = 10. Niemand käme auf die Idee, mitten in einer Berechnung mit dieser Formel (z.B. **nach** der Berechnung von (n + 1) aber **vor** der Berechnung von n / 2) den Wert der Variablen n zu verändern.

Allgemein gilt: Den Variablen in einer mathematischen Formel darf man nur am **Anfang** einer Berechnung Werte zuordnen. Während man dann die Formel auswertet, darf man die Werte der Variablen **nicht** mehr verändern.

Natürlich hindert einen nichts daran, mit **einer** Formel nacheinander **mehrere** Berechnungen durchzuführen und am Anfang jeder Berechnung erneut Werte für die Variablen festzulegen. In diesem Sinne sind Variablen in Formeln **variabel**, d.h. veränderbar. Aber wenn wir einer Variablen einen Wert zugeordnet haben, dann sollte dieser Wert bis zum Ende der betreffenden Berechnung unverändert bleiben.

Schon mit den ersten Computern wollte und konnte man "Berechnungen nach mathematischen Formeln" durchführen. Dazu mußte man unter anderem die **Variablen** in solchen Formeln technisch realisieren. Noch heute bezeichnet man die entsprechenden Bauteile von Computern manchmal als **Speicherzellen**.

Die Speicherzellen eines Computers haben die Eigenschaft, daß man praktisch **beliebig oft** und **jederzeit** einen neuen Wert hineinschreiben kann, und nicht nur "am Anfang einer Berechnung". Speicherzellen sind also auf Grund ihrer technischen Eigenschaften deutlich "variabler" als die Variablen in einer mathematischen Formel.

Der Unterschied zwischen "**Formel-Variablen**" und "**Speicherzellen-Variablen**" mag auf den ersten Blick klein erscheinen. Die Bedeutung dieses "kleinen Unterschieds" kann aber kaum überschätzt werden. Er ist unter anderem die Grundlage dafür, zwei völlig verschiedene Arten von Programmiersprachen zu unterscheiden:

Sprachen, bei denen man jeder Variablen nur **einmal** einen Wert zuordnen und diesen Wert dann nicht mehr ändern darf, bezeichnet man als **funktionale Programmiersprachen**. Eine **prozedurale Programmiersprache** ist dagegen eine, in der es eine sogenannte **Zuweisungs-Anweisung** gibt, mit der man den Wert jeder Variablen **beliebig oft** verändern darf.

Fast alle allgemein bekannten und verbreiteten Programmiersprachen sind **prozedurale** Sprachen, z.B. Cobol, Fortran, C, C++, Pascal, Modula, Ada und Java.

Lisp, Miranda und die Datenbanksprache SQL sind dagegen **funktionale** Sprachen.

Beispiel 3.1.1.: In Ada (und in vielen anderen Sprachen) kann eine Zuweisungs-Anweisung z.B. so aussehen:

```
X := X + Y;
```

Diese Zuweisung bewirkt, daß die momentanen Werte der Variablen X und Y addiert werden. und das Ergebnis zum neuen Wert der Variablen X gemacht wird. Wenn **vor** Ausführung dieser Anweisung X den Wert 5 und Y den Wert 3 hat, dann hat X **nach** Ausführung der Anweisung den Wert 8. Das, was rechts vom Zuweisungszeichen ":=" steht (im Beispiel: **X + Y**) ist einer mathematischen **Formel** sehr ähnlich und wird in Programmiersprachen als **Ausdruck** (expression) bezeichnet. Links vom Zuweisungszeichen muß der **Name einer Variablen** stehen (im Beispiel: **X**). Die Bedeutung der obigen Zuweisung beschreibt man auch so: Der **Wert** des Ausdrucks X + Y wird der Variablen X **zugewiesen**. **o**

Eine Variable, deren Wert man beliebig oft verändern kann (z.B. mit einer Zuweisungsanweisung) wird im folgenden häufig auch als **Wertebehälter** bezeichnet. Ganz allgemein gilt:

1. Ein Wertebehälter enthält **immer** einen Wert, d.h. ein Wertebehälter kann nicht "leer" sein.
2. Wenn man einem Wertebehälter einen neuen Wert zuweist, dann wird dadurch der **alte** Wert des Behälters zerstört und ist anschließend nicht mehr verfügbar.

Daß Wertebehälter nicht **leer** sein können, mag auf den ersten Blick unplausibel erscheinen. Genau genommen können aber auch andere Behälter nicht wirklich leer sein. Z.B. kann es vorkommen, daß ein Bierglas kein Bier enthält. Dann ist es aber meistens mit Luft gefüllt. Wenn man auch die entfernt, dann enthält das Bierglas ein Vakuum. Auch in diesem Fall ist das Bierglas nicht wirklich "leer", denn ein Vakuum enthält z.B. elektromagnetische Wellen und hat auch sonst viele interessante physikalische Eigenschaften.

Um ehrlich zu sein: Im Alltag ist es durchaus sinnvoll, z.B. von einem **leeren Bierglas** zu sprechen. Daß es **nicht** sinnvoll ist, von einem **leeren Wertebehälter** zu sprechen, hängt mit den technischen Eigenschaften von Speicherzellen zusammen. Eine Speicherzelle besteht aus einer Reihe von **Bits**. Ein Bit kann nur die Werte **Null** oder **Eins** annehmen. Eine Speicherzelle enthält also immer eine Reihe von Nullen und Einsen und kann nicht leer sein. **o**

Zusammenfassung 3.1.: In jeder Programmiersprache nennt man bestimmte Gebilde **Variablen**. Man muß aber zwei grundverschiedene Arten von Variablen unterscheiden: Solche, denen man nur **einmal** einen Wert zuordnen darf und solche, deren Wert man **beliebig oft** verändern darf (z.B. mithilfe einer **Zuweisungs-Anweisung**). Die letzteren werden hier auch als **Wertebehälter** bezeichnet. Entsprechend unterscheidet man **funktionale** Programmiersprachen (**ohne** Zuweisungs-Anweisung) und **prozedurale** Programmiersprachen (**mit** Zuweisungs-Anweisung). Die meisten Programmiersprachen sind **prozedural**. **o**

Aufgabe 3.1.1.: Die meisten Computer sind heutzutage mit einer **Festplatte** ausgerüstet. Eine Festplatte kann man als eine große **Variable** verstehen. Hat eine Festplatten-Variable eher die Eigenschaften einer **Formel-Variablen** oder die Eigenschaften einer **Speicherzellen-Variablen** (d.h. eines **Wertebehälters**)? Wie verhält es sich mit Disketten und Wechselplatten?

Aufgabe 3.1.2.: Eine sogenannte WROM-CD ist eine CD (compact disc), die man einmal beschreiben und dann nur noch lesen kann (write once read only). Mit welcher Art von **Variablen** haben solche WROM-CDs große Ähnlichkeit (mit **Formel-Variablen** oder mit **Speicherzellen-Variablen**)?

3.2. Das Konzept eines Typs

Beim Programmieren in einer **getypten Sprache** legt der Programmierer für jede Variable einen bestimmten **Typ** fest. Aus ihrem Typ folgt, welche **Werte** der Variablen zugewiesen werden dürfen und welche **Operationen** man auf sie anwenden darf.

Beispiel 3.2.1.: Hat der Programmierer für die Variablen **B1** und **B2** den Typ **boolean** festgelegt, dann darf man diesen Variablen nur die Wahrheitswerte **true** und **false** zuweisen und darf auf diese Variablen nur boolesche Operationen wie **and**, **or**, **not** etc. anwenden. Folgende Ausdrücke sind typenmäßig korrekt: **B1 or B2**, **B2 and B1**, **not B2** etc..
Man darf den Variablen **B1** und **B2** aber keine Zahlen wie **5** oder **-7** zuweisen, und man darf keine arithmetischen Operationen wie **+**, **-**, *****, **/** etc. auf sie anwenden. Folgende Ausdrücke sind typenmäßig **nicht** korrekt, d.h. sie enthalten formale Fehler: **B1 + B2**, **B2 * B1**, **- B2** etc.. **o**

Beispiel 3.2.2.: Hat der Programmierer für die Variablen **G1** und **G2** den Typ **integer** festgelegt, dann darf man diesen Variablen nur Ganzzahlwerte wie z.B. **5** oder **-7** oder **+13** etc. zuweisen und man darf auf diese Variablen nur arithmetische Operationen wie **+**, **-**, *****, **/** etc. anwenden. Folgende Ausdrücke sind typenmäßig korrekt: **G1 + G2**, **G2 * G1**, **- G2** etc..
Man darf den Variablen **G1** und **G2** aber keine Wahrheitswerte (**true** bzw. **false**) zuweisen und darf keine booleschen Operationen wie **and**, **or**, **not** etc. auf sie anwenden. Folgende Ausdrücke sind typenmäßig **nicht** korrekt , d.h. sie enthalten formale Fehler: **G1 or G2**, **G2 and G1**, **not G2** etc..
o

Zum Typ **boolean** gehören also die beiden Werte **true** und **false** und die Operationen **and**, **or**, **not** etc.. Zum Typ **integer** gehören die Werte **0**, **+1**, **-1**, **+2**, **-2**, **+3**, **-3**, ... und die Operationen **+**, **-**, *****, **/** etc.. Die genauen Grenzen des Typs **integer** sind von Sprache zu Sprache, und häufig auch von Ausführer zu Ausführer ("von Computer zu Computer") verschieden. Allgemein gilt: Ein **Typ** besteht aus einer **Menge von Werten** und aus einer **Menge von Operationen**.

Beim Programmieren in einer **ungetypten** Sprache darf der Programmierer jeder Variablen jeden Wert zuweisen und jede Operation auf jede Variable anwenden. Z.B. darf er einer Variablen **B1** den Wert **-7** und einer Variablen **G2** den Wert **true** zuweisen und anschließend den Wert des Ausdrucks **B1 + G2** berechnen lassen. Meistens entstehen solche Ausdrücke aus Versehen und sind dann zwar **inhaltlich** falsch, enthalten aber keinen **formalen** Fehler.

Der Unterschied zwischen **inhaltlichen Fehlern** und **formalen Fehlern** ist hier entscheidend. Formale Fehler sind solche, die der **Ausführer** entdecken kann und muß. Inhaltliche Fehler können höchstens vom **Programmierer** (oder vom **Kollegen-2**) entdeckt werden. Da die Rolle des Ausführers meistens mit einem **Computer** besetzt wird, die Rollen des Programmierers und des Kollegen-2 aber von **Menschen** übernommen werden, ist das Aufdecken von **formalen** Fehlern in aller Regel sehr viel **billiger**, als das Aufdecken von **inhaltlichen** Fehlern. Dazu trägt auch bei, daß Computer beim Aufdecken von Fehlern um Größenordnungen zuverlässiger und erfolgreicher sind als Menschen.

Hier wurde nur die **Grundidee** eines Typensystems skizziert. Die meisten Programmiersprachen sind mehr oder weniger getypt, aber einige erheblich mehr als andere. Z.B. sind die Typensysteme der Sprachen Fortran und C relativ schwach, die Typensysteme von Ada und Java dagegen deutlich stärker ausgeprägt.

Für den Programmierer ist ein Typensystem erst einmal eine **Einschränkung**. Er darf bestimmte Ausdrücke nicht in sein Programm schreiben, weil sie Typenfehler enthalten und vom Ausführer als formal falsch abgelehnt werden. Viele Programmierer lernen aber schnell, die **Vorteile** eines starken Typensystems ("weniger menschliche Arbeit bei der Fehlersuche") zu schätzen. Insbesondere die Arbeit des Kollegen-2 (der die Programme des Programmierers lesen, verstehen, verbessern, verändern und erweitern muß) wird durch ein starkes Typensystem erheblich erleichtert. Projektleiter, die für die Kosten einer Programmentwicklung, für die Zuverlässigkeit des entwickelten Programms und für die Kosten der Wartung des Programms zuständig sind, schätzen starke Typensysteme in aller Regel als sehr wertvoll ein.

Zusammenfassung 3.2.:

- Für eine **ungetypte** Variable gilt: Man darf ihr **jeden** Wert zuweisen und **jede** Operation auf sie anwenden.
- Für eine **getypte** Variable gilt dagegen: Man darf ihr nur **bestimmte** Werte zuweisen und nur **bestimmte** Operationen auf sie anwenden.
- Ein Typ besteht aus einer **Menge von Werten** und einer **Menge von Operationen**.
- Der Sinn eines Typensystems ist es, bestimmte **Flüchtigkeitsfehler** des Programmierers zu **formalen Fehlern** zu machen, die der **Ausführer** entdecken kann.
- Die meisten Programmiersprachen sind getypt, die Typensysteme einiger Sprachen (z.B. die Typensysteme von C und Ada) **unterscheiden sich** aber **erheblich** voneinander.

3.3. Das Konzept eines Unterprogramms

Ein **Programm** ist eine **Folge von Befehlen** (die der Programmierer aufgeschrieben hat, damit der Ausführer sie auf Veranlassung des Benutzers ausführt). Jedes Programm hat einen **Namen**.

Beispiel 3.3.1.: Skizze eines Programms (in einer Phantasiesprache geschrieben):

```
Programm OTTO:
      Mache-dieses-1
      Mache-jenes-1
      ...
Ende von OTTO
```

Hier soll "OTTO" der Name eines Programms sein, welches aus den Befehlen "Mache-dieses-1", "Mache-jenes-1" und evtl. weiteren Befehlen besteht. **o**

Sobald der Ausführer ein Programm akzeptiert hat, ist der **Name** des Programms ein neuer **Befehl**, den man dem Ausführer geben kann. Z.B. kann der **Benutzer** den Ausführer mit dem Befehl "Führe-OTTO-aus" dazu auffordern, das Programm OTTO auszuführen. Wie der Befehl "Führe-OTTO-aus" konkret aussieht, ist von Ausführer zu Ausführer verschieden. Bei vielen heutigen Computern braucht man nur den Namen "OTTO" **einzutippen** oder **anzuklicken**. In Zukunft wird es vielleicht genügen, den Namen "OTTO" **auszusprechen** oder konzentriert **daran zu denken**.

Aber nicht nur der Benutzer, auch der **Programmierer** kann den Namen eines Programms als Befehl verwenden.

Beispiel 3.3.2.: Skizze eines weiteren Programms (in einer Phantasiesprache geschrieben)

```
Programm EMIL:
      Mache-dieses-2
      Führe-OTTO-aus
      Mache-jenes-2
      Führe-OTTO-aus
      ...
Ende von EMIL
```
o

Im Programm EMIL gibt der Programmierer dem Ausführer unter anderem zweimal den Befehl, das Programm OTTO auszuführen. Man sagt auch: Der Programmierer hat im Programm EMIL das **Unterprogramm** OTTO **aufgerufen**.

Statt im Programm EMIL (das Programm) OTTO als Unterprogramm **aufzurufen**, hätte der Programmierer auch die Befehlsfolge, für die der Name "OTTO" steht, mehrmals in das Programm EMIL **kopieren** können, etwa so:

Beispiel 3.3.3.: Das Programm EMIL', eine Variante von EMIL:

```
Programm EMIL':
      Mache-dieses-2
      Mache-dieses-1
      Mache-jenes-1
      ...
      Mache-jenes-2
      Mache-dieses-1
      Mache-jenes-1
      ...
      ...
Ende von EMIL'
```
o

Diese Vorgehensweise hätte aber erhebliche Nachteile:

1. Das Programm EMIL' ist **schwerer zu verstehen** als EMIL. Denn daß in EMIL mehrmals der gleiche Befehl ("Führe-OTTO-aus") vorkommt, ist leicht zu erkennen. Daß in EMIL' mehrmals die

gleiche Befehlsfolge vorkommt, ist weniger leicht zu erkennen (insbesondere, wenn OTTO aus vielen Befehlen besteht).

2. Das Programm EMIL' ist **schwerer zu warten** als EMIL. Wenn der Nachfolger des Programmierers das Programm OTTO verbessert, verändert oder erweitert, wirken sich diese Wartungsarbeiten automatisch auch auf EMIL aus, und zwar an allen Stellen, an denen OTTO aufgerufen wird. Im Gegensatz dazu müssen im Programm EMIL' alle Wartungsarbeiten an der OTTO-Befehlsfolge an **mehreren** Stellen vorgenommen werden. Das ist mühsam und vor allem fehlerträchtig.

3. Das Programm EMIL' ist **länger** als das Programm EMIL

Faustregel: Wenn der Programmierer eine bestimmte Befehlsfolge **mehrmals** braucht, sollte er sie **einmal** in Form eines Unterprogramms aufschreiben und dieses Unterprogramm mehrmals **aufrufen**. Indem er seinen Unterprogrammen **sprechende Namen** gibt (z.B. BERECHNE_LOHNSTEUER, LOESCHE_DATEN_DES_VORJAHRES, ERMITTLE_NULLSTELLEN etc.) kann er die Lesbarkeit seines Programms erheblich verbessern und damit seinen Kollegen die Arbeit erleichtern. Daß eine bestimmte Folge von 137 Befehlen die Daten des Vorjahres löscht, ist für den Leser eines Programms in aller Regel nur sehr schwer zu erkennen. Wenn die gleichen 137 Befehle zu einem Unterprogramm namens LOESCHE_DATEN_DES_VORJAHRES zusammengefasst sind, kommt der Leser in aller Regel viel schneller auf die Bedeutung dieser Befehle.

In einigen Sprachen besteht ein **Programm** aus einem **Hauptprogramm** und beliebig vielen **Unterprogrammen** und das Hauptprogramm **sieht anders aus** als ein Unterprogramm (das ist z.B. in Fortran und Pascal der Fall). In andern Sprachen wird die Unterscheidung zwischen einem Hauptprogramm und einem Unterprogramm nicht so ernst genommen. In Ada wird ein Unterprogramm nur durch die Art und Weise, wie es vom Programmierer und vom Benutzer **benutzt** wird, zum Hauptprogramm, sieht aber genau so aus, wie andere Unterprogramme auch. Ein Programm besteht in Ada aus einem Unterprogramm **H** und all den weiteren Programmteilen, die man zur Ausführung von **H** benötigt. Dazu gehören unter anderem alle Unterprogramme, die in **H** aufgerufen werden.

Es gibt **zwei Arten** von Unterprogrammen: **Prozeduren** und **Funktionen**. Der Unterschied zwischen den beiden Arten wird später erläutert.

Unterprogramme (Prozeduren und Funktionen) gibt es in praktisch allen Programmiersprachen.

Zusammenfassung 3.3: Das Konzept eines **Unterprogramms** gibt dem Programmierer die Möglichkeit, bestimmten Befehlsfolgen einen Namen zu geben und diesen Namen als neuen Befehl zu verwenden. Mithilfe von Unterprogrammen kann man ein großes Programm in kleinere, übersichtliche Teile gliedern und die Lesbarkeit und Wartbarkeit des Programms verbessern.

Zusammenfassung 3.: Variablen, **Typen** und **Unterprogramme** sind die drei wichtigsten Grundkonzepte der meisten Programmiersprachen.

4. Drei Arten von Befehlen

Eine Programmiersprache umfaßt **Befehle**, die der Programmierer in ein Programm schreiben kann, damit der Ausführer sie auf Veranlassung des Benutzers ausführt. Trotz der Fülle unterschiedlicher Befehle, die im Rahmen verschiedener Sprachen erfunden wurden, gibt es im Grunde genommen nur **drei Arten** von Befehlen: **Vereinbarungen**, **Ausdrücke** und **Anweisungen**. Wenn man von einem Befehl nur weiß, zu welcher Art er gehört, dann weiß man schon ungefähr, "was er bewirkt". Im folgenden werden die drei Befehlsarten näher erläutert.

4.1. Vereinbarungen

Eine **Vereinbarung** (declaration) ist ein Befehl des Programmierers an den Ausführer, etwas zu **erzeugen**, z.B. eine Variable oder ein Unterprogramm. Welche "Dinge" man sonst noch vom Ausführer erzeugen lassen kann, ist von Sprache zu Sprache verschieden.

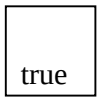
Beispiel 4.1.1.: Eine Vereinbarung einer Variablen (d.h. eines Wertebehälters) in Ada

```
B1 : boolean := true;
```

Diesen Befehl kann man etwa so ins Deutsche übersetzen: Erzeuge einen Behälter namens B1 für Werte vom Typ **boolean** und lege als Anfangswert den Wert **true** in diesen Behälter. **o**

Heute übliche Computer **erzeugen** einen Wertebehälter meistens, indem sie einen entsprechenden kleinen oder großen Abschnitt ihres Hauptspeichers reservieren und ihn (nicht zum Speichern von Befehlen, sondern) als Wertebehälter benützen. Ein **menschlicher** Ausführer könnte die Variable B1 z.B. dadurch erzeugen, daß er mit Bleistift einen Kasten auf Papier zeichnet, davor den Namen **B1** und darunter den Typ **boolean** schreibt und in den Kasten den Anfangswert **true** einzeichnet. Sollte dem Wertebehälter **B1** später ein anderer Wert zugewiesen werden, könnte der menschliche Ausführer den bisherigen Wert **true** ausradieren und den neuen Wert einzeichnen.

Beispiel 4.1.2.: Eine von einem menschlichen Ausführer erzeugte Variable namens B1 vom Typ boolean mit dem momentanen Wert true.



boolean

o

Wenn der Programmierer eine Variable vereinbart, ohne dabei einen **Anfangswert** anzugeben, hat die Variable nach ihrer Erzeugung **irgend einen Wert** (denn eine Variable kann ja grundsätzlich nicht "leer" sein) und der Programmierer muß auf jeden (sinnvollen oder unsinnigen) Wert in dieser Variablen gefaßt sein. Es kann sogar passieren, daß der Wert der Variablen nicht zu ihrem Typ gehört. Wenn der Programmierer das vermeiden möchte, sollte er einen Anfangswert festlegen.

Beispiel 4.1.3.: Eine Variablen-Vereinbarung ohne Anfangswert

B2 : boolean;

Wenn ein Mensch diese Vereinbarung "mit Papier und Bleistift" ausführt, kann er die erzeugte Variable z.B. so darstellen:

boolean

Der eingezeichnete Wert **chrkpt** soll ausdrücken, daß die Variable **irgendeinen Wert** enthält. Man beachte, daß der Wert **chrkpt** nicht zum Typ **boolean** gehört. **o**

Aufgabe 4.1.1.: Übersetzen Sie die folgenden beiden Vereinbarungen ins Deutsche, und führen Sie sie aus:

```
G1: integer := 17;
G2: integer;
```
o

Zum Typ **character** gehören in Ada genau 256 Werte. Einzelne Werte vom Typ **character** kann man durch **Zeichenliterale** wie z.B. 'A', 'B', 'C', ..., 'a', 'b', 'c', ..., '0', '1', '2', ..., '!', '?', '%', '§', '" etc. beschreiben. Weitere Einzelheiten werden später erläutert.

Aufgabe 4.1.2.: Übersetzen Sie die folgenden beiden Vereinbarungen ins Deutsche und führen Sie sie aus:

```
CLAUDIA: character := 'A';
CLAUS  : character;
```
o

Hier noch ein wichtiger Hinweis: Was man alles vereinbaren (d.h. vom Ausführer erzeugen lassen) kann, ist von Sprache zu Sprache verschieden. Aber **Werte** (z.B. Wahrheitswerte wie **true** und **false** oder Zahlen wie **17** oder **-5** oder Zeichenwerte wie **'A'** oder **'!'**) kann man grundsätzlich **nicht vereinbaren**. Einen **Wert** kann man nur **berechnen** lassen oder durch einen Ausdruck beschreiben, wie im nächsten Abschnitt näher erläutert wird.

4.2. Ausdrücke

Ein **Ausdruck** (expression) ist ein Befehl des Programmierers an den Ausführer, einen bestimmten **Wert zu berechnen**. Typischerweise muß der Ausführer dazu in Wertebehälter "hineinschauen". Von bestimmten Ausnahmen abgesehen werden die Inhalte dieser Behälter aber **nicht verändert**. Es gehört sozusagen zum tieferen Wesen von normalen Ausdrücken, daß sie **keine Behälterinhalte verändern**.

Beispiel 4.2.1.: Um den Wert des Ausdrucks **G1 + G2 + 8** zu berechnen, muß der Ausführer "nachsehen", welche Werte sich momentan in den Behältern **G1** und **G2** befinden. Dadurch werden die Inhalte dieser Behälter aber nicht verändert. **o**

Ausdrücke übersetzen wir "ganz pauschal" dadurch ins Deutsche, daß wir "**Berechne den Wert des Ausdrucks**" davor setzen. Der Ausdruck im vorigen Beispiel (4.2.1.) wird also in den folgenden deutschen Satz übersetzt: "Berechne den Wert des Ausdrucks G1 + G2 + 8".

Aufgabe 4.2.1.: Angenommen, **G1** und **G2** sind Variablen vom Typ **integer** und haben momentan die Werte **17** bzw. **-5**. Übersetzen Sie die folgenden Ausdrücke ins Deutsche und führen Sie sie aus. Berechnen Sie die Werte der Ausdrücke im Kopf oder schreibenen Sie sie auf einen Schmierzettel, aber **nicht** in einen der Behälter **G1** oder **G2**.

```
G1 + G2 + 8
10 + G1 * 3
```
o

Ausdrücke und **Werte** haben viel miteinander zu tun, sind aber völlig verschiedene "Dinge". Ein Ausdruck ist ein **Befehl**, einen Wert zu berechnen, ein Ausdruck ist aber kein Wert. In einem Wertebehälter kann immer nur ein **Wert** stehen und nie ein Ausdruck (darum heißen diese Behälter Wertebehälter und nicht Ausdrucksbehälter).

Aufgabe 4.2.2.: Übersetzen Sie die folgenden **Vereinbarungen** ins Deutsche, und führen Sie sie aus ("mit Papier und Bleistift"). Beachten Sie dabei, daß in einem Wertebehälter immer nur ein **Wert** stehen kann, und kein **Ausdruck**.

```
G3 : integer := +3;
G4 : integer := G3 * 5;
G5 : integer;
B2 : boolean := false;
B3 : boolean := not B2;
B4 : boolean;
B5 : boolean := (G3 < G4);
```

Um den richtigen Anfangswert für die Variable B5 herauszufinden, muß man den Wert des Ausdrucks **(G3 < G4)** berechnen. Ist G3 kleiner als G4 (**true**) oder ist G3 nicht kleiner als G4 (**false**)? Statt **(G3 < G4)** hätte man auch einfach **G3 < G4** (ohne Klammern) schreiben können. **o**

4.3. Anweisungen

Eine **Anweisung** (statement) ist ein Befehl des Programmierers an den Ausführer, die **Inhalte bestimmter Wertebehälter zu verändern**.

Am offensichtlichsten trifft diese Definition auf die **Zuweisungsanweisung** (kurz: Zuweisung) zu. In einer Zuweisung gibt man einen **Ausdruck** und einen **Wertebehälter** an. Der Wert des Ausdrucks wird in den Wertebehälter gelegt.

Beispiel 4.3.1. Eine Zuweisung

```
G3 := G3 + G4 + 7;
```

Diese Zuweisung könnte man etwa so ins Deutsche übersetzen:
Berechne den Wert des Ausdrucks **G3 + G4 + 7** und lege ihn in den Wertebehälter G3. **o**

Die Zuweisung **G3 := G3 + G4 + 7;** bewirkt, daß der Inhalt des Behälters G3 verändert wird. Der Ausdruck **G3 + G4 + 7** ist ein Bestandteil dieser Zuweisung. Aber es ist **nicht der Ausdruck**, der

die Veränderung des G3-Behälterinhaltes bewirkt, sondern die **Zuweisung**. Während der Ausführer den Wert des Ausdrucks berechnet, verändert er (noch) keinen Behälterinhalt. Erst **nachdem** er den Wert des Ausdrucks fertig berechnet hat, legt er ihn in den Behälter G3. Wenn man diesen etwas subtilen Punkt übersieht, könnte man zu der falschen Ansicht gelangen, daß hier durch die Auswertung eines Ausdrucks der Inhalt eines Behälters verändert wird.

Bisher haben wir nur **Variablen** als Wertebehälter bezeichnet. Jetzt soll die Bedeutung des Wortes "Wertebehälter" erweitert werden, so daß auch **Ein- und Ausgabegeräte** wie z.B. Bildschirme, Tastaturen, Drucker, Magnetplatten etc. als **Wertebehälter** gelten. Wenn man einen Wert "in einen Bildschirm legt" (d.h. zum Bildschirm ausgibt), dann erscheint ein Abbild dieses Wertes auf dem Bildschirm. Einen Wert "aus einer Tastatur holten" soll heißen, diesen Wert von der Tastatur einlesen etc..

Damit wird klar, daß alle Ein- und Ausgabebefehle **Anweisungen** sind. Eine Ausgabeanweisung legt die auszugebenden Werte in ein Ausgabegerät (z.B. in einen Bildschirm, in einen Drucker, in eine Magnetplatte etc.). Eine Eingabeanweisung holt Werte aus einem Eingabegerät (z.B. aus einer Tastatur oder aus einer Magnetplatte etc.) und legt sie in eine Variable. In jedem Fall werden bestimmte **Werte** in bestimmte **Behälter** getan.

Es gibt einige Befehle (des Programmierers an den Ausführer) deren Anweisungs-Charakter nicht so offensichtlich ist, wie bei der Zuweisung oder bei Ein- und Ausgabe-Anweisungen.

Aufgabe 4.3.1.: (Nur für Leser mit umfangreichen Vorkenntnissen!): In vielen Sprachen gibt es eine sogenannte **goto-Anweisung**. Inwiefern ist das eine **Anweisung**? Den Inhalt von welchem Behälter verändert eine solche Anweisung?

Zusammenfassung 4.: Es gibt drei Arten von Befehlen:

- Mit einer **Vereinbarung** (declaration) befiehlt der Programmierer dem Ausführer, etwas zu **erzeugen**, z.B. eine Variable oder ein Unterprogramm.
- Mit einem **Ausdruck** (expression) befiehlt der Programmierer dem Ausführer, einen Wert zu **berechnen**.
- Mit einer **Anweisung** (statement) befiehlt der Programmierer dem Ausführer, den Inhalt gewisser **Wertebehälter** (Variablen oder Ein-/Ausgabegeräte) **zu verändern**.

Um ehrlich zu sein: Es gibt ein paar Befehle, die man nicht ohne weiteres einer der drei Befehlsarten (Vereinbarung, Ausdruck, Anweisung) zuordnen kann. Ein Beispiel für solche "schwer zu klassifizierende Befehle" ist die **with-Klausel** in Ada (siehe Abschnitt 6). **o**

5. Organisation und Form der Beispielprogramme

Zu diesem Skript gehört eine Diskette mit etwa 200 meist kleineren **Beispielprogrammen**. Viele dieser Programme lesen ein paar Daten von der Tastatur ein (z.B. zwei Zahlen oder drei Zeichen etc.) und geben ein paar Zeilen zum Bildschirm aus. Einige lesen keine Daten ein, sondern geben gleich ein paar Zeilen aus. Eine gute Übung besteht darin, sich die Ausgaben dieser Programme anhand des Programmtextes (und der Eingaben) genau zu erklären ("Warum gibt das Programm gerade diese Daten aus?"). Eine noch bessere Übung besteht darin, sich zuerst nur den Programmtext anzusehen und die Ausgaben des Programms selbst "mit Papier und Bleistift" zu berechnen. Anschließend kann man das Programm dann von einem Computer ausführen lassen und prüfen, ob dabei die gleichen Ausgaben herauskommen.

Einige der Programme geben Daten in Dateien aus. Diese Ausgabedateien haben immer den gleichen Namen wie das Programm und die Erweiterung **.AUS**. Z.B. heißt die Ausgabedatei des Programms **EATXT_03** einfach **EATXT_03.AUS**.

Die **Namen** der Beispielprogramme sind (mit ganz wenigen Ausnahmen) genau 8 Zeichen lang, so daß man sie auch auf älteren DOS-Systemen ohne Probleme umwandeln und ausführen lassen kann. Die ersten 5 Zeichen eines Programmnamens geben einen Hinweis darauf, mit welchem Stoffgebiet das Programm vor allem zu tun hat: "GANZT" bedeutet "Ganzzahltypen", "AUFZT" steht für "Aufzählungstypen", "IFANW" für "if-Anweisungen", "CASEA" für "case-Anweisungen", "LOOPS" für Schleifen etc..

Den Text eines Unterprogramms namens **HALLO_01** findet man auf der Diskette in der **Datei** namens **HALLO_01.adb**. Den Text eines Paketes oder einer Schablone namens **SCHAB_01** findet man in den beiden Dateien **SCHAB_01.ads** und **SCHAB_01.adb**.

Einige dieser Beispielprogramme sind ganz oder teilweise im Text des Skripts wiedergegeben. Dabei wurden die Zeilen des Programmtextes grundsätzlich **numeriert** und wichtige Worte **fett** hervorgehoben. Die Zeilennummern und die fetten Hervorhebungen gehören aber nicht zum Programmtext und sind in den Programmen auf der Diskette nicht vorhanden.

Die Beispielprogramme auf der Diskette wurden alle von mindestens einem validierten Ada-Compiler akzeptiert und sollten deshalb keine formalen Fehler enthalten. Die im Skript wiedergegebenen Programmtexte sollte man aber grundsätzlich mit ein bißchen Skepsis lesen, da sie zumindest "von Hand nachbearbeitet" wurden. Dabei können sich leicht ein paar Fehler eingeschlichen haben. In Zweifelsfällen sind die Programmtexte auf der Diskette zuverläßiger als die Texte im Skript.

6. Das unvermeidliche Hallo-Programm

Hier ein erstes vollständiges Ada-Programm. Die Zeilennummern am linken Rand und die fetten Hervorhebungen gehören **nicht** zum Programmtext.

```
01 with ada.text_io;
02 procedure HALLO_01 is
03    VOR  : character;
04    NACH : character;
05 begin
06    ada.text_io.put(item => "Bitte geben Sie Ihre Initialen (z.B. AB) ein: ");
07    ada.text_io.get(item =>  VOR);
08    ada.text_io.get(item =>  NACH);
09    ada.text_io.put(item => "Hallo " & VOR & NACH & ", wie geht es?");
10    ada.text_io.new_line;
11 end HALLO_01;
```

6.1. Erläuterungen zur Ausführung des Programms HALLO_01

Angenommen, der Programmierer hat dem Ausführer den obigen Programmtext übergeben, und der Ausführer hat ihn geprüft und akzeptiert. Was macht der Ausführer, wenn der Benutzer ihn dann auffordert, das Programm **HALLO_01** auszuführen?

1. Er erzeugt zwei Wertebehälter namens **VOR** und **NACH** vom Typ **character** (Zeile 03 und 04).
2. Er gibt die Zeichenkette "**Bitte geben Sie Ihre Initialen (z.B. AB) ein:** " (ohne die Anführungszeichen, aber mit dem angegebenen Blank am Ende) zum Bildschirm aus (Zeile 06).
3. Er liest ein Zeichen von der Tastatur ein und legt es in den Behälter **VOR** (Zeile 07)
4. Er liest ein Zeichen von der Tastatur ein und legt es in den Behälter **NACH** (Zeile 08)
5. Er verknüpft die vier Textstücke **"Hallo "**, **VOR**, **NACH** und **", wie geht es?"** zu **einem** Text und gibt diesen zum Bildschirm aus. Falls **VOR** den Wert **'A'** und **NACH** den Wert **'B'** enthält, wird also die Meldung "**Hallo AB, wie geht es?"** (ohne die Anführungszeichen) ausgegeben (Zeile 09). Mit dem Konkatenationsoperator **&** kann man zwei Texte konkatenieren, d.h. zu **einem** Text verknüpfen.
7. Er beginnt auf dem Bildschirm eine neue Zeile, d.h. er schiebt den Bildschirmzeiger (cursor) an den Anfang der nächsten Zeile (Anweisung **new_line** in Zeile 10).
8. Er zerstört die beiden Wertebehälter namens **VOR** und **NACH**, die er im 1. Schritt erzeugt hat. Solche "Zerstörungen" führt der Ausführer immer durch, bevor er ein Unterprogramm (z.B. eine Prozedur wie **HALLO_01**) beendet. Der Programmierer braucht ihm dazu keinen Befehl zu geben.

6.2. Erläuterungen zum Paket ada.text_io:

Das Paket **ada.text_io** ist ein **Standardpaket**. D.h. es muß sich "schon immer" in der Bibliothek eines jeden Ada-Ausführers befinden.

Das Paket **ada.text_io** enthält vor allem **Unterprogramme** zum Ein- und Ausgeben von Texten. Ein Text besteht aus einzelnen Zeichen (vom Typ **character**) bzw. aus Zeichenketten (vom Typ **string**). Das Paket **ada.text_io** enthält **unter anderem** ein Unterprogramm namens **put**, mit dem man eine Zeichenkette vom Typ **string** zur aktuellen Ausgabe ausgeben kann. Die **aktuelle Ausgabe** ist in aller Regel der **Bildschirm**, vor dem der Benutzer sitzt. Der volle Name des **put**-Unter-

programms lautet **ada.text_io.put**. Im Programm **HALLO_01** wird dieses Unterprogramm zweimal aufgerufen (in den Zeilen 06 und 09).

Außerdem enthält das Paket **ada.text_io** ein Unterprogramm namens **get**, mit dem man ein einzelnes Zeichen vom Typ **character** von der aktuellen Eingabe (das ist in der Regel die Tastatur) einlesen kann. Die aktuelle Eingabe ist in aller Regel die Tastatur, vor der der Benutzer sitzt. Im Programm **HALLO_01** wird das Unterprogramm **ada.text_io.get** ebenfalls zweimal aufgerufen (in den Zeilen 07 und 08).

Anmerkung: Das Paket **ada.text_io** enthält noch viele weitere Unterprogramme, mit denen man z.B. eine ganze Zeichenkette einlesen oder ein einzelnes Zeichen ausgeben kann. Einige dieser weiteren Unterprogramme werden wir später noch kennenlernen. Eine vollständige Beschreibung des Pakets **ada.text_io** findet man im (ARM A.10).

Weil der Programmierer in der Prozedur **HALLO_01** Unterprogramme aus dem Paket **ada.text_io** aufruft, mußte er **vor** die Prozedur die Kontextklausel **with ada.text_io;** schreiben (in Zeile 01). **Kontextklausel** soll bedeuten: Die Prozedur HALLO_01 kann nur im **Kontext** des Paketes **ada.text_io** (d.h. im Zusammenhang mit dem Paket **ada.text_io**) ausgeführt werden. Die Kontextklausel **with ada.text_io;** werden wir sehr häufig benutzen.

6.3. Erläuterungen zur Struktur des Programms HALLO_01:

Ein Ada-**Programm** besteht aus **Programmeinheiten**, und zwar aus einem **Unterprogramm** und seinem **Kontext** (der durch Kontext-Klauseln vor dem Unterprogramm beschrieben wird).

Das **Programm** HALLO_01 besteht aus 2 **Programmeinheiten** (program units), nämlich aus dem **Unterprogramm** HALLO_01 (Zeile 02 bis 11) und dem **Paket** ada.text_io. Man beachte den feinen (aber manchmal wichtigen) Unterschied zwischen dem **Unterprogramm** HALLO_01 und dem **Programm** gleichen Namens. Das Unterprogramm HALLO_01 ist eine **Prozedur** (und nicht etwa eine Funktion).

Die Prozedur HALLO_01 besteht aus zwei Teilen, nämlich aus einem **Vereinbarungsteil** (declaration part, Zeile 03 bis 04) und einem **Anweisungsteil** (statement part, Zeile 06 bis 11). Diese beiden Teile werden durch das Schlüsselwort **begin** in Zeile 05 voneinander getrennt. Im Vereinbarungsteil dürfen nur **Vereinbarungen** stehen und im Anweisungsteil nur **Anweisungen**. **Ausdrücke** sind in Ada keine "selbständigen" Befehle. Sie können nur als Teil einer Vereinbarung (im Vereinbarungsteil) oder als Teil einer Anweisung (im Anweisungsteil) vorkommen. Z.B. steht in Zeile 09 hinter **item =>** der Ausdruck **" Hallo " & VOR & NACH & ", wie geht es?"**.

Durch eine entsprechende **Einrückung** hat der Programmierer versucht, die Struktur der Prozedur HALLO_01 leicht erkennbar zu machen. Nur die drei Zeilen, die mit den Worten **procedure**, **begin** und **end** beginnen, sind **nicht** eingerückt (die Kontextklausel **with ada.text_io** ist eine Art "Vorspann" vor der eigentlichen Prozedur und wird auch nicht eingerückt). Alle anderen Zeilen sind um eine Stufe eingerückt. So erkennt man "mit einem Blick" den Vereinbarungsteil (zwischen **procedure** und **begin**) und den Anweisungsteil (zwischen **begin** und **end**).

6.4. Übersetzung der Prozedur HALLO_01 ins Deutsche

Im Abschnitt 6.1. wurde die Bedeutung des Programms **HALLO_01** beschrieben, indem die Aktionen erläutert wurden, die der Ausführer beim Ausführen des Programms durchführen muß. Jetzt soll die Bedeutung von **HALLO_01** noch auf eine andere Weise verdeutlicht werden: Indem die einzelnen Befehle des Programms **ins Deutsche übersetzt** werden. Grundlage dieser Übersetzung ist die die folgende Vorstellung: Ein Programm besteht aus Befehlen des Programmierers an den Ausführer. Es gibt drei Arten von Befehlen: Vereinbarungen ("Erzeuge ..."), Ausdrücke ("Berechne den Wert des Ausdrucks ...") und Anweisungen ("Tue den Wert ... in den Behälter ..."). Befehle können **strukturiert** sein, d.h. ein Befehl kann andere Befehle enthalten.

Die Kontext-Klausel (**with ada.text_io;** in Zeile 01) ist nicht ganz leicht zu übersetzen, und wir ignorieren sie vorerst.

Die Zeilen 02 bis 11 stellen zusammengenommen **einen** einzigen (strukturierten) Befehl dar, und zwar eine **Vereinbarung**. Vereinbart wird dadurch eine Prozedur namens HALLO_01. Die Prozedurvereinbarung **enthält** zwei weitere **Vereinbarungen** (in Zeile 02 und 03) und fünf **Anweisungen** (Zeile 06 bis 10).

Die gesamte Prozedurvereinbarung (Zeile 02 bis 11) kann man etwa so ins Deutsche übersetzen:

Erzeuge eine Prozedur namens **HALLO_01**, die aus den folgenden sieben Befehlen (zwei Vereinbarungen und fünf Anweisungen) besteht:
1. Erzeuge eine Variable namens **VOR** vom Typ character.
2. Erzeuge eine Variable namens **NACH** vom Typ character.
3. Nimm den Wert des Zeichenkettenliterals **"Bitte geben Sie Ihre Initialen (z.B. AB) ein: "** als item-Parameter und führe damit die Prozedur namens **put** aus dem Paket **ada.text_io** aus.
4. Nimm die Variable **VOR** als item-Parameter und führe damit die Prozedur **get** aus dem Paket **ada.text_io** aus.
5. Nimm die Variable **NACH** als item-Parameter und führe damit die Prozedur **ada.text_io.get** aus.
6. Berechne den Wert des Ausdrucks **"Hallo " & VOR & NACH & ", wie geht es?"**, nimm diesen Wert als item-Parameter und führe damit die Prozedur **ada.text_io.put** aus.
7. Führe die Prozedur **ada.text_io.new_line** aus.

Man sieht: Auch die Übersetzung der Prozedur ist **strukturiert** und besteht aus einem **umfassenden** Befehl ("Erzeuge eine Prozedur namens HALLO_01, die ...") und sieben darin enthaltenen, **inneren** Befehlen.

Wenn der Programmierer dem Ausführer die Prozedurvereinbarung übergibt, führt dieser nur den umfassenden Befehl aus, d.h. er erzeugt eine Prozedur namens HALLO_01 und legt sie in seiner Bibliothek ab.

Wenn danach der Benutzer den Ausführer dazu auffordert, das Programm **HALLO_01** auszuführen, holt dieser die Prozedur **HALLO_01** und das Paket **ada.text_io** aus seiner Bibliothek und führt die sieben inneren Befehle der Prozedur aus, d.h. er erzeugt eine Variable namens **VOR** vom Typ **character**, erzeugt dann eine Variable namens **NACH** ebenfalls vom Typ character, gibt die Zeichenkette **"Bitte geben Sie Ihre Initialen (z.B. AB) ein: "** zum Bildschirm aus, ... etc. und zerstört schließlich die Variablen **VOR** und **NACH** wieder.

Eigentlich müßte man die Übersetzung der (Vereinbarung der) Prozedur **HALLO_01** noch ergänzen um die Übersetzungen der Prozeduren **put**, **get** und **new_line** aus dem Paket **ada.text_io**. Diese Übersetzungen werden hier aber weggelassen, in der Hoffnung, daß die Hinweise zur Wirkung von **put**, **get** und **new_line** im Abschnitt 6.1. vorläufig ausreichen.

Man beachte, daß es zu einigen Worten der Prozedur HALLO_01 in der Übersetzung keine direkte Entsprechung gibt, und zwar zu **is** in Zeile 01, **begin** in Zeile 05 und **end** in Zeile 11. Diese Worte dienen dazu, die einzelnen Teile der Prozedur leicht erkennbar zu machen und brauchen nicht übersetzt zu werden. Im Englischen werde solche Worte (mit einem Hauch von Geringschätzung) als **syntactic sugar** bezeichnet.

6.5. Aktionen eines menschlichen Ausführers

Angenommen, die LeserIn ist bereit, die drei Hauptrollen der Programmierung (d.h. die Rollen des Programmierers, des Ausführers und des Benutzers) selbst zu übernehmen. Wie könnte sie dann die charakteristischen Tätigkeiten der einzelnen Rollen (insbesondere die Tätigkeiten des Ausführers) konkret durchführen?

Um ehrlich zu sein: Ein **menschlicher** Ausführer sollte nicht versuchen, einen **maschinellen** Ausführer in allen Einzelheiten möglichst genau nachzuahmen. Vielmehr sollte er mit der Ausführung eines Programms das Ziel verfolgen, sich bestimmte Grundprinzipien der Programmausführung oder die genaue Bedeutung des ausgeführten Programms anschaulich klar zu machen. Von einigen Tätigkeiten genügt es, daß man sie sich vorstellt und im Prinzip versteht, ohne daß man sie in allen Einzelheiten wirklich durchführt.

Als Vorbereitung sollte die LeserIn sich (in der Rolle des **Ausführers**) **eine Bibliothek einrichten** (z.B. in Form eines Kartons, auf dem "Bibliothek" steht, oder indem sie einen bestimmten Bereich ihres Schreibtischs dafür reserviert). In dieser Bibliothek (so sollte sich die LeserIn vorstellen) befinden sich alle Standard-Programmeinheiten, darunter das Standard-Paket **ada.text_io**. Im Paket **ada.text_io** befinden sich (unter anderem) Prozeduren namens **put** (zum Ausgeben einer Zeichenkette zum Bildschirm), **get** (zum Einlesen eines einzelnen Zeichens von der Tastatur) und **new_line** (zum Vorschieben des Bildschirmzeigers zum Anfang der nächsten Zeile).

Wer will, kann das Paket **ada.text_io** auch durch ein Blatt Papier realisieren, auf dem die Prozeduren **put**, **get**, **new_line** etc. beschrieben werden. Dieses Papier muß man in seine Bibliothek legen, **bevor** der Programmierer einem einen Programmtext übergibt und ehe der Benutzer einen zur Ausführung eines Programms auffordert.

In der Rolle des **Programmierers** könnte die LeserIn dann die Vereinbarung der Prozedur **HALLO_01** (siehe oben, Zeile 01 bis 11) auf ein (weiteres) Blatt Papier schreiben (oder kopieren etc.) und das Papier dem **Ausführer** übergeben.

Daraufhin müßte der Ausführer den übergebenen Programmtext eigentlich gründlich und nach genau festgelegten Regeln auf formale Fehler prüfen. Da diese Tätigkeit aber kaum tiefe Einsichten bringt, sollte man sie einfach überspringen und durch andere Maßnahmen sicherstellen, daß der übergebene Programmtext keine formalen Fehler enthält (z.B. indem man den Programmtext besonders sorgfältig aus einer verläßlichen Quelle abschreibt oder ihn nebenbei von einem maschinellen Ausführer prüfen läßt).

Anmerkung: Alle Beispielprogramme auf der Begleitdiskette zu diesem Skript sind von mindestens einem maschinellen Ada-Ausführer geprüft und akzeptiert worden. Die Programmtexte im Skript sollten mit einem gewissen Mißtrauen gelesen werden, da sie (nach der Prüfung durch einen Ada-Ausführer) noch "von Hand überarbeitet" wurden.

Wir gehen hier also davon aus, daß die Vereinbarung der Prozedur HALLO_01 keine formalen Fehler enthält, daß der Ausführer sie somit akzeptiert und die Prozedur HALLO_01 in seine Bibliothek legt.

Grundlegend wichtig ist der folgende Sachverhalt: Der Text, den der Programmierer aufgeschrieben hat, ist genaugenommen keine **Prozedur**, sondern "nur" eine **Prozedurvereinbarung**, d.h. ein Befehl an den Ausführer, eine Prozedur (namens HALLO_01 die ... etc.) zu erzeugen. Erst wenn der Ausführer den Text akzeptiert und den Erzeugungsbefehl ausgeführt hat, existiert die Prozedur. Kurz: Die **Prozedur** HALLO_01 ist das, was in der Bibliothek des Ausführers steht, nicht die **Vereinbarung** "auf dem Blatt Papier des Programmierers".

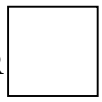
Wir werden später Beispielprogramme kennenlernen, bei denen der Unterschied zwischen einer **Prozedurvereinbarung** und der vereinbarten **Prozedur** ganz konkret wichtig ist. Aber schon jetzt sollte die LeserIn (in der Rolle des Ausführers) diesen Unterschied deutlich machen, indem sie die Vereinbarung der Prozedur HALLO_01 nicht völlig unverändert in ihre Bibliothek legt, sondern sie z.B. mit der Aufschrift "Vereinbarung akzeptiert und Prozedur erzeugt am ..." versieht (ein typischer maschineller Ausführer nimmt noch viel stärkere "Umformungen" vor, wenn er aufgrund einer Vereinbarung eine Prozedur erzeugt).

Bisher hat also der Programmierer dem Ausführer die Vereinbarung der Prozedur HALLO_01 übergeben, der Ausführer hat sie akzeptiert und die Prozedur HALLO_01 in seiner Bibliothek abgelegt. Wenn jetzt der **Benutzer** den Ausführer **auffordert**, das Programm HALLO_01 **auszuführen**, holt dieser die Prozedur HALLO_01 und das Paket **ada.text_io** aus seiner Bibliothek und führt die (sieben inneren Befehle der) Prozedur HALLO_01 aus.

Dazu sollte die Leserin (in der Rolle des **Ausführers**) drei Blätter Papier (oder drei Bereiche auf einem Blatt) verwenden. Das erste Blatt dient als **Bildschirm**, das zweite als **Speicher** und das dritte Blatt als Schmierzettel für **Zwischenrechnungen**.

Die sieben Befehle der Prozedur HALLO_01 kann der (menschliche) Ausführer etwa so ausführen:

1. Eine Variable namens VOR vom Typ character erzeugt er, indem er auf dem Speicher-Blatt ein Kästchen zeichnet und davor den Namen **VOR** und darunter den Typ **character** schreibt, etwa so:



character

2. Ganz entsprechend erzeugt er dann eine Variable namens **NACH**. Damit sind die beiden Vereinbarungen in der Prozedur HALLO_01 abgearbeitet.
3. Der **Ausführer** schreibt die Zeichenkette **"Bitte geben Sie Ihre Initialen (z.B. AB) ein: "** auf sein Bildschirm-Blatt.

Unmittelbar dahinter sollte der **Benutzer** (mithilfe einer gedachten Tastatur, tatsächlich mit einem Bleistift) zwei Buchstaben schreiben, z.B. **AB**. Indem er in Gedanken auf eine gedachte Return-Taste drückt, schließt der Benutzer seine Eingabe ab und schiebt den Bildschirmzeiger (cursor) an den Anfang der 2. Zeile (das ist die übliche Wirkung der Return-Taste).
4. Der **Ausführer** liest den ersten Buchstaben (z.B. **A**) von der Tastatur und schreibt ihn in den Kasten **VOR** (auf dem Speicher-Blatt).
5. Den zweiten Buchstaben des Benutzers (z.B. **B**) schreibt der **Ausführer** ganz entsprechend in die Variable **NACH**.
6. Den Wert des Ausdrucks **"Hallo " & VOR & NACH & ", wie geht es?"** berechnet der **Ausführer** (im Kopf oder auf seinem Schmierzettel, aber ohne irgendwelche Variablen in seinem Speicher zu verändern), indem er die Zeichenkette **"Hallo "**, den Inhalt der Variablen **VOR**, den Inhalt der Variablen **NACH** und die Zeichenkette **", wie geht es?"** zu einer einzigen Zeichenkette verknüpft. Das Ergebnis könnte z.B. der Wert **"Hallo AB, wie geht es?"** sein (der genaue Ergebniswert hängt natürlich vom Inhalt der Variablen **VOR** und **NACH** ab). Diesen Wert gibt der Ausführer zum Bildschirm aus, d.h. er schreibt die Zeichenkette in die zweite Zeile seines Bildschirm-Blattes.
7. Dann schiebt der **Ausführer** den Bildschirmzeiger (cursor) an den Anfang der nächsten Zeile. Zum Schluß zerstört er die Variablen **VOR** und **NACH** wieder, d.h. er radiert sie (auf seinem Speicher-Blatt) aus oder streicht sie deutlich durch.

So könnte ein menschlicher Ausführer das Programm **HALLO_01** konkret ausführen.

Aufgabe 6.5.1.: Schreiben Sie ein Programm namens **HALLO_02**, welches zwei Zeichen von der aktuellen Eingabe (d.h. von der Tastatur) einliest (z.B. **XY**) und sie in umgekehrter Reihenfolge und jedes Zeichen zweimal zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt (z.B. so: **YYXX**).

Aufgabe 6.5.2.: Übersetzen Sie die Prozedur **HALLO_02** ins Deutsche.

Aufgabe 6.5.3.: Führen Sie das Programm **HALLO_02** ("mit Papier und Bleistift") aus.

6.6. Lexikalische Regeln für Ada Programme

Ein Programmtext ist eine Folge von Zeichen. Gewisse Teilfolgen einer solchen Zeichenfolge ("Zeichen die gemeinsam eine sinnvolle Einheit bilden") bezeichnet man als **lexikalische Elemente**. Z.B. sind **HALLO_01**, **begin** und **&** drei lexikalische Elemente, aber die ersten sechs Zeichen **HALLO_** von HALLO_01 bilden **kein** lexikalisches Element.

Der Ada-Ausführer unterscheidet **nicht** zwischen großen und kleinen Buchstaben. Statt **HALLO_01** könnte man auch **Hallo_01** oder **hallo_01** oder sogar **hAlLo_01** schreiben und statt **begin** auch **BEGIN** oder sogar **BeGiN**. Im Interesse seiner Kollegen (und auch in seinem eigenen Interesse) sollte der Programmierer sich aber eine vernünftige und vor allem **einheitliche** Schreibweise angewöhnen. Z.B. sollte er immer **begin** oder immer **Begin**, aber nicht manchmal **begin** und manchmal **Begin** schreiben.

Groß- und Kleinschreibung: In diesem Skript wird folgende Regel eingehalten: In allen lexikalischen Elementen, deren Bedeutung durch die Sprache **Ada** festgelegt ist, werden alle

Buchstaben **klein** geschrieben, z.B. **procedure**, **begin**, **ada**, **text_io**, **character**, **boolean**, **integer** etc.. In allen lexikalischen Elementen, die der **Programmierer** erfunden hat, werden alle Buchstaben **groß** geschrieben, z.B. **HALLO_01**, **VOR**, **NACH** etc..

Praktisch bedeutet das: Wenn ein Leser ein kleingeschriebenes Wort nicht versteht wie z.B. **begin**, **procedure** oder **integer**, kann er im Ada Reference Manual (ARM) nachsehen (z.B. im Index). Wenn er dagegen von einem großgeschriebenen Wort wie z.B. **HALLO_01** oder **NACH** nicht weiß, was es bedeutet, muß er den entsprechenden Programmtext noch einmal sorgfältig durchlesen (vor allem den Vereinbarungsteil), den Programmierer befragen oder in der Dokumentation zu dem Programm nachlesen (falls Dokumentation vorhanden ist).

Lexikalische Regel Nr. 1: Jedes lexikalische Element muß **vollständig auf einer Zeile** stehen. D.h. eine lexikalisches Element darf nicht auf einer Zeile anfangen und auf einer anderen Zeile fortgesetzt bzw. beendet werden.

Für die **maximale Zeilenlänge** ist im ARM nur eine untere Grenze festgelegt: Jeder Ada-Ausführer **muß** Zeilen bis zu einer Länge von **200** Zeichen akzeptieren. Ein Ada-Ausführer **darf** aber auch längere Zeilen akzeptieren.

Praktisch bedeutet das: Der Programmierer sollte eine Zeilenlänge von 200 Zeichen möglichst nie überschreiten. Im Zusammenhang mit bestimmten Bildschirmen ist es häufig sehr empfehlenswert, sich freiwillig auf eine Zeilenlänge von etwa **75 Zeichen** zu beschränken.

Die folgenden "Feinheiten" kann man bei einem ersten Lesen kurz überfliegen und später bei Bedarf genauer ansehen.

Man unterscheidet in Ada **sieben Arten** von lexikalischen Elementen:

1. **Reservierte Wörter** (reserved words) wie z.B. **procedure**, **with**, **begin**, **is**, **end**. Eine vollständige Liste aller 69 reservierten Wörter findet man im (ARM 2.9).

2. **Bezeichner** (identifier) wie z.B. **HALLO_01**, **VOR**, **NACH**, **text_io**, **ada**, **new_line**, **get**, **put**, **character**, **boolean**, **integer**. Ein Bezeichner muß mit einem Buchstaben anfangen und darf ansonsten nur aus Buchstaben, Ziffern und Unterstrichen bestehen. Ein Unterstrich '_' darf nur **zwischen** zwei anderen Zeichen (Buchstaben oder Ziffern) stehen. Ein Bezeichner darf nicht mit einem reservierten Wort übereinstimmen.

3. **Numerische Literale** (numeric literals) wie z.B. die Ganzzahlliterale **12**, **0**, **1E6**, **123_456**, **1_000_000_000** oder die Bruchzahlliterale (real literals) **12.0**, **0.0**, **0.456**, **3.14159_25**. Mit Unterstrichen kann man numerische Literale **lesbarer** gestalten, sie haben aber keinerlei Einfluß auf den **Wert** des Literals. Z.B. haben die folgenden numerischen Literale alle den gleichen Wert: **1000000000**, **10000_00000**, **1_000_000_000**, **1_0_0_0_0_0_0_0_0_0**, **1E9**, **100e7**. Ein Unterstrich in einem numerischen Literal darf nur **zwischen** zwei Ziffern stehen.

4. **Zeichenliterale** (character literals) wie z.B. **'A'**, **'Z'**, **'a'**, **'z'**, **'0'**, **'9'**, **'!'**, **'?'**, **'"'**, **'+'**, **'-'**, **'*'** etc.. In einem Zeichenliteral unterscheidet der Ada-Ausführer zwischen großen und kleinen Buchstaben (entgegen dem, was oben vorläufig und vereinfachend behauptet wurde). Z.B. sind **'A'** und **'a'** verschiedene Zeichen.

5. **Zeichenkettenliterale** (string literals) wie z.B. **"Hallo!"**, **"abc???"**, **"+"**, **""**. Die Zeichenkette **""** besteht aus **null** Zeichen und wird auch **leere Zeichenkette** genannt. Die leere Zeichenkette spielt beim Umgang mit Zeichenketten eine ähnliche Rolle wie die 0 beim Rechnen mit Zahlen, sie ist also nicht etwa nutzlos, sondern im Gegenteil sehr nützlich und wichtig. Die **leere Zeichenkette** hat nichts zu tun mit Zeichenketten wie z.B. " " oder " ", die aus einem oderer mehreren sogenannten **Leerzeichen** bestehen. Um eine Verwechslung der leeren Zeichenkette mit Zeichenketten-die-aus-Leerzeichen-bestehen zu vermeiden, wird das Zeichen, "welches keine Druckerschwärze verbraucht" in diesem Skript als **Blank** bezeichnet (und nicht als Leerzeichen). Konkret kann man sich merken: Auf einem üblichen Bildschirm ist ein **Blank** ungefähr 2 bis 3 Millimeter breit, die **leere Zeichenkette** dagegen nimmt exakt **null** Millimeter in Anspruch.

Auch ein Zeichenkettenliteral muß, wie alle lexikalischen Elemente in Ada, vollständig auf **einer** Zeile stehen. Beliebig lange Zeichenketten kann man durch **Ausdrücke** darstellen, in denen mehrere Zeichenkettenliterale mit dem Konkatenationsoperator **&** verknüpft werden, z.B. so:

```
"Der kleine Prinz stieg auf einen hohen Berg. Die " &
"einzigen Berge, die er kannte, waren die drei Vul"
& "kane und sie reichten ihm nur bis ans Kni"
&
"e, und den erloschenen Vulkan benutzte er als Schemel"
```

Die Zerlegung der gesamten Zeichenkette in mehrere Literale wurde in diesem Beispiel (wenn man es freundlich ausdrückt) **nicht besonders schön** gelöst, soll aber verschiedene Möglichkeiten bei der Zerlegung der Zeichenkette und bei der Plazierung des Operators **&** (nach einem Literal, vor dem nächsten Literal, allein auf einer Zeile) demonstrieren.

Eine **Zeichenkette**, die aus **einem einzigen** Zeichen besteht (z.B. **"A"**, **"z"** oder **"?"**) ist etwas ganz anderes, als das betreffende einzelne **Zeichen** (im Beispiel **'A'**, **'z'** bzw. **'?'**). Dieser Unterschied ist in vielen Programmiersprachen wichtig (z.B. in Pascal, C und Java) und hat mit dem **Typensystem** zu tun (**character** und **string** sind zwei verschiedene Typen).

Auch in Zeichenkettenliteralen unterscheidet der Ada-Ausführer kleine und große Buchstaben (entgegen dem, was oben vorläufig und vereinfachend behauptet wurde). Z.B. sind **"Hallo!"** und **"hallo!"** zwei verschiedene Zeichenketten.

6. **Begrenzer** (delimiter), nämlich die 16 **Sonderzeichen** (special characters)
& ' ( ) + - * / , . ; : < = > |
und die zehn **zusammengesetzten Begrenzer** (compound delimiter, von denen jeder aus genau zwei Zeichen besteht): **=> .. ** := /= >= <= << >> <>**

Lexikalische Regel Nr. 2: Zwischen je zwei lexikalische Elemente **darf** man beliebig viele **Trennzeichen** (Blank, Tabulatorzeichen oder Zeilenende) einfügen. Bestimmte lexikalische Elemente **müssen** durch mindestens ein **Trennzeichen** voneinander getrennt werden. Trennzeichen werden im Englischen einzeln als **separator** oder gemeinsam als **white space** bezeichnet (z.B. in "White space after a program unit has no influence on the meaning of the program unit").

Z.B. muß man das reservierte Wort **procedure** und den nachfolgenden Bezeichner **HALLO_01** durch mindestens ein Trennzeichen trennen (ohne Trennzeichen würde **procedureHALLO_01**

einfach als ein neuer Bezeichner gelten, der aus 17 Zeichen besteht), und statt **ada.text_io** darf man auch
ada . text_io oder sogar
ada
.
text_io
schreiben. Siehe dazu auch das Beispielprogramm HALLO_03.

Weitere Einzelheiten (z.B. eine nicht ganz einfache Antwort auf die Frage: Welche Zeichen zählen als Buchstaben?) findet man im (ARM 2).

Zum Abschluß dieses Abschnitts noch ein kleiner praktischer Tip: Die Umlaute (**ä**, **ö**, **ü**) und das Esszett (**ß**) sind in Ada95-Programmen zwar erlaubt, aber man sollte ihre Verwendung trotzdem möglichst vermeiden. Denn Ada-Ausführer stellen alle Zeichen entsprechend dem internationalen Standard **ISO 10646 BMP** (basic multilingual plain) dar, aber Betriebssysteme wie DOS, Windows, Unix und MacOS halten sich (noch?) nicht an diesen Standard. Mit der Verwendung der Umlaute und des Esszett handelt man sich bei diesen Betriebsystemen also leicht Ärger ein, den man tunlichst vermeiden sollte.

6.7. Ein Programm welches aus mehreren Unterprogrammen besteht.

Eine Prozedur wie **HALLO_01** kann der **Benutzer** als "Hauptprozedur" starten, indem er den Namen HALLO_01 z.B. über eine Tastatur eintippt. Die gleiche Prozedur kann der Programmierer z.B. in einer anderen Prozedur als Unterprogramm aufrufen.

Das folgende Beispiel zeigt ein Programm, welches aus einem Hauptunterprogramm **HALLO_10**, zwei weiteren "echten" Unterprogrammen **HALLO_11** und **HALLO_12** und dem Paket **ada.text_io** besteht.

Beispiel 6.7.: Das Programm HALLO_10:

```
01 with ada.text_io;
02 procedure HALLO_11 is
03 begin
04    ada.text_io.put_line(item => "Hier ist HALLO_11!");
05 end HALLO_11;


06 with ada.text_io;
07 procedure HALLO_12 is
08 begin
09    ada.text_io.put_line(item => "Hier ist HALLO_12!");
10 end HALLO_12;

11 with HALLO_11, HALLO_12, ada.text_io;
12 procedure HALLO_10 is
13 begin
14    ada.text_io.put_line(item => "Hier ist HALLO_10, Stelle 1");
15    HALLO_11;
16    ada.text_io.put_line(item => "Hier ist HALLO_10, Stelle 2");
17    HALLO_11;
18    ada.text_io.put_line(item => "Hier ist HALLO_10, Stelle 3");
19    HALLO_12;
20    ada.text_io.put_line(item => "Hier ist HALLO_10, Stelle 4");
```

```
21 end HALLO_10;
```

Ein Aufruf der Prozedur **put_line** (in Zeile 04, 09, 14 etc.) leistet das Gleiche, wie ein Aufruf von **put** und ein nachfolgender Aufruf von **new_line** ("**put_line** ist gleich **put** plus **new_line**").

Die **Kontextklause** vor der Prozedur **HALLO_10** (in Zeile 11) drückt aus, daß **HALLO_10** nur zusammen mit den Unterprogrammen **HALLO_11** und **HALLO_12** und dem Paket **ada.text_io** ausgeführt werden kann. Deshalb muß man dem Ausführer die Unterprogramme **HALLO_11** und **HALLO_12** auch zeitlich **vor** dem Unterprogramm **HALLO_10** übergeben.

Aufgabe 6.7.1.: Was gibt das Programm **HALLO_10** zum Bildschirm aus? **o**

Aufgabe 6.7.2.: Schreiben Sie ein Programm, welches aus drei Unterprogrammen namens **HALLO_20**, **HALLO_21** und **HALLO_22** (und dem Paket **ada.text_io**) besteht. **HALLO_21** soll nur die Zeichenkette "----------" (d.h. 20 Minuszeichen) ausgeben. **HALLO_22** soll nur die Zeichenkette "| |" (d.h. zwei senkrechte Striche mit 18 Blanks dazwischen) ausgeben. Im Hauptunterprogramm **HALLO_20** sollen **HALLO_21** und **HALLO_22** so aufgerufen werden, daß auf dem Bildschirm ein **Rechteck** erscheint (20 Spalten breit und 5 Zeilen hoch), etwa so:

```
--------------------
|                  |
|                  |
|                  |
--------------------
o
```

Zentraldokument: Nach Filialdokument A95-01-06

7. Mit Ganzzahlen rechnen

Dieser Abschnitt hat Ähnlichkeit mit einer Kletterpartie durch eine sehr steile Felswand. Ganz oben, im Abschnitt 7.3. winkt ein kleines Beispielprogramm namens **GANZT_01**, welches zwei Ganzzahlen einliest, addiert und die Summe ausgibt. Um dieses (einfache?) Programm nicht nur oberflächlich zu verstehen, müssen wir uns vorher hinablassen und im Abschnitt 7.1. ein paar grundlegende **elektrotechnische Eigenarten** heutiger Computer betrachten. Diese Eigenarten muß man kennen, um (in irgendeiner Sprache) Programme zu schreiben, die gleichzeitig **portabel** ("auf verschiedenen Computern ausführbar") und **effizient** ("für den Ausführer nicht mit unnötig viel Arbeit verbunden") sind. Im Abschnitt 7.2. werden dann erste Einzelheiten über **Typen** und **Schablonen** in Ada erläutert. Der Aufstieg wird sehr steil und schwierig sein, aber von oben hat man eine sehr gute Übersicht über eine wichtige "Gegend der Programmierung".

7.1. Mathematik und Elektrotechnik

In die Konstruktion heutiger (1998) Computer sind zahlreiche **mathematische Ideen** eingegangen, die zum Teil schon sehr alt sind, z.B. die Idee der **ganzen Zahlen** und daß man sie **addieren** und **multiplizieren** kann. Einige dieser mathematischen Ideen sind aber durch die Möglichkeiten und Grenzen der **Elektrotechnik** beeinflußt und modifiziert worden. Ein wichtiges Beispiel für diese Vermischung von mathematischen und elektrotechnischen Einflüssen ist die Art, wie heute übliche Computer **Zahlen darstellen** und mit ihnen **rechnen**.

In der Mathematik ist **jede ganze Zahl** gleichzeitig auch eine **rationale Zahl** (d.h. eine Zahl, die sich als Bruch zweier ganzer Zahlen darstellen läßt) und mit der Additionsfunktion "+" für **rationale** Zahlen kann man somit insbesondere auch **ganze** Zahlen addieren. Z.B. sind **17**, **17/1**, **34/2** und **17.0** nur verschiedene Darstellungen oder Beschreibungen derselben Zahl **siebzehn** und ob man **17 + 5** oder **17.0 + 5.0** schreibt, ist in mathematischen Zusammenhängen in aller Regel unwesentlich.

Bei heute üblichen Computern ist das ein bißchen anders: Alle Zahlen werden dort durch bestimmte **Bitketten** dargestellt, aber für **Ganzzahlen** verwendet man ganz andere Bitcodes als für **Bruchzahlen**. Z.B. wird die **Ganzzahl 17** durch eine völlig andere Bitkette dargestellt, als die **Bruchzahl 17.0**. Entsprechend sind in einen Computer mehrere Maschinenbefehle zum Addieren eingebaut, z.B. einer zum Addieren von Ganzzahlen und ein anderer zum Addieren von Bruchzahlen.

Im folgenden werden ein paar wichtige **Eigenarten** heutiger Computer skizziert, die speziell mit der Darstellung von **Ganzzahlen** und dem Rechnen mit ihnen zu tun haben. Mit diesen Eigenarten muß man vertraut sein, um die Behandlung von Ganzzahltypen in Ada und insbesondere das Programm GANZT_01 wirklich zu verstehen. Allgemeiner muß man diese Eigenarten kennen, um (in irgendeiner Sprache) Programme schreiben zu können, die **portabel** und **effizient** sind. Ein Programm ist **portabel**, wenn man es (ohne daran manuelle Veränderungen und Anpaßungen vorzunehmen) von verschiedenen Ausführern (d.h. Computern) ausführen lassen kann. Ein Programm ist **effizient**, wenn es den Ausführer nicht zur Verschwendung von Rechenzeit oder Speicherplatz veranlaßt. Wenn z.B. ein Programm vorschreibt, daß bestimmte Berechnungen mit zwanzigstelligen Zahlen durchzuführen sind, obwohl man diese Berechnungen auch mit zehnstelligen Zahlen durchführen könnte (und der Ausführer dazu in der Lage wäre), dann ist das Programm **nicht effizient**.

Im Prinzip kann man jeden Coputer so programmieren, daß er auch mit sehr großen Ganzzahlen rechnen kann, z.B. mit Ganzzahlen, die aus ein paar Hundert oder ein paar Tausend Ziffern bestehen. "Besonders gut" kann ein (heute üblicher) Computer aber nur mit Ganzzahlen rechnen, die in bestimmten **Hardwareformaten** dargestellt sind. Die folgenden Hardwareformate sind heute besonders weit verbreitet: **B08** (8-Bit-Binärzahlen), **B12**, **B16**, **B24**, **B32** und **B48** (48-Bit-Binärzahlen).

Es gibt heute sehr viele verschiedene **Rechnerarchitekturen**, die sich unter anderem durch die Hardwareformate ihrer Ganzzahlen unterscheiden. Das macht es schwierig, wirklich **portable** Programme zu schreiben. Um diese Schwierigkeit klar darstellen zu können, sollen hier als Beispiel zwei erfundene, aber realistische Rechnerarchitekturen mit den einprägsamen Namen **Archy_8/16/32** und **Archy_24/48** skizziert werden.

Archy_8/16/32 kann besonders gut mit Ganzzahlen der Formate **B08**, **B16** und **B32** umgehen. **Archy_24/48** kann besonders gut mit Ganzzahlen der Formate **B24** und **B48** umgehen.

Hier die Bereiche von Ganzzahlen, die zu den einzelnen Hardwareformaten von **Archy_8/16/32** und **Archy_24/48** gehören:

Rechner-Architektur	Hardware-format	In dem Hardwareformat darstellbare Ganzzahlen
Archy_8/16/32	B08	-128..+127
	B16	-32_768..+32_767
	B32	-2_147_483_648..+2_147_483_647
Archy_24/48	B24	-8_388_608..+8_388_607
	B48	-140_737_488_355_328..+140_737_488_355_327

"Besonders gut umgehen können" soll bedeuten: **Archy_24/48** kann mit **einem einzigen** Maschinenbefehl z.B. zwei Ganzzahlen im B24-Format oder zwei Ganzzahlen im B48-Format addieren, subtrahieren, multiplizieren oder dividieren. Entsprechend kann **Archy_8/16/32** mit einem einzigen Maschinenbefehl zwei Zahlen im B08-, B16- oder B32-Format miteinander verrechnen.

Im Prinzip kann **Archy_8/16/32** auch mit Zahlen in dem ihm fremden B48-Format rechnen, aber er kann mit solchen Zahlen **nicht besonders gut umgehen**. Denn um z.B. zwei B48-Zahlen zu **addieren**, muß **Archy_8/16/32** jede dieser beiden Zahlen in eine 32-Bit-Teilzahl und eine 16-Bit-Teilzahl zerlegen, muß diese Teilzahlen getrennt addieren und bei der zweiten Teiladdition einen eventuellen Überlauf aus der ersten Teiladdition berücksichtigen. Kurz: Um zwei B48-Zahlen zu addieren, braucht **Archy_24/48** deutlich **mehr als einen** Maschinenbefehl (z.B. drei Maschinenbefehle oder mehr).

Jetzt könnte man vermuten, daß **Archy_24/48** mit seinem großen B48-Format es leichter hat, die kleineren Formate von **Archy_8/16/32** "nachzumachen". Dem ist aber nicht so. Auch "der mächtige" **Archy_24/48** kann mit den ihm fremden Formaten B08, B16 und B32 **nicht besonders gut umgehen**.

Um zu sehen, woran das liegt, betrachten wir die Aufgabe, drei B32-Zahlen G1, G2 und G3 zu addieren. Dabei müssen wir mit folgendem Problemfall rechnen: Zwar ist die Summer aller drei Zahlen G1 + G2 + G3 eine B32-Zahl, aber zur Darstellung des Zwischenergebnisses G1 + G2 sind

33 Bit nötig. Bei der Addition von G1 und G2 wird das B32-Format überschritten. Man sagt auch: es tritt ein (B32-) **Überlauf** auf.

Archy_8/16/32 würde einen solchen Überlauf ohne Zeitverlust und automatisch beim Addieren von G1 und G2 erkennen (durch bestimmte Prüfschaltungen, die bei jeder Addition "mitlaufen" und keine extra Zeit kosten) und würde als Ergebnis der gesamten Berechnung (G1 + G2 + G3) ebenfalls einen Überlauf herausbekommen.

Archy_24/48 müßte jede der drei B32-Zahlen zu einer B48-Zahl "aufblasen" und sie dann mit seiner B48-Addition addieren. Die B48-Addition ist aber "zu gut", um einen B32-Überlauf automatisch zu erkennen (die Prüfschaltungen der B48-Addition erkennen nur B48-Überläufe). Würde **Archy_24/48** "einfach so" alle drei Zahlen addieren, käme (in dem angenommenen Problemfall) insgesamt kein Überlauf heraus (wie bei **Archy_8/16/32**) sondern eine Zahl, die sich auch im **B32**-Format darstellen läßt. Um genau das selbe Ergebnis zu produzieren wie **Archy_8/16/32**, muß **Archy_24/48** nach jeder Addition ein paar **zusätzliche Maschinenbefehle** ausführen und damit prüfen, ob sein B48-Ergebnis sich noch als B32-Zahl darstellen läßt.

Für die Addition der drei B32-Zahlen, die **Archy_8/16/32** mit **zwei** Maschinenbefehlen erledigen kann, braucht **Archy_24/48** insgesamt etwa vier bis acht Machinenbefehle (abhängig von Einzelheiten, die hier nicht wichtig sind). Man sieht: Ein größeres Format ist nicht immer besser als ein kleineres, denn um die Überläufe eines kleineren Formats zu erkennen, muß man zusätzliche Maschinenbefehle ausführen. Das hier skizzierte Problem verschwindet also nicht, wenn demnächst Computer üblich werden, die besonders gut mit **B64**- oder mit **B96**-Zahlen rechnen können.

Fazit: Wenn man von einem heute üblichen Computer verlangt, mit Ganzzahlen eines ihm **fremden Hardwareformates** zu rechnen, erhöht man dadurch die Anzahl der auszuführenden Maschinenbefehle erheblich (z.B. um einen Faktor von zwei oder drei oder mehr). Das ist selbst dann der Fall, wenn das fremde Hardwareformat **kleiner** ist, als ein eigenes Format des Computers.

Mit dieser Erkenntnis über die Hardware heutiger Computer können wir wieder zu unserem eigentlichen Thema (Grundkonzepte von Programmiersprachen, insbesondere Grundkonzepte von Ada) zurückkehren. Stellen wir uns vor, wir entwickeln selbst eine neue Sprache namens **TMP** (wie "Typische moderne Programmiersprache"). Wir haben schon beschlossen, daß es darin drei **vordefinierte Ganzzahltypen** namens **short_integer**, **integer** und **long_integer** geben soll. Wir haben außerdem fest geplant, unsere Sprache **TMP** mindestens auf den beiden Rechnerarchitekturen **Archy_8/16/32** und **Archy_24/48** zu implementieren (d.h. **TMP**-Programme sollen mindestens von Rechnern dieser beiden Architekturen ausgeführt werden können).

Die wichtige Frage, auf die dieser Abschnitt 7.1. hinzielt, ist die folgende:

Frage: Wie soll festgelegt werden, welche Ganzzahlen zu den einzelnen Ganzzahltypen **short_integer**, **integer** und **long_integer** gehören?

Antwort1: **Die Sprache** TMP ordnet jedem Typ ein **bestimmtes Hardwareformat fest zu** (z.B. dem Typ **short_integer** das Hardwareformat **B08**, dem Typ **integer** das Format **B16** und dem Typ **long_integer** das Format **B32**, oder so ähnlich).

Antwort2: Jeder einzelne **TMP-Ausführer** (**Archy_8/16/32**, **Archy_24/48**, ...) darf den Typen der Sprache **TMP** die Hardwareformate zuordnen, mit denen er besonders gut umgehen kann (z.B. könnte **Archy_8/16/32** die in der **Antwort1** angegebene Zuordnung wählen, aber **Archy_24/48**

würde dem Typ **short_integer** das Format **B24**, dem Typ **long_integer** das Format **B48** und dem Typ **integer** wahlweise auch **B24** oder auch **B48** zuordnen).

Aufgabe 7.1.1.: Welche Vor- und Nachteile hat die **Antwort1**?

Aufgabe 7.1.2.: Welche Vor und Nachteile hat die **Antwort2**?

Aufgabe 7.1.3.: Diskutieren Sie die folgende These: "Wenn demnächst ein Pentium-Prozessor nur noch einen Dollar oder einen Euro kostet, wird es egal sein, wieviele Maschinenbefehle für eine bestimmte Ganzzahlrechnung ausgeführt werden müssen, d.h. Effizienz spielt dann keine Rolle mehr".

Zusammenfassung 7.1.: Wenn eine Programmiersprache **nur vordefinierte Ganzzahltypen** enthält, kann man in dieser Sprache kaum Programme schreiben, die gleichzeitig **portabel** und **effizient** sind. Denn falls die **Sprache** den vordefinierten Typen bestimmte Hardwareformate fest zuordnet, kann sie auf einigen Rechnerarchitekturen nicht effizient implementiert werden. Falls dagegen jeder einzelne **Ausführer** den Typen seine eigenen, speziellen Hardwareformate zuordnen darf, kann man kaum portable Programme schreiben.

7.2. Ganzzahltypen und -untertypen in Ada

Die Regeln über Ganzzahltypen sind in Ada komplizierter als in anderen Sprachen. Dafür leisten sie auch etwas mehr.

In Ada unterscheidet man **Typen** (types) und **Untertypen** (subtypes). Ein **Typ** besteht aus einer Menge von **Werten** und einer Menge von **Operationen**, die man auf diese Werte anwenden kann. Ein **Untertyp** besteht aus einem **Typ** und einer **Einschränkung** (constraint). Die Einschränkung dient dazu, eine **Teilmenge** der Werte (des Typs) zu beschreiben.

Merkwürdigerweise haben **Typen** in Ada grundsätzlich **keine Namen**. Nur ein Untertyp kann einen Namen besitzen. Weil es schwer ist, über namenlose Dinge klar zu sprechen, wird in diesem Skript folgende **Konvention** benützt: Falls ein Typ einen Untertyp namens OTTO besitzt, dann bezeichnen wir den Typ mit dem **Hilfsnamen** !OTTO. Der Untertyp OTTO besteht also aus dem Typ !OTTO und einer bestimmten Einschränkung (ein Beispiel kommt gleich).

Die Hilfsnamen wie z.B. !OTTO, mit denen in diesem Skript Typen bezeichnet werden, sind natürlich **keine richtigen Namen** und dürfen in einem Ada-Programm **nicht** verwendet werden (richtige Ada-Namen müssen mit einem Buchstaben beginnen). Die Hilfsnamen sollen uns nur dabei helfen, in Diskussionen über Ada und Ada-Programme möglichst einfach und klar zwischen **Untertypen** und **Typen** zu unterscheiden.

In Ada kann der Programmierer **eigene Ganzzahltypen** vereinbaren, um damit seine Programme portabel und effizient zu machen.

Beispiel 7.2.1.: Vereinbarung eines **signierten** ("vorzeichenbehafteten") **Ganzzahltyps**:

```
type OTTO is range -5_000_000..+1_000_000_000;
```

Diese Vereinbarung kann man etwa so ins Deutsche übersetzen: Erzeuge einen neuen **signierten Ganzzahltyp** !OTTO und einen **Ganzzahluntertyp** namens OTTO. Der Untertyp OTTO soll aus dem Typ !OTTO und der Einschränkung -5_000_000..+1_000_000_000 bestehen. **o**

Der Untertyp **OTTO** wird auch als der **erste Untertyp** (first subtype) von !OTTO bezeichnet. Im Abschnitt 8.2. wird gezeigt, wie man **weitere Untertypen** desselben Typs vereinbart.

Entscheidend wichtig ist, daß diese Vereinbarung **nicht festlegt**, welche Werte zum **Typ** !OTTO gehören sollen. Nur für den **Untertyp** OTTO wird ein bestimmter Bereich festgelegt. Das läßt jedem einzelnen Ada-Ausführer die Möglichkeit, dem **Typ** !OTTO ein Hardwareformat zuzuordnen, mit dem er besonders gut umgehen kann. Natürlich muß der Typ !OTTO **mindestens** all die Werte umfassen, die zum **Untertyp** OTTO gehören sollen (also alle Ganzzahlen im Bereich von -5_000_000 bis +1_000_000_000). Aber für einen maschinellen Ausführer ist es in aller Regel günstig, die Wertemenge des **Typs** !OTTO größer zu wählen.

Aufgabe 7.2.1.: Welches Hardwareformat würde der Ausführer **Archy_8/16/32** dem Typ !OTTO zuordnen? Welche **Wertemenge** gehört zu diesem Hardwareformat (und damit dann auch zum Typ !OTTO)? Welches Hardwareformat und welche Wertemenge würde der Ausführer **Archy_24/48** für den Typ !OTTO wählen?

Nachdem man einen (Typ und einen) Untertyp vereinbart hat, kann man **Variablen** von diesem Untertyp vereinbaren, etwa so:

Beispiel 7.2.2.: Variablen vom Untertyp OTTO vereinbaren:

```
EIN   : OTTO := 0;
G27_F : OTTO;
```

Diese Variablenvereinbarungen kann man etwa so ins Deutsche übersetzen:
Erzeuge eine Variable namens **EIN** vom Untertyp **OTTO** und gib ihr als Anfangswert den Wert **0** (oder etwas kürzer: und **initialisiere** sie mit 0).
Erzeuge eine Variable namens **G27_F** vom Untertyp **OTTO**. **o**

Die Variablen **EIN** und **G27_F** gehören nicht nur zum **Untertyp** OTTO, sondern gleichzeitig auch zum **Typ** !OTTO (weil OTTO eine Untertyp von !OTTO ist). Aber weil sie zum **Untertyp** OTTO gehören, wird der Ausführer ihnen nie einen Wert zuweisen, der nicht zu diesem Untertyp gehört. Sollte der Programmierer dem Ausführer z.B. den Befehl **EIN := -500_001;** ("weise der Variablen EIN den Wert -500_001 zu") geben, dann wird der Ausführer diesen Befehl nicht ausführen und stattdessen eine sogenannte **Ausnahme** (exception) namens **constraint_error** ("eine Einschränkung wurde verletzt") auslösen. Praktisch bedeutet das (zumindest vorläufig): Die Ausführung des betreffenden Programms wird abgebrochen und der Ausführer gibt eine Fehlermeldung aus, in der das Wort **constraint_error** vorkommt. Im Abschnitt 13. wird besprochen, wie der Programmierer Ausnahmen so behandeln kann, daß sie nicht den Abbruch der betreffenden Programmausführung bewirken.

Mit dem neuen Ganzzahltyp !OTTO erzeugt der Ausführer automatisch bestimmte **Operationen**, die man auf die Werte dieses Typs anwenden kann, darunter die üblichen Rechenoperationen **"+"** (Addition), **"-"** (Subtraktion), **"*"** Multiplikation und **"/"** (Ganzzahldivision, d.h. 4 / 3 ist gleich 1 und nicht etwa gleich 1.3 oder gleich 1.333333333 oder so).

Operationen, mit denen man Werte des Typs !OTTO (oder zumindest Werte des Untertyps OTTO) **einlesen** oder **ausgeben** kann, erzeugt der Ausführer aber **nicht** automatisch. Wenn der Programmierer solche Ein- und Ausgabeoperationen benützen möchte, muß er sie vorher ausdrücklich **vereinbaren** (d.h. vom Ausführer erzeugen lassen), etwa so:

Beispiel 7.2.3.: Vereinbarung eines Paketes, welches Ein- und Ausgabeoperationen für Werte des Untertyps OTTO enthält:

```
package OTTO_EA is new ada.text_io.integer_io(num => OTTO);
```

Diese Paketvereinbarung kann man etwa so ins Deutsche übersetzen: Erzeuge ein Paket namens **OTTO_EA** als Abdruck der Paketschablone **integer_io**, die sich im Paket **ada.text_io** befindet. Alle Befehle im neuen Paket **OTTO_EA** (z.B. die Prozeduren **get** und **put**) sollen "auf den Untertyp OTTO abgestimmt" sein, d.h. sie sollen die Ein- und Ausgabe von Werten dieses Untertyps ermöglichen. **o**

Paketschablonen wie **ada.text_io.integer_io** gehören zu den kompliziertesten, gleichzeitig aber auch mächtigsten und elegantesten Werkzeugen, die Ada dem Programmierer zur Verfügung stellt. Wenn der LeserIn das Beispiel 7.2.3. und die nachfolgenden Erläuterungen dunkel und unverständlich erscheinen, sollte sie sich davon nicht beunruhigen lassen und einfach darüber hinweglesen (d.h: "weiterklettern").

Durch die Vereinbarung im Beispiel 7.2.3. wird ein Paket namens **OTTO_EA** erzeugt. Mit dem Namen **OTTO_EA** will der Programmierer deutlich machen, daß dieses Paket "etwas mit dem Untertyp **OTTO** zu tun hat". Hätte er das Paket z.B. **EMIL** genannt oder **XF57_A**, dann wäre dieser Zusammenhang nicht so leicht erkennbar. **EA** soll an "Ein- und Ausgabe" erinnern.

Daß das Paket **OTTO_EA** etwas mit dem Untertyp **OTTO** zu tun hat, folgt aber nicht aus seinem Namen, sondern aus der Angabe **num => OTTO** (ganz rechts in der Vereinbarung). Hätte der Programmierer hinter **num =>** den Namen eines anderen Ganzzahluntertyps angegeben (z.B. **natural**), dann hätte das Paket mit diesem anderen Ganzzahluntertyp zu tun.

Die Paketschablone **integer_io** (im Paket **ada.text_io**) sieht ganz ähnlich aus wie ein Paket, enthält aber an mehreren Stellen den "Platzhalter" **num**, wo eigentlich der Name eines Ganzzahluntertyps stehen müßte. Das Paket **OTTO_EA** ist sozusagen eine **Kopie** von **integer_io**, in der aber jedes Vorkommen des Platzhalters **num** durch **OTTO** ersetzt wurde. Die Angabe **num => OTTO** bedeutet also einfach: "Ersetze jedes Vorkommen von **num** (in integer_io) durch **OTTO**". Man sagt auch: Die Paketschablone **integer_io** wurde mit dem Ganzzahluntertyp OTTO **parametrisiert**, um das Paket OTTO_EA zu erhalten, und das Paket OTTO_EA ist eine **Instanz** der Schablone **integer_io** (**Instanz** ist die offizielle Bezeichnung für einen **Abdruck**).

Wer Lust hat, kann sich die Paketschablone **integer_io** mal ansehen im (ARM A.10.1(52)). Sie enthält zwei Variablen und sechs Prozeduren.

Das Paket OTTO_EA enthält unter anderem eine Prozedur namens **get**, mit der man einen Wert des Untertyps OTTO von der aktuellen Eingabe (d.h. von der Tastatur) **einlesen** kann. Der volle Name dieser Prozedur lautet: **OTTO_EA.get**. Außerdem enthält OTTO_EA eine Prozedur namens **put**, mit der man einen Wert des Untertyps OTTO zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgeben kann. Der volle Name dieser Prozedur lautet natürlich **OTTO_EA.put**.

Die Prozeduren **OTTO_EA.get** und **OTTO_EA.put** sind **auf den Untertyp OTTO abgestimmt**. Das bedeutet: Wenn der Ausführer die Prozedur **OTTO_EA.get** ausführt (d.h. wenn er versucht, einen Wert des Untertyps OTTO von der der Tastatur einzulesen) und der Benutzer einen zu großen oder zu kleinen Wert oder eine unsinnige Zeichenkette eingibt, dann löst der Ausführer die Ausnahme **data_error** ("Eingabedaten sind fehlerhaft") aus und bricht die Programmausführung ab (mit einer Fehlermeldung, in der das Wort **data_error** vorkommt). Und wenn der Programmierer versucht, mit der Prozedur **OTTO_EA.put** z.B. die Ganzzahl +1_000_000_001 auszugeben, dann löst der Ausführer die Ausnahme **constraint_error** ("eine Einschränkung wurde verletzt") aus und bricht die Programmausführung ab (mit einer Fehlermeldung, in der das Wort **constraint_error** vorkommt).

Um ehrlich zu sein: Die Prozeduren **OTTO_EA.get** und **OTTO_EA.put** sind komplizierter, als bisher zugegeben wurde. Denn mit einer Tastatur kann man nur **Texte** (einzelne Zeichen oder ganze Zeichenketten) eingeben, aber keine **Ganzzahlwerte**, und zu einem Bildschirm kann man nur **Texte** ausgeben, aber keine **Ganzzahlwerte**. **Texte** werden im Speicher eines heute üblichen Computers durch ganz andere Bitketten dargestellt, als **Ganzzahlwerte** (z.B. hat die Bitkette für den **Text "+123"** keinerlei Ähnlichkeit mit der Bitkette für den **Ganzzahlwert +123**). Die Prozedur **OTTO_EA.get** liest also einen **Text** ein und versucht, diesen in einen **Ganzzahlwert** vom Untertyp **OTTO** umzuwandeln. Wenn ihr das nicht gelingt, wird die Ausnahme **data_error** ausgelöst. Die Prozedur **OTTO_EA.put** wandelt den Ganzzahlwert, den sie als **item**-Parameter bekommt, in einen entsprechenden Text um, und gibt diesen Text zum Bildschirm aus. Manchmal ist es nützlich zu wissen, daß bei der Ein- und Ausgabe von Ganzzahlen solche Umwandlungen zwischen **externen Texten** und **internen Ganzzahlwerten** stattfinden müssen. **o**

Zusammenfassung 7.2.:

- In Ada kann der Programmierer **neue Ganzzahltypen** und **eigene Ganzzahluntertypen** vereinbaren.
- Welche Werte zu einem neuen **Ganzzahltyp** gehören, bestimmt der jeweilige **Ausführer**.
- Welche Werte zu einem **Ganzzahluntertyp** gehören, bestimmt der **Programmierer**.
- Zu jedem Ganzzahltyp gehören bestimmte **Rechenoperationen**, z.B. +, -, *, / etc..
- Wenn der Programmierer Ganzzahlen **einlesen** oder **ausgeben** will, muß er eine entsprechende Instanz der Paketschablone **integer_io** vereinbaren.

7.3. Das Beispielprogramm GANZT_01

Hier endlich das (einfache?) Beispielprogramm **GANZT_01**:

```
01 with ada.text_io;
02 procedure GANZT_01 is
03    type    OTTO    is range -500_000_000 .. +1_000_000_000;
04    package OTTO_EA is new ada.text_io.integer_io(num => OTTO);
05    EIN1 : OTTO;
06    EIN2 : OTTO;
07 begin
08    ada.text_io.put(item => "Bitte geben Sie 2 Ganzzahlen ein: ");
09    OTTO_EA.get    (item =>  EIN1);
10    OTTO_EA.get    (item =>  EIN2);
11    ada.text_io.put(item => "Die Summe der beiden Zahlen ist ");
12    OTTO_EA.put    (item =>  EIN1 + EIN2);
13    ada.text_io.new_line;
14 end GANZT_01;
```

Die Prozedur **GANZT_01** kann man etwa so ins Deutsche übersetzen (eine Übersetzung der Kontextklausel in Zeile 01 wird ebenfalls angegeben, aber vorerst nicht weiter erläutert):

Binde das Paket **ada.text_io** in jedes Programm ein, in das du die folgende Programmeinheit (d.h. die Prozedur **GANZT_01**) einbindest (Zeile 01).

Erzeuge eine Prozedur namens **GANZT_01**, die aus den folgenden vier Vereinbarungen und sechs Anweisungen (insgesamt also aus zehn Befehlen) besteht (Zeile 02 bis 14) :

1. Erzeuge einen neuen Ganzzahltyp !OTTO und seinen ersten Untertyp **OTTO**, der aus dem neuen Typ !OTTO und der Einschränkung -500_000_000 .. +1_000_000_000 besteht (Zeile 03).

2. Erzeuge ein Paket namens **OTTO_EA** als Abdruck der Paketschablone **integer_io**. Alle Befehle im Paket **OTTO_EA** sollen "auf den Untertyp **OTTO** abgestimmt" sein (Zeile 04).

3. Erzeuge eine Variable namens **EIN1** vom Untertyp **OTTO** (Zeile 05).
4. Erzeuge eine Variable namens **EIN2** vom Untertyp **OTTO** (Zeile 06).

5. Nimm den Wert des Zeichenkettenliterals **"Bitte geben Sie zwei Ganzzahlen ein: "** als item-Parameter und führe damit die Prozedur **put** im Paket **ada.text_io aus** (Zeile 08).

6. Nimm die Variable **EIN1** als item-Parameter und führe damit die Prozedur **get** im Paket **OTTO_EA** aus (Zeile 09).
7. Nimm die Variable **EIN2** als item-Parameter und führe damit die Prozedur **OTTO_EA.get** aus (Zeile 10).

8. Nimm den Wert des Zeichenkettenliterals **"Die Summe der beiden Zahlen ist "** als item-Parameter und führe damit die Prozedur **ada.text_io.put** aus (Zeile 11).

9. Berechne den Wert des Ausdrucks **EIN1 + EIN2**, nimm diesen Wert als item-Parameter und führe damit die Prozedur **OTTO_EA.put** aus (Zeile 12).

10. Führe die Prozedur **ada.text_io.new_line** aus. **o**

Aufgabe 7.3.1.: Übergeben Sie (in der Rolle des Programmierers) die Vereinbarung der Prozedur **GANZT_01** (Zeile 01 bis 14) einem maschinellen Ada-Ausführer und fordern Sie (in der Rolle des Benutzers) den Ausführer wiederholt dazu auf, das Programm **GANZT_01** auszuführen. Geben Sie dabei (als Benutzer) die Daten ein, die in der ersten Spalte der folgenden Tabelle stehen und tragen Sie die Ergebnisse, die auf dem Bildschirm erscheinen, in die zweite Spalte der Tabelle ein. Wenn der Ausführer den Text "Die Summe der beiden Zahlen ist " und dahinter eine Zahl ausgibt, dann genübt es, diese **Zahl** als Ergebnis in die Tabelle einzutragen. Wenn der Ausführer eine Fehlermeldung ausgibt, in der das Wort **data_error** oder **constraint_error** vorkommt, dann brauchen Sie nur dieses Wort in die Tabelle einzutragen. Wenn der Ausführer eine andere Fehlermeldung ausgibt, sollten Sie die Meldung vollständig und genau abschreiben und mit einem Experten für maschinelle Ada-Ausführer besprechen.

Tabelle (mit Eingaben für GANZT_01 und Platz für die Ergebnisse):

Eingaben für GANZT_01	**Ergebnisse** von GANZT_01

5 3	
-17 +34	
+500_000_000 +500_000_000	
-200_000_000 -300_000_000	
0 1_000_000_000	
1 1_000_000_000	
-5_000_000 -1	
-5_000_001	
+1_000_000_001	
ABC	
17_	
_17	
1___7	
18 0	
2#10010# 0	
3#200# 0	
4#102# 0	
7#24# 0	
8#22# 0	
10#18# 0	
12#16# 0	
16#12# 0	
17#11# 0	
1#00# 0	

o

Aufgabe 7.3.2.: Verändern Sie in der Prozedur **GANZT_01** die **Grenzen des Untertyps OTTO** (in Zeile 03), und übergeben Sie dann die Prozedur erneut einem maschinellen Ada-Ausführer. Wenn Sie die Untergrenze zu klein oder die Obergrenze zu groß wählen, wird der Ausführer die Prozedur **ablehnen** und eine Fehlermeldung ausgeben. Versuchen Sie herauszufinden, welches die kleinste Untergrenze und die größte Obergrenze ist, die Ihr maschineller Ada-Ausführer noch akzeptiert. Probieren Sie als Untergrenze Zweierpotenzen wie z.B. **-2**15** ("minus zwei hoch fünfzehn"), **-2**31** oder **-2**63** und als Obergrenzen Zahlen wie z.B. **+2**15-1** ("plus zwei hoch fünfzehn minus eins"), **+2**31-1** oder **+2**63-1**. Sie dürfen auch im Ada-Programm Ausdrücke wie **-2**15** oder **+2**63-1** etc. verwenden. **o**

Aufgabe 7.3.3.: Ersetzen Sie in der Prozedur **GANZT_01** den Ausdruck **EIN1 + EIN2** (in Zeile 12, hinter **item** =>) z.B. durch **EIN1 * EIN2** oder **EIN1 / EIN2 + 27** oder **EIN1**2 + EIN1**3** etc.. Übergeben Sie die Prozedur **GANZT_01** dann jeweils wieder einem maschinellen Ada-Ausführer und rufen Sie das Programm **GANZT_01** wiederholt auf. Entsprechen die Ergebnisse Ihren Erwartungen? **o**

Aufgabe 7.3.4.: Schreiben Sie eine Prozedur namens **GANZT_02**, in der ein Typ **!EMIL**, sein erster Untertyp **EMIL** mit dem Bereich -30_000..+30_000 und drei Variablen namens **EIN1**, **EIN2** und **EIN3** vom Untertyp **EMIL** vereinbart werden. Die Prozedur soll drei Zahlen (in die drei Variablen) einlesen und den Wert des Ausdrucks **EIN1 + EIN2 + EIN3** ausgeben. Was passiert, wenn man das Programm **GANZT_02** ausführen läßt, und die folgenden Zahlen eingibt:
30_000 2_000 -2_000
30_000 3_000 -3_000

Können Sie die Ausgaben des Programms erklären? Berücksichtigen Sie dabei, daß Operationen wie +, -, *, / etc. immer zu einem **Typ** gehören, und nicht nur zu einem **Untertyp**, und daß der Wertebereich eines **Typs** vom **Ausführer** festgelegt wird, nicht vom Programmierer. **o**

Es ist empfehlenswert, einige Varianten des Programms **GANZT_01** (insbesondere das Programm **GANZT_02**) mit Hilfe eines **maschinellen** Ada-Ausführers **auszuprobieren**. Auf diese Weise kann man viele Einzelheiten lernen, die mühsam zu beschreiben und noch mühsamer nachzulesen wären. Die Methode des **Ausprobierens an einem Computer** ist aber auch mit sehr ernsthaften **Gefahren** verbunden. Zum einen kostet sie in aller Regel **viel Zeit**. Man sollte sie also nur dann anwenden, wenn es keine deutlich schnellere Lernmethode gibt (z.B. Nachlesen im ARM oder in anderen Schriften). Von grundsätzlicherer Bedeutung ist aber die Tatsache, daß man durch **Ausprobieren** nur die Eigenschaften **eines bestimmten Ada-Ausführers** (den man gerade benutzt) erkunden kann, nicht aber die **allgemeinen Regeln** der Sprache Ada.

Wie könnte ein **menschlicher** Ausführer das Programm **GANZT_01** ausführen? Die meisten Befehle in diesem Programm kamen schon in **HALLO_01** vor und ihre Ausführung wurde im Abschnitt 6.5. besprochen (zur Erinnerung: man braucht dazu drei Blätter Papier, die man als **Bildschirm**, als **Speicher** und als **Schmierzettel** benützt). Wirklich neu sind im Programm **GANZT_01** eigentlich nur die **Typvereinbarung** (in Zeile 03) und die **Paketvereinbarung** (in Zeile 04).

Praktisch genügt es für einen menschlichen Ausführer, wenn er diese Vereinbarungen sorgfältig durchliest und sich merkt, was die Namen **OTTO** und **OTTO_EA** bezeichnen (**OTTO** bezeichnet einen **Ganzzahluntertyp** mit dem Wertebereich -5_000_000..+1_000_000_000 und **OTTO_EA** bezeichnet ein **Paket**, welches unter anderem eine **get-** und eine **put**-Prozedur enthält, mit denen man einen Wert des Untertyps OTTO von der aktuellen Eingabe einlesen bzw. zur aktuellen Ausgabe ausgeben kann). Indem der menschliche Ausführer sich so die Bedeutung der Namen OTTO und OTTO_EA merkt, hat er (in seinem Kopf) einen Untertyp namens OTTO und ein Paket namens OTTO_EA erzeugt (und damit die Vereinbarungen in Zeile 03 und 04 ausgeführt).

Wer ein bißchen formaler vorgehen will, kann den Untertyp OTTO und das Paket OTTO_EA auch dadurch erzeugen, daß er ihre Vereinbarungen auf sein **Speicherblatt** abschreibt. Diese Vorgehensweise hat den Vorteil, daß alle vom Ausführer erzeugten Größen (Typen, Pakete, Variablen etc.) klar erkennbar auf dem Speicherblatt stehen (und nicht zum Teil nur im Kopf des Ausführers existieren). Allgemein gilt: Je einfacher ein Programm ist und je mehr Erfahrung man im Ausführen hat, desto mehr kann man die Ausführung "im Kopf durchführen" und desto weniger muß man ausdrücklich auf seine Blätter schreiben.

8. Mehrere Ganzzahltypen und Untertypen in einem Programm

Wenn der vorige Abschnitt einer steilen Kletterpartie glich, dann ist dieser Abschnitt eher ein Spaziergang auf einem sanft ansteigenden Weg. Hier soll der praktische Nutzen von **Typen** anhand einfacher Beispielen demonstriert und der Unterschied zwischen **Typen** und **Untertypen** etwas genauer dargestellt werden.

8.1. Mehrere Ganzzahltypen in einem Programm

Häufig muß man in einem Programm verschiedene Berechnungen durchführen lassen, die nichts miteinander zu tun haben. Das soll bedeuten: In jeder Berechnung werden bestimmte Variablen benützt, aber keine Variable kommt in mehreren Berechnungen vor. Durch entsprechende **Typvereinbarungen** kann der Programmierer dem Ausführer und seinen Kollegen gegenüber klar ausdrücken, welche Variablen "zusammengehören" und welche nicht.

Beispiel 8.1.1.: Zwei Ganzzahltypen

```
01 type AEPFEL is range 0..30_000;
02 type BIRNEN is range 0..30_000;
03 A1, A2 : AEPFEL := 0;
04 B1, B2 : BIRNEN := 0;
```

Diese Vereinbarungen kann man etwa so ins Deutsche übersetzen:
Erzeuge einen **neuen** Ganzzahltyp !AEPFEL und seinen ersten Untertyp namens AEPFEL, der aus dem Typ !AEPFEL und der Einschränkung 0..30_000 besteht (Zeile 01)
Erzeuge einen **neuen** Ganzzahltyp !BIRNEN und seinen ersten Untertyp namens BIRNEN, der aus dem Typ !BIRNEN und der Einschränkung 0..30_000 besteht (Zeile 02).
Erzeuge zwei Variablen namens A1 und A2, die (zum **Typ** !AEPFEL und) zum **Untertyp** AEPFEL gehören und initialisiere sie mit dem Wert 0 (Zeile 03).
Erzeuge zwei Variablen namens B1 und B2, die (zum **Typ** !BIRNEN und) zum **Untertyp** BIRNEN gehören und initialisiere sie mit dem Wert 0 (Zeile 04). **o**

Der entscheidende Punkt ist hier: **!AEPFEL** und **!BIRNEN** sind zwei **verschiedene** Typen! Praktisch bedeutet das: "Man darf Äpfel und Birnen nicht einfach so addieren" oder sonstwie miteinander verrechnen. Z.B. enthält jeder der folgenden vier Ausdrücke einen **Typenfehler**:
A1 + B1, **A2 - B1**, **A1 * B2**, **A2 / B2**.
Dagegen sind die folgenden vier Ausdrücke typenmäßig korrekt:
A1 + A2, **A2 - A1**, **B1 * B2**, **B2 / B1**.

Jede **Variable** gehört fest und unveränderlich zu einem bestimmten **Typ**. Dagegen sind **Ganzzahlliterale** wie z.B. **0**, **17** oder **123_456** etc. angenehm **anpassungsfähig**, was ihren Typ betrifft:: Ihre Werte werden automatisch dem Typ angepaßt, der in einem bestimmten Zusammenhang gebraucht wird. Z.B. wird der Wert des Ganzzahlliterals **5** in dem Ausdruck **A1 + 5** automatisch in einen Wert des Typs **!AEPFEL** umgewandelt, im Ausdruck **5 * B2** dagegen in einen **!BIRNEN**-Wert. Entsprechendes gilt für Zuweisungen wie z.B. **A1 := 5;** und **B1 := 5;** (der Variablen **A1** wird der Wert **5** des Typs **!AEPFEL** zugewiesen, der Variablen **B1** dagegen der Wert **5** des Typs **!BIRNEN**).
Praktisch bedeutet das: Man kann Ganzzahlliterale im Zusammenhang mit verschiedenen Ganzzahltypen "**einfach so**" verwenden und braucht dabei keine Angst vor Typenfehlern zu haben.

Im Kern beruht das Konzept eines **Typs** auf der Vorstellung, daß **der Wert 5 des Typs !AEPFEL** und **der Wert 5 des Typs !BIRNEN** zwei **verschiedene** Werte sind (die sich nur **entsprechen**, aber **nicht** gleich sind).

Wenn der Programmierer dem Ausführer aus Versehen doch einmal den Befehl gibt, "Äpfel und Birnen zu addieren", z.B. mit einem Ausdruck wie **A1 + B1** oder **B2 * A1** etc., dann ist das ein **formaler Fehler**, den der Ausführer schon bei der **Übergabe** des betreffenden Programmtextes ("zur Compilezeit") meldet, und nicht erst bei der **Ausführung** des Programms ("zur Laufzeit"). Mit solchen Typen kann man deshalb die Zuverlässigkeit und Sicherheit von Programmen deutlich erhöhen.

Was ist, wenn der Programmierer zwar erstmal mit Äpfeln und sauber davon getrenn mit Birnen rechnen will, aber dann (z.B. am Ende eines Programms) doch eine Summe von Äpfeln und Birnen ermitteln möchte? In diesem Fall kann er mit einer **Typkonversion** einen Wert des Typs !AEPFEL in den **entsprechenden** Wert des Typs !BIRNEN umwandeln und dann mit den beiden !AEPFEL-Werten rechnen (oder umgekehrt, den !BIRNEN-Wert in einen !AEPFEL-Wert umwandeln und dann mit den beiden !AEPFEL-Werten rechnen), wie z.B. in den folgenden Ausdrücken:

BIRNEN(A1) + B1	oder	**B2 * BIRNEN(A2)**	etc..
A1 - AEPFEL(B2)	oder	**AEPFEL(B1) / A2**	etc..

Falls es bei einer Typkonversion keinen **entsprechenden Wert des Zieltyps** gibt, löst der Ausführer die Ausnahme **constraint_error** aus, wie das folgende Beispiel zeigen soll:

Beispiel 8.1.2.: Typkonversionen: Die eine funktioniert, die andere nicht:

```
05 type ZITRONEN is range 0..40_000;
06 Z1 : ZITRONEN := 25_000;
07 Z2 : ZITRONEN := 35_000;
07 A3 : AEPFEL   := AEPFEL(Z1); -- Typkonversion ist o.k.
08 A4 : AEPFEL   := AEPFEL(Z2); -- Typkonversion loest constraint_error aus!
o
```

Man sieht: Indem der Programmierer verschiedene Ganzzahltypen vereinbart hat, verhindert er nur, daß er **aus Versehen** "Äpfel und Birnen addiert". Wenn er es ausdrücklich will, kann er mit Hilfe von Typkonversionen wie **BIRNEN(A1)** oder **AEPFEL(B2)** einen ganz entsprechenden Effekt erzielen.

Zusammenfassung 8.1.:

- Werte verschiedener Ganzzahl**typen** kann man **nicht** miteinander verrechnen.
- Mit einer **Typkonversion** kann man einen Wert eines Ganzzahltyps in den entsprechenden Wert eines anderen Ganzzahltyps umwandeln (falls es diesen entsprechenden Wert gibt).

8.2. Mehrere Ganzzahluntertypen in einem Programm

Nachdem man mit einer **Typvereinbarung** einen **Typ** (und einen ersten Untertyp) vereinbart hat, kann man mit **Untertypvereinbarungen** beliebig viele weitere **Untertypen** dieses Typs erzeugen lassen.

Beispiel 8.2.1.: Vier **Untertypen** (Fortsetzung von Beispiel 8.1.1.)

```
09 subtype AEPFEL_PRO_WOCHE is AEPFEL           range 0..70;
10 subtype AEPFEL_PRO_TAG   is AEPFEL_PRO_WOCHE range 0..10; -- entweder so
11 subtype BIRNEN_PRO_WOCHE is BIRNEN           range 0..70;
12 subtype BIRNEN_PRO_TAG   is BIRNEN           range 0..10; -- oder so!
13 AW1, AW2 : AEPFEL_PRO_WOCHE := 0;
14 AT1, AT2 : AEPFEL_PRO_TAG   := 0;
15 BW1, BW2 : BIRNEN_PRO_WOCHE := 0;
16 BT1, BT2 : BIRNEN_PRO_TAG   := 0;
```

Die vier Untertypvereinbarungen (in Zeile 09 bis 12) kann man etwa so ins Deutsche übersetzen:
Erzeuge einen **Untertyp** namens AEPFEL_PRO_WOCHE, der aus dem **Typ** !AEPFEL und der Einschränkung 0..70 besteht (Zeile 09).
Erzeuge einen **Untertyp** namens AEPFEL_PRO_TAG, der aus dem **Typ** !AEPFEL und der Einschränkung 0..10 besteht (Zeile 10).
Erzeuge einen **Untertyp** namens BIRNEN_PRO_WOCHE, der aus dem **Typ** !BIRNEN und der Einschränkung 0..70 besteht (Zeile 11).
Erzeuge einen **Untertyp** namens BIRNEN_PRO_TAG, der aus dem **Typ** !BIRNEN und der Einschränkung 0..10 besteht (Zeile 12). **o**

Man beachte, daß der Untertyp **AEPFEL_PRO_TAG** ein Untertyp des **Typs** !AEPFEL ist, obwohl in seiner Vereinbarung (in Zeile 10) hinter dem reservierten Wort **is** der **Untertyp AEPFEL_PRO_WOCHE** erwähnt wird.

Allgemein gilt: Es gibt in Ada nur **Untertypen von Typen**, keine **Untertypen von Untertypen**.

Trotzdem spielt es eine Rolle, ob man in einer Untertypvereinbarung z.B. **is AEPFEL_PRO_TAG** oder **is AEPFEL_PRO_WOCHE** angibt. Denn man darf den Bereich des hinter **is** angegebenen Untertyps zwar noch weiter **einschränken**, man darf ihn aber **nicht erweitern**:

Beispiel 8.2.2.: Erlaubte und verbotene **Untertypvereinvarungen**:

```
17 subtype AEPFEL_PRO_STUNDE_1    is AEPFEL_PRO_TAG;                  -- erlaubt
18 subtype AEPFEL_PRO_STUNDE_2    is AEPFEL_PRO_TAG     range 3..7;   -- erlaubt
19 subtype AEPFEL_PRO_STUNDE_3    is AEPFEL_PRO_TAG     range 0..20;  -- verboten!
20 subtype AEPFEL_PRO_STUNDE_4    is AEPFEL_PRO_WOCHE   range 0..20;  -- erlaubt
```

Der Untertyp **AEPFEL_PRO_STUNDE_1** hat denselben Bereich wie der Untertyp **AEPFEL_-PRO_TAG** (nämlich den Bereich 0..10), der Untertyp **AEPFEL_PRO_STUNDE_2** hat den kleineren Bereich 3..7. **o**

Besonders wichtig ist: Werte, die zum selben **Typ** gehören, darf man beliebig miteinander "verrechnen", unabhängig davon, zu welchem **Untertyp** sie gehören:

Beispiel 8.2.3.: Die folgenden drei Ausdrücke sind **typenmäßig korrekt**:

A1 + AW1 + AT1, **B2 * BT1 - BW2**, **BT1 + 1_000 + BT2**

Die Rechnungen finden sozusagen "innerhalb des **Typs**" statt. Erst wenn ein Wert einer **Variablen zugewiesen** wird, prüft der Ausführer, ob der Wert zum **Untertyp** der Variablen gehört und löst gegebenenfalls die Ausnahme **constraint_error** ("eine Einschränkung wurde verletzt") aus:

Beispiel 8.2.4.: Angenommen, die Variablen A1, A2, AW1, AW2, AT1 und AT2 haben irgendwelche Werte aus dem Bereich ihres jeweiligen Untertyps (**A1** und **A2** aus dem Bereich **0..30_000**, **AW1** und **AW2** aus dem Bereich **0..70**, **AT1** und **AT2** aus dem Bereich **0..10**). Dann gilt:

```
25 -- Folgende Zuweisungen loesen garantiert keine Ausnahme aus:
26 A1  := AW1;
27 A1  := 10_000 + AW1 + AW2 + AT1 + AT2;
28 AW1 := 2 * AT1 + 3 * AT1;
29 AT1 := A1 - A1 + AT2;
30 -- Folgende Zuweisungen loesen evtl. einen constraint_error aus:
31 AT1 := A1 + AW2 * AT1 - A2;
32 AT1 := A2;
33 A1  := A1 + 1;
34 AT1 := A2 + 30_000 - 30_000;
35 -- Folgende Zuweisungen loesen garantiert einen constraint_error aus:
36 AT1 := 11;
37 A1  := (A1 + 2) * 20_000;
```
o

Das folgende Beispiel soll noch einmal verdeutlichen: Jede **Ganzzahlvariable** gehört zu einem bestimmten **Typ** und zu einem bestimmten **Untertyp**. Die **Rechenoperationen** wie +, -, *, / etc. gehören nur zu einem **Typ**, aber nicht zu irgendeinem Untertyp dieses Typs.

Beispiel 8.2.5.: Ausführung einer Zuweisungs-Anweisung:

```
40 AW1 := AT1 * 1_000 - A1 + 57;
```

Diese Zuweisung kann man etwa so ins Deutsche übersetzen:
Berechne (zuerst) den Wert des Ausdrucks **AT1 * 1_000 - A1 + 57** und weise diesen Wert (dann) der Variablen **AW1** zu.

Die Ausnahme **constraint_error** wird ausgelöst, wenn
1. bei der Auswertung des Ausdrucks ein Zwischenergebnis nicht mehr zum **Typ** !AEPFEL gehört.
oder wenn
2. der fertig berechnete Wert des Ausdrucks nicht zum **Untertyp** der Variablen AW1 gehört. **o**

Zusammenfassung 8.2.:
- Wenn der Wert eines Ganzzahlausdrucks berechnet wird, spielen die **Untertypen** der beteiligten Werte keine Rolle (nur der **Typ** der Werte ist wesentlich).
- Wenn einer Variablen ein Wert **zugewiesen** wird, spielt der **Untertyp** der Variablen eine wichtige Rolle.

8.3. Mehrere Untertypen eines Typs nützlich anwenden

Das Programm **GANZT_01** liest zwei Ganzzahlten ein und gibt ihre Summe aus. Bei der **Benutzung** des Programms können an zwei Stellen Fehler auftreten: 1. Beim **Einlesen** der Zahlen (weil der Benutzer ungeeignete Daten eingibt) und 2. beim **Berechnen** der Summe (wenn die Summe nicht mehr zum Untertyp **OTTO** gehört).

Für den Benutzer günstiger wäre es, wenn Fehler nur beim **Einlesen** auftreten könnten, aber nicht mehr im Verlaufe der **Berechnungen**, die das Programm durchführt. Das gilt für allem bei Pro-

grammen, die mehrere komplizierte Berechnungen durchführen, ansonsten aber eine ähnliche Struktur haben wie **GANZT_01** (Eingabe, Verarbeitung, Ausgabe).

Indem wir **zwei Untertypen** desselben Ganzzahltyps vereinbaren, können wir solche Programme benutzerfreundlicher gestalten. Der eine Untertyp beschreibt alle Zahlen, die der Benutzer **eingeben** darf und der andere alle Zahlen, die während der **Berechnungen** als Zwischenergebnisse auftreten können bzw. ausgegeben werden sollen. Welchen Bereich dieser zweite Untertyp umfassen muß, hängt natürlich von den Berechnungen ab, die man durchführen will. Den umfangreicheren der beiden Untertypen vereinbaren wir in einer **Typvereinbarung** und den anderen mit einer **Untertypvereinbarung**.

Beispiel 8.3.1.: Das Programm **GANZT_04** soll zwei Zahlen aus dem Bereich -1_000..+3_000 einlesen und ihre Summe ausgeben. Alle Fehler sollen schon beim Einlesen der Zahlen erkannt werden, d.h. beim Berechnen der Summe soll keine Ausnahme mehr auftreten können. Die Vereinbarungen dafür könnten z.B. so aussehen:

```
03    type    GANZ         is      range -2_000..+6_000;
04    subtype GANZ_EINGABE is GANZ range -1_000..+3_000;
05    package GANZ_EA      is new ada.text_io.integer_io(num => GANZ);
06    EIN1, EIN2 : GANZ_EINGABE;
```

Wenn das **Einlesen** von zwei Zahlen nach **EIN1** und **EIN2** keine Ausnahme auslöst, dann funktioniert auch das **Ausgeben** ihrer Summe **EIN1 + EIN2** ohne Ausnahme. o

Von den **Anweisungen** der Prozedur **GANZT_04** sind vor allem die beiden **Einlesebefehle** interessant. Der erste davon sieht so aus:

```
09    GANZ_EA.get(item => EIN1);
```

Wenn dieser Befehl ausgeführt wird und der Benutzer z.B. die Zahl **+3_000** eingibt, geht alles gut. Wenn der Benutzer stattdessen z.B. die Zahl **+7_000** eingibt, wird die Ausnahme **data_error** ausgelöst. Aber was passiert, wenn der Benutzer z.B. die Zahl **+5_000** eingibt?

Zu beachten ist, daß die Befehle im Paket **GANZ_EA** "auf den größeren Untertyp **GANZ** abgestimmt" sind (siehe Zeile 05: **num => GANZ**), und nicht nur auf den kleineren Untertyp **GANZ_EINGABE**. Die Prozedur **GANZ_EA.get** kann den Wert **+5_000** also ohne weiteres einlesen und löst dabei keinen **data_error** aus. Erst der Versuch (noch innerhalb der **get**-Prozedur) diesen Wert der Variablen EIN1 **zuzuweisen**, wird die Ausnahme **constraint_error** ausgelöst.

Im Programm **GANZT_04** werden also alle Fehler schon beim Einlesen der Eingaben erkannt und gemeldet, einige als **data_error** und andere als **constraint_error**. Wie der Programmierer Ausnahmen behandeln kann (so daß sie z.B. zu einheitlicheren und benutzerfreundlicheren Fehlermeldungen führen) wird im Abschnitt 13. gezeigt.

Aufgabe 8.3.1.: Schreiben Sie ein Programm namens **GANZT_05**, welches eine Ganzzahl aus dem Bereich **-1_000..+1_000** einliest und das Quadrat dieser Zahl ausgibt. Alle Benutzerfehler sollen schon beim Einlesen der Zahlen erkannt und gemeldet werden.

Aufgabe 8.3.2.: Schreiben Sie ein Programm namens **GANZT_06**, welches eine Ganzzahl aus dem Bereich **-1_000..+1_000** einliest, durch zehn dividiert und das Ergebnis ausgibt. Alle Benutzerfehler sollen schon beim Einlesen der Zahlen erkannt und gemeldet werden.

Zusammenfassung 8.3.: Mit Untertypen kann man unter anderem **Eingabedaten** auf elegante Weise prüfen und Programme benutzerfreundlich gestalten.

9. Vertiefendes zu Ganzzahltypen und Untertypen

In diesem Abschnitt werden am Beispiel von **Ganzzahluntertypen** ein paar Einzelheiten behandelt, die zum Teil auch bei anderen Typen (die später eingeführt werden) wichtig sind.

9.1. Konstanten

Eine Konstante besteht aus einem **Namen** und einem **Wert** und gehört (ähnlich wie eine Variable) zu einem bestimmten **Typ**. Der Programmierer muß eine Konstante **vereinbaren** ("vom Ausführer erzeugen lassen"), ehe er sie als Ausdruck (oder als Teil eines Ausdrucks) verwendet.

Beispiel 9.1.1.: Vier **Konstanten** (und eine Variable zum Vergleich):

```
01 type OTTO is range -5_000_000 .. + 1_000_000_000;
02 MEHR_WERT_STEUER_SATZ : constant OTTO := 17;
03 ANZAHL_KOENIGE        : constant OTTO :=  3;
04 ANZAHL_SAMURAI        : constant OTTO :=  7;
05 TAGE_PRO_WOCHE        : constant OTTO :=  7;
06 TAGE_PRO_MONAT        :          OTTO := 30; -- initialisierte Variable
```

Die Konstantenvereinbarung in Zeile 02 kann man etwa so ins Deutsche übersetzen: Erzeuge eine Konstante namens **MEHR_WERT_STEUER_SATZ** vom Untertyp **OTTO** mit dem Wert **17**.
o

Eine **Konstantenvereinbarung** unterscheidet sich nur durch das reservierte Wort **constant** von der **Vereinbarung einer Variablen mit Anfangswert** (siehe oben Zeile 06).

Eine Konstante **leistet** deutlich **mehr**, als eine Variable, die der Programmierer "bestimmt nie verändert". Daß eine **Konstante** (während einer Programmausführung) nicht verändert wird, garantiert der **Ausführer**. Daß eine **Variable** nicht verändert wird, ist höchstens ein Versprechen des **Programmierers**. Bei der **Fehlersuche** in einem Programm kann man sich grundsätzlich nicht auf Versprechungen des Programmierers verlassen. Deshalb ist insbesondere der Kollege-2 (der die Programme des Programmierers unter anderem von kleinen Fehlern befreien soll) sehr dankbar, wenn in einem Programm **Konstanten** verwendet werden, statt "nie veränderte" **Variablen**.

Ein Programm wird **änderungsfreundlich**, wenn man Literale wie **17**, **3**, **7** etc. nur in **Vereinbarungen** von **Konstanten** und **Typen** verwendet, überall sonst aber die **Namen** der Konstanten bzw. **Attribute** (siehe nächsten Abschnitt 9.2.) benutzt. Wenn dann z.B. der Mehrwertsteuersatz von 17 auf 12 % gesenkt wird, braucht der Kollege-2 nur **eine** leicht erkennbare Stelle im Programm zu verändern (die Vereinbarung der Konstanten MEHR_WERT_STEUER_SATZ), statt **mehrere** schwer zu findende Vorkommen des Literals **17** (von denen einige möglicherweise nichts mit der Mehrwertsteuer zu tun haben). Außerdem sieht man z.B. dem Literal **7** nicht ohne weiteres an, ob es gerade eine Anzahl von Tagen oder von Schwertkämpfern bezeichnet. Die Konstantennamen "TAGE_PRO_WOCHE" und "ANZAHL_SAMURAI" sind dagegen deutlich informativer.

9.2. Attribute

In Ada gibt es zu verschiedenen Größen (zu Untertypen, zu Variablen und zu Unterprogrammen) bestimmte **Attribute**. Hier ein paar Beispiele:

Beispiel 9.2.1.: Attribute eines **Ganzzahltuntertyps**:

```
01 type OTTO is range -1_000 .. +2_000;
02 V1 : OTTO := OTTO'first;                        -- Anfangswert -1_000
03 V2 : OTTO := OTTO'last / 2 - 1;                 -- Anfangswert   +999
04 V3 : OTTO := OTTO'min(V1, V2);                  -- Anfangswert -1_000
05 V4 : OTTO := OTTO'max(2 * V2, V1 + 2_800);      -- Anfangswert +1_898
```

Das Attribut **OTTO'first** ist eine **Konstante** und bezeichnet den **ersten** (d.h. kleinsten) Wert des Untertyps OTTO. Entsprechend bezeichnet **OTTO'last** den **letzten** (d.h. größten) Wert des Untertyps OTTO. Das Attribut **OTTO'min** ist eine **Funktion** mit zwei Parametern vom Typ !OTTO. Sie liefert den Wert des kleineren Parameters ("das Minimum der beiden Parameterwerte") als Ergebnis. Entsprechend liefert die Funktion **OTTO'max** den größeren der beiden Parameter ("das Maximum der beiden Parameterwerte"), auf die man sie anwendet. Natürlich darf man die beiden Funktionen **OTTO'min** und **OTTO'max** nicht nur beim Initialisieren von Variablen benützen (wie in obigem Beispiel), sondern "überall" in einem Programm (aber erst **nach** der Vereinbarung des Untertyps OTTO). **o**

Attributnamen erkennt man immer an dem **Apostroph** in der Mitte. Rechts davon steht der Attributbezeichner (attribute designator) und links der Name der Größe, zu der das Attribut gehört (z.B. der Untertyp OTTO). Im Anhang K des ARM werden **alle Attribute** beschrieben.

Das Attribut **OTTO'base** bezeichnet den **Basisuntertyp** (base subtype) des Typs !OTTO. Dieser Untertyp besteht aus dem Typ !OTTO und der **leeren Einschränkung**, die "keinerlei Einschränkung bewirkt". Das bedeutet praktisch: Alle Werte des **Typs** !OTTO (nicht nur die Werte des Untertyps OTTO!) gehören auch zu seinem **Basisuntertyp**. Zur Erinnerung: Welche Werte zum **Typ** !OTTO gehören, wird vom jeweiligen Ausführer festgelegt und kann von Ausführer zu Ausführer verschieden sein.

Da OTTO'base ein **Untertyp** ist, hat er auch alle **Attribute** eines Untertyps. Z.B. bezeichnet das Attribut **OTTO'base'first** den kleinsten Wert des Untertyps **OTTO'base**. Gleichzeitig ist das auch der kleinste Wert des **Typs** !OTTO. Entsprechendes gilt für das Attribut **OTTO'base'last**.

Man beachte, daß man die Attributfunktion **OTTO'min** auf beliebige Werte des **Typs** !OTTO, und nicht nur auf Werte des **Untertyps** OTTO anwenden kann. Genauso wie die Rechenoperationen +, -, *, / etc. gehören auch **Attributfunktionen** immer zu einem **Typ**, und nicht nur zu einem bestimmten **Untertyp**. Daraus folgt: Das Attribut **OTTO'base'min** des Untertyps **OTTO'base** ist genau dieselbe Funktion wie das Attribut **OTTO'min** des Untertyps **OTTO**.

Aufgabe 9.2.1.: Welche Werte gehören zum Untertyp **OTTO'base'base**? Und welche Werte gehören zum Untertyp **OTTO'base'base'base'base**?

Aufgabe 9.2.1.: Schreiben Sie ein Programm namens **GANZT_05**, in dem ein Ganzzahltyp und sein erster Untertyp namens **OTTO** vereinbart werden (wie z.B. in GANZT_01). Der größte und kleinste Wert des **Typs** !OTTO (nicht des **Untertyps** OTTO) soll zur aktuellen Ausgabe ausgegeben werden. Welchen Untertyp muß man dazu beim Instanziieren der Paketschablone **integer_io**:

```
package OTTO_EA is new ada.text_io.integer_io(num => ???);
```

wohl angeben?

9.3. Ausdrücke und Werte

Die Begriffe **Ausdruck**, **Literal**, **Konstante**, **Variable** und **Wert** gehören eng zusammen. Um Verwechslungen zu erschweren, werden sie hier gemeinsam erläutert und gegeneinander abgegrenzt. Einige wenige Details sind Ada-spezifisch. Die meisten Erläuterungen gelten auch für andere Sprachen.

Ein **Ausdruck** ist ein **Befehl** des Programmierers an den Ausführer, einen bestimmten **Wert** zu berechnen. Ein Ausdruck steht in einem Programm. Ein Ausdruck kann **nicht** in einem Wertebehälter (in einer Variablen) stehen.

Ein **einfacher Ausdruck** besteht nur aus einem **Literal** (z.B. 123, -1_000, "Hallo!", 'X' etc.), aus dem Namen einer **Konstanten** (z.B. MEHR_WERT_STEUER_SATZ, OTTO'first etc.) oder aus einem Namen einer **Variablen** (z.B. EIN1, V2 etc.).

Ein **zusammengesetzter Ausdruck** besteht aus (Teil-) Ausdrücken, die mit **Operatoren** wie +, -, *, /, & etc., **Klammern** und ähnlichen Mitteln "nach den üblichen Regeln" zusammengefügt wurden, z.B. so: **123 + EIN1**, **"Hallo!" & " Wie geht es?"**, **(V1 + V2) * V3** etc..

Ein **Wert** ist etwas, das bei der Auswertung eines Ausdrucks **als Ergebnis herauskommen** und einer Variablen **zugewiesen** werden kann. Zwar gibt man in einer Zuweisungsanweisung (rechts vom Zuweisungszeichen ":=") einen **Ausdruck** an, aber wirklich zugewiesen wird der **Wert** des Ausdrucks, nicht der Ausdruck selbst. **Verschiedene** Ausdrücke können **denselben** Wert haben und dem Wert sieht man nicht an, welcher Ausdruck seine Berechnung veranlaßt hat. Ein Wert ist etwas **abstrakteres** als ein Ausdruck.

Besonders naheliegend ist es, **Werte** und **Literale** zu verwechseln. Die folgenden beiden Sätze sollen zeigen, daß es (zumindest manchmal) sinnvoll ist, zwischen einem **Literal** und seinem **Wert** zu unterscheiden:

1. Die drei **Literale** 17, +1_7 und 16#11# unterscheiden sich offenbar voneinander, haben aber alle denselbe **Wert** siebzehn.
2. Das **Literal** 17 besteht aus zwei Dezimalziffern. Sein **Wert** wird von heute üblichen Computern durch eine bestimmte Bitkette dargestellt.

Ein **Literal** ist also etwas Ähnliches wie ein Name, der einen Wert bezeichnet. Eine **Konstante** ist ein Name, der einen Wert bezeichnet. Der Zusammenhang zwischen einem **Literal** und seinem **Wert** wird durch eine **Sprache** (z.B. durch Ada) festgelegt, der Zusammenhang zwischen dem **Namen** einer Konstanten und ihrem **Wert** durch den **Programmierer**. Wenn das **Literal** 17 nicht mehr den Wert siebzehn haben sollte, dann ist das ein Fehler des Ausführers ("ein Compilerfehler"). Wenn die **Konstante** MEHR_WERT_STEUER_SATZ nicht den Wert siebzehn haben sollte, dann hat der **Programmierer** sie mit einem anderen Wert vereinbart.

Ausdrücke, insbesondere **Literale**, Namen von **Konstanten** und Namen von **Variablen** sind **syntaktische Gebilde**, die in Programmen vorkommen und vom **Programmierer** direkt manipuliert werden können. Dagegen ist ein **Wert** ein **semantisches Gebilde**, das nur vom **Ausführer** mani-

puliert (berechnet, in einer Variablen gespeichert etc.) wird. Der Programmierer bekommt Werte eigentlich nie direkt zu Gesicht.

Zusammenfassung 9.3.:

- **Ausdrücke**, **Literale** und **Namen** von Konstanten und Variablen sind **syntaktische Gebilde**. Sie kommen in Programmen vor.
- **Werte** sind **semantische Gebilde** und werden erst während der Ausführung eines Programms vom Ausführer berechnet.

9.4. Der vordefinierte Untertyp integer

In Ada gibt es genau einen **vordefinierten** signierten Ganzzahltyp. Dieser Typ hat drei Untertypen namens **integer**, **natural** und **positive**. Davon ist **integer** der erste Untertyp. Deshalb bezeichnen wir den vordefinierten Ganzzahlttyp in diesem Skript mit dem Hilfsnamen **!integer**. Der Untertyp **integer** besteht aus dem Typ **!integer** und einer Einschränkung, die keinen Wert des Typs ausschließt. Zum **Untertyp** integer gehören also alle Werte des **Typs** !integer.

Der Ada-Standard legt **nicht** fest, welche Werte zum Typ !**integer** gehören, sondern überläßt es ausdrücklich jedem einzelnen **Ada-Ausführer**, diesem Typ ein Hardwareformat zuzuordnen, mit dem er (der Ausführer) "besonders gut umgehen" kann (siehe Abschnitt 7.1.). Der Standard verlangt nur, daß zum Typ !**integer** mindestens die Werte im Bereich **-32_767..+32_767** gehören. Das sind gerade die Ganzzahlwerte, die jeder heute übliche Computer mit 16 Bit darstellen kann (einige Computer können mit 16 Bit sogar noch eine weitere Zahl, **-32_768**, darstellen). Viele Ada-Ausführer gehen aber weit über diese Mindestforderung hinaus und ordnen dem Typ !**integer** den 32-Bit-Wertebereich **-2_147_483_648..+2_147_483_647** (d.h. **-2**31..+2*31-1**) zu.

Der Untertyp **natural** besteht aus dem Typ **!integer** und der Einschränkung **0..integer'last**, er umfaßt also alle nicht-negativen Werte des Typs !**integer**. Der Untertyp **positive** besteht aus dem Typ !**integer** und der Einschränkung **1..integer'last**. Der Typ **natural** umfaßt also genau einen Wert mehr als der Untertyp **positive**, nämlich den Wert 0.

Wenn ein Programmierer die Untertypen **integer**, **natural** oder **positive** in einem Programm benutzt (indem er z.B. Variablen dieser Untertypen vereinbart), dann weiß ein Leser des Programms **nicht** genau, welche Wertebereiche der Programmierer damit meint. Deshalb sollte man die Untertypen **integer**, **natural** und **positive** möglichst **selten benutzen**.

Es gibt Zusammenhänge, in denen ein Ada-Programmierer die drei Untertypen des vordefinierten Typs !**integer** benutzen **muß** (z.B. wenn er intensiv mit Zeichenketten vom vordefinierten Typ **string** arbeitet). Von diesen Ausnahmen abgesehen sollte der Programmierer möglichst immer "seine eigenen Ganzzahltypen" vereinbaren und damit arbeiten.

Um Werte des Untertyps **integer** einzulesen oder auszugeben kann man entweder das Standard-Paket **ada.integer_text_io** benützen oder, ganz entsprechend wie bei selbst vereinbarten Ganzzahltypen, die Paketschablone **integer_io** mit dem Untertyp integer als **num**-Parameter instanziieren, etwa so:

```
package INTEGER_EA is new ada.text_io.integer_io(num => integer);
```

Im Beispielprogramm **GANZT_09** wird das Standardpaket **ada.integer_text_io** benützt, in **GANZT_10** wird die Schablone **integer_io** mit **num => integer** instanziiert.

Zum Abschluß noch ein weiterer Grund, warum man selbst Ganzzahltypen vereinbaren soll, statt **integer** zu benützen: Viele heute übliche Computer können mit drei (oder sogar mehr) Hardware-formaten besonders gut umgehen, z.B. mit B08, B16 und B32 oder mit B12, B24 und B48. Auf solchen Computern wird dem Typ **!integer** häufig nur das mittlere Hardwareformat zugeordnet (also z.B. B16 oder B24), und nicht etwa das größte (B32 bzw. B48). Daraus folgt dann: Zum Typ **!integer** gehören gar nicht die größten Ganzzahlen, mit denen der betreffende Computer "besonders gut umgehen kann". Solche Ada-Ausführer kann man nur "voll ausreizen", indem man eigene Ganzzahltypen vereinbart.

9.5. Modulare Ganzzahltypen

Außer den bisher behandelten **signierten** ("vorzeichenbehafteten") **Ganzzahltypen** gibt es in Ada auch noch **modulare Ganzzahltypen**. Die modularen Typen braucht man vor allem dann, wenn man **maschinennah** programmieren ("mit den Bits eines bestimmten Rechners herumfummeln") muß. Die Werte eines modularen Typs haben **kein Vorzeichen** (d.h. sie gelten üblicherweise alle als positiv), und wenn beim Rechnen ein Zwischenergebnis **außerhalb** des Wertebereichs des Typs liegt, dann wird (so kann man sich zumindest vorstellen) solange eine bestimmte Zahl (der **Modulus** des Typs) subtrahiert oder addiert, bis das Ergebnis wieder **innerhalb** des Wertebereichs liegt.

Beispiel 9.4.1.: Vereinbarung eines **modularen Typs** und seines ersten Untertyps:

```
01 type MONIKA is mod 256; -- 256 ist der Modulus des Typs !MONIKA
02 CM1 : constant MONIKA := MONIKA'first +  1; -- Anfangswert 1
03 VM1 :          MONIKA := MONIKA'last  - 10: -- Anfangswert 245
04 VM2 :          MONIKA;
```

Die Typvereinbarung in Zeile 01 kann man etwa so ins Deutsche übersetzen: Erzeuge einen **modularen Ganzzahltyp** !MONIKA mit dem Modulus 256, und einen Untertyp namens MONIKA, der aus dem Typ !MONIKA und der Einschränkung 0..255 besteht.

CM1 ist eine **Konstante** vom Untertyp MONIKA. VM1 ist eine **initialisierte** und VM2 eine **nicht initialisierte Variable** vom Untertyp MONIKA. **o**

Mit dem Typ !MONIKA erzeugt der Ausführer auch die üblichen Rechenoperationen +, -, *, / etc., aber diese Operationen rechnen **modular**. Was das bedeutet, soll das nächste Beispiel deutlichen machen:

Beispiel 9.4.2.: Modulares Rechnen mit Werten des Typs !MONIKA:

```
250 + 5          ist gleich   255     -- wenig überraschend
250 + 6          ist gleich     0     --  256  -    1 * 256
250 + 10         ist gleich     4     --  260  -    1 * 256
200 + 200 +200   ist gleich    88     --  600  -    2 * 256
5 - 6            ist gleich   255     --   -1  +    1 * 256
5 - 10           ist gleich   251     --   -5  +    1 * 256
20 * 30          ist gleich    88     --  600  -    2 * 256
```

```
10 * 255        ist gleich   246      -- 2550  -   16 * 256
- 16            ist gleich   240      --  256  -   16
- 1             ist gleich   255      --  256  -    1
- 0             ist gleich     0      --  256  - 256
o
```

Außerdem erzeugt der Ausführer mit dem Typ !MONIKA auch logische Operationen namens **and**, **or**, **xor** und **not**, und mit diesen Operationen darf man Werte des Typs !MONIKA "bitweise" miteinander verknüpfen. Nähere Einzelheiten findet man z.B. in (ARM 3.5.4) und (ARM 4.5) sowie im Beispielprogramm GANZT_07.

9.6. Die Operationen "**", rem und mod für Ganzzahltypen

Die Beispiele in diesem Abschnitt beruhen auf folgenden Vereinbarungen:

```
01 type OTTO   is range -20_000..+30_000;       -- Ein signierter Ganzzahltyp
02 type MONIKA is mod 256;                      -- Ein modularer  Ganzzahltyp
03 G1, G2, G3 : OTTO;
04 M1, M2, M3 : MONIKA;
05 N1, N2, N3 : natural;
```

Für jeden (signierten oder modularen) **Ganzzahltyp** gibt es auch eine **Potenzierungsoperation "**"** (gesprochen "Doppelstern" oder "Stern Stern" oder einfach "hoch") und zwei **Restoperationen** namens **rem** und **mod**.

Die Potenzierungsoperation **"**"** für den Typ **!OTTO** hat nicht (wie man leicht vermuten könnte) zwei Parameter vom Typ **!OTTO** (z.B. so: **G1 ** G2**), sondern nur einen linken Parameter vom Typ **!OTTO** und einen rechten Parameter vom Untertyp **natural** (d.h. vom Typ !integer). Die folgenden Ausdrücke sind typenmäßig korrekt:

G1 ** 0, **G1 ** 1**, **G1 ** 2**, **G1 ** 3** etc., **G1 ** N1**, **(G1 + G2) ** N3**,
G1 ** (2 * N1 - 3) etc..

Dagegen enthalten die folgenden Ausdrücke Typenfehler und werden vom Ausführer abgelehnt:

G1 ** G2, **G1 ** (G2 + 3)**, **G1 ** (G2 + N1)**

Ganz Entsprechendes gilt für die Potenzierungsoperation "**" des modularen Typs MONIKA. Die folgenden Ausdrücke sind typenmäßig korrekt:

M1 ** 0, **M1 ** 1**, **M1 ** 2**, **M1 ** 3** etc., **M1 ** N1**, **(M1 + M2) ** N3**,
M1 ** (2 * N1 - 3) etc..

Dagegen enthalten die folgenden Ausdrücke Typenfehler und werden vom Ausführer abgelehnt:

M1 ** M2, **M1 ** (M2 + 3)**, **M1 ** (M2 + N1)**

Die Potenzierungsoperation **"**"** bewirkt (so kann man sich zumindestens vorstellen) folgende Rechnung: Eine Variable mit dem Anfangswert 1 wird so oft mit dem ersten Parameter **multipliziert**, wie der zweite Parameter (der immer zum Untertyp **natural** gehören muß) angibt. **o**

Für die **Ganzzahldivision "/"** gilt in Ada z.B.: **13 / 5** ist gleich **2**. Außerdem bleibt ein Rest von **3** übrig. Die Operation **rem** (wie "remainder") liefert diesen Rest. Z.B. ist **13 rem 5** gleich **3**. Hier ein paar weitere Beispiele dafür, "was die Operationen **"/"** und **rem** machen":

Beispiel 9.6.1.: Ein paar Ergebnisse der Operationen **"/"** und **rem**:

Wert der Variablen **G1**	Wert der Variablen **G2**	Wert des Ausdrucks **G1 / G2**	Wert des Ausdrucks **G1 rem G2**
9	5	1	4
10	5	2	0
11	5	2	1
12	5	2	2
13	5	2	3
14	5	2	4
15	5	3	0

o

Die Restoperation **mod** liefert dieselben Ergebnisse wie **rem**, wenn beide Parameter (z.B. G1 und G2) das **gleiche Vorzeichen** haben (d.h. wenn beide positiv oder beide negativ sind). Haben die beiden Parameter **unterschiedliche Vorzeichen**, liefern **rem** und **mod** verschiedene Ergebnisse (z.B. ist **-13 rem 5** gleich **-3**, aber **-13 mod 5** ist gleich **2**). Der Grund dafür ist, daß die Restoperation **rem** zu der (in Ada vordefinierten) Ganzzahldivision **"/"** gehört, die Restoperation **mod** dagegen zu einer etwas anderen Ganzzahldivision, die in Ada **nicht** vordefinierten ist, die man aber relativ leicht programmieren kann. Diese "etwas andere" Ganzzahldivision wird im folgenden mit **"&"** bezeichnet. Die **"/"**-Division "**rundet immer in Richtung 0**" (d.h. bei positiven Ergebnissen wird **abgerundet**, aber bei negativen Ergebnissen wird **aufgerundet**). Die **"&"**-Division dagegen "**rundet immer nach rechts**", z.B. so:

-11 / 5 ist gleich **11 / -5** ist gleich **-2**
-11 & 5 ist gleich **11 & -5** ist gleich **-3**

Entsprechend sind auch die Restoperationen (**rem** für **"/"** und **mod** für **"&"**) verschieden. Weitere Einzelheiten findet man z.B. im (ARM 4.5.5(22)). Wenn man die Operationen **rem** und **mod** genauer kennen lernen möchte, empfiehlt es sich, die Aufgabe 9.6.2. (siehe unten) zu lösen und/oder mit dem Programm **GANZT_11** "zu experimentieren".

Aufgabe 9.6.1.: Angenommen, die Variable **G1** hat den Wert **17**. Welchen Wert haben dann die Ausdrücke **0 rem G1** und **0 mod G1**? Und was passiert, wenn man dem Ausführer befiehlt, den Wert des Ausdrucks **G1 rem 0** oder den Wert des Ausdrucks **G1 mod 0** zu berechnen? **o**

Aufgabe 9.6.2.: Schreiben Sie ein Programm namens **GANZT_11**, welches zwei Ganzzahlen **G1** und **G2** einliest und die Werte der Ausdrücke **G1 / G2**, **G1 rem G2** und **G1 mod G2** ausgibt. **o**

9.7. Formatkontrolle beim Ein- und Ausgeben von Ganzzahlen

Dieser Abschnitt ist für ein tieferes Verständnis von Ada **völlig unwichtig** und kann getrost übersprungen werden. Die Beispiele in diesem Abschnitt gehen von folgenden Vereinbarungen aus:

```
01 type    OTTO    is range -20_000..+30_000;
02 package OTTO_EA is new ada.text_io.integer_io(num => OTTO);
03 G1, G2, G3, G4 : OTTO;
```

Außerdem wird angemommen, daß den Variablen **G1** bis **G4** irgendwelche Werte (ihres Untertyps OTTO) zugewiesen wurden. Die in diesem Abschnitt behandelten Möglichkeiten der Formatkontrolle werden auch im Beispielprogramm **GANZT_12** verwendet und vorgeführt.

Wenn man mit der Prozedur **OTTO_EA.put** einen Wert des Untertyps OTTO ausgibt, dann wird dieser Wert zuerst in eine entsprechende Zeichenkette der Länge **sechs** umgewandelt und diese Zeichenkette wird ausgegeben. Die Länge **sechs** kommt so zustande: Die "längsten Zahlen" des Untertyps OTTO bestehen in dezimaler Darstellung (ohne verzierende Unterstriche) aus **fünf** Ziffern und **einem** Vorzeichen (z.B. "-10000" oder "-20000" etc.). Bei positiven Zahlen wird als **Vorzeichen** ein **Blank** ausgegeben.

Wenn der Programmierer möchte, daß die ausgegebene Zeichenkette **länger** oder **kürzer** als sechs Zeichen sein soll, dann kann er im Aufruf der **put**-Prozedur einen **width**-Parameter angeben, z.B. so:

Beispiel 9.7.1.: Ein **put**-Befehl mit **width**-Parameter:

```
10 OTTO_EA.put(item => G1, width => 10);
11 OTTO_EA.put(item => G2, width =>  3);
12 OTTO_EA.put(item => G3, width =>  0);
```

Der Wert von G1 wird als **Zeichenkette der Länge 10** ausgegeben (links mit Blanks aufgefüllt). Der Wert von G2 wird in eine Zeichenkette umgewandelt, die **mindestens** 3 Zeichen lang ist. Der Wert von G2 wird aber auf jeden Fall "in voller Länge" ausgegeben, auch wenn dazu mehr als 3 Zeichen notwendig sind. Der Wert von G3 wird als Zeichenkette mit einer "Mindestlänge von 0 Zeichen" ausgegeben. Praktisch bedeutet das: Der Wert von G3 wird "**so kurz wie möglich**" (ohne irgendwelche Blanks) ausgegeben. **o**

Wenn man viele OTTO-Werte "in gleicher Länge" ausgeben will, braucht man diese Länge nicht in jedem **put**-Befehl erneut anzugeben, sondern kann sie durch eine Zuweisung an die Variable namens **default_length** im Paket **OTTO_EA** "bis auf Widerruf" festlegen, etwa so:

Beispiel 9.7.2.: Die Ausgabelänge für OTTO-Werte bis auf Widerruf festlegen:

```
20 OTTO_EA.default_width := 10;              -- Ausgabebreite auf 10 setzen
21 OTTO_EA.put(item => G1);                  -- Ausgabebreite 10
22 OTTO_EA.put(item => G2);                  -- Ausgabebreite 10
23 OTTO_EA.default_width := OTTO'width;      -- Ausgabelaenge auf 6 zuruecksetzen
23 OTTO_EA.put(item => G3);                  -- Ausgabebreite 6
24 OTTO_EA.put(item => G4, width => 8)       -- Ausgabebreite 8
```

G1 und **G2** werden jetzt in einer Länge von je **zehn** Zeichen ausgegeben. Das Attribut **OTTO'width** hat den Wert **sechs**, weil "die längsten OTTO-Zahlen" in dezimaler Darstellung aus **sechs** Zeichen bestehen. Der Wert von **G3** wird in einer Breite von sechs, **G4** in einer Breite von 8 Zeichen ausgegeben (ein ausdrücklicher **width**-Parameter "wirkt stärker", als die Variable **default_width**). **o**

Ganz ähnlich wie die Länge kann man auch das **Zahlensystem** beeinflussen, in dem die auszugebende OTTO-Zahl dargestellt werden soll. Wenn man keine besonderen Angaben macht, wird eine OTTO-Zahl vom Befehl **OTTO_EA.put** als **Dezimalzahl** ausgegeben. Man kann sie aber auch als Zahl eines anderen Zahlensystems mit einer Basis zwischen 2 und 16 ausgeben lassen, etwa so:

Beispiel 9.7.3.: OTTO-Zahlen in verschiedenen Zahlensystemen ausgeben:

```
30 OTTO_EA.put(item => G1, base => 2); -- Ausgabe im  2-er-System
31 OTTO_EA.put(item => G2, base => 8); -- Ausgabe im  8-er-System
32 OTTO_EA.put(item => G3, base =>16); -- Ausgabe im 16-er-System
```

Zahlen im 2-er-System (bzw. im 8-er- bzw. 16-er-System) werden auch als **Binärzahlen** (bzw. als **Oktalzahlen** bzw. als **Hexadezimalzahlen**) bezeichnet. **o**

Wenn man viele OTTO-Werte als Zahlen des selben Zahlensystems ausgeben will, braucht man die Basiszahl nicht in jedem **put**-Befehl erneut anzugeben, sondern kann sie durch eine Zuweisung an die Variable namens **default_base** im Paket **OTTO_EA** "bis auf Widerruf" festlegen, etwa so:

Beispiel 9.7.4.: Das Ausgabezahlensystem für OTTO-Werte bis auf Widerruf festlegen:

```
40 OTTO_EA.default_base := 2;          -- Ausgabezahlensystem: 2er-System
41 OTTO_EA.put(item => G1);
42 OTTO_EA.put(item => G2);
43 OTTO_EA.default_base := 10;         -- Ausgabezahlensystem: 10er-System
43 OTTO_EA.put(item => G3);
```

Die Werte von G1 und G2 werden als Binärzahlen, der Wert von G3 als Dezimalzahl ausgegeben.
o

Natürlich kann man **width**- und **base**-Parameter auch gleichzeitig angeben, etwa so:
`OTTO_EA.put(item => G1, base => 2, width => 20);`
Soviel zu den Formatierungsmöglichkeiten der **put**-Prozedur im Paket **OTTO_EA**.

Wenn der **Ausführer** aufgrund eines **get**-Befehls aus dem Paket **OTTO_EA** einen OTTO-Wert einliest (z.B. von der Tastatur), geht er **normalerweise** so vor: Er **überliest** alle Trennzeichen (Blanks, Tabulatorzeichen und Zeilenenden). Dann **liest** er alle Zeichen, die zu einer Ganzzahl gehören **könnten** (Vorzeichen, Ziffern etc.) und versucht, die gelesene Zeichenkette in einen Wert des Untertyps OTTO umzuwandeln.

Durch Angabe eines **width**-Parameters im **get**-Befehl kann der Programmierer dem Ausführer befehlen, nur eine genau festgelegte Anzahl von Zeichen einzulesen, z.B. so:

Beispiel 9.7.5.: Nur eine bestimmte Anzahl von Zeichen einlesen:

```
50 OTTO_EA.get(item => G1, width =>  3;        -- Genau  3 Zeichen einlesen
51 OTTO_EA.get(item => G2, width => 10;        -- Genau 10 Zeichen einlesen
52 OTTO_EA.get(item => G3, width =>  0;        -- wie "normalerweise" einlesen
```

Der erste **get**-Befehl (in Zeile 50) bewirkt, daß der Ausführer genau 3 Zeichen einliest und versucht, diese Zeichen in einen Wert des Untertyps OTTO umzuwandeln. Der zweite **get**-Befehl funktioniert entsprechend. Die Angabe **width => 0** bewirkt, daß der Ausführer beim Einlesen wie **normaler-weise** (siehe oben) vorgeht, d.h. er überliest alle Trennzeichen ... etc.. **o**

Der **width**-Parameter der **get**-Prozedur ist vor allem dann wichtig, wenn man Zahlen "aus alten Dateien" einliest, die z.B. mit Fortran-Programmen erstellt wurden. In den Datensätzen solcher Dateien sind Zahlen häufig "in festgelegten Spalten" und ohne Trennzeichen gespeichert (z.B. eine Zahl in den Spalten 2 bis 7 und die nächste sofort anschließend in den Spalten 8 bis 12)..

Die in diesem Abschnitt behandelten Möglichkeiten bestehen natürlich nicht nur für den Untertyp **OTTO**, sondern für **alle** (signierten oder modularen) **Ganzzahluntertypen**. Mit einem **modularen** Untertyp **MONIKA** muß man allerdings die Paketschablone **decimal_io** instanziieren (um ein Paket zu erhalten, das man z.B. **MONIKA_EA** nennen könnte), und nicht die Schablone **integer_io**, die nur für **signierte** Ganzzahluntertypen zuständig ist.

Achtung: Die Prozeduren namens **ada.text_io.get** und **ada.text_io.put**, mit denen man einzelne Zeichen vom Untertyp **character** bzw. ganze Zeichenketten vom Untertyp **string** einlesen bzw. ausgeben kann, haben **keine width**- oder **base**-Parameter!

Zusammenfassung 9.7.:

- Mit dem **width**-Parameter der Prozedur **OTTO_EA.put** kann man "die Breite" beeinflussen, in der eine OTTO-Zahl ausgegeben wird.
- Mit dem **base**-Parameter kann man das **Zahlensystem** festlegen, in dem eine OTTO-Zahl bei der Ausgabe dargestellt werden soll (Zweiersystem, Achtersystem, Zwölfersystem, ...).
- Mit dem **width**-Parameter der Prozedur **OTTO_EA.get** kann man genau festlegen, wieviel Zeichen der Ausführer einlesen soll.

10. Mit Aufzählungswerten umgehen

Mit einem Computer kann man nicht nur **Berechnungen** mit **Zahlen** durchführen, sondern auch ganz **andere Daten** ganz **anders verarbeiten**, z.B.

- **Farben** zusammenstellen, mischen, verändern etc.
- **Wochentage** bestimmen
- **Zustände** einer Maschine (z.B. IST_ABGESCHALTET, LAEUFT_OK, IST_BLOCKIERT, IST_KAPUTT, WIRD_REPARIERT etc.) aufzeichnen, melden etc.

Beim Programmieren in älteren Sprachen (z.B. in Fortran) mußte der Programmierer solche anderen Daten (Farben, Wochentage, Zustände etc.) durch **Zahlen** kodieren und in seinem Programm dann die entsprechenden Zahlen verarbeiten (z.B. 0 für ROT, 1 für GRUEN, 2 für BLAU etc.). Einige neuere Sprachen (z.B. Ada) nehmen dem Programmierer einen Teil dieser Codierungsarbeit ab und ermöglichen es ihm, Bezeichnungen wie ROT, GRUEN, ... oder MONTAG, DIENSTAG, ... etc. als sogenannte **Aufzählungsliterale** zu definieren und sie dann ganz ähnlich wie Zahlenliterale (17 oder 123_456 etc.) in seinem Programm zu verwenden.

Die **Werte** solcher Aufzählungsliterale werden im Speicher eines Computers genauso durch bestimmte **Bitcodes** dargestellt wie z.B. die Werte von Zahlenliteralen und alle anderen Werte. Der Programmierer braucht sich aber in aller Regel **nicht** um die konkreten Einzelheiten dieser Codierung zu kümmern. Er benützt in seinem Programm die von ihm selbst gewählten und (hoffentlich) leicht verständlichen Bezeichner **ROT**, **GRUEN**, ..., **MONTAG**, **DIENSTAG**, ... etc., und nicht die mehr oder weniger willkürlichen und schwer zu verstehenden Codes.

10.1. Gewöhnliche Aufzählungstypen

Ehe der Programmierer Bezeichner wie **MONTAG**, **DIENSTAG**, ... etc. als Literale verwenden kann, muß er einen entsprechenden **Aufzählungstyp** vereinbaren, z.B. so:

Beispiel 10.1.1.: Einen gewöhnlichen **Aufzählungstyp** vereinbaren:

```
01 type WOCHEN_TAG is (MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
02                     FREITAG, SAMSTAG, SONNTAG);
```

Diese Typvereinbarung kann man etwa so ins Deutsche übersetzen: Erzeuge einen **Aufzählungstyp** !WOCHEN_TAG und seinen **ersten Untertyp** WOCHEN_TAG. Zum Typ **!WOCHEN_TAG** sollen genau **sieben** Werte gehören, die (ab jetzt) von den sieben Literalen MONTAG, DIENSTAG, ... etc. bezeichnet werden. Der Untertyp **WOCHEN_TAG** soll aus dem **Typ** !WOCHEN_TAG und der **Einschränkung** MONTAG..SONNTAG ("alle sieben Wochentage von Montag bis Sonntag") bestehen. **o**

Weitere **Untertypen** des Typs !WOCHEN_TAG kann man (ganz ähnlich wie Untertypen eines Ganzzahltyps) z.B. so vereinbaren:

Beispiel 10.1.2.: Untertypen eines Aufzählungstyps vereinbaren:

```
03 subtype ARBEITS_TAG is WOCHEN_TAG range MONTAG..FREITAG;
04 subtype FEIER_TAG   is WOCHEN_TAG range SAMSTAG..SONNTAG;
05 subtype FRISOER_TAG is WOCHEN_TAG range DIENSTAG.. SAMSTAG;
```

Die Untertypvereinbarung in Zeile 03 kann man etwa so ins Deutsche übersetzen: Erzeuge einen **Untertyp** namens **ARBEITS_TAG**, der aus dem Typ **!WOCHEN_TAG** und der **Einschränkung** MONTAG..FREITAG besteht. **o**

Variablen und **Konstanten** der vereinbarten Untertypen kann man (ganz ähnlich wie bei Ganzzahluntertypen) z.B. so vereinbaren:

Beispiel 10.1.3.: Konstanten und **Variablen** von Aufzählungsuntertypen vereinbaren:

```
06 ZAHL_TAG      : constant WOCHEN_TAG := FREITAG; -- Konstante
07 AUSSCHLAF_TAG :          FEIER_TAG  := SAMSTAG; -- Variable mit  Anfangswert
08 BESPRECHUNG   :          ARBEITS_TAG;           -- Variable ohne Anfangswert
```

Diese Vereinbarungen kann man etwa so ins Deutsche übersetzen:
Erzeuge eine **Konstante** namens ZAHL_TAG vom Untertyp WOCHEN_TAG mit dem Wert FREITAG (Zeile 06).
Erzeuge eine **Variable** namens AUSSCHLAF_TAG vom Untertyp FEIER_TAG mit dem **Anfangswert** SAMSTAG (Zeile 07).
Erzeuge eine **Variable** namens BESPRECHUNG vom Untertyp ARBEITS_TAG. **o**

Wenn der Programmierer Werte eines Aufzählungsuntertyps **einlesen** oder **ausgeben** will, dann muß er die Paketschablone **enumeration_io** im Paket **ada.text_io** mit dem betreffenden Untertyp als enum-Parameter instanziieren, z.B. so:

Beispiel 10.1.4.: Ein Paket mit Ein- und Ausgabebefehlen vereinbaren:

```
09 package ARBEITS_TAG_EA is
10    new ada.text_io.enumeration_io(enum => ARBEITS_TAG);
```

Diese Paketvereinbarung kann man etwa so ins Deutsche übersetzen: Erzeuge ein **Paket** namens **ARBEITS_TAG_EA** als Instanz der Paketschablone **enumeration_io** im Paket **ada.text_io**. Die Befehle im Paket **ARBEITS_TAG_EA** sollen "auf den Untertyp **ARBEITS_TAG** abgestimmt" sein. **o**

Mit dem Paket **ARBEITS_TAG_EA** kann man nur Werte des Untertyps **ARBEITS_TAG** einlesen und ausgeben (**SAMSTAG** und **SONNTAG** gehören **nicht** dazu). Will der Programmierer alle Werte des Untertyps **WOCHEN_TAG** einlesen und ausgeben können, dann muß er beim Instanziieren der Paketschablone **enumeration_io** als enum-Parameter diesen Untertyp angeben (also **enum => WOCHEN_TAG** anstelle von **enum => ARBEITS_TAG**).

Mit der Prozedur **get** im **Paket ARBEITS_TAG_EA** kann man Werte des Untertyps **ARBEITS-_TAG** z.B. von der Tastatur einlesen und mit der Prozedur **put** kann man solche Werte z.B. zum Bildschirm ausgeben, etwa so:

Beispiel 10.1.5.: Werte des Untertyps ARBEITS_TAG **einlesen** und **ausgeben**:

```
10 begin
11    ada.text_io.put   (item => "Bitte geben Sie einen Wochentag ein: ");
```

```
12    ARBEITS_TAG_EA.get(item =>  BESPRECHUNG);
13    ada.text_io.put   (item => "Zahltag ist ");
14    ARBEITS_TAG_EA.put(item =>  ZAHL_TAG);
```

Wenn der **get**-Befehl in Zeile 12 ausgeführt wird, dann sollte der Benutzer eine Zeichenkette wie z.B. **MONTAG** oder **MITTWOCH** oder **FREITAG** (aber nicht **SAMSTAG** und schon gar nicht **SONNABEND** oder **SATURDAY**) eingeben. Große und kleine Buchstaben werden dabei vom Ausführer nicht unterschieden, d.h. der Benutzer kann auch z.B. **Montag** oder **montag** oder sogar **MoNtAg** eingeben (über Geschmack sollte man möglichst selten streiten).

Der **put**-Befehl in Zeile 14 bewirkt, daß der Wert der Konstanten **ZAHL_TAG** in eine entsprechende Zeichenkette umgewandelt wird und diese Zeichenkette ("FREITAG") zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgegeben wird. **o**

Mit jedem und **für** jeden Aufzählungstyp erzeugt der Ausführer eine Reihe von **Operationen**, aber keine Rechenoperationen wie +, -, *, / etc. und keine logischen Verknüpfungsoperationen wie **and**, **or**, **xor** etc.. Zu den interessanteren Operationen eines Aufzählungstyps gehören die beiden Attributfunktionen **'succ** (successor function, Nachfolgerfunktion) und **'pred** (predecessor function, Vorgängerfunktion). Die Funktion **WOCHEN_TAG'succ** hat einen Parameter vom Typ **!WOCHEN_TAG** (egal, von welchem Untertyp) und liefert den **nächsten** Wochentag. Dabei ist die **Reihenfolge** entscheidend, in der die Literale MONTAG, DIENSTAG etc. in der entsprechenden **Typvereinbarung** aufgezählt wurden (siehe oben Beispiel 10.1.1.). Entsprechend liefert die Funktion **WOCHEN_TAG'pred** den jeweils vorigen Wochentag, wie das folgende Beispiel verdeutlichen soll:

Beispiel 10.1.6.: Die Attributfunktionen **WOCHEN_TAG'succ** und **WOCHEN_TAG'pred**:

Ausdruck	**Wert des Ausdrucks**
`WOCHEN_TAG'succ(MONTAG)`	`DIENSTAG`
`WOCHEN_TAG'pred(DONNERSTAG)`	`MITTWOCH`
`WOCHEN_TAG'succ(ZAHL_TAG)`	`SAMSTAG`
`WOCHEN_TAG'pred(AUSSCHLAF_TAG)`	`FREITAG`
`WOCHEN_TAG'pred(WOCHEN_TAG'pred(ZAHL_TAG))`	`MITTWOCH`
`WOCHEN_TAG'succ(SONNTAG)`	`löst constraint_error aus!`
`WOCHEN_TAG'pred(MONTAG)`	`löst constraint_error aus!`

Solche Ausdrücke kann man z.B. in **Zuweisungsanweisungen** verwenden, etwa so:

```
15 begin
16    AUSSCHALF_TAG := WOCHEN_TAG'succ(ZAHL_TAG);
17     AUSSCHLAF_TAG := WOCHEN_TAG'pred(AUSSCHLAF_TAG);
```

Die Zuweisung in Zeile 17 löst die Ausnahme **constraint_error** aus, weil der Wert FREITAG nicht zum Untertyp FEIER_TAG der Variablen AUSSCHLAF_TAG gehört. **o**

Die in diesem Abschnitt behandelten Befehle für den Umgang mit Aufzählungstypen und Aufzählungswerten werden im Beispielprogramm **AUFZT_01** angewendet.

Aufgabe 10.1.1.: Schreiben Sie ein Programm namens **AUFZT_02**. Darin soll
1. Ein Aufzählungstyp **!FARBE** und sein erster Untertyp namens **FARBE** vereinbart werden. Zum Typ **!FARBE** sollen mindestens vier Werte gehören (nehmen Sie Ihre Lieblingsfarben).
2. Ein Wert vom Untertyp **FARBE** soll eingelesen und der Vorgänger und der Nachfolger dieses Wertes sollen ausgegeben werden.

10.2. Zeichentypen sind spezielle Aufzählungstypen

Als **Literal** eines Aufzählungstyps darf der Programmierer nicht nur **Bezeichner** wie MONTAG, DIENSTAG, ROT, WIRD_REPARIERT etc. definieren, sondern auch **Zeichenliterale** wie 'A', 'x', '?' etc. (siehe Abschnitt 6.6.), z.B. so:

Beispiel 10.2.1.: Aufzählungstypen mit **Zeichenliteralen**:

```
01 type VOKAL            is ('A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u');
02 type ROEMISCHE_ZIFFER is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
03 type STELLUNG         is (ANFANG, 'A', 'B', MITTE, 'C', 'D', ENDE);
o
```

Der Typ !STELLUNG soll zeigen, daß man **Bezeichner** (wie ANFANG, MITTE, etc.) und **Zeichenliterale** (wie 'A', 'B' etc.) beliebig **kombinieren** darf. Ein **Zeichentyp** ist ein Aufzählungstyp, in dessen Vereinbarung mindestens **ein** Zeichenliteral vorkommt.

Ein Aufzählungsliteral (Bezeichner oder Zeichenliteral) kann gleichzeitig zu **mehreren** Aufzählungstypen gehören. Z.B. gehört das Zeichenliteral **'A'** zum Typ **!VOKAL** zum Typ **!STELLUNG** und zum vordefinierten Typ **!character** (siehe unten). Solche Literale bezeichnet man als **überladen**, weil sie "mit mehreren verschiedenen Bedeutungen be- oder überladen" sind. Das Zeichenliteral **'A'** bezeichnet einen Wert des Typs **!VOKAL**, einen Wert des Typs **!STELLUNG** und einen Wert des Typs **!character**, hat also drei Bedeutungen. Der Wert **'A'** des Typs **!STELLUNG**, der Wert **'A'** des Typs **!VOKAL** und der Wert **'A'** des Typs **!character** sind drei verschiedene Werte (ganz ähnlich, wie auch der Wert **5** des Typs **!AEPFEL**, der Wert **5** des Typs **!BIRNEN** und der Wert **5** des Typs **!integer** drei verschiedene Werte sind).

In Ada gibt es zwei **vordefinierte Zeichentypen**. Zum Typ **!character** (und zu seinem ersten Untertyp **character**) gehören genau **256** Zeichen, darunter die großen Buchstaben 'A', 'B', ... 'Z', die Dezimalziffern '0', '1', ... '9', die kleinen Buchstaben 'a', 'b', ... , 'z', zahlreiche Sonderzeichen wie '!', '?', '/', '*', '%' etc.. und einige Zeichenwerte, für die es keine Zeichenliterale, sondern nur **Konstanten** gibt wie z.B. NUL, ESC, CR, LF etc.. Diese Konstanten sind im Standardpaket namens **ada.characters.latin_1** vereinbart. Nähere Einzelheiten findet man in (ARM A.3.3).

Zum Typ **!wide_character** (und zu seinem ersten Untertyp **wide_character**) gehören genau **65_536** Zeichen (so viele, wie man mit 16 Bit darstellen kann). Die ersten 256 Zeichen des Typs **!wide_character** entsprechen den Zeichen des Typs **!character**. Mit den Werten des Typs **!wide_character** kann man auch z.B. griechische, kyrillische und koreanische Buchstaben sowie chinesische und japanische Schriftzeichen und weitere Buchstaben und Schriftzeichen darstellen. In Ada-Programmen kann man solche Zeichen in Form von hexadezimalen Codes ein- und ausgeben und beliebig verarbeiten. Für ihre "graphische Ein- und Ausgabe" braucht man allerdings spezielle Hard- und Software.

Beim **Ein**- und **Ausgeben** von Werten eines **Zeichentyps** ist eine kleine **Besonderheit** zu beachten: Der Wert eines Zeichenliterals (z.B. der Wert von 'A') wird in Form von **drei** Zeichen ausgegeben: einem großen A mit je einem Apostroph links und rechts. Ganz entsprechend muß man für den Wert von 'A' drei Zeichen eingeben: ein Apostroph, ein großes A und noch ein Apostroph. Mit dem Programm AUFZT_03 kann man ausprobieren, wie das praktisch geht.

Um Werte des vordefinierten Untertyps **character** einzulesen oder auszugeben, braucht man keine Paketschablone zu instanziieren. Im Paket **ada.text_io** gibt es schon Prozeduren namens **get** und **put**, mit denen man einen Wert vom Untertyp **character** einlesen bzw. ausgeben kann, und zwar **ohne** irgendwelche **Apostrophs** (siehe z.B. das Programm HALLO_01).

Aufgabe 10.2.1.: Die folgenden Typvereinbarungen sind beide korrekt:

```
10 type ABC1 is ('A', 'B', 'C');
11 type ABC2 is ( A ,  B ,  C );
```

Erläutern Sie die Unterschiede zwischen den Untertypen ABC1 und ABC2. **o**

10.3. Der vordefinierte Typ !boolean

Der Typ **!boolean** ist (ähnlich wie der Typ **!character**) ein **vordefinierter** Typ, d.h. er existiert "schon immer". Der erste Untertyp von **!boolean** heißt natürlich **boolean**. Der Typ **!boolean** und sein erster Untertyp **boolean** (so kann man sich zumindest vorstellen) wurden "in Urzeiten" aufgrund der folgenden **Typvereinbarung** vom Ausführer erzeugt:

```
type boolean is (false, true);
```

Daraus folgt: Der Typ **!boolean** ist ein **Aufzählungstyp**, zu dem genau **zwei** Werte gehören, die man mit den Literalen **true** bzw. **false** bezeichnen kann. Außerdem gilt:

```
boolean'first          ist gleich false
boolean'last           ist gleich true
boolean'succ(false)    ist gleich true
boolean'pred(true)     ist gleich false
```

Die weiteren Beispiele in diesem Abschnitt gehen von den folgenden **Vereinbarungen** aus:

```
01 type OTTO is range -20_000..+30_000;      -- Ein Ganzzahltyp
02 G1, G2, G3 : OTTO;                        -- Drei Ganzzahlvariablen
03 B1, B2, B3 : boolean;                     -- Drei boolesche Variablen
```

Für den Typ **!OTTO** gibt es unter anderem die Rechenoperationen **+**, **-**, ***** und **/** (mit **zwei** Parametern vom Typ **!OTTO**) und die beiden Vorzeichenoperationen **+** und **-** (mit **einem** Parameter vom Typ **!OTTO**). Jede dieser sechs Operationen liefert ein Ergebenis vom Typ **!OTTO**. Wenn z.B. **G1** den Wert **3** hat, dann ist **G1 + 5** gleich **8** und **-(2 * G1)** ist gleich **-6** etc..

Ganz entsprechend gibt es für den Typ **!boolean** die logischen Operationen **and**, **or** und **xor** (mit **zwei** Parametern vom Typ **!boolean**) und die logische Operation **not** (mit **einem** Parameter vom Typ **!boolean**). Jede dieser vier Operationen liefert ein Ergebnis vom Typ **!boolean**. "Was diese Operationen bewirken" wird im (ARM 4.5.1) anhand einer Wertetabelle vollständig beschrieben. Hier nur ein paar Beispiele dazu:

Beispiel 10.3.1.: Boolesche Ausdrücke 1

Angenommen, die booleschen Variablen **B1**, **B2** und **B3** haben die Werte **true**, **true** bzw. **false**. Dann haben folgende Ausdrücke die danebenstehenden Werte:

Ausdruck	Wert
`true and false`	`false`
`B1 and false`	`false`
`true and B3`	`false`
`B1 and B3`	`false`
`B1 or B3`	`true`

Ausdruck	Wert
`B1 xor B3`	`true`
`not false`	`true`
`not B2`	`false`
`(B1 or B3) and B2`	`true`
`(B1 and B3) or B2`	`false`

Aufgabe 10.3.1.: Angenommen, die Variablen **B1**, **B2** und **B3** haben die Werte **true**, **true**, bzw. **false** (wie im vorigen Beispiel). Berechnen Sie die Werte der folgenden Ausdrücke:

```
B1 and B2 and B3,     not B1 or B2,       not (B1 or B2),    not B1 or not B2,
B1 or B2 or not B3,   B1 xor B2 xor B3,   not B2 xor not B3,
(B1 or B2) xor (B2 and B3).  o
```

Zum Ganzzahltyp **!OTTO** gehören nicht nur **Rechenoperationen** wie +, -, *, / etc., sondern auch sechs **Vergleichsoperationen** namens = ("gleich"), /= ("ungleich"), < ("kleiner"), <= ("kleiner-gleich"), >= ("größer-gleich") und > ("größer"). Genau genommen gehören diese sechs Operationen nicht nur zum Typ **!OTTO**, sondern gleichzeitig auch zum Typ **!boolean**. Denn jede dieser Operationen hat zwei **Parameter** vom Typ **!OTTO** und liefert ein **Ergebnis** vom Typ **!boolean**.

Beispiel 10.3.2.: Boolesche Ausdrücke 2:

Die folgenden Ausdrücke gehören zum Typ **!boolean** (obwohl die Parameter der darin vorkommenden Vergleichsoperationen zum Typ **!OTTO** gehören):

```
G1 < 5,   G2 >= G1,   G3 <= G1 + 17,   2 * G1 /= G3 + 4,   G1 < G2 and G2 < G3,
G3 > 0 or B1,         not(G1 < G2),    G1 /= G2 and G2 /= G3 and G3 /= G1  o
```

Zum Aufzählungstyp **!boolean** gehören nicht nur **logische Operationen** wie **and** und **or** etc., sondern auch sechs **Vergleichsoperationen** namens =, /=, <, <=, >= und >. Jede dieser sechs Operationen hat **zwei** Parameter vom Typ **!boolean** und liefert ein Ergebnis vom Typ **!boolean**. Aus der Vereinbarung des Typs **!boolean** (siehe oben) folgt, daß der Ausdruck **false < true** den Wert **true** hat. Die Werte aller anderen Vergleichsausdrücke mit booleschen Parametern kann man sich daraus ausrechnen.

Beispiel 10.3.3.: Boolesche Ausdrücke 3:

Angenommen, die booleschen Variablen **B1**, **B2** und **B3** haben die Werte **true**, **true** bzw. **false**, dann haben die folgenden Ausdrücke die danebenstehenden Werte:

Ausdruck	Wert
`false > true`	`false`
`false >= true`	`true`
`B1 <= true`	`true`

Ausdruck	Wert
`B1 /= B3`	`true`
`(B1 <= B3) = B2`	`false`
`B1 < B2 < B3`	`false`

Aufgabe 10.3.2.: Füllen Sie die folgende Tabelle aus:

Wert der Variablen B1	Wert der Variablen B2	Wert des Ausdrucks B1 /= B2	Wert des Ausdrucks B1 xor B2
true	true		
true	false		
false	true		
false	false		

Was kann man aus der (ausgefüllten) Tabelle über die Operationen **/=** und **xor** schließen? **o**

Den **Wert** eines **Ausdrucks** vom Typ **!OTTO** kann man einer **Variablen** vom Typ **!OTTO** zuweisen. Entsprechend kann man den **Wert** eines **Ausdrucks** vom Typ **!boolean** einer **Variablen** vom Typ **!boolean** zuweisen, etwa so:

Beispiel 10.3.4.: Zuweisungen

```
10 G1 := 10;
11 G2 := G1;
12 G3 := (12 - G1) * (G2**2 + 20);
13 B1 := true;
14 B2 := B1;
15 B3 := (true and not B2) or (B1 xor B2);
16 B1 := G1 < G2;
17 B2 := B1 < B2;
18 B3 := (G1 < G2) or (B1 < B2);
19 B1 := B2 and (B1 < (G1 < G2) and not B3;
```

o

Aufgabe 10.3.3.: Führen Sie die Zuweisungen im vorigen Beispiel (10.3.4.) mit Papier und Bleistift aus. Beachten Sie, daß die Reihenfolge der Zuweisungen wichtig ist, weil einige Zuweisungen die Werte von Ausdrücken in späteren Zuweisungen beeinflussen. **o**

Zum Typ **!boolean** gehören noch zwei weitere logische Operationen namens **and then** und **or else**, die den Operationen **and** bzw. **or** ähneln, sich aber gleichzeitig auf subtile und interessante Weise von diesen unterscheiden. Hier die Gemeinsamkeiten und Unterschiede:

Wenn die boolesche Variable **B1** den Wert **false** hat, dann hat auch der Ausdruck **B1 and (G1 < G2 or B3)** den Wert **false**, unabhängig davon, ob der Teilausdruck **(G1 < G2 or B3)** rechts von **and** den Wert **true** oder **false** hat. Trotzdem berechnet der Ada-Ausführer auch den Wert dieses rechten Teilausdrucks, wenn er den Wert des Gesamtausdrucks berechnet.

Das ist bei der Operation **and then** (z.B. im Ausdruck **B1 and then (G1 < G2 or B3)**) ein bißchen anders: Wenn der Parameter links von **and then** den Wert **false** hat, schaut der Ausführer den rechten Parameter "gar nicht an" und nimmt gleich **false** als Wert des Gesamtausdrucks. Nur wenn der linke Parameter den Wert **true** hat, berechnet der Ausführer auch den Wert des rechten Parameters.

In vielen Fällen liefern die Operationen **and** und **and then** genau die gleichen Ergebnisse, wenn man sie auf die gleichen Parameter anwendet, und die Operation **and then** ist höchstens ein bißchen schneller als **and**, weil sie manchmal einen Teilausdruck nicht auswertet. Besonders interessant sind aber die Fälle, wo die Operationen **and** und **and then** zu verschiedenen Ergebnissen führen. Hier ein typisches Beispiel:

Beispiel 10.3.5.: Die Operationen **and** und **and then** führen zu verschiedenen Ergebnissen:

```
20 B1 := (G1 /= 0) and then (100/G1 < 10);
21 B2 := (G1 /= 0) and      (100/G1 < 10);
```

Wenn **G1** den Wert **0** hat, wird der Variablen **B1** der Wert **false** zugewiesen. Die zweite Zuweisung löst dagegen die Ausnahme **constraint_error** aus, weil bei der Auswertung des Teilausdrucks **(100/G1 < 10)** durch 0 geteilt wird.

Wenn bei der Operation **or else** der linke Parameter den Wert **true** hat, wird der rechte Parameter **nicht** ausgewertet und **true** als Ergebnis des Gesamtausdrucks genommen. Nur wenn der linke Parameter den Wert **false** hat, wird auch der rechte Parameter ausgewertet. Hier ein Beispiel, bei dem **or else** und **or** manchmal verschiedene Ergebnisse liefern:

Beispiel 10.3.6.: Die Operationen **or** und **or else** führen zu verschiedenen Ergebnissen:

```
30 B1 := (G1 = 0) or else (100/G1 < 10);
31 B2 := (G1 = 0) or      (100/G1 < 10);
```

Wenn **G1** den Wert **0** hat, wird der Variablen **B1** der Wert **true** zugewiesen, aber die zweite Zuweisung löst (wegen der Division durch 0) die Ausnahme **constraint_error** aus. **o**

Die logischen Operationen **and then** und **or else** werden im Deutschen ein bißchen abschreckend als **Kurzauswertungsoperationen** und im Englischen auch nicht viel griffiger als **short circuit control forms** bezeichnet. Trotzdem ist es in den meisten Fällen günstiger, **and then** anstelle von **and** bzw. **or else** anstelle von **or** zu benützen.

Zwei Vergleichsoperationen namens = und /= ("gleich" und "ungleich") gibt es in Ada zu (fast) **jedem** Typ, egal, ob der Typ vordefiniert ist oder vom Programmierer vereinbart wurde. Weitere vier Vergleichsoperationen namens <, <=, >= und > ("kleiner", "kleiner-gleich", "größer-gleich" und "größer") gibt es nicht zu jedem Typ, aber unter anderem zu jedem **Ganzzahlttyp**, zu jedem **Bruchzahltyp**, zu jedem **Reihungstyp** und zu jedem **Aufzählungstyp** (Bruchzahltypen und Reihungstypen werden später behandelt). Alle diese Vergleichsoperationen gehören zu "ihrem" jeweiligen Typ (weil sie nur auf Parameter "ihres Typs" angewendet werden dürfen) und gleichzeitig zum Typ **!boolean**, weil sie ein Ergebnis vom Typ **!boolean** (**false** oder **true**) liefern. Zwei Werte, die zu verschiedenen Typen gehören, sind nicht nur verschieden, sondern sogar "unvergleichbar". Konkret bedeutet das: Ausdrücke wie **true /= 7** oder '**A' = 15** oder **G1 < B1** etc. enthalten Typenfehler und sind in Ada nicht erlaubt.

Mit der Operation **in** kann man testen, ob ein Wert zu einem bestimmten **Untertyp** oder **Bereich** gehört, z.B. so:

Beispiel 10.3.7.: Der **in**-Zugerhörigkeitstest (membership test):

```
01    type      OTTO is      range -5_000..+10_000;
02    subtype P_OTTO is OTTO range      1..20_000;  -- Positive Werte
03    subtype K_OTTO is OTTO range    -10..+20;     -- Kleine   Werte
04    G1, G2 : OTTO;
05    C1, C2 : character;
06    B1, B2, B3, B4, B5, B6, B7, B8 : boolean;
07 begin
08    ...    -- Werte nach G1, G2, C1 und C2 bringen
09    B1 := G1     in P_OTTO;
10    B2 := G2 not in K_OTTO;
11    B3 := G1     in -17..+25;
```

```
12    B4 := 17 not in  G1..G2;
13    B5 := G1+G2  in -17..+25;
14    B6 := C1     in 'A'..'Z';
15    B7 := C2 not in '0'..'9';
16    B8 := (C1 in 'a'..'z') or (C1 in 'A'..'Z');
```

Im folgenden wird für jede der Variablen **B1** bis **B8** kurz beschrieben, unter welcher Bedingung ihr in Zeile 09 bis 16 der Wert **true** zugewiesen wird. Wenn die Bedingung **nicht** erfüllt ist, wird der betreffenden Variablen der Wert **false** zugewiesen:

Variable Unter welcher **Bedingung** wird der Variablen der Wert **true** zugewiesen?

B1	Der Wert der Variablen **G1** gehört zum Untertyp **P_OTTO**.
B2	Der Wert der Variablen **G2** gehört nicht zum Untertyp **K_OTTO**.
B3	Der Wert der Variablen **G1** liegt im Bereich **-17..+25**.
B4	Die Zahl **17** liegt im Bereich **G1..G2**.
B5	Der Wert des Ausdrucks **G1+G2** liegt im Bereich **-17..+25**.
B6	Der Wert der Variablen **C1** liegt im Bereich **'A'..'Z'**.
B7	Der Wert der Variablen **C2** liegt nicht im Bereich **'0'..'9'**.
B8	Der Wert der Variablen **C1** ist ein (kleiner oder großer) Buchstabe.

Eine vollständige Beschreibung der Operationen **in** und **not in** findet man im (ARM 4.5.2). **o**

Boolesche Ausdrücke wie **false**, **true**, **B1**, **B1 or B2**, **G1 < G2**, **B3 and then (G1 = G2)** etc. werden häufig auch als **Bedingungen** bezeichnet, insbesondere im Zusammenhang mit **if-Anweisungen** und **Schleifenanweisungen** (siehe unten Abschnitt 12. bzw. 12.3. und 12.5.). Von einer **Bedingung** sagt man auch, daß sie **erfüllt** ist (d.h. den Wert **true** hat) oder **nicht erfüllt** ist (d.h. den Wert **false** hat).

Zum Schluß dieses Abschnitts noch ein kleiner **Tip**: Der boolesche Ausdruck **B1 = true** hat in allen Zusammenhängen genau den gleichen Wert wie der (einfachere) Ausdruck **B1**. Nur wenn der Programmierer z.B. pro Zeichen bezahlt wird, sollte er den komplizierteren Ausdruck anstelle des einfacheren verwenden.

Zusammenfassung 10.3.:

- Der Aufzählungstyp **!boolean** und sein erster Untertyp **boolean** sind vordefiniert.
- Zum Typ **!boolean** (und zu **boolean**) gehören genau zwei Werte, **false** und **true**, und **false** ist kleiner als **true**.
- Zum Typ **!boolean** gehören die logischen Operationen **and**, **or**, **xor**, **not**, **and then** und **or else**.
- Außerdem gehören zum Typ **!boolean** alle Vergleichsoperationen namens =, /=, <, <=, >= und > (egal, zu welchem anderen Typ diese Operationen auch noch gehören).
- Die Operation /= für den Typ **!boolean** und die Operation **xor** unterscheiden sich nur durch ihre Namen.
- Außerdem gehören zum Typ **!boolean** die Zugehörigkeitstests **in** und **not in**.
- Boolesche Ausdrücke werden häufig auch als **Bedingungen** bezeichnet.

10.4. Formatkontrolle beim Ausgeben von Aufzählungswerten

Dieser Abschnitt ist für ein tieferes Verständnis von Ada **völlig unwichtig** und kann getrost übersprungen werden. Die Beispiele in diesem Abschnitt beruhen auf folgenden Vereinbarungen:

```
01 type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, GELB, WEISS);
02 package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
03 FARBE1, FARBE2, FARBE3, FARBE4 : FARBE;
```

Außerdem wird angenommen, daß den Variablen **FARBE1** bis **FARBE4** irgendwelche Werte des Untertyps **FARBE** zugewiesen wurden. Die in diesem Abschnitt behandelten Möglichkeiten der **Formatkontrolle** werden auch im Beispielprogramm **AUFZT_04** verwendet und vorgeführt.

Beim **Einlesen** von Werten eines Aufzählungstyps ist **keine** Formatkontrolle möglich. Die Formatkontrolle beim **Ausgeben** von Aufzählungswerten (z.B. des Untertyps FARBE) erfolgt ganz ähnlich, wie beim Ausgeben von **Ganzzahlen** (siehe oben Abschnitt 9.7.).

Beim Aufrufen der Prozedur **FARBE_EA.put** kann man (zusätzlich zum **item**-Parameter) noch einen **width**-Parameter oder einen **set**-Parameter (oder beide) angeben. Mit dem **width**-Parameter kann man festlegen, wie viele Zeichen **mindestens** ausgegeben werden sollen. Das auszugebende FARBE-Literal wird aber auf jeden Fall in voller Länge ausgegeben. Falls es kürzer ist als hinter **width =>** angegeben wird es rechts mit Blankzeichen "verlängert".

Mit dem **set**-Parameter kann man angeben, ob die Ausgabe mit **kleinen** oder **großen** Buchstaben geschrieben werden soll ("set" soll an "type set" erinnern). Es sind nur zwei verschiedene Angaben möglich: **set => FORMAT_EA.lower_case** oder **set => FORMAT_EA.upper_case**.

Beispiel 10.4.1.: **width**- und **set**-Parameter in Aufrufen der Prozedur **FARBE_EA.put**:

```
10 FARBE_EA.put(item => FARBE1, width => 5);                        -- Mind. 5 Zeichen
11 FABER_EA.put(item => FARBE2, width => FARBE'width);              -- Genau 7 Zeichen
12 FARBE_EA.put(item => FARBE3, set => FARBE_EA.lower_case);        -- Kleine Buchstb.
13 FARBE_EA.put(item => FARBE4, set => FARBE_EA.upper_case);        -- Grosse Buchstb.
14 FARBE_EA.put(item => FARBE'succ(FARBE1),
15              set   => FARBE_EA.lower_case,                       -- Kleine Buchstb.
16              width => FARBE'width);                              -- Genau 7 Zeichen
```

Der Wert der Variablen **FARBE1** wird in einer Breite von **mindestens** fünf Zeichen ausgegeben (z.B. **"ROT "** (mit **zwei** Blanks am Ende), **"BLAU "** (mit **einem** Blank am Ende), **"GRUEN"** (**ohne** Blanks am Ende). Falls **FARBE1** den Wert **SCHWARZ** hat, werden trotzdem alle **sieben** Zeichen von "SCHWARZ" ausgegeben.
Der Wert der Variablen **FARBE2** wird in einer Länge von genau **sieben** Zeichen ausgegeben. Das Attribut **FARBE'width** hat den Wert **sieben**, weil das "längste Literal" des Untertyps **FARBE** (nämlich das Literal **SCHWARZ**) einen Bezeichner hat, der sieben Zeichen lang ist. Das Attribut **FARBE'width** ist z.B. dann sehr nützlich, wenn man verschiedene Farben untereinander in einer Spalte der Breite sieben ausgeben will. Statt einfach **FARBE'width** könnte man auch kompliziert **7** schreiben, aber dann wäre das Programm nicht so leicht lesbar und änderungsfreundlich.
Der Wert der Variablen **FARBE3** wird in **kleinen** Buchstaben ausgegeben.
Der Wert der Variablen **FARBE4** wird in **großen** Buchstaben ausgegeben.
In Zeile 14 bis 16 steht ein **put**-Befehl, der den auf den Wert von **FARBE1** folgenden FARBE-Wert (**FARBE'succ(FARBE1)**) in **kleinen** Buchstaben und als Zeichenkette der Länge **sieben** ausgibt. **o**

Wenn man viele Werte des Untertyps **FARBE** im gleichen Format ausgeben will (z.B. in **kleinen** Buchstaben und in einer Breite von **FARBE'width**), dann braucht man nicht in jedem **put**-Befehl

entsprechende **width**- und **set**-Parameter anzugeben, sondern kann durch Zuweisungen an die Variablen namens **default_width** bzw. **default_setting** im Paket **FARBE_EA** das Format "bis auf Widerruf" festlegen, etwa so:

Beispiel 10.4.2.: Formatkontrolle mit den Variablen **FARBE_EA.default_setting** und **FARBE_EA.default_width**:

```
20 FARBE_EA.default_setting := FARBE_EA.lower_case;
21 FARBE_EA.default_width   := FARBE'width;
22 FARBE_EA.put(item => FARBE1);      -- Kleine Buchstb., Breite 7
23 FABER_EA.put(item => FARBE2);      -- Kleine Buchstb., Breite 7
24 FARBE_EA.default_setting := FARBE_EA.upper_case;
25 FARBE_EA.put(item => FARBE3);      -- Grosse Buchstb., Breite 7
26 FARBE_EA.put(item => FARBE4, set => FARBE_EA.lower_case);
```

Die Werte der Variablen **FARBE1** und **FARBE2** werden in **kleinen** Buchstaben und in einer Breite von **FARBE'width** ausgegeben. Der Wert von **FARBE3** wird in **großen** Buchstaben (und in einer Breite von **FARBE'width**), der Wert von **FARBE4** dagegen in **kleinen** Buchstaben (und in einer Breite von **FARBE'width**) ausgegeben. **o**

Das letzte Beispiel in Zeile 26 soll zeigen: Ein **width**- oder **set**-Parameter im **put**-Befehl "**wirkt stärker**" als die Werte in den Variablen **default_width** bzw. **default_set**. im Paket FARBE_EA.

Aufgabe 10.4.1.: Lesen Sie im **ARM** folgende Stellen nach und beantworten Sie dann die untenstehenden Fragen:

A.10.1(1-7)	Die ersten zehn Zeilen des Paketes **ada.text_io**
A.10.1(78-83)	Die Paketschablone **enumeration_io** (ca 25 Zeilen)
A.10.10	Erläuterungen zur Paketschablone **enumeration_io** (ca. eine Seite)

1. Zu welchem Untertyp gehört die Variable **default_width** in der Schablone **enumeration_io**?
2. Wo ist dieser Untertyp vereinbart? Welche Werte gehören zu diesem Untertyp?
3. Zu welchem Untertyp gehört die Variable **default_setting** in der Schablone **enumeration_io**?
4. Wo ist dieser Untertyp vereinbart? Welche Werte gehören zu diesem Untertyp?

o

Aufgabe 10.4.2.: Schreiben Sie ein Programm namens **AUFZT_05**, in dem zwei Aufzählungstypen **!WOCHEN_TAG** und **!DAY_OF_WEEK** vereinbart und die Wochentage auf **Deutsch** und **Englisch** in zwei Spalten ausgegeben werden, etwa so:

```
MONTAG     monday
DIENSTAG   tuesday
MITTWOCH   wednesday
DONNERSTAG thursday
FREITAG    friday
SAMSTAG    saturday
SONNTAG    sunday
```

Natürlich soll in diesem Programm **nicht** die Prozedur **ada.text_io.put** verwendet werden. **o**

Zusammenfassung 10.4.:

- Die Prozedur **FARBE_EA.get** hat keine Formatparameter.
- Mit dem **width**-Parameter der Prozedur FARBE_EA.put kann man festlegen, wieviele Zeichen mindestens ausgegeben werden sollen.

- Mit dem **set**-Parameter kann man Groß- oder Kleinschreibung festlegen.
- Das Attribut **FARBE'width** gibt die Länge des längsten Literals des Untertyps FARBE an.

Zentraldokument: Nach Filialdokument A95-07-10

11. Die Klasse der diskreten Typen

Typen, die wichtige Eigenschaften gemeinsam haben, faßt man in Ada zu (Typ-) **Klassen** zusammen und vereinigt bestimmte solche Klassen zu höheren Klassen und diese zu noch höheren Klassen etc.. Die folgende Darstellung zeigt einen wichtigen Teil dieser Hierarchie von Typklassen:

Diskrete Typen	(discrete types)
Ganzzahltypen	(integer types)
Signierte ("vorzeichenbehaftete") Ganzzahltypen	(signed intger types)
!integer	
!OTTO, !AEPFEL, !BIRNEN, ... etc.	
Modulare Ganzzahltypen	(modular types)
!MONIKA ... etc.	
Aufzählungstypen	(enumeration types)
Gewöhnliche Aufzählungstypen	(ordinary enumeration types)
!WOCHEN_TAG, !FARBE, !COLOR, ... etc.	
Boolesche Typen	
!boolean	
Zeichentypen	(character types)
!character	
!wide_character	
!ROEMISCHE_ZIFFER, !VOKAL, ... etc.	

Alle bisher behandelten Typen gehören zur Klasse der **diskreten Typen**. Ein diskreter Typ ist entweder ein **Ganzzahltyp** oder ein **Aufzählungstyp** etc..

Das folgende Beispiel soll noch einmal zusammenfassen, wie die **Vereinbarungen** von Typen der verschiedenen Klassen aussehen. Mit jedem **Typ** wird automatisch auch sein **erster Untertyp** vereinbart.

Beispiel 11.1.: Vereinbarungen von Typen **verschiedener Klassen**:

```
10 type OTTO   is range -20_000..+30_000;            -- Signierter Ganzzahltyp
11 type MONIKA is mod 256;                           -- Modularer Ganzzahltyp
12 type FARBE  is (SCHWARZ, ROT, GRUEN, BLAU,        -- Gewoehnlicher Auf-
13                 GELB, WEISS);                     -- zaehlungstyp
14 type VOKAL  is ('A', 'E', 'I', 'O', 'U');         -- Zeichentyp
o
```

Für jeden **diskreten Typ** gibt es unter anderem zwei Attribute namens **'pos** und **'val**, die besonders bei **Aufzählungstypen** nützlich sind. Jedem Literal (bzw. Wert) eines Aufzählungstyps ist eine **Position** zugeordnet, und zwar dem **ersten** Literal (in der Typvereinbarung) die Position **0**, dem zweiten Literal die Position **1** u.s.w.. Z.B. hat der Wert **SCHWARZ** des Typs FARBE die Position **0** und **WEISS** hat die Position **5**. Das Attribut **FARBE'pos** ist eine Funktion, die man auf einen Wert des Typs **!FARBE** anwenden kann, und die die **Position** dieses Wertes liefert. **FARBE'pos(SCHWARZ)** ist also gleich **0** und **FARBE'pos(FARBE'succ(SCHWARZ))** ist gleich **1** u.s.w.. Das Attribut **FARBE'val** ist sozusagen "die Umkehrung" der Funktion **FARBE'pos**. Sie liefert, angewendet auf eine Ganzzahl (eines beliebigen Typs), den entsprechenden **Wert** des Typs **!FARBE** (falls es einen solchen Wert gibt). Z.B. ist **FARBE'val(5)** gleich **WEISS** und **FARBE'val(6)** löst die Ausnahme **constraint_error** aus, weil es keine Farbe mit der

Position 6 gibt. Ist **FARBE1** eine Variable vom Typ **!FARBE** und mit irgendeiner Farbe initialisiert, dann gilt: **FARBE'val(FARBE'pos(FARBE1))** ist gleich **FARBE1**. Und andersherum gilt: Ist **G1** irgendeine Ganzzahlvariable (von einem beliebigen Ganzzahltyp) mit einem Wert zwischen 0 und 5, dann ist **FARBE'pos(FARBE'val(G1))** gleich **G1**.

Für **Ganzzahltypen** sind die Attribute **'pos** und **'val** weniger interessant. Z.B. hat der Ausdruck OTTO'pos(17) den Wert 17 und OTTO'val(17) liefert den Wert 17 des Typs !OTTO.

12. Einfache und zusammengesetzte Anweisungen

Eine **Anweisung** (statement) ist ein Befehl des Programmierers an den Ausführer, die Inhalte bestimmter Wertebehälter zu verändern. Die Wertebehälter können **Variablen** sein oder **Ein-/Ausgabegeräte** wie z.B. Tastaturen, Bildschirme, Festplatten etc..

Die konzeptionell wichtigste Anweisung ("die anweisungste aller Anweisungen") ist die **Zuweisung**. Alle anderen Anweisungen kann man als **Varianten** ("verkappte Sonderformen") der Zuweisung verstehen.

Man unterscheidet **einfache Anweisungen** (simple statements) und **zusammengesetzte Anweisungen** (compound statements), je nachdem aus welchen syntaktischen Bestandteilen sie bestehen (siehe dazu auch (ARM 5)). Eine **einfache** Anweisung kann aus **Ausdrücken** und **Variablennamen** bestehen, die mit "syntaktischem Zucker verklebt sind" (siehe unten), darf aber keine anderen **Anweisungen** enthalten. Eine **zusammengesetzte** Anweisung muß mindestens eine andere Anweisung als Bestandteil enthalten ("muß unter anderem aus Anweisungen zusammengesetzt sein").

Beispiel 12.1.: Eine **einfache** Anweisung: Die **Zuweisung**

```
10 X := Y + 1;
```

besteht im wesentlichen aus dem **Variablennamen** X und dem **Ausdruck** Y + 1 (das Zuweisungszeichen ":=" und das abschließende Semikolon ";" sind nur syntaktischer Zucker). Allgemein gilt: Eine Zuweisung enthält keine anderen **Anweisungen** und ist somit eine **einfache Anweisung**. **o**

Beispiel 12.2.: Eine **zusammengesetzte** Anweisung:

Sogenannte **if-Anweisungen** werden im Abschnitt 12.3. genauer behandelt. Hier als Vorgriff eine einfache aber typische if-Anweisung:

```
20 if X < Y then
21    X := Y + 1;
22 else
23    Y := X - 1;
24 end if;
```

Sie besteht im wesentlichen aus dem **Ausdruck** X < Y, der **Anweisung** X := Y + 1; und der **Anweisung** Y := X - 1; (die restlichen Bestandteile sind syntaktischer Zucker). Allgemein gilt: Eine if-Anweisung ist unter anderem aus **Anweisungen** zusammengesetzt und gehört somit zu den **zusammengesetzten** Anweisungen. **o**

12.1. Einfache Anweisungen

Außer der Zuweisung ist auch jeder **Aufruf einer Prozedur** eine **einfache Anweisung**.

Aufgabe 12.1.1.: Die folgenden **einfachen** Anweisungen verändern (wie alle Anweisungen) die Inhalte bestimmter Wertebehälter. Nennen oder beschreiben Sie diese Wertebehälter.

```
10 ada.text_io.put(item => "Hallo, wie geht es?");
```

```
11 ada.text_io.get(item =>  NACH);
12 ada.text_io.new_line;
o
```

Die **null-Anweisung** (geschrieben: **null;**) ist eine einfache Anweisung, die "nichts bewirkt". Mit ihr kann der Programmierer deutlich machen, daß er an bestimmten Stellen eines Programms **absichtlich** keine anderen Anweisungen hingeschrieben hat und es **kein Versehen** ist, daß dort keine anderen Anweisungen stehen. Um solche "Versehen" möglichst zu vermeiden, gilt in Ada-Programmen folgende Regel: Überall da, wo Anweisungen stehen **dürfen** (z.B. im Anweisungsteil einer Prozedur), **muß** mindestens **eine** Anweisung stehen (gegebenenfalls die **null**-Anweisung).

Weitere einfache Anweisungen (**return**, **exit**, **raise** etc.) werden später behandelt.

12.2. Zusammengesetzte Anweisungen

Angenommen, eine Prozedur **P** besteht nur aus (Vereinbarungen und) **einfachen Anweisungen**. Dann wird bei jeder Ausführung von **P** jeder Befehl (d.h. jede Vereinbarung und jede Anweisung von P) genau **einmal** ausgeführt. Dies ist z.B. bei allen bisher behandelten Beispielprogrammen (HALLO_01, GANZT_01, AUFZT_01 etc.) der Fall.

Mit **zusammengesetzten Anweisungen** kann der Programmierer bewirken, daß bestimmte (in den zusammengesetzten Anweisungen enthaltene) Anweisungen **mehr als einmal** oder **weniger als einmal** (also nicht) ausgeführt werden. Zusammengesetzte Anweisungen werden auch als **Kontrollstrukturen** bezeichnet, weil der Programmierer damit kontrollieren kann, wie oft (die in den Kontrollstrukturen enthaltenen) Anweisungen ausgeführt werden (nullmal, einmal oder mehrmals).

Mit einer **if-Anweisung** oder einer **case-Anweisung** kann der Programmierer bewirken, daß bestimmte Anweisungen **weniger als einmal** (also **nicht**) ausgeführt werden. Mit **Schleifen** (loop-Anweisungen) kann er erreichen, daß bestimmte Anweisungen **mehr als einmal** ausgeführt werden. Diese zusammengesetzten Anweisungen werden in den folgenden Abschnitten genauer beschrieben. Die **block-Anweisung** gehört ebenfalls zu den zusammengesetzten Anweisungen, fällt aber ein bißchen aus dem Rahmen. Sie wird zuletzt (im Abschnitt 12.8.) behandelt.

12.3. Die if-Anweisung

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type OTTO is range -20_000..+30_000;
02 G1, G2, G3 : OTTO;
03 ALLES_OK   : boolean;
04 ZEICHEN1   : character;
```

Außerdem wird angenommen, daß den Variablen **G1**, **G2**, **G3**, **ALLES_OK** und **ZEICHEN1** irgendwelche Werte (ihres Untertyps) **zugewiesen** wurden.

Von der **if-Anweisung** gibt es (in Ada ganz ähnlich wie in vielen anderen Sprachen) mehrere **Varianten**. Die **if-then-Variante** bewirkt, daß eine bestimmte Folge von Anweisungen unter bestimmten Bedingungen **nicht** ausgeführt wird.

Beispiel 12.3.1.: Eine **if-then-Anweisung**:

```
10 if G1 < G2 then
11    G2 := 0;
12    ada.text_io.put(item => "G1 ist kleiner als G2!");
13 end if;
```

Die Anweisungsfolge zwischen **then** und **end if** (Zeile 11 bis 12) wird nur ausgeführt, wenn die Bedingung **G1 < G2** erfüllt ist. **o**

Die **if-then-else-Variante** bewirkt, daß von zwei Anweisungsfolgen eine **ausgeführt** und die andere **nicht ausgeführt** wird.

Beispiel 12.3.2.: Eine **if-then-else-Anweisung**:

```
20 if G1 < G2 then
21    G2 := 0;
22    ada.text_io.put(item => "G1 ist kleiner als G2!");
23 else
24    G3:= 1;
25 end if;
```

Die Anweisungsfolge zwischen **then** und **else** (Zeile 21 bis 22) wird auch als **then-Teil** bezeichnet und die Anweisungsfolge zwischen **else** und **end if** (Zeile 24) als **else-Teil**. Wenn die Bedingung **G1 < G2** erfüllt ist, wird der **then-Teil** ausgeführt und der **else-Teil** nicht ausgeführt. Ist die Bedingung **G1 < G2** nicht erfüllt, wird umgekehrt der **then-Teil** nicht ausgeführt und der **else-Teil** ausgeführt. **o**

In der **if-then-elsif-Varianten** ist die if-Anweisung aus **mehreren** Bedingungen und **mehreren** then-Teilen zusammengesetzt. Der **erste** then-Teil, dessen Bedingung erfüllt ist, wird ausgeführt, die anderen then-Teile werden **nicht** ausgeführt. Falls keine der Bedingungen erfüllt ist, wird keiner der then-Teile ausgeführt.

Beispiel 12.3.3.: Eine **if-then-elsif-Anweisung**:

```
30 if    G1 < G2 then
31    G2 := 0;
32    ada.text_io.put(item => "G1 ist kleiner als G2!");
33 elsif G1 = G2 then
34    G2 := 2 * G1;
35    G3:= 0;
36 elsif G3 > G1 + G2 then
37    ada.text_io.put(item => "Was ist los?");
38 end if;
```

Wenn die **erste** Bedingung (G1 < G2 in Zeile 30) erfüllt ist, wird nur der **erste** then-Teil (Zeile 31 bis 32) ausgeführt. Wenn die erste Bedingung **nicht** erfüllt ist, aber die **zweite** Bedingung (G1 = G2 in Zeile 33) erfüllt ist, wird nur der **zweite** then-Teil (Zeile 34 bis 35) ausgeführt. Wenn die ersten beiden Bedingungen nicht erfüllt sind, aber die **dritte** Bedingung (G3> G1 + G2 in Zeile 36) erfüllt ist, wird nur der **dritte** then-Teil (Zeile 37) ausgeführt. Wenn alle drei Bedingungen **nicht** erfüllt sind, wird **keiner** der then-Teile ausgeführt. **o**

Die if-Anweisung im vorigen Beispiel (12.3.3.) enthält zwei **elsif-Abschnitte** (Zeile 33 bis 35 und Zeile 36 bis 37). Im allgemeinen darf eine if-then-elsif-Anweisung **beliebig viele** elsif-Abschnitte enthalten.

In der **if-then-elsif-else-Varianten** enthält die if-Anweisung zusätzlich zu beliebig vielen **elsif-Abschnitten** auch noch einen **else-Teil**.

Beispiel 12.3.4.: Eine **if-then-elsif-else-Anweisung**:

```
40 if    G1 < G2 then
41    G2 := 0;
42    ada.text_io.put(item => "G1 ist kleiner als G2!");
43 elsif G1 = G2 then
44    G2 := 2 * G1;
45    G3:= 0;
46 elsif G3> G1 + G2 then
47    ada.text_io.put(item => "Was ist los?");
48 else
49    ada.text_io.put(item => "Fehler 17!");
50    ada.text_io.new_line;
51 end if;
```

Wie eine solche **if**-Anweisung ausgeführt wird, ergibt sich (hoffentlich) aus den vorigen beiden Beispielen. **o**

Die **Anweisungsfolgen** in einer if-Anweisung (hinter **then** bzw. hinter **else**) dürfen **beliebig lang**, aber **nicht leer** sein. Wenn der Programmierer will, daß der Ausführer an einer solchen Stelle "nichts macht", kann er z.B. die **null-Anweisung** (**null;**) hinschreiben. Noch besser ist es meistens, die **if**-Anweisung so umzuformulieren, daß der entsprechende **then**- oder **else-Teil** wegfällt, wie das nächste Beispiel zeigt:

Beispiel 12.3.5.: Zwei if-Anweisungen mit gleicher Bedeutung:

if-Anweisung-1

```
60 if G1 = G2 then
61    null;
62 else
63   G3 := G3 + 1;
64 end if;
```

if-Anweisung-2

```
70 if G1 /= G2 then
71    G3 := G3 + 1;
72 end if;
```

Welche ist leichter zu lesen und zu verstehen? **o**

Im Beispielprogramm **IFANW_01** kann man verschiedene Varianten der if-Anweisungen im Rahmen eines vollständigen Programms ansehen und "ausprobieren".

Aufgabe 12.3.1.: Jede **if-Anweisung** enthält **Anweisungsfolgen** (mindestens einen **then-Teil**, eventuell mehrere **then-Teile** und eventuell einen **else-Teil**). Betrachten Sie die folgenden drei Aussagen:

1. Mindestens eine Anweisungsfolge wird ausgeführt.
2. Höchstens eine Anweisungsfolge wird ausgeführt.
3. Genau eine Anweisungsfolge wird ausgeführt.

Welche dieser Aussagen trifft auf die **if-then-Variante** der if-Anweisung zu? Ebenso für die **if-then-else**-Variante, die **if-then-elsif**-Variante und die **if-then-elsif-else**-Variante. **o**

Wenn eine der Anweisungsfolgen einer if-Anweisung eine if-Anweisung enthält, dann bezeichnet man diese als (einfach) **geschachtelte** if-Anweisung. Wenn eine einfach geschachtelte if-Anweisung ihrerseits eine weitere if-Anweisung enthält, dann bezeichnet man diese als **doppelt geschachtelt**, etc..

Beispiel 12.3.6.: Zwei einfach geschachtelte if-Anweisungen:

```
80 if G1 < G2 then
81    if G2 < G3 then          -- Eine einfach geschachtelte if-Anweisung
82       ada.text_io.put(item => "Sortierung ist o.k.!");
83    else
84       ada.text_io.put(item => "G2 >= G3");
85    end if;
86 else
87    ada.text_io.put(item => "G1 >= G2");
88    if G2 < G3 then          -- Noch eine einfach geschachtelte if-Anweisung
89       ada.text_io.put(item => "G2 und G3 sind o.k.");
90    else
91       ada.text_io.put(item => "G2 >= G3");
92    end if;
93 end if;
o
```

Viele Menschen (darunter der Autor dieses Skripts) haben **Mühe**, einfach geschachtelte if-Anweisungen zu lesen und **sehr große Mühe**, doppelt geschachtelte if-Anweisungen zu verstehen. Der Programmierer sollte if-Anweisungen deshalb **möglichst wenig** schachteln. Das Beispielprogramm **IFANW_02** soll zeigen, wie schwer doppelt geschachtelte if-Anweisungen zu verstehen sind. Die folgenden beiden Aufgabe sollen dazu anregen, durch scharfes Nachdenken geschachtelte if-Anweisungen zu vermeiden. Im Beispiel 12.3.7. (weiter unten) kommt eine (vermeidbare) einfach geschachtelte if-Anweisung vor.

Aufgabe 12.3.2.: Schreiben Sie ein Programm namens **IFANW_03**, welches drei Ganzzahlen in drei Variable namens **G1**, **G2** und **G3** einliest und die größte der drei Zahlen ausgibt. Benützen Sie zur Lösung nur **ungeschachtelte** if-Anweisungen. **o**

Aufgabe 12.3.3.: Schreiben Sie ein Programm namens **IFANW_04**, welches fünf Ganzzahlen in fünf Variable namens **G1**, **G2**, **G3**, **G4** und **G5** einliest und die größte der fünf Zahlen ausgibt. Diese Aufgabe kann man praktisch nur dann lösen, wenn man die vorige Aufgabe **ohne** geschachtelte if-Anweisungen gelöst hat. **o**

Man beachte, daß das reservierte Wort **elsif** nur **ein** 'e' enthält (und nicht etwa **elseif** geschrieben wird). Man könnte statt **elsif** auch **else if** (mit mindestens einem Trennzeichen zwischen **else** und **if**) schreiben, hätte damit aber eine geschachtelte if-Anweisung und müßte für jedes **if** ein dazugehöriges **end if** notieren (und durch eine entsprechende **Einrückung** deutlich machen, welches **end if** zu welchem **if** gehört). Wenn möglich sollte man deshalb anstelle von **else if** einfach **elsif** verwenden.

Beispiel 12.3.7.: Zwei **if**-Anweisungen mit gleicher Bedeutung:

if-Anweisung-1 **if-Anweisung-2**

```
10 if    G1 = 0 then
11    G3 := 1;
12 elsif G1 = G2 then
13    G3 := 2;
14 elsif G1 < G2 then
15    G3 := 3;
16 else
17    G3 := 4;
18 end if;
```

```
20 if G1 = 0 then
21    G3 := 1;
22 else
23    if G1 = G2 then
24       G3 := 2;
25    else
26       if G1 < G2 then
27          G3 := 3;
28       else
29          G3 := 4;
30       end if;
31    end if;
32 end if;
```

Welche Version finden Sie leichter zu lesen und zu verstehen? **o**

Als **Bedingung** darf man in einer if-Anweisung (zwischen **if** und **then** bzw. **elsif** und **then**) einen beliebigen **booleschen Ausdruck** angeben. Dieser Ausdruck kann z.B. aus mehreren Teilausdrücken zusammengesetzt sein oder aber nur aus dem **Namen einer booleschen Variablen** bestehen, wie das folgende Beispiel deutlich machen soll:

Beispiel 12.3.8.: Zwei komplizierte und eine besonders einfache Bedingung:

```
40 if (G1 < G2 and G2 < G3) or not ALLES_OK                    then ...
41 if ZEICHEN1 in 'A' .. 'Z' or ZEICHEN1 in 'a' .. 'z'         then ...
42 if ALLES_OK                                                 then ...
```

In Zeile 42 könnte man statt einfach **ALLES_OK** auch komplizierter **ALLES_OK = true** schreiben, aber das wäre nicht besonders schön. **o**

Aufgabe 12.3.4.: Schreiben Sie ein Programm namens **IFANW_06**, welches drei Ganzzahlen einliest und die größere der drei Zahlen wieder ausgibt. Benützen Sie in diesem Programm aber keine zusammengesetzten Befehle (insbesondere **keine if-Anweisungen**). Erinnern Sie sich noch an die **Attribute** eines Ganzzahluntertyps, die im Abschnitt 9.2. eingeführt wurden? **o**

Aufgabe 12.3.5.: Übersetzen Sie die **if-Anweisungen** in den **Beispielen 12.3.1.** bis **12.3.3.** ins Deutsche.

Zusammenfassung 12.3.:

- Von der if-Anweisung gibt es verschiedene Varianten (**if-then**, **if-then-else**, **if-then-elsif** und **if-then-elsif-else**).
- Man sollte if-Anweisungen **möglichst wenig schachteln**.
- Manchmal kann man if-Anweisungen durch noch leichter lesbare Konstrukte ersetzen, z.B. durch Funktionsaufrufe (siehe Aufgabe 12.3.4).

12.4. Die case-Anweisung

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type OTTO  is range -20_000 .. +30_000;
02 type FARBE is (SCHWARZ, ROT, GRUEN, BLAU, GELB, WEISS);
03 G1, G2, G3     : OTTO;
```

```
04 FARBE1, FARBE2 : FARBE;
```

Außerdem wird angenommen, daß den Variablen G1, G2, G3, FARBE1 und FARBE2 irgendwelche Werte (ihres Untertyps) zugewiesen wurden.

Ein **case-Anweisung** besteht hauptsächlich aus einem **Ausdruck** und beliebig vielen **Anweisungsfolgen**. Abhängig vom Wert des Ausdrucks wird **genau eine** der Anweisungsfolgen ausgewählt und ausgeführt (alle anderen Anweisungsfolgen werden **nicht** ausgeführt).

Beispiel 12.4.1.: Eine case-Anweisung:

```
10 case FARBE1 is
11    when SCHWARZ =>
12       ada.text_io.put(item => "Zu dunkel!");
13       FARBE1 := GELB;
14    when ROT | GRUEN =>
15       ada.text_io.put(item => "Nicht modisch genug!");
16    when BLAU..WEISS =>
17       ada.text_io.put(item => "Schon besser!");
18       G1 := G2 + G3;
19 end case;
```

Wenn der Ausdruck **FARBE1** (zwischen **case** und **is** in Zeile 10) den Wert **SCHWARZ** hat, wird nur die **erste** Anweisungsfolge (Zeile 12 bis 13) ausgeführt.
Wenn FARBE1 den Wert **ROT** oder **GRUEN** hat, wird nur die **zweite** Anweisungsfolge (Zeile 15) ausgeführt.
Wenn FARBE1 einen Wert im Bereich **BLAU** bis **WEISS** hat, wird nur die **dritte** Anweisungsfolge (Zeile 17 bis 18) ausgeführt. **o**

Die **case-Anweisung** in diesem Beispiel besteht aus dem **Ausdruck** FARBE1 (zwischen **case** und **is** in Zeile 10) und drei **Alternativen**, die jeweils mit **when** beginnen (Zeile 11 bis 13, Zeile 14 bis 15 und Zeile 16 bis 18). Jede Alternative besteht aus einer **Wahlliste** (zwischen **when** und dem Pfeil =>) und einer **Anweisungsfolge** (zwischen dem Pfeil => und dem Ende der Alternativen). Die **Wahlliste** der ersten Alternativen (siehe Zeile 11) enthält als einzigen **Wahleintrag** das Literal SCHWARZ. Die **Wahlliste** der dritten Alternativen (siehe Zeile 16) enthält als einzigen **Wahleintrag** die **Bereichsangabe** BLAU..WEISS. Die **Wahlliste** der zweiten Alternativen (siehe Zeile 14) enthält **zwei** Wahleinträge: das Literal ROT und das Literal GRUEN.

Eine **Wahlliste** kann im Prinzip beliebig viele (durch senkrechte Striche "|" voneinander getrennte) **Wahleinträge** enthalten. Jeder Wahl-Eintrag bezeichnet entweder **einen einzigen Wert** (z.B. SCHWARZ) oder einen ganzen **Bereich** (z.B. BLAU..WEISS). Den senkrechten Strich "|" liest man als "**oder**", man sollte ihn aber nicht mit der logischen Operation **or** verwechseln, mit der er höchstens sehr entfernt ("um viele Ecken herum") verwandt ist.

In Ada müssen **case**-Anweisungen **vier wichtige Regeln** erfüllen:

Regel 1: Der **Ausdruck** der case-Anweisung (der zwischen **case** und **is** steht) muß zu einem **diskreten Typ** (d.h. zu einem Ganzzahltyp oder zu einem Aufzählungstyp) gehören. Er darf z.B. nicht zu einem Bruchzahltyp wie **float** gehören und auch nicht zu einem Reihungstyp wie **string** (Bruchzahltypen und Reihungstypen werden später behandelt).

Regel 2: Alle Wahleinträge müssen **statisch** sein. Das heißt: Der Ausführer muß die Werte der Wahleinträge schon dann ermitteln können, wenn der Programmierer ihm das Programm **übergibt**

("zur Compilezeit"), und nicht erst, wenn er es **ausführt** ("zur Laufzeit"). Variablennamen (z.B. FARBE2) sind deshalb als Wahleintrag grundsätzlich **nicht** erlaubt. Erlaubt sind aber z.B. **Attribute** wie **FARBE'first** oder **FARBE'last** und sogar Ausdrücke wie **FARBE'succ(BLAU)** oder **FARBE'pred(FARBE'pred(FARBE'last))**.

Regel 3: Eine **case**-Anweisung muß **vollständig** sein. Das hat mit dem **Untertyp** des **Ausdrucks** (der zwischen **case** und **is** steht) zu tun. **Jeder Wert dieses Untertyps** muß von einem Wahleintrag (in einer Auswahlliste einer Alternativen) beschrieben werden. Praktisch bedeutet das für obiges Beispiel: Für jede Farbe muß es eine entsprechende Alternative geben. Wenn bei bestimmten Werten "nichts gemacht" werden soll, kann man das durch die **null-Anweisung** ausdrücken.

Regel 4: Eine **case**-Anweisung muß **eindeutig** sein. Das bedeutet: **Kein** Wert des betreffenden Untertyps darf von **mehr als einem** Wahleintrag beschrieben werden. Praktisch bedeutet das für obiges Beispiel: Für keine Farbe darf es mehr als eine Alternative geben.

Aufgabe 12.4.1.: Die folgenden drei case-Anweisungen sind **fehlerhaft**. Warum?

```
20 case FARBE1 is
21    when ROT | FARBE'last =>
22       G1 := 1;
23    when FARBE'first .. GELB =>
24       null;
25 end case;
26
27 case FARBE1 is
28    when SCHWARZ..ROT | BLAU..GELB =>
29       null;
30    when WEISS =>
31       G2 := 2;
32 end case;
33
34 case FARBE1 is
35    when FARBE'first..FARBE2 =>
36       G3 := 1;
37    when FARBE'succ(FARBE2)..FARBE'last =>
38       G3 := 2;
39 end case;
o
```

Manchmal ist die **Vollständigkeit** einer case-Anweisung schwer zu erreichen. Für diese Fälle gibt es als Erleichterung die **others-Alternative**. Sie muß die **letzte** Alternative sein und darf als **einzigen** Wahleintrag in ihrer Wahlliste nur das Wort **others** ("irgendein anderer Wert") haben.

Beispiel 12.4.2.: Eine case-Anweisung mit **others-Alternative**:

```
40 case G1 is
41    when OTTO'first..-10_000 | -5_000 | -2_500 =>
42       ada.text_io.put(item => "Ziemlich negativ!");
43    when -100..-20 | +20..+100 =>
44       ada.text_io.put(item => "Kleine Fische!");
45    when 2 | 3 | 5 | 7 | 13 | 17 =>
46       ada.text_io.put(item => "Kleine Primzahlen!");
47    when others =>
48       ada.text_io.put(item => "Wieso gerade diese Zahl?");
49 end case;
```

In diesem Beispiel wäre es ziemlich aufwendig, den "pauschalen" Wahleintrag **others** ("den Resteverwerter") durch "spezifische" Wahleinträge zu ersetzen. **o**

Als **Ausdruck** wird in einer case-Anweisung (zwischen **case** und **is**) in aller Regel nur ein **Variablenname** angegeben (z.B. FARBE1 oder G1 etc.). Falls man doch einmal einen komplizierteren, **zusammengesetzten Ausdruck** angibt, muß man folgende **Feinheit** beachten: Der Ausdruck gehört nicht unbedingt zum selben **Untertyp**, wie die Variablen, die in ihm vorkommen. Z.B. gehört die Variable **G1** zum Untertyp **OTTO**, der Ausdruck **G1 + 1** gehört aber zum (umfangreicheren) Untertyp **OTTO'base**. Das hängt damit zusammen, daß man mit der Operation "+" alle Werte des **Typs** !OTTO addieren kann, und nicht nur die Werte irgendeines **Untertyps** von !OTTO (z.B. die Werte des Untertyps **OTTO**). Das folgende Beispiel soll diesen seltenen Fehlerfall illustrieren:

Beispiel 12.4.3.: Eine **unvollständige** case-Anweisung:

```
50 case G1 + 1 is
51    when OTTO'first..0 =>
52       ada.text_io.put(item => "Wert ist negativ oder null!");
53    when 1..OTTO'last  =>
54       ada.text_io.put(item => "Wert ist positiv!");
55 end case;
```

Der Ausdruck **G1 + 1** gehört zum Untertyp **OTTO'base**. Welche Werte zum Typ **!OTTO** und damit zum Untertyp **OTTO'base** gehören, bestimmt der **Ausführer**, nicht der **Programmierer** (aus Gründen der Effizienz). Wenn zum Untertyp **OTTO'base** mehr Werte gehören, als zum Untertyp **OTTO**, ist die case-Anweisung in diesem Beispiel **unvollständig**. Mit einer **others-Alternativen** könnte man sie aber leicht vervollständigen. Eine andere Lösung besteht darin, den Wert des Ausdrucks einer Variablen zuzuweisen, z.B. **G2 := G1 + 1;** und dann **case G2 is ...** zu schreiben. **o**

Die **case-Anweisung** kann man als einen **Spezialfall** der **if-then-elsif-else-Anweisung** verstehen. Der wesentliche Unterschied besteht darin, daß man in einer if-Anweisung **mehrere Bedingungen** angegeben kann (hinter **if** und hinter jedem **elsif**), die nichts miteinander zu tun zu haben brauchen. Alle diese Bedingungen können aber Einfluß darauf haben, welche Anweisungsfolge ausgeführt wird. Dagegen erfolgt bei der case-Anweisung die Auswahl einer Alternativen anhand **eines einzigen Ausdrucks**. Praktisch bedeutet das: Man kann jede **case-Anweisung** relativ leicht und nach einem festen Schema in eine entsprechende **if-then-elsif-else-Anweisung** übersetzen. Eine Übersetzung in der umgekehrten Richtung (von **if** nach **case**) ist dagegegen nicht so einfach und allgemein möglich.

Das Beispielprogramm **CASEA_01** zeigt eine **case**-Anweisung im Rahmen eines vollständigen kleinen Programms.

Aufgabe 12.4.2.: Übersetzen Sie die **case-Anweisung** im Beispiel 12.4.1. in eine entsprechende **if-Anweisung**. **o**

Aufgabe 12.4.3.: Schreiben Sie ein Programm namens **CASEA_02**, welches ein einzelnes Zeichen einliest, untersucht, ob es sich um einen kleinen Vokal (a, e, i, o, u), um einen großen Vokal (A, E, I, O, U), um einen kleinen Konsonant, um einen großen Konsonant oder um sonst ein Zeichen handelt und eine entsprechende Meldung (z.B. "Das Zeichen ist ein kleiner Konsonant") ausgibt. **o**

Aufgabe 12.4.4.: Übersetzen Sie die **case-Anweisungen** in den **Beispielen 12.4.1.** und **12.4.2.** ins Deutsche. **o**

Zusammenfassung 12.4.:
- Der Ausdruck zwischen **case** und **is** muß zu einem **diskreten Typ** gehören.
- Eine case-Anweisung muß **vollständig** und **eindeutig** sein.
- Die **Wahleinträge** in den Wahllisten müssen **statisch** sein
- Manche Probleme lassen sich mit einer case-Anweisung **eleganter** lösen, als mit einer entsprechenden if-Anweisung.

12.5. Bedingte Schleifen

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type    OTTO     is range -20_000 .. +30_000;
02 type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, GELB, WEISS);
03 package OTTO_EA  is new ada.text_io.integer_io    ( num => OTTO);
04 package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
05 G1, G2, G3     : OTTO;
06 SUMME          : OTTO;
07 FARBE1, FARBE2 : FARBE;
```

Außerdem wird angenommen, daß den Variablen **G1**, **G2**, **G3**, **FARBE1** und **FARBE2** geeignete Werte (ihres Untertyps) zugewiesen wurden.

Jede **Schleifenanweisung** besteht unter anderem aus einer **Anweisungsfolge**, die man auch als den **Rumpf** der Schleife bezeichnet. **Eine** Ausführung der Schleife bewirkt, daß ihr Rumpf **mehrmals** (in Sonderfällen auch nur **einmal** oder sogar **nullmal**, d.h. nicht) ausgeführt wird. Manchmal besteht der Rumpf einer Schleife auch aus **mehreren Anweisungsfolgen** (siehe unten).

In Ada ist der **Rumpf** einer Schleife immer in die reservierten Wörter **loop** und **end loop** eingeschlossen.

Beispiel 12.5.1.: Eine **loop-exit-Schleife**:

```
10 loop
11    ada.text_io.put(item => "Eine Ganzzahl (0 zum Beenden)? ");
12    OTTO_EA.get(item => G1);
13 exit when G1 = 0;
14    SUMME := SUMME + G1;
15    ada.text_io.put(item => "Ihre Ganzzahl wurde zur Summe addiert!");
16 end loop;
```

Diese Schleife besteht im wesentlichen aus dem **Ausdruck** G1 = 0 (in Zeile 13) und zwei **Folgen von Anweisungen**. Die **erste** Anweisungsfolge (in Zeile 11 bis 12) steht zwischen **loop** und dem Anfang der **exit**-Anweisung in Zeile 13, die **zweite** (in Zeile 14 bis 15) beginnt hinter der **exit**-Anweisung und geht bis **end loop**. Die Einrückung soll es dem Leser leicht machen, diese beiden Anweisungsfolgen zu erkennen.

Die Anweisung in diesem Beispiel kann man etwa so ins Deutsche übersetzen:
Um die **Schleife** (in Zeile 10 bis 16) **einmal** auszuführen, sollst du die folgenden Schritte unternehmen:

1. Führe die **erste Anweisungsfolge** (in Zeile 11 bis 12) aus.
2. Berechne den Wert des booleschen Ausdrucks **G1 = 0**.
3. Wenn dieser Wert gleich **true** ist, dann hast du die Schleife fertig ausgeführt (mache hinter **end loop** weiter).
4. Führe sonst (wenn der Wert des Ausdrucks **false** ist) die **zweite Anweisungsfolge** (in Zeile 14 bis 15) aus und
5. Führe die Schleife (in Zeile 10 bis 16) erneut aus. Wie? Siehe oben: "Um die Schleife (in Zeile 10 bis 16) einmal auszuführen, sollst du ..."). **o**

Die Schleife im vorhergehenden Beispiel veranlaßt den Ausführer, solange Ganzzahlen von der aktuellen Eingabe (Tastatur) einzulesen und zur **SUMME** zu addieren, bis der Benutzer die Zahl 0 eingibt (oder bis eine Ausnahme wie **data_error** oder **constraint_error** ausgelöst wird).

Aufgabe 12.5.1.: Wie oft wird (im vorigen Beispiel 12.5.1.) die **erste Anweisungsfolge** (in Zeile 11 bis 12) **mindestens** ausgeführt? Wie oft wird die zweite Anweisungsfolge (in Zeile 14 bis 15) **mindestens** ausgeführt? **o**

Die **exit-Anweisung** ist eine **einfache Anweisung** (siehe Abschnitt 12.1.) und darf nur im Rumpf einer Schleife (zwischen **loop** und **end loop**) verwendet werden. Ein Schleifenrumpf darf beliebig viele **exit**-Anweisungen enthalten:

Beispiel 12.5.2. Eine loop-exit-Schleife mit **zwei** exit-Anweisungen:

```
20 loop
21    PAUL_EA.get(item => G1);
22 exit when G1 = 0;
23    PAUL_EA.get(item => G2);
24 exit when G2 = 0;
25    PAUL_EA.put(item => G1 + G2);
26 end loop;
```

Diese Schleife bewirkt, daß der Ausführer wiederholt zwei Ganzzahlen einliest und ihre Summe ausgibt, bis der Benutzer eine 0 eingibt. **o**

Aufgabe 12.5.2.: Was sind die wesentlichen Bestandteile (Ausdrücke, Anweisungsfolgen) der Schleife im vorigen Beispiel 12.5.2.? Übersetzen Sie diese Schleife ins Deutsche. **o**

Aufgabe 12.5.3.: Betrachten Sie die folgenden Anweisungen:

```
30 OTTO_EA.get(item => G1);
31 G2 := 0;
32 loop
33 exit when G1 <= 1; -- "<=" bedeutet "kleiner oder gleich"
34    G1 := G1 / 2;
35    G2 := G2 + 1;
36 end loop;
```

1. Welcher Zusammenhang besteht zwischen dem Wert, der anfangs in die Variable **G1** eingelesen wird (Zeile 30) und dem Wert, der am Schluß (wenn die Schleife fertig ausgeführt ist) in **G2** steht? Führen Sie die Anweisungen (Zeile 30 bis 36) mehrmals mit Papier und Bleistift aus, und füllen Sie die folgende Tabelle aus:

Anfangs in G1	Zum Schluß in G2	Anfangs in G1	Zum Schluß in G2	Anfangs in G1	Zum Schluß in G2

0		5		-1	
1		10		-137	
2		50			
3		64			
4		128			

2. Was sind die wesentlichen Bestandteile (Ausdrücke, Anweisungsfolgen) der Schleifenanweisung (in Zeile 32 bis 36)?
3. Übersetzen Sie die Schleifenanweisung ins Deutsche. **o**

Wenn eine Schleife nur **eine** exit-Anweisung enthält und diese im Rumpf **vor** allen anderen Anweisungen steht, dann bezeichnet man die Schleife auch als **vorprüfende** (oder: kopfgesteuerte) Schleife (englisch: pre check loop). Steht die einzige exit-Anweisung im Rumpf **nach** allen anderen Anweisungen, spricht man von einer **nachprüfenden** (oder: fußgesteuerten) Schleife (englisch: post check loop). Die Schleife in der vorigen Aufgabe 12.5.3. ist eine vorprüfende Schleife.

Beispiel 12.5.4.: Eine nachprüfende Schleife:

```
40 OTTO_EA.get(item => G1);
41 G2 := 0;
42 loop
43    G1 := G1 / 2;
44    G2 := G2 + 1;
45 exit when G1 = 0;
46 end loop;
o
```

Aufgabe 12.5.4.: Betrachten Sie die Anweisungen im vorigen Beispiel.

1. Welcher Zusammenhang besteht zwischen dem Wert, der anfangs in die Variable **G1** eingelesen wird (Zeile 40) und dem Wert, der am Schluß (wenn die Schleife fertig ausgeführt ist) in **G2** steht? Führen Sie die Anweisungen (Zeile 40 bis 46) mehrmals mit Papier und Bleistift aus, und füllen Sie die folgende Tabelle aus:

Anfangs in G1	**Zum Schluß in G2**	**Anfangs in G1**	**Zum Schluß in G2**	**Anfangs in G1**	**Zum Schluß in G2**
0		5		-1	
1		10		-137	
2		50			
3		64			
4		128			

2. Was sind die wesentlichen Bestandteile (Ausdrücke, Anweisungsfolgen) der Schleifenanweisung (in Zeile 42 bis 46)?
3. Übersetzen Sie die Schleifenanweisung ins Deutsche. **o**

Für **vorprüfende** Schleifen gibt es in Ada eine zusätzliche Notation, nämlich die Notation als **vorprüfende while-Schleife**:

Beispiel 12.5.5.: Eine **while-Schleife**:

```
50 OTTO_EA.get(item => G1);
51 G2 := 0;
52 while G1 > 1 loop
53    G1 := G1 / 2;
54    G2 := G2 + 1;
55 end loop;
```

Diese **while-Schleife** (Zeile 52 bis 55) hat genau die gleiche Bedeutung, wie die **loop-exit-Schleife** in Aufgabe 12.5.3.. Man beachte aber, daß die Bedingung **G1 > 1** hinter **while** die **Verneinung** der Bedingung **G1 <= 1** hinter **exit when** ist. **o**

Für LeserInnen, die mit anderen Programmiersprachen vertraut sind: Insgesamt gibt es **vier** Spezialfälle von Schleifen, für die in der einen oder anderen Programmiersprache (z.B. in C oder in Pascal etc.) spezielle Notationen zur Verfügung gestellt werden. Der "syntaktische Zucker" dieser Notationen ist von Sprache zu Sprache verschieden und nicht besonders wichtig. Worauf es ankommt, ist die **Struktur** der einzelnen Schleifenarten. Diese Struktur wird hier durch die Worte **while**, **until**, **loop** und **end loop** kurz dargestellt ("charakterisiert"):

Bezeichnung der Schleifenart	**Charakterisierung**	**Anmerkung**
Vorprüfende while-Schleifen	while...loop...end loop	gibt es in Ada
Nachprüfende while-Schleifen	loop...while...end loop	gibt es nicht in Ada
Vorprüfende until-Schleifen	until...loop...end loop	gibt es nicht in Ada
Nachprüfende until-Schleifen	loop...until...end loop	gibt es nicht in Ada

Diese vier **speziellen Formen von Schleifen** kann man relativ leicht mit **loop-exit-Schleifen** nachbilden und mit **loop-exit-Schleifen** kann man einige Probleme **eleganter** lösen als mit den speziellen Schleifenformen. Im ersten Beispiel in diesem Abschnitt (Beispiel 12.5.1.) wird ein solches Problem gelöst ("Zahlen einlesen, bis der Benutzer nicht mehr will").

Alle bisher besprochenen Schleifen (loop-exit-Schleifen und die speziellen Schleifenformen) werden gemeinsam auch als **bedingte Schleifen** bezeichnet, weil es von einer **Bedingung** (d.h. von einem **booleschen Ausdruck**) abhängt, ob der Rumpf der Schleife noch einmal ausgeführt wird oder nicht. Bei einer **while**-Schleife muß die Bedingung **erfüllt** sein (d.h. den Wert **true** liefern), damit der Rumpf erneut ausgeführt wird. Bei einer **loop-exit**-Schleife (und ebenso bei einer **until**-Schleife) muß die Bedingung **nicht-erfüllt** sein (d.h. den Wert **false** liefern), damit der Ausführer noch "weiter in der Schleife drinbleibt".

Bedingte Schleifen sind sehr mächtige Instrumente, gleichzeitig aber auch mit einer grundlegenden **Gefahr** verbunden: Sie können zu **Endlosschleifen** werden. Ein bißchen vereinfacht gesagt gilt: Bei einer Endlosschleife nimmt die kontrollierende **Bedingung** (der boolesche Ausdruck hinter **while** bzw. hinter **exit when**) **nie** den Wert an, der für eine **Beendigung** der Schleife nötig wäre (**false** bei einer while-Schleife bzw. **true** bei einer loop-exit-Schleife).

Beispiel 12.5.6.: Eine relativ einfache, aber manchmal **endlose** Schleife:

```
60 OTTO_EA.get(item => G1);
61 while G1 < 100 loop
62    G1 := 2 * G1;
63 end loop;
```

Wenn der Benutzer eine **positive** Zahl eingibt (siehe Zeile 60), wird der Rumpf der Schleife (Zeile 62) höchstens siebenmal ausgeführt (probieren Sie es mit Papier und Bleistift). Wenn der Benutzer eine **negative** Zahl eingibt, wird die Schleife nach ein paar Rumpfausführungen durch eine Ausnahme **constraint_error** abgebrochen. Aber wenn der Benutzer eine **0** eingibt, dann wird die Schleife zur **Endlosschleife**. **o**

In aller Regel ist eine **Endlosschleife** in einem Programm ein schwerwiegender (manchmal ein katastrophaler) **Fehler** und der Programmierer muß durch sorgfältiges Programmieren und mindestens ebenso sorgfältiges Testen seiner Programme so gut es geht sicherstellen, daß keine **bedingte Schleife** in seinem Programm zu einer **Endlosschleife** werden kann. Siehe dazu auch den nächsten Abschnitt.

Zusammenfassung 12.5.:

- In Ada gibt es **zwei Notationen** für bedingte Schleifen: Die allgemeine **loop-exit-Schleife** und die spezielle **while-Schleife**.
- Mit **loop-exit-Schleifen** kann man auch andere Schleifenformen nachbilden, für die es in Ada keine speziellen Notationen gibt (z.B. vor- und nachprüfende **until**-Schleifen).
- **exit**-Anweisungen dürfen nur im **Rumpf einer Schleife** vorkommen
- Bedingte Schleifen können zu **Endlosschleifen** werden. Der Programmierer muß aufpassen.

12.6. Zählschleifen

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type    OTTO     is range -20_000 .. +30_000; -- OTTO ist auch negativ!
02 type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, GELB, WEISS);
03 package OTTO_EA  is new ada.text_io.integer_io    ( num => OTTO);
04 package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
05 G1, G2, G3     : OTTO;
06 FARBE1, FARBE2 : FARBE;
07 ZEICHEN1       : character;
```

Außerdem wird angenommen, daß den Variablen **G1**, **G2**, **G3**, **FARBE1** und **FARBE2** geeignete Werte (ihres Untertyps) zugewiesen wurden.

Weil **bedingte Schleifen** nicht nur sehr mächtig und nützlich, sondern auch gefährlich sind, hat man in Ada noch eine radikal andere Art von Schleifen eingeführt: **Zählschleifen**. Bei einer Zählschleife garantiert der **Ada-Ausführer** (und nicht der **Programmierer**), daß der Rumpf nur endlich oft ausgeführt wird. Selbst wenn der Programmierer beim Schreiben des Programms **Fehler** macht und der Benutzer bei der Ausführung des Programms **falsche Daten** eingibt, kann eine Zählschleife nicht zu einer **Endlosschleife** werden. Nur wenn der **Ada-Ausführer** fehlerhaft ist und die Regeln der Sprache Ada nicht korrekt einhält, kann eine Zählschleife "durchdrehen". Dieser Fall ist nicht grundsätzlich ausgeschlossen, aber relativ zur Häufigkeit von Programmiererfehlern sehr selten.

Im wesentlichen funktioniert eine **Zählschleife** so: Wenn der Ausführer mit der Ausführer der Schleife beginnt, berechnet er zuerst einmal (aus Angaben des Programmierers am Anfang der Schleife) **eine natürliche Zahl n**. Anschließend führt er den Rumpf der Schleife aus, und zwar normalerweise **genau** n-mal. Aufgrund von **exit**-Anweisungen (im Rumpf der Zählschleife) und ähnlichen Dingen kann es passieren, daß der Rumpf **weniger** als n-mal ausgeführt wird. Aber auf keinen Fall wird er **mehr** als n-mal ausgeführt. Man sagt auch: Die Anzahl der Rumpfausführungen ist durch die Zahl n **nach oben begrenzt**.

Beispiel 12.6.1.: Eine Zählschleife:

```
70 for F in FARBE loop
71    FARBE_EA.put(item => F);
72 end loop;
```

Im wesentlichen besteht diese Schleife aus der **Spezifikation eines Schleifenparameters** (**F in FARBE** in Zeile 70) und dem **Rumpf** der Schleife (Zeile 71). Man kann diese Schleife etwa so ins Deutsche übersetzen:
1. Erzeuge eine neue Variable namens F vom Untertyp FARBE.
2. Weise dieser Variablen der Reihe nach jeden Wert des Untertyps FARBE zu (in aufsteigender Reihenfolge, erst SCHWARZ, dann ROT, dann GRUEN, dann BLAU, dann GELB und schließlich WEISS) und führe nach jeder solchen Zuweisung den **Rumpf** der Schleife (einmal) aus.
3. Zerstöre die Variable F wieder. **o**

Im Beispiel wird der Rumpf der Zählschleife genau **sechsmal** ausgeführt, weil zum Untertyp FARBE **sechs** Werte gehören. Die Namen der sechs Farben werden zur aktuellen Ausgabe (zum Bildschirm) ausgegeben.

Die **Variable** namens **F** (der **Schleifenparameter** der Zählschleife) wird also zu Beginn einer Ausführung der Schleife erzeugt und vor Beendigung dieser Schleifenausführung wieder zerstört. In Befehlen **außerhalb** des Schleifenrumpfes darf man den Schleifenparameter F **nicht** erwähnen, denn während diese Befehle ausgeführt werden, existiert der Schleifenparameter noch nicht oder nicht mehr. Sollte der Programmierer "weiter oben im Programm" schon eine Variable namens **F** vereinbart haben, dann stört diese nicht weiter, hat aber nichts mit dem Schleifenparameter namens **F** zu tun. Solche Situationen, in denen zur gleichen Zeit zwei verschiedene Variablen mit gleichem Namen existieren, sollte man aber möglichst vermeiden, weil sie den Kollegen das Lesen des Programms möglicherweise ein bißchen erschweren.

Der Schleifenparameter F ist nur **für den Ausführer** eine **Variable** (der er der Reihe nach die Werte SCHWARZ, ROT, ..., WEISS zuweist). **Für den Programmierer** ist F eine **Konstante**. D.h. der Programmierer darf dem Ausführer **nicht** befehlen, den Wert von F zu verändern (etwa durch Zuweisungsanweisungen oder Aufrufe der Prozedur **FARBE_EA.get** oder ähnliche Befehle). Falls der Programmierer gegen diese Regel verstoßen sollte (z.B. mit einer Zuweisung wie **F := FARBE'pred(F)**), dann meldet der Ausführer schon bei der Übergabe des Programms ("zur Compilezeit") einen entsprechenden Fehler. Der Programmierer kann mit dem **Schleifenparameter** F aber alles machen, was er auch mit anderen **Konstanten** machen darf, z.B. ihren Wert ausgeben lassen oder ihren Namen in Ausdrücken verwenden (z.B. **FARBE1 := F;** oder **if F = FARBE1 then ...end if;** oder **while FARBE'succ(F) < GELB loop ... end loop;** etc.).

Falls der Schleifenparameter **nicht alle Werte** seines Untertyps durchlaufen soll, kann man eine entsprechende **Bereichseinschränkung** angeben, wie das folgende Beispiel zeigt:

Beispiel 12.6.2.: Zwei Zählschleifen mit **Bereichseinschränkungen**:

```
80 for F in FARBE range GRUEN .. WEISS loop
81    ...           -- irgendein Schleifenrumpf
82 end loop;
83
84 for I in OTTO range OTTO'first + 1 .. OTTO'last / 2 loop
85    ...           -- irgendein Schleifenrumpf
86 end loop;
```

In der ersten Zählschleife bewirkt die Bereichseinschränkung **range GRUEN..WEISS**, daß der Schleifenparameter **F** nur die Werte GRUEN, BLAU, GELB und WEISS annimmt. Die Bereichseinschränkung **range OTTO'first + 1 .. OTTO'last / 2** schränkt den Schleifenparameter I auf den Bereich -19_999..15_000 ein.

Der **Schleifenparameter** einer Zählschleife muß zu einem **diskreten Typ** (d.h. zu einem Ganzzahltyp oder einem Aufzählungstyp) gehören. Er darf z.B. nicht zu einem Bruchzahltyp wie **float** oder zu einem Reihungstyp wie **string** gehören. Bruchzahltypen und Reihungstypen werden später behandelt.

Normalerweise werden dem Schleifenparameter seine Werte in **aufsteigender Reihenfolge** zugewiesen. Der Programmierer kann aber auch eine **absteigende Reihenfolge** verlangen, indem er hinter **in** das reservierte Wort **reverse** angibt, z.B. so:

Beispiel 12.6.3.: Zwei Zählschleifen mit **Bereichseinschränkungen**:

```
90 for F in reverse FARBE range GRUEN .. WEISS loop
91    ...           -- irgendein Schleifenrumpf
92 end loop;
93
94 for I in reverse OTTO range 5..8 loop
95    ...           -- irgendein Schleifenrumpf
96 end loop;
```

Der Schleifenparameter **F** nimmt jetzt die Werte WEISS, GELB, BLAU und GRUEN (in dieser "umgekehrten" Reihenfolge) an und I durchläuft die Werte 8, 7, 6, und 5 (in dieser Reihenfolge).

Es liegt intuitiv nahe, eine Umkehrung der Reihenfolge dadurch zu verlangen, daß man in einer Bereichseinschränkung wie z.B. **range 5..8** die Grenzen vertauscht und **range 8..5 schreibt**. Das hätte aber einen ganz **anderen** als den angestrebten **Effekt**. Die Bereichsangabe **range 8..5** ist in Ada zwar erlaubt, beschreibt aber einen **leeren Bereich** (zu dem **keine** Werte gehören). Begründung: Es gibt genau **null** Werte, die gleichzeitig größer als 8 und kleiner als 5 sind.

Für diese Festlegung (die auf den ersten Blick möglicherweise befremdend wirkt) gibt es einsichtige Gründe. **Bereiche** spielen in Ada an sehr **vielen Stellen** eine Rolle, z.B. in **Vereinbarungen** von Ganzzahltypen oder Untertypen, bei der **is-in**-Operation (z.B. **if ZEICHEN is in 'A'..'Z' then ...**), bei **Zählschleifen** und an anderen Stellen, die noch behandelt werden. An **allen** diesen Stellen der Sprache Ada ist ein **leerer Bereich** als Grenzfall sinnvoll (etwa so, wie die Zahl 0 als Grenzfall einer Anzahl sinnvoll ist). Ein Bereich "in umgekehrter Reihenfolge" ist nur an **einer** Stelle sinnvoll, nämlich bei **Zählschleifen**. Deshalb hat man dem **wichtigen** leeren Bereich die "einfache" Notation **range 8..5** etc. zugeordnet und drückt die **weniger wichtige** "umgekehrte Reihenfolge bei Zählschleifen" durch das zusätzliche Wort **reverse** aus. Durch diese Festlegungen hat man andere wenig natürliche Regeln überflüssig gemacht (z.B. "In einer Typvereinbarung ist die Bereichs-

angabe **range 8..5** verboten, aber in einer Zählschleife ist sie erlaubt." oder "Die beiden Bereichsangaben **range 5..8** und **range 8..5** haben in einer **Typvereinbarung** die **gleiche** Bedeutung, in einer **Zählschleife** aber **verschiedene** Bedeutungen." etc.). Leere Bereiche werden wir später noch häufig benutzen. Hier noch ein konkretes Beispiel für die Anwendung des reservierten Wortes **reverse**:

Beispiel 12.6.4.: Zählschleifen ohne und mit "Rückwärtsgang" (**reverse**):

```
10 for Z in character range 'A'..'C' loop
11    ada.text_io.put(item => Z);
12 end loop;
13
14 for Z in character range 'C'..'A' loop
15    ada.text_io.put(item => Z);
16 end loop;
17
18 for Z in reverse character range 'A'..'C' loop
19    ada.text_io.put(item => Z);
20 end loop;
21
22 for Z in reverse character range 'C'..'A' loop
23    ada.text_io.put(item => Z);
24 end loop;
```

Die erste Schleife gibt die ersten **drei** Buchstaben des Alphabets in aufsteigender Reihenfolge aus, die dritte Schleife gibt sie in absteigender Reihenfolge aus. Die zweite und vierte Schleife geben jede **null** Zeichen aus (in aufsteigender bzw. absteigender Reihenfolge, aber das macht bei null Zeichen keinen Unterschied). **o**

Die Grenzen des Bereichs, den der Schleifenparameter durchlaufen soll, können durch beliebige **Ausdrücke** (z.B. durch **Variablen**) beschrieben werden. Der Ausführer berechnet die Werte dieser beiden Ausdrücke nur einmal (wenn er mit der Ausführung der Schleife beginnt) und "merkt sie sich". Die wichtigen Konsequenzen dieser Regelung soll das nächste Beispiel deutlich machen:

Beispiel 12.6.5.: Ein Bereich mit "variablen Grenzen":

```
30 OTTO_EA.get(item => G1);
31 OTTO_EA.get(item => G2);
32 for I in OTTO range G1 .. 2 * G2 - 3 loop
33    ...
34    G1 := G1 + 5;
35    G2 := G2 - 7;
36    ...
37 end loop;
```

Die Zuweisungen in Zeile 34 und 35 haben **keinen** Einfluß darauf, wie oft der Rumpf der Schleife ausgeführt wird. Ehe der Ausführer sie zum ersten Mal ausführt, hat er sich ausgerechnet und "gemerkt", wie oft er den Rumpf der Schleife (höchstens) ausführen soll. **o**

In Ada durchläuft der Schleifenparameter einer Zählschleife stets **alle** Werte eines bestimmten Bereichs, d.h. man kann keine Werte "überspringen lassen". Diese Eigenschaft beschreibt man auch so: In Ada hat eine Zählschleife immer die **Schrittweite** 1 (**ohne** reverse) oder -1 (**mit** reverse).

Die Entwickler von Ada haben bewußt **keine** besondere Notation für andere Schrittweiten in die Sprache aufgenommen. Statistische Untersuchungen von vielen Programmen stützen die Vermutung, daß andere Schrittweiten als 1 und -1 in der Praxis nur sehr selten gebraucht werden.

Falls nötig, kann man allgemeine Schrittweiten mit "normalen Ada-Befehlen" (ohne eine spezielle Notation) realisieren, wie das folgende Beispiel zeigen soll:

Beispiel 12.6.6.: Allgemeine Schrittweiten in Zählschleifen:

```
40 for I in OTTO range 10 .. 20 loop
41    G1 :=  3 * I;
42    G2 := -2 * I + 5;
43    ...
44 end loop;
```

Der Schleifenparameter I durchläuft die Werte 10, 11, 12, ..., 20 (Schrittweite 1). Die Variable G1 durchläuft die Werte 30, 33, 36, ..., 60 (Schrittweite 3). Die Variable G2 durchläuft die Werte -15, -17, -19, ..., -35 (Schrittweite -2). **o**

Um eine allgemeine Schrittweite wie +3 oder -2 zu realisieren, braucht man eine Multiplikationsoperation. Im vorigen Beispiel steht eine solche zur Verfügung, weil der Schleifenparameter zu einem **Ganzzahltyp** (!OTTO) gehört. Für **Aufzählungstypen** gibt es **keine** Multiplikation. Trotzdem kann man auch für einen Aufzählungstyp eine allgemeine Schrittweite realisieren (indirekt über einen Ganzzahltyp und mit den Attributen '**pos** und '**val**), wie das nächste Beispiel zeigt:

Beispiel 12.6.7.: Allgemeine Schrittweite bei einem **Aufzählungstyp**:

```
50 G1 := character'pos('A');  -- 'A' ist der erste Wert fuer ZEICHEN1
51 for I in OTTO range 0 .. 6 loop
52    ZEICHEN1 := character'val(G1 + 3 * I);
53    ...
54 end loop;
```

Die character-Variable **ZEICHEN1** nimmt der Reihe nach die Werte 'A', 'D', 'G', 'J', 'M', 'P' und 'S' an ("jeden dritten Buchstaben ab 'A'"). **o**

Zusammenfassung 12.6.:

- In Ada können Zählschleifen garantiert **nicht** zu **Endlosschleifen** werden.
- Der **Parameter** einer Zählschleife wird am Anfang jeder Ausführung der Schleife neu **erzeugt** und am Ende der Ausführung wieder zerstört.
- Der Parameter muß zu einem **diskreten** Typ gehören.
- Soll ein Wertebereich **rückwärts** durchlaufen werden, muß man **reverse** angeben (**nicht** die Grenzen des Bereichs "vertauschen"!).

12.7. Schleifen vorzeitig beenden und andere Feinheiten

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type    OTTO     is range -20_000 .. +1_000_000_000; -- OTTO ist gewachsen!
02 type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, GELB, WEISS);
03 type    COLOR    is (ORANGE, RED, YELLOW, LEMON, GREEN, OLIVE);
04 type    FRUIT    is (APPLE, LEMON, ORANGE, AVOCADO);
05 package OTTO_EA  is new ada.text_io.integer_io(num => OTTO);
06 G1, G2, G3  : OTTO;
07 B1          : boolean;
08 ZEICHEN1    : character;
```

Außerdem wird angenommen, daß den Variablen **G1**, **G2**, **G3**, **B1** und **ZEICHEN1** irgendwelche Werte (ihres Untertyps) zugewiesen wurden. Man beachte, daß die Aufzählungsliterale **ORANGE** und **LEMON** überladen sind, d.h. gleichzeitig zu mehreren Typen (**!COLOR** und **!FRUIT**) gehören. Ganz entsprechend sind Zeichenliterale wie **'A'** oder **'?'** überladen, weil sie (mindestens) zu den Typen **!character** und **!wide_character** gehören.

Für **Zählschleifen** gibt es eine kleine **Abkürzung**, die hier kurz beschrieben werden soll. In einigen der bisher angeführten Beispiele stand am Anfang einer Zählschleife hinter **in** bzw. hinter **in reverse** ein **Untertyp** (z.B. **FARBE**) und eine **Bereichseinschränkung** (z.B. **range GRUEN..WEISS**), etwa so:

Beispiel 12.7.1.: Zählschleifen mit Angabe eines Untertyps und einer Bereichseinschränkung:

```
10 for F in         FARBE     range BLAU..WEISS loop ...
11 for I in         OTTO      range    5..8     loop ...
12 for Z in         character range  'A'..'Z'   loop ...
13 for F in reverse FARBE     range BLAU..WEISS loop ...
14 for I in reverse OTTO      range    5..8     loop ...
15 for Z in reverse character range  'A'..'Z'   loop ...
o
```

Da die **Schleifenparameter** (in den Beispielen **F**, **I** bzw. **Z**) durch ihr Vorkommen hinter **for** erst **vereinbart** werden, ist es günstig, ihren **Untertyp** ausdrücklich anzugeben (ähnlich wie in einer normalen Variablenvereinbarung, z.B. **F1 : FARBE;** oder **G1 : OTTO;**).

In bestimmten Fällen kann man den **Untertyp** und die **Bereichseinschränkung** weglassen und stattdessen nur einen **Bereich** angeben (z.B. **BLAU..WEISS** oder **OTTO'first..2 * G1**, ohne **range** davor). Den **Bereich** beschreibt man (wie immer) durch zwei **Ausdrücke** (z.B. **BLAU** und **WEISS**) mit zwei Punkten **..** dazwischen. Aus diesen Ausdrücken versucht der Ausführer dann (nach bestimmten Regeln) den **Untertyp des Schleifenparameters** zu **schließen**. Wenn kein **eindeutiger** Schluß möglich ist, lehnt der Ausführer das betreffende Programm mit einer Fehlermeldung ab, sonst akzeptiert er es. Hier ein paar **empfehlenswerte**, ein paar **fragwürdige** (aber noch erlaubte) und ein paar **verbotene** Beispiele für diese abgekürzte Notation von Zählschleifen:

Beispiel 12.7.2.: Zählschleifen nur mit **Bereichsangabe**:

```
20 -- Empfehlenswerte Beispiele:                    -- Typ des Schleifenparameters:
21 for F in FARBE'first..GRUEN      loop ...        -- FARBE
22 for C in COLOR'first..LEMON      loop ...        -- COLOR
23 for F in FRUIT'first..FRUIT'last loop ...        -- FRUIT
24 for I in        100 .. OTTO'last loop ...        -- OTTO
25 -- Fragwuerdige Beispiele:
26 for F in      GRUEN .. WEISS     loop ...        -- FARBE
27 for I in         G1 .. G2        loop ...        -- OTTO
28 for I in         10 .. G2        loop ...        -- OTTO
29 for I in         G1 .. 20        loop ...        -- OTTO
30 for N in         10 .. 20        loop ...        -- integer (nicht OTTO!!)
31 -- Fehlerhafte Beispiele:
32 for F in     ORANGE .. LEMON     loop ...        -- COLOR oder FRUIT?
33 for Z in        'A' .. 'Z'       loop ...        -- character oder wide_character?
```

In Zeile 21 und 22 können die Kollegen und der Ausführer (und der Programmierer selbst) **leicht erkennen**, zu welchem Untertyp der Schleifenparameter gehören soll. In Zeile 23 bis 27 kann der Ausführer zwar noch eindeutig einen Untertyp festlegen, aber die Kollegen haben möglicherweise **Mühe**, diesen Untertyp herauszufinden. In den Zeilen 29 bis 30 kann selbst der Ausführer **nicht**

mehr eindeutig erkennen, welcher Untertyp gemeint ist und wird das betreffende Programm mit einer Fehlermeldung ablehnen. **o**

Wenn man Zweifel hat, sollte man hinter **in** bzw. **in reverse** lieber **ausdrücklich** einen **Untertyp** (und evtl. eine Bereichseinschränkung) angeben, statt sich und anderen durch "komplizierte Abkürzungen" das Lesen des Programms zu erschweren.

In Ada darf eine **exit-Anweisung** nicht nur in einer **loop-exit-Schleife** vorkommen, sondern auch in einer **while-Schleife** oder in einer **Zählschleife**. Bei **while**- und **Zählschleifen** empfiehlt es sich, den Leser durch einen **Kommentar** am Schleifenanfang darauf hinzuweisen, daß die Schleife evtl. frühzeitig mit einer **exit**-Anweisung verlassen wird, etwa so:

Beispiel 12.7.3: Eine **Zählschleife** mit **exit**-Anweisung und **Kommentar**:

```
10 OTTO_EA.get(item => G1)
11 for I in OTTO range 1..10 loop
12 -- Wird evtl. fruehzeitig mit exit verlassen!!
13    OTTO_EA.put(item => G1);
14    ada.text_io.new_line;
15 exit when G2 > OTTO'last / 2;
16    G1 := G1 / 2;
17 end loop;
```

Ein Hinweis wie der in Zeile 12 ist **besonders** dann empfehlenswert, wenn der Text der Schleife nicht vollständig auf **eine** Seite paßt, auf **mehrere** Seiten lang ist. **o**

Eine **exit**-Anweisung ohne **when** (einfach nur **exit;**) ist ein Befehl, die umgebende Schleife (sofort und unbedingt) zu verlassen. Eine solche unbedingte **exit**-Anweisung ist gleichbedeutend mit der bedingten Anweisung **exit when true;** und die Anweisung **exit when B1;** bedeutet das Gleiche wie **if B1 then exit; end if;**. Hier ein Beispiel dazu:

Beispiel 12.7.4: Zwei **Zählschleifen** mit gleicher Bedeutung:

```
20 loop                          30 loop
21    ...                        31    ...
22 exit when B1;                 32    if B1 then exit; end if;
23    ...                        33    ...
24 end loop;                     34 end loop;
```

Welche Form finden Sie leichter lesbar? **o**

Die Ausführung einer Schleife kann nicht nur durch eine **exit-Anweisung**, durch eine "**while-Bedingung**" oder durch eine "**for-Zählung**", sondern auch durch eine **return-Anweisung** oder durch das Auftreten einer **Ausnahme** beendet werden.

Eine **return-Anweisung** darf nur innerhalb eines **Unterprogramms** stehen (z.B. innerhalb einer Prozedur, aber nicht im Anweisungsteil eines Paketes. Pakete werden später behandelt). Sie beendet die Ausführung des umgebenden Unterprogramms. Ein Unterprogramm darf **beliebig viele** return-Anweisungen enthalten. Im folgenden Beispiel beendet die **return**-Anweisung "nebenbei" auch noch eine Schleife:

Beispiel 12.7.5: Eine Prozedur mit **return**-Anweisung:

```
40 with ada.text_io;
41 procedure QUADR_01 is
42    type    GANZ    is      range -1_000..+1_000_000;
43    subtype EINGABE is GANZ range -1_000..+1_000;
44    package GANZ_EA is new ada.text_io.integer_io(num => GANZ);
45    G1 :    EINGABE;
46 begin

47    loop
48       ada.text_io.put(item => "Eine Zahle (-1_000..+1_000)? ");
49       GANZ_EA.get(item => G1);
50       if G1 = 0 then
51          return;     -- Beende die Ausfuehrung der Prozedur QUADR_01!
52       end if;
53       GANZ_EA.put(item => G1 ** 2);
54    end loop;
55 end QUADR_01;
```

Dieses Programm liest wiederholt eine Ganzzahl ein und gibt ihr Quadrat aus, bis der Benutzer eine 0 eingibt. Die **return-Anweisung** in Zeile 51 beendet die Prozedur und "nebenbei" natürlich auch die Schleife, in der sie steht. **o**

Das folgende Beispiel ist eine Variante des vorigen. Es enthält eine Schleife **ohne exit**- oder **return**-Anweisung im Rumpf. Trotzdem handelt es sich **nicht** um eine echte Endlosschleife:

Beispiel 12.7.6: Eine **Schleife**, die nur durch eine **Ausnahme** beendet werden kann:

```
60 with ada.text_io;
61 procedure QUADR_02 is
62    type    GANZ    is      range -1_000..+1_000_000;
63    subtype EINGABE is GANZ range -1_000..+1_000;
63    package GANZ_EA is new ada.text_io.integer_io(num => GANZ);
64    G1 :    EINGABE;
65 begin
66    loop
67       ada.text_io.put(item => "Eine Zahle (-1_000..+1_000)? ");
68       GANZ_EA.get    (item =>  G1);   -- Loest evtl. eine Ausnahme aus!
69       GANZ_EA.put    (item =>  G1 ** 2);
70    end loop;
71 end QUADR_02;
```

Wenn der Benutzer anstelle einer **Ganzzahl** z.B. irgendeinen **Buchstaben** eingibt, löst der **get**-Befehl in Zeile 68 die Ausnahme **data_error** aus. Dadurch wird das ganze Unterprogramm **QUADR_02** (und "nebenbei" natürlich auch die Schleife) beendet. Dieses Beispiel soll einen wichtigen Effekt von Ausnahmen deutlich machen, stellt aber keine Empfehlung dar, so zu programmieren. **o**

Die grundlegende **arbeitsökonomische Bedeutung** von Schleifenanweisungen beruht auf der folgenden simplen (aber sehr wichtigen) Tatsache: Mithilfe einer Schleife kann der Programmierer mit **wenig** (menschlicher) **Schreibarbeit** den Ausführer zu **sehr viel** (maschineller) **Ausführungsarbeit** veranlassen, etwa so:

Beispiel 12.7.7.: Wenig Programmiererarbeit, viel Ausführerarbeit:

```
80 for I in OTTO range 1..1_000_000_000 loop
81    ...
82 end loop;
```
o

Bei **if-** und **case-**Anweisungen ist das Verhältnis eher **umgekehrt**: Der Programmierer muß **mehr** Befehle hinschreiben, als der Ausführer später dann ausführt.

Aufgabe 12.7.1.: Angnommen der Rumpf der Schleife im vorigen Beispiel (12.7.7.) besteht nur aus der Zuweisung **G1 := G1 + 1;**. In welcher Größenordnung liegt die Zeit, die ein heute üblicher PC für die Ausführung (der entsprechenden Maschinenbefehle) vermutlich braucht? Millisekunden? Sekunden? Minuten? Stunden? Tage? **o**

Aufgabe 12.7.2.: Skizzieren Sie ein Programm, welches dem Ausführer befiehlt, ungefähr 1_000_000_000-mal (auf ein paar Millionen mehr oder weniger kommt es hier nicht an) die Zuweisung **G1 := G1 + 1;** auszuführen. Das Programm soll nicht länger als ca. 200 bis 300 Zeilen (10_000 bis 15_000 Zeichen) sein. Der Haken: Das Programm soll **keine einzige** Schleife enthalten (darf aber aus beliebig vielen Prozeduren bestehen). **o**

Beim **Sprechen und Diskutieren über Schleifen** sollte man möglichst sorgfältig zwischen einer **Schleife** und ihrem **Rumpf** unterscheiden. Das folgende Beispiel soll diese Empfehlung begründen:

Beispiel 12.7.8.: Zwei geschachtelte Schleifen:

```
10 ada.text_io.put(item => "Vor Schleife1");
11 for I1 in OTTO range 1..100 loop
12    ada.text_io.put(item => "In Schleife1, vor Schleife2");
13    for I2 in OTTO range 1..200 loop
14       ada.text_io.put(item => "In Schleife2");
15       ...
16    end loop;
17 end loop;
```

Wenn die **Schleife1** (in Zeile 11 bis 17) **einmal** ausgeführt wird, wird ihr **Rumpf** (in Zeile 12 bis 16) genau **100-mal** ausgeführt. Wenn die **Schleife2** (in Zeile 13 bis 16) **einmal** ausgeführt wird, wird ihr **Rumpf** (in Zeile 14 bis 15) genau **200-mal** ausgeführt. Da die Schleife2 zum Rumpf der Schleife1 gehört, wird sie (die Schleife2) genau 100-mal ausgeführt und ihr Rumpf wird (100 * 200 gleich) **20_000-mal ausgeführt**. Konkret bedeutet das: Die Meldung "Vor Schleife1" wird **einmal**, die Meldung "In Schleife1, vor Schleife2" wird **100-mal** und die Meldung "In Schleife2" wird **20_000-mal** ausgegeben. **o**

Die richtigen Antworten auf die beiden Fragen "Wie oft wird die **Schleife2** ausgeführt?" und "Wie oft wird **der Rumpf der Schleife2** ausgeführt?" unterscheiden sich deutlich voneinander (100 bzw. 20_000). Das spricht dafür, auch die Fragen sorgfältig zu unterscheiden.

Zusammenfassung 12.7.:

- In einer Zählschleife muß man den **Untertyp** des Schleifenparameters nicht immer **ausdrücklich angeben**, aber häufig sollte man es trotzdem tun.
- Auch die Ausführung einer **while**- oder einer **Zählschleife** kann "vorzeitig" durch eine **exit**-Anweisung, eine **return**-Anweisung oder eine **Ausnahme** abgebrochen werden.
- Schleifen sind wichtig, weil sie mit **wenig** Programmiererarbeit **viel** Ausführerarbeit veranlassen können.
- Theoretisch sollte man eine **Schleife** und ihren **Rumpf** sorgfältig unterscheiden (praktisch werden die beiden häufig verwechselt).

12.8. Die Blockanweisung

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type    OTTO     is range -20_000 .. +30_000; -- OTTO ist wieder geschrumpft!
02 G1, G2, G3  : OTTO;
```

Außerdem wird angenommen, daß den Variablen **G1**, **G2**, **G3**, **B1** und **ZEICHEN1** irgendwelche Werte (ihres Untertyps) zugewiesen wurden.

Die bisher vorgestellten **zusammengesetzten Anweisungen** (if-, case- und loop-Anweisungen) dienen dazu, bestimmte Anweisungsfolgen **mehr als einmal** oder **weniger als einmal** (und als Sonderfall auch genau einmal) ausführen zu lassen. Die **Blockkanweisung** hat einen ganz anderen Zweck. Sie faßt eine Folge von **Vereinbarungen** und eine Folge von **Anweisungen** zu **einer Anweisung** zusammen, etwa so:

Beispiel 12.8.1.: Eine **Blockanweisung**:

```
10 declare
11    TMP : constant OTTO := G1;
12 begin
13    G1 := G2;
14    G2 := TMP;
15 end;
```

Die Blockanweisung besteht aus einem **Vereinbarungsteil** (zwischen **declare** und **begin**, Zeile 11) und einem **Anweisungsteil** (zwischen **begin** und **end**, Zeile 13 bis 14). Sie wird ausgeführt, indem zuerst alle **Vereinbarungen** (im Vereinbarungsteil) und dann alle **Anweisungen** (im Anweisungsteil) ausgeführt werden. Zum Schluß **zerstört** der Ausführer alle im Block vereinbarten Größen wieder (im Beispiel ist das nur die eine Konstante namens TMP). Die Blockanweisung in diesem Beispiel bewirkt, daß die Werte der Variablen G1 und G2 vertauscht werden. **o**

Die im Vereinbarungsteil vereinbarten Größen bezeichnet man auch als **lokale Größen** des Blocks. Sie existieren nur, während der Block ausgeführt wird. Davor und danach gibt es sie noch nicht bzw. nicht mehr. Wenn der Block in obigem Beispiel z.B. **siebenmal** ausgeführt wird (etwa, weil er zum Rumpf einer Schleife gehört), dann wird **siebenmal** eine Konstante namens TMP erzeugt (und wieder zerstört). Diese sieben Konstanten können unterschiedliche Werte haben, aber keine ändert ihren Wert im Laufe ihres kurzen Lebens.

In einem Block kann man im Prinzip **alle** Größen vereinbaren, die man in Ada überhaupt vereinbaren kann, z.B. Typen, Untertypen, Konstanten, Variablen, ... etc..

Ein Block kann auch nur aus einem **Anweisungsteil** (ohne **Vereinbarungsteil** davor) bestehen, z.B. so:

Beispiel 12.8.2.: Eine Blockanweisung **ohne Vereinbarungsteil**:

```
20 begin
21    G3 := G1;
22    G1 := G2;
23    G2 := G3;
24 end;
```

Auch diese Blockanweisung bewirkt, daß die Werte der Variablen G1 und G2 vertauscht werden. Als "Zwischenlager" wird diesmal aber keine lokale Konstante verwendet, sondern die globale (d.h. nicht im Block vereinbarte) Variable G3. **o**

Warum es manchmal sinnvoll und notwendig ist, Anweisungen "in einen Block zu packen", statt sie "einfach so" (ohne **begin** und **end**) hinzuschreiben, kann erst im Zusammenhang mit **Ausnahmen** (im Abschnitt 13.) erläutert werden. Als Andeutung soll hier genügen: Mit einem Block kann man die Umgebung des Blocks vor unerwünschten Folgen einer Ausnahme (die im Block ausgelöst wird), "**beschützen**". Z.B. kann man bewirken, daß aufgrund einer Ausnahme im Block nur der Block (und nicht das gesamte Programm) abgebrochen wird.

Der Programmierer kann einem Block auch einen **Namen** geben, z.B. so:

Beispiel 12.8.3.: Zwei Blockanweisungen **mit Namen**:

```
30 BERND: declare                          40 BERTA: begin
31    TMP : constant OTTO := G1;           41    G3 := G1;
32 begin                                   42    G1 := G2;
33    G1 := G2;                            43    G2 := G3;
34    G2 := TMP;                           44 end BERTA;
35 end BERND;
```

Der Name des Blocks **muß** hinter **end** wiederholt werden. **o**

In einem Programm mit vielen (oder langen) Blöcken können **Blocknamen** deutlich machen, welches **end** zu welchem Blockanfang (**declare** bzw. **begin**) gehört. Außerdem kann ein gut gewählter Name dem Leser eine Idee davon vermitteln, "was der Block bewirkt". Im Beispiel wären Namen wie z.B. **VERTAUSCHE** oder **SWAP** informativer als **BERND** bzw. **BERTA**.

Zusammenfassung 12.8.:

- Ein **Block** ist eine **Anweisung**, die eine **Anweisungsfolge** und möglicherweise auch **Vereinbarungen** von lokalen Größen enthält.
- Blöcke werden vor allem zum **Behandeln von Ausnahmen** gebraucht (siehe Abschnitt 13).

Zentraldokument: Nach Filialdokument A95-11-12

13. Ausnahmen vereinbaren, auslösen und behandeln

Viele Ada-Befehle lösen in bestimmten Situationen eine **Ausnahme** aus. Z.B. löst die Additionsoperation **"+"** des Typs **!OTTO** die Ausnahme **constraint_error** aus, wenn das Ergebnis der Addition nicht mehr zum Typ **!OTTO** gehört, und die Prozedur **get** aus dem Paket **OTTO_EA** löst die Ausnahme **data_error** aus, wenn der Benutzer eine "ungeeignete Zeichenkette" eingibt.

Außerdem kann der Programmierer mit einer **raise-Anweisung** dem Ausführer ausdrücklich befehlen, eine bestimmte Ausnahme auszulösen (z.B. **raise constraint_error;** oder **raise data_error;** etc.).

In Ada gibt es **vier vordefinierte** Ausnahmen (**constraint_error**, **programm_error**, **storage_error** und **tasking_error**). **Sieben** weitere Ausnahmen (darunter **data_error**) stehen zur Verfügung, wenn man das Standardpaket **ada.text_io** in ein Programm einbindet (mit der Kontextklausel **with ada.text_io;**). Die Namen dieser sieben Ausnahmen findet man im (ARM A.10.1(84)).

Der Programmierer kann weitere Ausnahmen **vereinbaren** (d.h. vom Ausführer erzeugen lassen) und sie dann mit der **raise**-Anweisung **auslösen**, z.B. so:

Beispiel 13.1.: Eine Ausnahme **vereinbaren**, verschiedene Ausnahmen **auslösen**:

```
01 with ada.text_io;
02 procedure AUSNA_01 is
03    DAS_WAR_EIN_A       : exception;
04    ZEICHEN1            : character;
05 begin
06    ada.text_io.put(item => "Bitte geben Sie ein Zeichen ein: ");
07    ada.text_io.get(item => ZEICHEN1);
08    case ZEICHEN1 is
09       when  'A'   => raise DAS_WAR_EIN_A;
10       when  'B'   => raise constraint_error;
11       when  'C'   => raise ada.text_io.data_error;
12       when others => null;
13    end case;
14    ada.text_io.put_line(item => "Normalfall, keine Ausnahme!");
15 end AUSNA_01;
```

In Zeile 03 wird eine **Ausnahme** namens DAS_WAR_EIN_A **vereinbart**. In Zeile 09 wird diese Ausnahme mit einer **raise**-Anweisung **ausgelöst**. Wenn (bei einer Ausführung des Programms AUSNA_01) der Benutzer einen Buchstaben zwischen 'A' und 'C' eingibt, wird eine Ausnahme **ausgelöst** (in Zeile 09, 10 bzw. 11) und das Programm mit einer entsprechenden Fehlermeldung (in der das Wort "**DAS_WAR_EIN_A**" bzw. "**constraint_error**" bzw. "**data_error**" vorkommt) abgebrochen. Gibt der Benutzer irgendein anderes Zeichen des Untertyps **character** ein (z.B. 'X'), wird die Meldung "Normalfall, keine Ausnahme!" ausgegeben (siehe Zeile 14). **o**

Die **Vereinbarung der Ausnahme** (in Zeile 03) sieht ganz ähnlich aus, wie die **Vereinbarung einer Variablen** (siehe z.B. Zeile 04). Man beachte aber, daß **exception** kein Typ ist, sondern "ein spezielles reserviertes Wort zum Vereinbaren von Ausnahmen". Entsprechend ist **DAS_WAR_EIN_A** auch keine Variable, sondern eine Ausnahme.

Mit der **raise**-Anweisung kann der Programmierer also auch solche Ausnahmen auslösen, die er nicht selbst vereinbart hat, z.B. die vordefinierte Ausnahme **constraint_error** (siehe Zeile 10) oder die im Paket **ada.text_io** vereinbarte Ausnahme **data_error** (siehe Zeile 11).

Im Programm **AUSNA_01** wird **eine** Ausnahme **vereinbart** (DAS_WAR_EIN_A) und **drei** Ausnahmen werden **ausgelöst**. Im folgenden Programm **AUSNA_02** werden **drei** Ausnahmen vereinbart, ausgelöst und außerdem auch noch **behandelt**.

Beispiel 13.2.: Ausnahmen vereinbaren, auslösen und **behandeln**:

```
01 with ada.text_io;
02 procedure AUSNA_02 is
03    DAS_WAR_EIN_A: exception;
04    DAS_WAR_EIN_B: exception;
05    DAS_WAR_EIN_C: exception;
06    ZEICHEN1     : character;
07 begin
08    ada.text_io.put(item => "Bitte geben Sie ein Zeichen ein: ");
09    ada.text_io.get(item => ZEICHEN1);
10    case ZEICHEN1 is
11       when  'A'   => raise DAS_WAR_EIN_A;
12       when  'B'   => raise DAS_WAR_EIN_B;
13       when  'C'   => raise DAS_WAR_EIN_C;
14       when others => null;
15    end case;
16    ada.text_io.put(item => "Normalfall, keine Ausnahme!");
17 exception
18    when DAS_WAR_EIN_A =>
19       ada.text_io.put_line(item => "Sie haben ein A eingegeben!");
20       ada.text_io.put_line(item => "Versuchen Sie mal B oder C!");
21    when DAS_WAR_EIN_B | DAS_WAR_EIN_C =>
22       ada.text_io.put_line(item => "Sie haben B oder C eingegeben!");
23 end AUSNA_02;
```

Der **Anweisungsteil** der Prozedur AUSNA_02 (Zeile 08 bis 22) wird durch das reservierte Wort **exception** (in Zeile 17) in zwei Teile geteilt, und zwar in einen **normalen Anweisungsteil** (zwischen **begin** und **exception**, Zeile 08 bis 16) und einen **Ausnahmeteil** (zwischen **exception** und **end**, Zeile 18 bis 22). Der **Ausnahmeteil** enthält in diesem Beispiel zwei **Ausnahmebehandler** (in Zeile 18 bis 20 bzw. in Zeile 21 bis 22). Der erste ist nur für die eine Ausnahme **DAS_WAR_-EIN_A** zuständig, der zweite für die beiden Ausnahmen **DAS_WAR_EIN_B** und **DAS_WAR_-EIN_C**.

Ein **Ausnahmebehandler** besteht (ähnlich wie eine Alternative einer **case**-Anweisung, siehe Beispiel 12.4.1.) aus einer **Wahlliste** (zwischen **when** und dem Pfeil =>) und einer **Anweisungsfolge** (zwischen dem Pfeil => und dem Ende des Ausnahmebehandlers).

Angenommen, der Benutzer gibt ein **'A'** ein (siehe Zeile 09). Dann wird die Ausnahme **DAS_-WAR_EIN_A** ausgelöst (durch die **raise**-Anweisung in Zeile 11). Daraufhin verläßt der Ausführer den **normalen Anweisungsteil** (Zeile 08 bis 16), sucht im **Ausnahmeteil** (nach dem Wort **exception**) einen zuständigen **Behandler**, findet einen (in Zeile 18 bis 20), führt dessen **Anweisungsfolge** (Zeile 19 bis 20) aus und beendet danach "ganz normal" die Ausführung der Prozedur AUSN_02. **o**

Aufgabe 13.1.: Was passiert im Programm AUSNA_02, wenn der Benutzer ein **'C'** oder wenn er ein **'X'** eingibt? Beschreiben Sie die Aktionen des Ausführers, und benutzen Sie dabei möglichst viele Fachbegriffe wie **normaler Anweisungsteil**, **Ausnahmeteil**, (Ausnahme-) **Behandler**, **Wahlliste**, **Anweisungsfolge eines Behandlers** etc.. **o**

Aufgabe 13.2.: Lassen Sie das Programm AUSNA_02 **viermal** von einem maschinellen Ausführer ausführen, und geben Sie (als Benutzer) dabei die Buchstaben **'A'**, **'B'**, **'C'** bzw. **'X'** ein. Notieren

Sie sich alle **Ausgaben** des Ausführers, und **erklären** Sie anhand des Programmtextes, "warum gerade diese Ausgaben herauskommen". **o**

Ausnahmen haben die folgende sehr wichtige und nützliche Eigenschaft: Sie können an **einer** Stelle in einem Programm erkannt und **ausgelöst** und an einer ganz **anderen** Stelle **behandelt** werden. Selbst wenn an dieser anderen Stelle die Namen der möglichen Ausnahmen nicht alle bekannt sind, ist eine Behandlung doch möglich. Normalerweise steht am Anfang eines Behandlers (zwischen **when** und dem Pfeil =>) eine **Wahlliste**, in der alle Ausnahmen genannt werden müssen, für die dieser Behandler **zuständig** sein soll. Ein Behandler, der mit **when others =>** beginnt, ist für alle Ausnahmen zuständig, die von keinem anderen Behandler behandelt werden. Im folgenden Beispiel werden Ausnahmen in **einem** Unterprogramm erkannt und ausgelöst (in **AUSNA_01**, siehe oben) und in einem **anderen** Unterprogramm behandelt (in **AUSNA_03**, siehe unten). Das Standardpaket **ada.exceptions** (siehe ARM 11.4.1) enthält einige nützliche Unterprogramme zur Bearbeitung von Ausnahmen (insbesondere von Ausnahmen, deren Namen man nicht kennt).

Beispiel 13.3.: Anonyme Ausnahmen mit einem **when-others-Behandler** abfangen:

```
01 with AUSNA_01, ada.text_io, ada.exceptions;
02 procedure AUSNA_03 is
03 begin
04    AUSNA_01; -- Loest evtl. eine Ausnahme aus.
05    ada.text_io.put_line(item => "AUSNA_01 wurde normal beendet!");
06 exception
07    when constraint_error =>
08       ada.text_io.put_line(item => "AUSNA_01 loeste constraint_error aus!");
09    when A: others =>
10       ada.text_io.put_line(item => "AUSNA_01 loeste folg. Ausnahme aus:");
11       ada.text_io.put_line(item => ada.exceptions.exception_name(X => A));
12 end AUSNA_03;
```

Das Programm **AUSNA_03** besteht aus den beiden Unterprogrammen **AUSNA_03** (siehe Zeile 02 bis 12) und **AUSNA_01** (siehe Zeile 01) und aus den beiden Paketen **ada.text_io** und **ada.exceptions** (siehe Zeile 01). In Zeile 03 wird das Unterprogramm **AUSNA_01** aufgerufen (siehe oben Beispiel 13.1.). Dadurch wird möglicherweise eine der Ausnahmen **DAS_WAR_EIN_A**, **constraint_error** oder **ada.text_io.data_error** ausgelöst. Diese Ausnahmen behandelt der Ausführer mit den Behandlern im Ausnahmeteil (Zeile 07 bis 11) von AUSNA_03.

Der erste Behandler (Zeile 07 bis 08) ist nur für die Ausnahme **constraint_error** zuständig. Der zweite Behandler (Zeile 09 bis 11) ist für **alle anderen** Ausnahmen zuständig (insbesondere also für **DAS_WAR_EIN_A** und **ada.text_io.data_error**).

Der zweite Behandler beginnt mit **when A: others =>**. Dabei ist "A" ein vom Programmierer gewählter Name ("A" wie "Ausnahme"), der die zu behandelnde Ausnahme bezeichnet (technisch genauer ausgedrückt: A ist eine Konstante vom Typ **exception_occurrence** und bezeichnet das **Ausnahmevorkommen**, welches gerade behandelt wird). Wenn die Funktion **exception_name** (aus dem Paket **ada.exceptions**) auf A angewendet wird, bekommt man den **Namen** der Ausnahme A (als Zeichenkette vom Typ **string**). Diesen kann man z.B. ausgeben, wie es in Zeile 11 geschieht.

Man beachte, daß die Ausnahme **DAS_WAR_EIN_A** im Unterprogramm **AUSNA_01** vereinbart wurde und somit in AUSNA_03 **nicht sichtbar** ist (d.h. nicht mit dem Namen AUSNA_01 bezeichnet werden kann). Trotzdem kann diese Ausnahme in AUSNA_03 **behandelt** werden (mit dem when-others-Behandler in Zeile 09 bis 11) und ihr **Name** kann z.B. ausgegeben werden (mit Hilfe der Funktion **exception_name** aus dem Paket **ada.exceptions**). **o**

Aufgabe 13.3.: Was passiert im Programm AUSNA_03, wenn der Benutzer ein **'A'** bzw. ein **'X'** eingibt? Beschreiben Sie die Aktionen des Ausführers und benutzen Sie dabei möglichst viele Fachbegriffe wie **normaler Anweisungsteil**, **Ausnahmeteil**, (Ausnahme-) **Behandler**, **Wahlliste**, **Anweisungsfolge eines Behandlers** etc.. **o**

Aufgabe 13.4.: Lassen Sie das Programm AUSNA_03 **viermal** von einem maschinellen Ausführer ausführen, und geben Sie (als Benutzer) dabei die Buchstaben **'A'**, **'B'**, **'C'** bzw. **'X'** ein. Notieren Sie sich alle **Ausgaben** des Ausführers, und **erklären** Sie anhand des Programmtextes, "warum gerade diese Ausgaben herauskommen". **o**

In den Programmen **AUSNA_02** und **AUSNA_03** werden Ausnahmen nicht nur vereinbart und ausgelöst, sondern auch **behandelt**. Trotzdem bewirkt das Auftreten einer Ausnahme immer noch (wie bei **AUSNA_01**), daß die Ausführung des Programms **beendet** wird (nachdem der zuständige Ausnahmebehandler ausgeführt wurde). Konkret erkennt man das daran, daß im Falle einer Ausnahme die Meldung "**Normalfall, keine Ausnahme**!" (siehe Zeile 16 in **AUSNA_02**) bzw. die Meldung "**AUSNA_01 wurde normal beendet!**" (siehe Zeile 05 in **AUSNA_03**) **nicht** ausgegeben wird.

Im folgenden Beispiel wird das Auslösen und das Behandeln von Ausnahmen in eine **Blockanweisung** "eingepackt". Falls eine Ausnahme auftritt, wird dann nur die Ausführung des Blocks beendet, aber nicht die Ausführung des ganzen Programms.

Beispiel 13.4.: Ausnahmen in einem **Block** auslösen und behandeln:

```
01 with ada.text_io;
02 procedure AUSNA_10 is
03    DAS_WAR_EIN_A: exception;
04    DAS_WAR_EIN_B: exception;
05    DAS_WAR_EIN_C: exception;
06    ZEICHEN1     : character;
07 begin
08    ada.text_io.put(item => "Bitte geben Sie ein Zeichen ein: ");
09    ada.text_io.get(item => ZEICHEN1);
10    AUSNA_11: begin
11       case ZEICHEN1 is
12          when  'A'   => raise DAS_WAR_EIN_A;
13          when  'B'   => raise DAS_WAR_EIN_B;
14          when  'C'   => raise DAS_WAR_EIN_C;
15          when others => null;
16       end case;
17       ada.text_io.put_line(item => "Normalfall, keine Ausnahme!");
18    exception
19       when DAS_WAR_EIN_A =>
20          ada.text_io.put_line(item => "Sie haben ein A eingegeben!");
21       when DAS_WAR_EIN_B | DAS_WAR_EIN_C =>
22          ada.text_io.put_line(item => "Sie haben ein B oder C eingegeben!");
23    end AUSNA_11;
24    ada.text_io.put(item => "AUSNA_11 wurde normal beendet!");
25 end AUSNA_10;
```

Der **Block** namens **AUSNA_11** (Zeile 10 bis 23) besteht nur aus einem **Anweisungsteil** (ohne Vereinbarungsteil davor). Dieser Anweisungsteil wird durch das Wort **exception** (in zwei Zeile 18) in einen **normalen Anweisungsteil** (Zeile 11 bis 17) und einen **Ausnahmeteil** (Zeile 19 bis 22) geteilt.

Wenn im Block **AUSNA_11** z.B. die Ausnahme **DAS_WAR_KEIN_B** ausgelöst wird (durch die **raise**-Anweisung in Zeile 13), sucht der Ausführer im Ausnahmeteil **des Blocks** nach einem zuständigen Behandler, findet einen (in Zeile 21 bis 22), führt dessen **Anweisungsfolge** (nur eine Anweisung in Zeile 24) aus und beendet dann "ganz normal" die Ausführung des **Blocks** AUSNA_11 (**nicht** die Ausführung der **Prozedur** AUSNA_10!). Dann wird noch die Meldung "**AUSNA_11 wurde normal beendet!**" (siehe Zeile 24) ausgegeben und erst danach die Prozedur **AUSNA_10** (ebenfalls ganz normal) beendet.

Aufgabe 13.5.: Was passiert im Programm **AUSNA_10**, wenn der Benutzer ein **'A'** oder wenn er ein **'X'** eingibt? Beschreiben Sie die Aktionen des Ausführers, und benutzen Sie dabei möglichst viele Fachbegriffe wie **normaler Anweisungsteil**, **Ausnahmeteil**, (Ausnahme-) **Behandler**, **Wahlliste**, **Anweisungsfolge eines Behandlers** etc.. **o**

Aufgabe 13.6.: Lassen Sie das Programm AUSNA_10 **viermal** von einem maschinellen Ausführer ausführen, und geben Sie (als Benutzer) dabei die Buchstaben **'A'**, **'B'**, **'C'** bzw. **'X'** ein. Notieren Sie sich alle **Ausgaben** des Ausführers, und **erklären** Sie anhand des Programmtextes, "warum gerade diese Ausgaben herauskommen". **o**

Aufgabe 13.7.: Zur Erinnerung: Wenn man versucht, mit einem Befehl wie **OTTO_EA.get**, eine Ganzzahl des Untertyps **OTTO** einzulesen, der Benutzer aber "falsche Daten" eingibt, wird die Ausnahme **data_error** ausgelöst. Schreiben Sie ein Programm namens **AUSNA_04**, welches solange versucht, von der aktuellen Eingabe eine Ganzzahl einzulesen, bis der Benutzer Daten eingibt, die **nicht** die Ausnahme **data_error** auslösen.
Hinweis1: Es empfiehlt sich, im Programm **AUSNA_04** eine boolesche Variable mit einem Namen wie z.B. **EINGABE_NOCH_NICHT_OK** zu vereinbaren. Welchen Anfangswert sollte man dieser Variablen geben? Wann sollte man ihren Wert ändern?
Hinweis2: Nachdem der Benutzer "falsche Daten" eingegeben hat und bevor Sie erneut versuchen, eine Ganzzahl einzulesen, müssen Sie die "falschen Daten" des Benutzers mit der Anweisung **ada.text_io.skip_line;** "wegwerfen" (d.h. "aus dem Eingabepuffer des Betriebssystems entfernen"), sonst gerät der Ausführer in eine Endlosschleife. **o**

Das folgende Beispielprogramm **AUSNA_20** enthält zwei **Blöcke** namens **AUSNA_21** und **AUSNA_22**, die **geschachtelt** sind (d.h. der Block AUSNA_22 ist eine Anweisung **im** Block AUSNA_21). Das Programm soll deutlich machen, daß eine Ausnahme, die in einem **inneren** Block ausgelöst wurde, in diesem **inneren** oder in einem **weiter außen liegenden** Block behandelt werden kann.

Beispiel 13.5.: Ausnahmen in einem Rahmen auslösen und in verschiedenen Rahmen behandeln:

```
01 with ada.text_io;
02 procedure AUSNA_20 is
03    DAS_WAR_EIN_A: exception;
04    DAS_WAR_EIN_B: exception;
05    DAS_WAR_EIN_C: exception;
06    ZEICHEN1     : character;
07 begin
08    ada.text_io.put(item => "Bitte geben Sie ein Zeichen ein: ");
09    ada.text_io.get(item => ZEICHEN1);
10    AUSNA_21: begin
11       AUSNA_22: begin
12          case ZEICHEN1 is
13             when  'A'   => raise DAS_WAR_EIN_A;
```

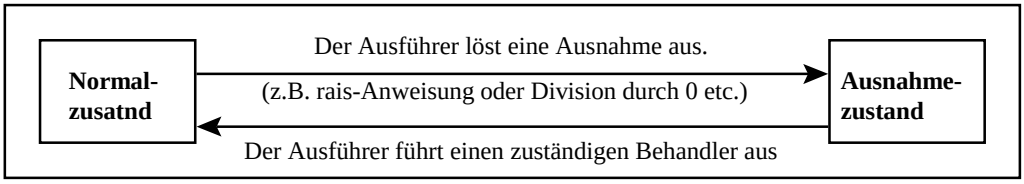
```
14             when  'B'   => raise DAS_WAR_EIN_B;
15             when  'C'   => raise DAS_WAR_EIN_C;
16             when others => null;
17          end case;
18          ada.text_io.put_line(item => "Normalfall, keine Ausnahme!");
19       exception -- AUSNA_22
20          when DAS_WAR_EIN_A =>
21             ada.text_io.put_line(item => "Sie haben ein A eingegeben!");
22       end AUSNA_22;
23       ada.text_io.put_line(item => "AUSNA_22 wurde normal beendet!");
24    exception -- AUSNA_21
25       when DAS_WAR_EIN_B =>
26          ada.text_io.put_line(item => "Sie haben ein B eingegeben!");
27    end AUSNA_21;
28    ada.text_io.put_line(item => "AUSNA_21 wurde normal beendet!");
29 exception -- AUSNA_20
30    when DAS_WAR_EIN_C =>
31       ada.text_io.put_line(item => "Sie haben ein C eingegeben!");
32 end AUSNA_20;
```

Wenn der Benutzer ein **'B'** eingibt (siehe Zeile 09), wird die Ausnahme **DAS_WAR_EIN_B** ausgelöst (durch die **raise**-Anweisung im inneren Block **AUSNA_22** in Zeile 14). Daraufhin führt der Ausführer folgende Aktionen durch:
1. Er beendet die Ausführung des **normalen Anweisungsteils** von **AUSNA_22** (Zeile 12 bis 18) und
2. sucht im **Ausnahmeteil** von **AUSNA_22** (Zeile 20 bis 21) nach einem **zuständigen Behandler**.
3. Er **findet keinen**. Daraufhin **bricht** er die Ausführung von AUSNA_22 **ab** und **propagiert** die Ausnahme **DAS_WAR_EIN_B**, d.h. er **löst** sie unmittelbar **nach** AUSNA_22 (sozusagen zwischen Zeile 22 und 23) erneut **aus**.
4. Daraufhin beendet er die Ausführung des **normalen Anweisungteils** von **AUSNA_21** (Zeile 11 bis 23) und
5. sucht im **Ausnahmeteil** von **AUSNA_21** (Zeile 25 bis 26) einen **zuständigen Behandler**.
6. Er **findet einen** (in Zeile 25 bis 26), führt dessen Anweisungsfolge (Zeile 26) aus und **beendet** "ganz normal" den Block AUSNA_21.
7. Zum Schluß gibt er die Meldung "**AUSNA_21 wurde normal beendet!**" aus. **o**

Aufgabe 13.8.: Was passiert im Programm **AUSNA_20**, wenn der Benutzer ein **'A'** oder wenn er ein **'X'** eingibt? Beschreiben Sie die Aktionen des Ausführers, und benutzen Sie dabei möglichst viele Fachbegriffe wie **normaler Anweisungsteil**, **Ausnahmeteil**, (Ausnahme-) **Behandler**, **Wahlliste**, **Anweisungsfolge eines Behandlers** etc.. **o**

Aufgabe 13.9.: Lassen Sie das Programm AUSNA_20 **viermal** von einem maschinellen Ausführer ausführen, und geben Sie (als Benutzer) dabei die Buchstaben **'A'**, **'B'**, **'C'** bzw. **'X'** ein. Notieren Sie sich alle **Ausgaben** des Ausführers, und **erklären** Sie anhand des Programmtextes, "warum gerade diese Ausgaben herauskommen". **o**

Im Grunde genommen funktionieren Ausnahmen in Ada folgendermaßen: Der Ausführer befindet sich in jedem Moment einer Programmausführung entweder im **Normalzustand** oder im **Ausnahmezustand**. Wenn er sich im **Normalzustand** befindet und eine Ausnahme auslöst, geht er in den **Ausnahmezustand** über. Wenn er sich im **Ausnahmezustand** befindet und einen (für die zuletzt ausgelöste Ausnahme zuständigen) Behandler ausführt, geht er damit wieder in den **Normalzustand** zurück. Graphisch kann man diesen einfachen Grundmechanismus etwa so darstellen:



Im Hinblick auf **Ausnahmen** bezeichnet man **Blöcke**, **Prozeduren** und ähnliche Konstrukte (z.B. Funktionen und Pakete, die später behandelt werden) auch als **Rahmen** (frames). Jeder Rahmen hat einen **Anweisungsteil**, der zwischen den reservierten Wörtern **begin** und **end** steht. Im vorigen Beispiel enthält der Rahmen **AUSNA_20** den Rahmen **AUSNA_21** und der enthält seinerseits den Rahmen **AUSNA_22**. Ausnahmen, die im innersten Rahmen **AUSNA_22** ausgelöst werden, können wahlweise an **drei** verschiedenen Orten **behandelt** werden:
1. Im innersten Rahmen AUSNA_22 (wie die Ausnahme DAS_WAR_EIN_A) oder
2. im umgebenden Rahmen AUSNA_21 (wie die Ausnahme DAS_WAR_EIN_B) oder
3. im äußersten Rahmen AUSNA_20 (wie die Ausnahme DAS_WAR_EIN_C).

Allgemein verhält der Ausführer sich **Ausnahmen** gegenüber wie folgt:
1. Wenn in einem **Rahmen** eine **Ausnahme** ausgelöst wird, sucht er in diesem Rahmen nach einem zuständigen Behandler.
2. Wenn er keinen findet, **bricht** er diesen Rahmen **ab** und **propagiert** die Ausnahme, d.h. er löst sie im **umgebenden Rahmen** aus (und macht dann oben bei 1. weiter). Wenn es keinen umgebenden Rahmen (mehr) gibt, wird die Ausführung des ganzen **Programms** mit einer kleinen Fehlermeldung abgebrochen.
3. Wenn er (bei seiner Suche nach einem Behandler in einem Rahmen) einen zuständigen Behandler findet, führt er dessen Anweisungsfolge aus, **beendet** diesen Rahmen und macht dahinter **normal weiter**.

Das folgende Beispielprogramm **AUSNA_30** verhält sich dem Benutzer gegenüber genauso wie das vorige Programm **AUSNA_20**. Es ist intern aber anders strukturiert: Anstelle der geschachtelten Blöcke enthält es zwei **Prozeduren** namens **AUSNA_31** und **AUSNA_32**. Die drei Prozeduren **AUSNA_30**, **AUSNA_31** und **AUSNA_32** sind in folgender Weise miteinander verbunden: Im Anweisungsteil von **AUSNA_30** wird **AUSNA_31** aufgerufen und im Anweisungsteil von **AUSNA_31** wird **AUSNA_32** aufgerufen. Das hat einen ähnlichen Effekt wie die "Verschachtelung" der Blöcke im vorigen Beispiel.

Beispiel 13.6.: Ausnahmen in einem Rahmen auslösen und in verschiedenen Rahmen behandeln:

```
01 with ada.text_io;
02 procedure AUSNA_30 is
03    DAS_WAR_EIN_A: exception;
04    DAS_WAR_EIN_B: exception;
05    DAS_WAR_EIN_C: exception;
06    ------------------------------------------------------------------
07    procedure AUSNA_32 is
08       ZEICHEN1     : character;
09    begin  -- Anweisungsteil von AUSNA_32
10       ada.text_io.put(item => "Bitte geben Sie ein Zeichen ein: ");
11       ada.text_io.get(item => ZEICHEN1);
12       case ZEICHEN1 is
13          when  'A'   => raise DAS_WAR_EIN_A;
14          when  'B'   => raise DAS_WAR_EIN_B;
15          when  'C'   => raise DAS_WAR_EIN_C;
16          when others => null;
17       end case;
```

```
18       ada.text_io.put_line(item => "Normalfall, keine Ausnahme!");
19    exception -- Ausnahmebehandler im Rahmen AUSNA_32
20       when DAS_WAR_EIN_A =>
21          ada.text_io.put_line(item => "Sie haben ein A eingegeben!");
22    end AUSNA_32;
23    ------------------------------------------------------------------
24    procedure AUSNA_31 is
25    begin  -- Anweisungsteil von AUSNA_31
26       AUSNA_32; -- Fuehre die Prozedur AUSNA_32 aus!
27       ada.text_io.put_line(item => "AUSNA_32 wurde normal beendet!");
28    exception -- Ausnahmebehandler im Rahmen AUSNA_31
29       when DAS_WAR_EIN_B =>
30          ada.text_io.put_line(item => "Sie haben ein B eingegeben!");
31    end AUSNA_31;
32    ------------------------------------------------------------------
33 begin -- Anweisungsteil von AUSNA_30
34    AUSNA_31;  -- Fuehre die Prozedur AUSNA_31 aus!
35    ada.text_io.put_line(item => "AUSNA_31 wurde normal beendet!");
36 exception  -- Ausnahmebehandler im Rahmen AUSNA_30
37    when DAS_WAR_EIN_C =>
38       ada.text_io.put_line(item => "Sie haben ein C eingegeben!");
39 end AUSNA_30;
```

Aufgabe 13.10.: Analysieren Sie das Programm **AUSNA_30**, und beantworten Sie dann die folgenden Fragen:

1. Welche Zeilen belegt der **Anweisungsteil** der Prozedur AUSNA_30?
2. Welche Zeilen belegt der **normale Anweisungsteil** von AUSNA_30?
3. Welche Zeilen belegt der **Ausnahmeteil** von AUSNA_30?

4.-6. Ebenso für die Prozedur AUSNA_31.

7.-9. Ebenso für die Prozedur AUSNA_32.

10. Welche Ausnahme(n) werden im Rahmen AUSNA_30 behandelt?
11. Welche Ausnahme(n) werden im Rahmen AUSNA_31 behandelt?
12. Welche Ausnahme(n) werden im Rahmen AUSNA_32 behandelt?

Aufgabe 13.11.: Lassen Sie das Programm AUSNA_30 **viermal** von einem maschinellen Ausführer ausführen, und geben Sie (als Benutzer) dabei die Buchstaben **'A'**, **'B'**, **'C'** bzw. **'X'** ein. Notieren Sie sich alle **Ausgaben** des Ausführers, und **erklären** Sie anhand des Programmtextes, "warum gerade diese Ausgaben herauskommen". **o**

Das Programm **AUSNA_40** (hier im Skript nicht wiedergegeben) verhält sich dem Benutzer gegenüber ganz ähnlich wie **AUSNA_20** und **AUSNA_30**. Es besteht aus den **Unterprogrammen** AUSNA_40, AUSNA_41 und AUSNA_42 sowie dem **Paket** AUSNA_43. Die Unterprogramme AUSNA_41 und AUSNA_42 sind selbständige **Bibliothekseinheiten** (die dem Ausführer unabhängig voneinander übergeben werden können und von ihm in seiner Bibliothek abgelegt werden). Sie sind **nicht** innerhalb von **AUSNA_40** vereinbart (wie **AUSNA_31** und **AUSNA_32** in **AUSNA_30** vereinbart sind). Das Paket **AUSNA_43** ist besonders einfach. Es besteht nur aus einer Spezifikation, die die Vereinbarungen der drei Ausnahmen DAR_WAR_EIN_A, DAS_WAR_-EIN_B und DAS_WAR_EIN_C enthält.

Zusammenfassung 13.:

- Ausnahmen werden von bestimmten Befehlen **ausgelöst** (z.B. von **"+"** oder **get**).
- Mit der **raise**-Anweisung kann der Programmierer Ausnahmen **auslösen**.
- Vier Ausnahmen sind **vordefiniert** (z.B. **constraint_error**), sieben weitere im Paket **ada.text_io vereinbart** (z.B. **data_error**).
- Der Programmierer kann weitere Ausnahmen **vereinbaren**.
- Am Ende eines Anweisungsteils kann man **Ausnahmebehandler** angeben.

- Am Anfang eines Ausnahmebehandlers steht, für welche **Ausnahme**(**n**) er **zuständig** ist.
- Ein **when-others-Behandler** ist für **alle** Ausnahmen zuständig, für die kein anderer Behandler zuständig ist.
- Wird eine Ausnahme **ausgelöst**, dann sucht der Ausführer "**von innen nach außen**" nach einem zuständigen Behandler. Wenn er **keinen** findet, **bricht** er das ganze Programm mit einer Fehlermeldung **ab**. Sonst führt er den Behandler aus und macht ab da "**normal** weiter".

14. Reihungstypen und -untertypen

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type    GROESSE       is range 34..48; -- 15 Schuhgroessen
02 type    ANZAHL        is range  0..100;
03 type    SEMESTER      is (SS90, WS90, SS91, WS91, SS92, WS92, SS93, WS93);
04 package ANZAHL_EA     is new ada.text_io.integer_io(num => ANZAHL);
05 GR1,  GR2   : GROESSE;
06 ANZ1, SUMME : ANZAHL;
07 ALLE_SIND_GERADE, IST_SORTIERT, KEIN_WERT_DOPPELT : boolean;
```

Angenommen, in einem Programm sollen **15 Variablen vom Typ ANZAHL** vereinbart und verarbeitet werden (für jede Schuhgrösse zwischen 34 und 48 eine Variable). Mit den bisher vorgestellten Ada-Befehlen wäre das ziemlich **mühsam**. Der Programmierer müßte 15 verschiedene Namen erfinden und als Bestandteil einer Variablenvereinbarung hinschreiben, z.B. so:

Beispiel 14.1.: Viele Variablen vereinbaren (auf die mühsame Weise):

```
01 declare
02    ANZ_34, ANZ_35, ANZ_36, ANZ_37, ANZ_38 : ANZAHL;
03    ANZ_39, ANZ_40, ANZ_41, ANZ_42, ANZ_43 : ANZAHL;
04    ANZ_44, ANZ_45, ANZ_46, ANZ_47, ANZ_48 : ANZAHL;
```

Um z.B. in jede Variable einen Wert einzulesen, müßte er 15 **get**-Befehle hinschreiben, z.B. so:

```
05 begin
06    ANZAHL_EA.get(item => ANZ_34);
07    ANZAHL_EA.get(item => ANZ_35);
...      ...
20    ANZAHL_EA.get(item => ANZ_48);
o
```

Mit den bisher vorgestellten Ada-Befehlen wäre es nicht möglich, diese 15 Anweisungen durch eine **Schleife** zu ersetzen, deren Rumpf 15 Mal ausgeführt wird. Entsprechendes gilt auch dann, wenn man z.B. jeder Variablen den Wert 0 zuweisen oder den Wert jeder Variablen um 1 erhöhen will etc..

14.1. Eingeschränkte Reihungsuntertypen

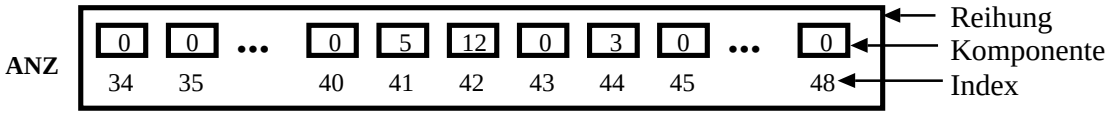
Mit einem **Reihungstyp** (engl. array type, im Deutschen auch Feldtyp genannt) kann der Programmierer die hier skizzierten Probleme mit weniger Befehlen und deutlich eleganter lösen, etwa so:

Beispiel 14.1.2.: Vereinbarung eines **Reihungstyps** und einer **Reihungsvariablen**

```
01 declare
02    type  BESTAND is array(GROESSE) of ANZAHL;
03    ANZ : BESTAND := (41 => 5, 42 => 12, 44 => 3, others => 0);
```

In Zeile 02 wird ein **Reihungstyp** !BESTAND und sein **erster Untertyp** namens **BESTAND** vereinbart. Jede Variable des Untertyps **BESTAND** ist ein Behälter, der 15 Wertebehälter des

Untertyps **ANZAHL** enthält. Die Variable **ANZ**, die in Zeile 03 vereinbart wird, kann man sich bildlich etwa so vorstellen:



Jedes **kleine** Kästchen ist eine Variable vom Untertyp **ANZAHL**, der **große** längliche Kasten ist die Variable namens **ANZ**. Die Variable **ANZ** wird auch als **Reihungsvariable** bezeichnet und die 15 Variablen vom Typ **ANZAHL** (die kleinen Kästchen) als **Komponentenvariablen** oder als **die Komponenten** der Variablen **ANZ**. Eine **Reihung** enthält also "eine **Reihe** von Komponenten". Jede Komponente hat (wie jede Variable) einen **Wert** und (anders als andere Variablen) einen **Index**. In bildlichen Darstellungen wird der **Wert** einer Komponente **in** das betreffende Kästchen eingezeichnet und der **Index** wird **unter** das Kästchen geschrieben. Bei der Reihung **ANZ** gehören die **Werte** der Komponenten zum Untertyp **ANZAHL** und die **Indizes** zum Untertyp **GROESSE**. Die erste Komponente hat den **Index 34** und den **Wert 0**, die Komponente mit dem **Index 41** hat den **Wert 5** etc..

Die Reihungsvariable **ANZ** wird auch als eine **zusammengesetzte Variable** bezeichnet (weil sie aus Komponenten zusammengesetzt ist), und ihr Wert wird als **zusammengesetzter Wert** bezeichnet (weil er aus 15 Werten vom Untertyp ANZAHL zusammengesetzt ist).

Die Angabe **(41 => 5, 42 => 12, 44 => 3, others => 0)** am Ende der Vereinbarung von **ANZ** (siehe oben Zeile 03) ist ein (Reihungs-) **Aggregat**. Es beschreibt den **Anfangswert** der Reihungsvariablen **ANZ**. Dieser Reihungswert ist aus **15** Werten des Untertyps **ANZAHL** zusammengesetzt. **o**

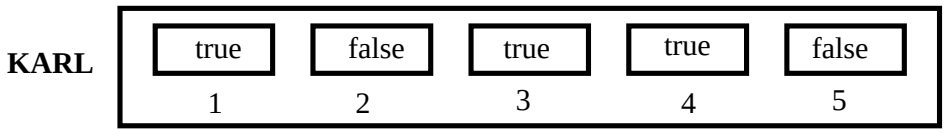
Die Klasse der **Reihungstypen** gehört zur Klasse der **zusammengesetzten Typen** (compound types). Die Klasse der **diskreten Typen** gehört zur Klasse der **einfachen Typen** (simple types). Variablen und Werte eines **zusammengesetzten Typs** haben **Komponenten**, Variablen und Werte eines einfachen Typs haben **keine** Komponenten.

Aufgabe 14.1.1.: Betrachten Sie die folgenden Vereinbarungen:

```
01 declare
02    type JAHR   is range 1995..1999;
03    type BETRAG is range    0..1_000;
04    type PLAN   is array(JAHR) of BETRAG;
05    PLAN_A : PLAN := (1996 => 300; 1999 => 850, others => 500);
```

1. Stellen Sie die Reihung **PLAN_A** bildlich dar (etwa so, wie oben ANZ dargestellt wurde).
2. Von welchem Untertyp sind die Indizes der Reihung **PLAN_A**?
3. Von welchem Untertyp sind die Komponenten der Reihung **PLAN_A**?
4. Von welchem Untertyp ist die Reihung **PLAN_A**? **o**

Aufgabe 14.1.2.: Betrachten Sie die folgende bildliche Darstellung einer Reihung:



Wie könnten die **Vereinbarungen** aussehen, die den Ausführer zum Erzeugen dieser Variablen veranlaßt haben? Bevor Sie die Variable namens KARL vereinbaren können, müssen Sie wohl zwei bis drei Typvereinbarungen hinschreiben (ähnlich wie in der vorigen Aufgabe). **o**

Praktische Merkregel: In der **Vereinbarung eines Reihungstyps** (siehe z.B. oben Zeile 11 oder Zeile 17) steht der **Indexuntertyp** immer **in Klammern** (z.B. **(GROESSE)**) und der **Komponentenuntertyp** steht hinter dem Wort **of** (z.B. **of ANZAHL**).

Der wichtigste Nutzen einer Reihungsvariablen wie **ANZ** hängt mit der **Notation** zusammen, mit der der Programmierer ihre **Komponenten** bezeichnen kann. Der Ausdruck **ANZ(34)** bezeichnet die **erste** Komponente von **ANZ** (d.h. die Komponente mit dem Index 34). Entsprechend bezeichnet **ANZ(35)** die **zweite** Komponente von **ANZ** etc. und **ANZ(48)** die **letzte** Komponente (die mit dem Index 48). Hier ein paar einfache Beispiele dafür, wie der Programmierer solche Bezeichnungen (für Komponenten einer Reihung) in einem Programm verwenden kann:

Beispiel 14.1.3.: Auf **Komponenten** von **ANZ** zugreifen:

```
01 begin
02    ANZ1    := ANZ(41);           -- "Lesender     Zugriff" auf ANZ(41)
03    ANZ(41) := 15;                -- "Schreibender Zugriff" auf ANZ(41)
04    ANZ(48) := ANZ1 + ANZ(41);    -- ANZ(41) lesen, nach ANZ(48) schreiben
05    ANZ(48) := ANZ(48) + 1;       -- ANZ(48) lesen und beschreiben
```
o

Mit der **einen** Vereinbarung

```
03    ANZ : BESTAND := (41 => 5, 42 => 12, 44 => 3, others => 0);
```

vereinbart der Programmierer nicht nur eine **Reihungsvariable** namens **ANZ** vom Reihungsuntertyp **BESTAND**, sondern gleichzeitig auch **15 Komponentenvariablen** namens **ANZ(34)**, **ANZ(35)**, ..., **ANZ(48)**. Die **eine** Vereinbarung in Zeile 03 leistet also mindestens ebensoviel, wie die Vereinbarungen der 15 Variablen im Beispiel 14.1. (siehe ziemlich weit oben).

Auf die Komponenten der Reihungsvariablen **ANZ** kann man nicht nur mit **statischen Indizes** zugreifen (mit Ausdrücken wie **ANZ(34)**, **ANZ(43)** etc.), sondern auch auf **dynamische** Weise mit Ausdrücken wie z.B. **ANZ(GR1)**. Dabei soll **GR1** eine **Variable** des Indexuntertyps **GROESSE** sein (wie am Anfang dieses Abschnitts in Zeile 05 vereinbart). Je nachdem, welchen Wert die Variable **GR1** gerade enthält, bezeichnet der Ausdruck **ANZ(GR1)** mal die eine und mal eine andere Komponente von **ANZ**, wie das folgende Beispiel zeigt:

Beispiel 14.1.4.: Mit **dynamischen Indizes** auf Komponenten von **ANZ** zugreifen:

```
01 begin
02    GR1       := 34;
03    ANZ(GR1)  := ANZ(GR1) + 1;      -- wirkt wie ANZ(34) := ANZ(34) + 1;
04    GR1       := 42;
05    ANZ(GR1)  := ANZ(GR1) + 1;      -- wirkt wie ANZ(42) := ANZ(42) + 1;
```

Der entscheidende Punkt ist hier: Obwohl die Anweisungen in Zeile 03 und 04 exakt **gleich** aussehen, haben sie **verschiedene** Wirkungen. Die beiden **gleichen** Anweisungen bearbeiten **verschiedene** Komponenten der Reihung **ANZ** "auf gleiche Weise". **o**

Im vorigen Beispiel wurde die dynamische Notation **ANZ(GR1)** als Ersatz für die statischen Notationen **ANZ(34)** bzw. **ANZ(42)** verwendet, aber ein wirklicher **Nutzen** der dynamischen Notation war noch nicht erkennbar. Das ist im nächsten Beispiel (hoffentlich) anders:

Beispiel 14.1.5.: Ein **dynamischer** und **nützlicher** Zugriff auf Komponenten von **ANZ**:

```
01 begin
02    for G in GROESSE loop
03       ANZ(G) := ANZ(G) + 1;
04    end loop;
```

Die Anweisung in Zeile 03 wird **15mal** ausgeführt und "macht jedesmal ein bißchen was anderes". Insgesamt wird **jede** Komponente von **ANZ** um 1 erhöht. Die **drei** Zeilen in diesem Beispiel sind also ein "eleganter Ersatz" für **15** verschiedene Zuweisungen. **o**

Aufgabe 14.1.3.: Geben Sie **Anweisungen** an, die den Ausführer dazu veranlassen, in jede Komponente der Reihung **ANZ** eine Anzahl **einzulesen** (mit der Anweisung **ANZAHL_EA.get**). **o**

Aufgabe 14.1.4.: Geben Sie **Anweisungen** an, nach deren Ausführung in der Variablen **SUMME** die Summe aller Komponenten von **ANZ** steht. Eine "Lösung mit roher Gewalt") besteht aus einer einzigen, ziemlich langen Zuweisungsanweisung, etwa so:
SUMME := ANZ(34) + ANZ(35) + ... + ANZ(48).
Können Sie eine elegantere Lösung angeben? **o**

Aufgabe 14.1.5.: Geben Sie **Anweisungen** an, die jede Komponente **ANZ(G)** von **ANZ** wie folgt bearbeiten: Falls der Wert von **ANZ(G)** eine **gerade** Zahl ist (d.h. falls er sich ohne Rest durch 2 teilen läßt), soll dieser Wert **halbiert** werden. Falls **ANZ(G)** dagegen eine **ungerade** Zahl ist, soll sie **um 1 vermindert** werden. Zur Erinnerung: Der Ausdruck **ANZ1 mod 2 = 0** hat genau dann den Wert **true**, wenn die Variable **ANZ1** einen **geraden** Wert hat. **o**

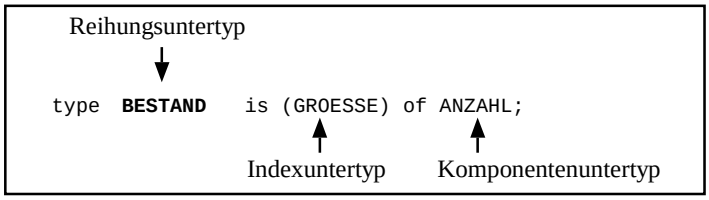
Aufgabe 14.1.6.: Geben Sie **Anweisungen** an, die der Variablen **ALLE_SIND_GERADE** den Wert **true** bzw. **false** zuweisen, je nachdem, ob **alle** Komponenten von ANZ **gerade** (d.h. ohne Rest durch 2 teilbar) sind oder ob mindestens eine Komponente einen **ungeraden** Wert hat (zu **gerade** siehe auch Aufgabe 14.1.4.). **o**

Aufgabe 14.1.7.: Geben Sie **Anweisungen** an, die der Variablen **IST_SORTIERT** den Wert **true** bzw. **false** zuweisen, je nachdem, ob die Komponentenwerte von ANZ **aufsteigend sortiert** sind oder nicht. "Aufsteigend sortiert" heißt hier: Von zwei (benachbarten) Komponenten darf die mit dem **kleineren Index** keinen **größeren Wert** haben, als die mit dem größeren Index. **Hinweis**: Wenn die Variable **G** einen Wert zwischen **GROESSE'first** und **GROESSE'last-1** hat, dann bezeichnen die Ausdrücke **ANZ(G)** und **ANZ(G+1)** zwei nebeneinanderliegende Komponenten der Reihung **ANZ**. **o**

Aufgabe 14.1.8.: Geben Sie **Anweisungen** an, die der Variablen **KEIN_WERT_DOPPELT** den Wert **true** bzw. **false** zuweisen, je nachdem, ob alle Komponenten von ANZ verschiedene Werte haben oder nicht (d.h. der Variablen KEIN_WERT_DOPPELT soll der Wert **true** zugewiesen werden, wenn in ANZ **kein** Komponentenwert **doppelt** vorkommt). **Hinweis**: Zur Lösung dieses

Problems genügt es, wenn man jede Komponente **ANZ(G1)** mit allen Komponenten **ANZ(G2)** vergleicht, die "weiter rechts liegen", für die also **G1 < G2** gilt. **o**

Um einen **Reihungstyp** (und seinen ersten Untertyp) **zu vereinbaren**, braucht man "als Rohmaterial" **zwei Untertypen**: einen für die **Indizes** und einen für die **Komponenten**. Hier noch einmal die Vereinbarung des Reihungstyps !BESTAND und seines ersten Untertyps **BESTAND**:



In dieser **Vereinbarung des Reihungstyps !BESTAND** spielt der Untertyp **GROESSE** die Rolle des **Indexuntertyps** und **ANZAHL** die Rolle des **Komponentenuntertyps**.

!BESTAND ist ein **eindimensionaler** Reihungstyp. Das erkennt man daran, daß in seiner Typvereinbarung in den Klammern nach **array** nur **ein** Indexuntertyp angegeben wurde. Die Vereinbarung eines z.B. **fünfdimensionalen** Reihungstyps könnte z.B. so aussehen:

Beispiel 14.1.6.: Vereinbarung eines **fündimensionalen** Reihungstyps **!BESTAND5**:

```
01 type GRUPPE       is (FRAUEN, MAENNER, KINDER);                --  3 Gruppen
02 type FARBE        is (SCHWARZ, BRAUN, WEISS, BUNT);            --  4 Farben
03 type REGAL        is range  1..3;                              --  3 Regale
04 type ART          is (WANDER, TURN, HAUS, STRASSEN, EDEL);     --  5 Arten
05 type GROESSE      is range 34..48;                             -- 15 Schuhgroessen
06 type ANZAHL       is range  0..100;
07 type BESTAND5     is array(GRUPPE, FARBE, REGAL, ART, GROESSE) of ANZAHL;
```

Jede Reihung des Untertyps **BESTAND5** besteht aus 3 * 4 * 3 * 5 * 15 gleich 2_700 Komponenten des Untertyps **ANZAHL** (weil zu den fünf Indexuntertypen des Untertyps **BESTAND5** genau 3, 4, 3, 5 bzw. 15 Werte gehören). **o**

Mehrdimensionale Reihungen werden erst im Abschnitt 16 genauer behandelt. Die folgende Regel 1 ist aber schon allgemein für Reihungstypen mit **beliebig vielen Dimensionen** formuliert (und nicht nur für **eindimensionale** Reihungstypen).

Regel 1 für Reihungstypen: Jeder **Indextyp** muß ein **diskreter** Typ sein (d.h. ein Ganzzahltyp oder ein Aufzählungstyp).

Regel 2 für Reihungstypen: Der **Komponentenuntertyp** darf **irgendein** Untertyp sein.

Beispiel 14.1.7.: Weitere Vereinbarungen von **Reihungstypen**, ihren ersten **Untertypen** und von **Variablen** dieser Untertypen:

```
01 declare
02    type TEILNEHMER    is array(SEMESTER)  of ANZAHL;
03    type ZEICHEN_MENGE is array(character) of boolean;
04    PR1 : TEILNEHMER := (     SS91 => 43, SS92 => 38,
05                              SS93 => 40, others => 0);
06    PR2 : TEILNEHMER := (     others => 0);
07    PR3 : TEILNEHMER := (     35, 42, 33, 38, 43, 31, 40, 43);
```

```
08    IST_ZIFFER    : ZEICHEN_MENGE := (       '0'..'9' => true, others => false);
09    IST_BUCHSTABE : ZEICHEN_MENGE := (       'A'..'Z' => true,
10                                             'a'..'z' => true, others => false);
```

In dem Aggregat in Zeile 07 werden nur die **Werte** der einzelnen Komponenten, aber keine **Indizes** (und keine Pfeile "=>") angegeben. Diese schlichtere Notation ist manchmal (aber nicht immer) **leichter zu lesen**, als die "volle Notation" **mit** Index und Pfeil.
o

Aufgabe 14.1.9:
1. Wie heißt der **Indexuntertyp** des Reihungsuntertyps TEILNEHMER?
2. Wie heißt der **Komponentenuntertyp** des Reihungsuntertyps TEILNEHMER?
3. **Wieviele Komponenten** hat jede Reihung des Reihungsuntertyps TEILNEHMER?
4. Stellen Sie eine Reihung vom Untertyp **TEILNEHMER** (z.B. die Reihung PR1) **bildlich** dar (durch "Kästchen in einem Kasten", wie oben die Reihung ANZ dargestellt wurde).
5.-8. Lösen Sie die Teilaufgaben 1.-4. auch für den Reihungsuntertyp **ZEICHEN_MENGE**. **o**

Mit **Aggregaten** kann man Werte eines **zusammengesetzten Typs** bezeichnen, so wie man mit **Literalen** Werte eines **einfachen Typs** bezeichnen kann. Mit **Reihungsaggregaten** wie z.B**. (41 => 5, 42 => 12, 44 => 3, others => 0)** oder **('0'..'9' => true, others => false)** kann man Werte eines **Reihungstyps** bezeichnen. Man könnte **Reihungsaggregate** auch **Literale für Reihungstypen** oder kurz **Reihungsliterale** nennen.

Regel 1 für Aggregate: Ein Aggregat muß immer **vollständig** sein, d.h. es muß für **jede** Komponente der betreffenden Reihung einen Wert festlegen (entweder direkt oder zumindest indirekt mit einer Angabe wie **others => 0** oder **others => false** etc.).

Regel 2 für Aggregate: Ein Aggregat muß immer **eindeutig** sein, d.h. es darf für keine Komponente **mehr als einmal** einen Wert festlegen. Damit ist auch verboten, für eine Komponente mehrmals den **gleichen** Wert festzulegen (z.B. so: **(..., 41 => 5, ..., 40..42 => 5, ...)**).

Regel 3 für Aggregate: In einem Aggregat kann man für eine Komponente wahlweise nur einen **Wert** angeben (z.B. **43** oder **true**) oder den **Index** der Komponente, einen Pfeil => und dann den **Wert** (z.B. **SS90 => 43**). Wenn man **keine** Indizes angibt, muß man die Werte "genau in der **richtigen Reihenfolge**" aufführen. Wenn man Indizes angibt, kann man die einzelnen Komponenten in **beliebiger Reihenfolge** beschreiben (nur die Angabe **when others => ...** muß, wenn sie vorhanden ist, immer am **Ende** eines Aggregates stehen).

Regel 4 für Aggregate: Wenn man in einem Aggregat **Indizes angibt**, dann muß man sie durch **statische** Ausdrücke beschreiben, d.h. durch Ausdrücke, deren Wert der Ausführer schon bei der **Übergabe** des Programms ("zur Compilezeit") ermitteln kann (und nicht erst, während er das Programm ausführt). Alle **Literale** wie 0, 17, ..., 'A', 'B', ..., **false**, **true**, ROT, SCHWARZ, ... etc. sind **statische** Ausdrücke. **Variablen** sind **keine** statischen Ausdrücke. Eine genaue und verbindliche Beschreibung aller **statischen Ausdrücke** findet man im (ARM 4.9). **o**

Beispiel 14.1.8.: Richtige und falsche **Reihungsaggregate**:

```
01 declare
02    -- Richtige Aggregate
03    ENGLISCH1 : TEILNEHMER := (     WS90 => 67, WS91 => 58, WS92 => 62,
04                                    WS93 => 71, others => 0);
05    SPANISCH1 : TEILNEHMER := (     0, 67, 0, 58, 0, 62, 0, 71);
```

```
06    RUSSISCH1 : TEILNEHMER := (     others => 0);
07    MATHE1    : TEILNEHMER := (     WS90 | SS93 | WS93 => 25, SS91..WS92 => 31,
08                                    SS90 => 17);
09    MATHE2    : TEILNEHMER := (     17, 25, 31, 31, 31, 31, 25, 25);

10    -- Falsche Aggregate
11    ENGLISCH2 : TEILNEHMER := ( 10, 20, 30);
12    SPANISCH2 : TEILNEHMER := (     SS90..WS91 => 20, WS91..WS93 => 20);
13    RUSSISCH2 : TEILNEHMER := (     43 => 20, SS91 => 25, others => 30);
```

Die Regeln für **Reihungsaggregate** ähneln stark den Regeln für Wahllisten und Wahleinträgen in **case-Anweisungen** (siehe Abschnitt 12.4.).

Die Reihungen **ENGLISCH1** und **SPANISCH1** sind **gleich** (d.h. jede Komponente von **ENGLISCH1** ist gleich der entsprechenden Komponenten von **SPANISCH1**). Die Komponenten der Reihung **RUSSISCH1** haben alle den **Anfangswert 0**. Die Reihungen **MATHE1** und **MATHE2** sind ebenfalls gleich.

Das Aggregat in Zeile 11 ist **unvollständig** (es fehlen Werte für die Komponenten mit den Indizes **WS91** bis **WS93**). Das Aggregat in Zeile 12 ist **nicht eindeutig** (es legt für die Komponente mit dem Index **WS91** zweimal den Wert **20** fest). Das Aggregat in Zeile 13 ist falsch, weil der Wert **43** nicht zum Untertyp SEMESTER gehört. SEMESTER ist der **Indexuntertyp** des **Reihungstyps** !TEILNEHMER. **o**

Aufgabe 14.1.10.: Vereinbaren Sie
1. einen **Ganzzahltyp** !ISADORA, zu dessen erstem Untertyp **ISADORA** die Ganzzahlen im Bereich **0..100** gehören.
2. einen **Ganzzahltyp** !KLAUS, zu dessen erstem Untertyp **KLAUS** die Ganzzahlen im Bereich **-10_000..+10_000** gehören.
3. Einen **Reihungstyp** !REIMUND und seinen ersten Untertyp **REIMUND**, der den Untertyp ISADORA als **Indexuntertyp** und den Untertyp KARL als **Komponentenuntertyp** hat. Zur Erinnerung: In einer Reihungstypvereinbarung steht der Indexuntertyp in **Klammern**, der Komponentenuntertyp steht hinter **of**.
4. zwei **Reihungsvariablen** namens **RV1** und **RV2** vom Untertyp **REIMUND**. Beide Variablen sollen mit Hilfe von Aggregaten initialisiert werden. **o**

Reihungen können nicht nur **komponentenweise** bearbeitet werden, sondern auch "im Ganzen", wie das folgende Beispiel zeigen soll:

Beispiel 14.1.9.: Reihungen im Ganzen bearbeiten:

```
01 begin
02    RUSSISCH1 := (20, 25, 17, 23, 28, 31, 25, 27);
03    SPANISCH1 := RUSSISCH1;
04    if ENGLISCH1 /= SPANISCH1 then ... end if;
05    if MATHE1 = MATHE2 then ... end if;
```

Der Reihung **RUSSISCH1** wird der Wert eines Aggregates zugewiesen. Dieser Wert ist aus **acht** Werten vom Untertyp ANZAHL zusammengesetzt, denn zum Indexuntertyp **SEMESTER** gehören die acht Werte **SS90** bis **WS93**. Der Reihung **SPANISCH1** wird der Wert der Reihung

RUSSISCH1 zugewiesen. Diese **eine** Zuweisung zwischen **Reihungen** leistet das Gleiche wie die folgenden **acht** Zuweisungen zwischen den entsprechenden **Komponenten**:

```
11    RUSSISCH1(SS90) := SPANISCH1(SS90);
12    RUSSISCH1(WS90) := SPANISCH1(WS90);
      ...
18    RUSSISCH1(WS93) := SPANISCH1(WS93);
```

oder wie die folgende Schleife:

```
21    for SEM in SEMESTER loop
22       RUSSISCH1(SEM) := SPANISCH1(SEM);
23    end loop;
```

Der Ausdruck **ENGLISCH1 /= SPANISCH1** (oben in Zeile 04) hat genau dann den Wert **true**, wenn **mindestens eine** Komponente der Reihung ENGLISCH1 **nicht gleich** der **entsprechenden** Komponente von SPANISCH1 ist. Ganz ähnlich hat der Ausdruck **MATHE1 = MATHE2** (oben in Zeile 05) genau dann den Wert **true**, wenn **jede** Komponente der Reihung MATHE1 **gleich** der **entsprechenden** Komponenten von MATHE2 ist. In diesem einfachen Fall **entsprechen** sich zwei Komponenten verschiedener Reihungen, wenn sie den **gleichen Index** haben. **o**

Das vorige Beispiel sollte zeigen: Wenn zwei Reihungen zum selben Typ gehören, dann kann man sie "im Ganzen" einander **zuweisen** und miteinander **vergleichen** und dabei die **gleichen Operationszeichen** ("**:=**", "**=**" und "**/=**") verwenden, mit denen man auch andere Zuweisungen und Vergleiche programmiert (z.B. Zuweisungen an **Ganzzahlvariablen** oder Vergleiche zwischen **Aufzählungswerten**).

In Ada kann man Reihungen also wahlweise **komponentenweise** oder **im Ganzen** bearbeiten. Dazwischen gibt es noch eine **dritte** Möglichkeit, nämlich den Zugriff auf **Teilreihungen** einer Reihung. Z.B. bezeichnet der Ausdruck **RUSSISCH1(WS90..SS92)** eine Teilreihung mit vier Komponenten der Reihung **RUSSISCH1**. Solche Teilreihungen kann man z.B. in Zuweisungen und Vergleichen verwenden, etwa so:

Beispiel 14.1.10.: **Teilreihungen** einer Reihe bearbeiten:

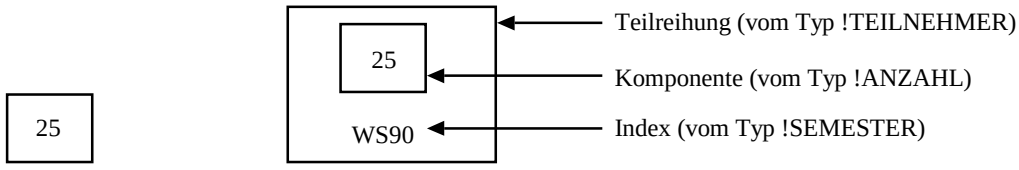
```
01 begin
02    RUSSISCH1(WS90..SS92) := (25, 17, 23, 28);
03    SPANISCH1(WS90..SS92) := RUSSISCH1(WS91..SS93);
04    if ENGLISCH1(WS91..SS93) /= SPANISCH1(WS90..SS92) then ... end if;
05    if MATHE1(SS90..WS90) = MATHE2(SS90..SS91) then ... end if;
```

In einer **Zuweisung** wie in Zeile 02 oder 03 müssen die Teilreihungen (links und rechts vom Zuweisungszeichen ":=") zum selben **Typ** gehören und **gleich lang** sein (d.h. aus **gleich vielen Komponenten** bestehen). Die **Indizes** der beiden Teilreihungen brauchen dagegen **nicht** übereinzustimmen.

In einem **Vergleichsausdruck** (wie in Zeile 04 und 05 jeweils zwischen **if** und **then**) müssen die beteiligten Teilreihungen ebenfalls zum selben **Typ** gehören. Sie müssen aber **nicht** den gleichen Indexbereich haben oder gleich lang sein (siehe Zeile 05). Allerdings gilt: Zwei Teilreihungen unterschiedlicher Länge sind immer **ungleich**, egal, welche Werte ihre Komponenten haben. Verglichen werden jeweils nur **entsprechende Komponenten** (die erste Komponente mit der ersten, die zweite mit der zweiten etc., unabhängig von den konkreten Indizes dieser Komponenten). Zwei Reihungen **gleicher** Länge, aber mit **verschiedenen** Indexbereichen, können also **gleich** sein. **o**

Man beachte die wichtigen **Gemeinsamkeiten** und **Unterschiede** zwischen einer **Reihung**, einer **Teilreihung** und einer **Komponente**:
1. Die **Reihung RUSSISCH1** gehört zum **Typ** !TEILNEHMER **und** zu seinem ersten **Untertyp** TEILNEHMER. Alle Werte des Indexuntertyps SEMESTER sind Indizes der Reihung RUSSISCH1.
2. Eine **Teilreihung** wie **RUSSISCH(WS90..SS92)** gehört zwar auch zum **Typ** !TEILNEHMER, aber **nicht** zu seinem ersten **Untertyp** TEILNEHMER. Denn einige Werte des Indexuntertyps SEMESTER sind **nicht** Indizes dieser Teilreihung (z.B. der Indexwert SS90).
3. Eine **Komponente** wie **RUSSISCH(WS90)** gehört **nicht** zum Reihungs**typ** !TEILNEHMER und schon gar nicht zu seinem ersten **Untertyp** TEILNEHMER, sondern zu dessen **Komponenten-untertyp** ANZAHL (und zu dessen Typ !ANZAHL).

Hier noch ein paar **Extremfälle**: Die Teilreihung **RUSSISCH1(SS90..WS93)**, die man eleganter auch mit dem Ausdruck **RUSSISCH1(SEMESTER'first..SEMESTER'last)** bezeichnen kann, ist eine **unechte Teilreihung**, d.h. sie gehört (anders als die **echten** Teilreihungen) auch zum **Untertyp** TEILNEHMER. Die Teilreihung **RUSSISCH1(WS90..WS90)** ist zwar nur aus einer einzigen Komponenten "zusammengesetzt", gehört aber trotzdem zu einem ganz anderen Typ als die Komponente **RUSSISCH1(WS90)**. Zur Verdeutlichung hier eine bildliche Darstellung:



RUSSISCH1(WS90) (eine Komponente der Reihung RUSSISCH1)

RUSSISCH1(WS90..WS90) (eine Teilreihung der Reihung RUSSISCH1)

Die Komponente **RUSSISCH1(WS90)** und die Teilreihung **RUSSISCH1(WS90..WS90)** sind nicht nur deutlich verschieden voneinander, sondern sogar **unvergleichbar**. Das soll heißen: Ein **Vergleichsausdruck** wie **RUSSISCH1(WS90) = RUSSISCH1(WS90..WS90)** enthält einen **Typenfehler** und wird vom Ausführer (schon bei der Übergabe des betreffenden Programms, "zur Compilezeit") **abgelehnt**. Entsprechendes gilt für **Zuweisungen** wie z.B. **RUSSISCH1(WS90) := RUSSISCH1(WS90..WS90)** oder
RUSSISCH1(WS90..WS90) := ENGLISCH1(SS93) etc..

Hier noch ein letzter, aber besonders wichtiger **Extremfall einer Teilreihung**: Die Teilreihung **RUSSISCH1(WS90..SS90)** ist aus **null** Komponenten zusammengesetzt. Denn der Indexbereich **WS90..SS90** ist ein **leerer Bereich** ("es gibt null Werte des Untertyps SEMESTER, die gleichzeitig größer als **WS90** und kleiner als **SS90** sind"). Solche Teilreihungen werden auch als **leere** (Teil-) Reihungen bezeichnet. Leere (Teil-) Reihungen werden im nächsten Abschnitt (insbesondere im Zusammenhang mit dem vordefinierten Reihungstyp **!string**) noch etwas genauer behandelt. Beim Bearbeiten von Reihungen sind **leere (Teil-) Reihungen** genauso **nutzlos** wie die Zahl 0 beim Rechnen (d.h. sie sind **sehr nützlich**).

Alle in diesem Abschnitt behandelten Reihungsuntertypen (BESTAND, PLAN, TEILNEHMER, ZEICHEN_MENGE und REIMUND) sind **eingeschränkte** Reihungsuntertypen. Das bedeutet z.B. für den Untertyp **BESTAND**: Alle Reihungen dieses Reihungsuntertyps haben genau die gleichen

Indizes (Schuhgrössen von 34 bis 48) und damit auch genau gleich viele Komponenten (nämlich 15).

Für jeden **eingeschränkten Reihungsuntertyp** und für jede **Reihungsvariable** gibt es vier Attribute namens **'first**, **'last**, **'range** und **'length**. Vor das Apostroph muß man jeweils den **Namen** des betreffenden **Untertyps** oder der betreffenden **Reihung** schreiben. Diese Attribute werden hier am Beispiel des Untertyps **TEILNEHMER** und der Variablen **RUSSISCH1** erläutert:

Das Attribut **RUSSISCH1'first** ist eine Konstante vom Untertyp **SEMESTER** und bezeichnet den **ersten Index** der Reihung RUSSISCH1, also den Wert **SS90**.

Ganz entsprechend bezeichnet das Attribut **RUSSISCH1'last** den **letzten Index** der Reihung RUSSISCH1, also den Wert **WS93**.

Das Attribut **RUSSISCH1'range** bezeichnet den **Bereich aller Indizes** der Reihung RUSSISCH1 und hat die gleiche Bedeutung wie **RUSSISCH1'first..RUSSISCH1'last** oder **SEMESTER'first..SEMESTER'last** oder **SS90..WS93**.

Das Attribut **RUSSISCH1'length** bezeichnet die **Länge** der Reihung RUSSISCH1, d.h. die Anzahl ihrer Komponenten, nämlich **acht** (weil zum Indexuntertyp SEMESTER **acht** Werte von **SS90** bis **WS93** gehören). Der **Typ** des Ausdrucks **RUSSISCH1'length** wird, ähnlich wie der Typ des **Literals 8**, automatisch "der Umgebung angepaßt, in der der Programmierer den Ausdruck verwendet".

Hier noch einmal die Attribute der **Reihung** RUSSISCH1 in einer Tabelle zusammengefaßt:

Attribut	**Bedeutung**	**Erläuterung**
RUSSISCH1'first	SS90	Erster Index der Reihung RUSSISCH1
RUSSISCH1'last	WS93	Letzter Index der Reihung RUSSISCH1
RUSSISCH1'range	SS90..WS93	Indexbereich der Reihung RUSSISCH1
RUSSISCH1'length	8	Länge der Reihung RUSSISCH1

Für die Attribute des eingeschränkten **Untertyps** TEILNEHMER gilt ganz entsprechend:

Attribut	**Bedeutung**	**Erläuterung**
TEILNEHMER'first	SS90	Erster Index des Untertyps TEILNEHMER
TEILNEHMER'last	WS93	Letzter Index des Untertyps TEILNEHMER
TEILNEHMER'range	SS90..WS93	Indexbereich des Untertyps TEILNEHMER
TEILNEHMER'length	8	Länge der Reihungen, die zum Untertyp TEILNEHMER gehören

Man beachte aber: Obwohl die **Reihung** RUSSISCH1 und der **Untertyp** TEILNEHMER die **gleichen** Attribute haben, sind Reihungen und Untertypen doch grundsätzlich sehr **verschiedene** "Dinge". Die Vereinbarung eines (Reihungs-) Untertyps sieht **ganz anders** aus, als die Vereinbarung einer Reihung und eine Reihung kann **zu einem Untertyp** gehören, aber ein Untertyp kann nicht zu einer Reihung gehören.

Beispiel 14.1.11.: Mit **Attributen** von Reihungsvariablen die Lesbarkeit erhöhen:

```
01 begin
02    for SEM in RUSSISCH1'range loop
03       RUSSISCH1(SEM) := ENGLISCH1(SEM) + 2;
04    end loop;
05
06    for G in ANZ'first .. ANZ'last - 1 loop
07       ANZ(G) := ANZ(G + 1);
08    end loop;
```

Aus der Angabe **RUSSISCH1'range** in Zeile 02 kann man klar erkennen, daß der Schleifenparameter SEM **alle Indizes der Reihung RUSSISCH1** durchläuft. Hätte der Programmierer stattdessen **for SEM in SEMESTER loop** oder nur **for SEM in SS90..WS93 loop** geschrieben, wäre dieser Zusammenhang zwischen **SEM** und **RUSSISCH1** nicht ganz so offensichtlich und leicht erkennbar. Die Formulierung in Zeile 02 kann aber noch verbessert werden, indem auch der **Untertyp des Schleifenparameters** G ausdrücklich angegeben wird:

```
02a for SEM in SEMESTER range RUSSISCH1'range loop ...
```

Zu Deutsch etwa: Erzeuge eine Variable vom Untertyp **SEMESTER** und weise ihr der Reihe nach jeden Wert aus dem Bereich **RUSSISCH1'range** zu.

In Zeile 06 bis 08 läuft der Schleifenparameter **G** vom **ersten** bis zum **vorletzten** Index der Reihung **ANZ**. Statt **ANZ'last - 1** hätte der Programmierer auch kürzer **47** schreiben können, damit wäre aber nicht so deutlich ausgedrückt, daß der **vorletzte** Index von **ANZ** gemeint ist. Auch in Zeile 06 könnte der Programmierer dem Leser den Untertyp von **G** noch einmal ausdrücklich vor Augen führen und folgendes schreiben:

```
06a for G in GROESSE range ANZ'first .. ANZ'last - 1 loop ...
```
o

Die hier vorgeschlagenen alternativen Formulierungen von Schleifen (Zeile 02a und 06a) mögen auf den ersten Blick kompliziert erscheinen. Vor allem das zweimalige Vorkommen des Wortes **range** in Zeile 02a ist gewöhnungsbedürftig. Aber wenn man sich mit ein bißchen Übung an die ausführlicheren Formulierungen gewöhnt hat, kann man sich damit häufig sehr viel **klarer** ausdrücken, als mit kürzeren Formulierungen.

Aufgabe 14.1.11.: Geben Sie die Bedeutung der Attribute **'first**, **'last**, **'range** und **'length** für die Reihungsuntertypen **BESTAND** (vereinbart im Beispiel 14.1.2.) und **ZEICHEN_MENGE** (vereinbart im Beispiel 14.1.7. in Zeile 03) an. **o**

Man beachte, daß die Attribute **'first**, **'last** und **'range** bei **Reihungs**untertypen und **Reihungen** eine **andere** Bedeutung haben, als bei **skalaren** Untertypen (scalar types, zu denen insbesondere die **diskreten Typen** gehören). Z.B. bezeichnet das Attribut **SEMESTER'first** den ersten **Wert** des diskreten Untertyps **SEMESTER**. Dagegen bezeichnet **TEILNEHMER'first** den ersten **Index** des Reihungsuntertyps **TEILNEHMER**, und nicht den ersten **Wert** dieses Untertyps. Der Index **TEILNEHMER'first** ist zwar auch ein Wert, aber kein Wert des Untertyps **TEILNEHMER**, sondern ein Wert des Indexuntertyps **SEMESTER**. Kurz: Der Wert **SEMESTER'first** gehört zum Untertyp **SEMESTER**, aber der Wert **TEILNEHMER'first** gehört **nicht** zum Untertyp **TEILNEHMER**, sondern zum **Indexuntertyp** von **TEILNEHMER**.

Aufgabe 14.1.12.: **PLAN** ist ein Reihungsuntertyp und **ANZAHL** ein diskreter (und damit auch skalarer) Untertyp. Ergänzen Sie die folgenden beiden Sätze:
PLAN'last ist der letzte ... des Untertyps PLAN.
ANZAHL'last ist der letzte ... des Untertyps ANZAHL.

Das Attribut **'range** hat bei **skalaren** Untertypen (insbesondere also bei **diskreten** Untertypen) **keine** wichtige Bedeutung: **SEMESTER'range** bedeutet das gleiche wie **SEMESTER** (ohne Attribut), nämlich den Bereich aller **Werte** des Untertyps SEMESTER. Z.B. haben die beiden Schleifen **for SEM in SEMESTER'range loop ...** und **for SEM in SEMESTER loop ...** genau die gleiche Bedeutung. Bei einem **Reihungs**untertyp wie TEILNEHMER bedeutet das Attribut **TEILNEHMER'range** dagegen den Bereich aller **Indizes**, und nicht die Menge aller **Werte** des Untertyps TEILNEHMER. In einer Schleife wie **for SEM in TEILNEHMER'range loop ...** durchläuft der Parameter **SEM** alle Indizes von **TEILNEHMER**, aber eine Schleife wie **for T in TEILNEHMER loop ...** ist **nicht erlaubt**. weil **TEILNEHMER** kein diskreter Typ ist.

Das Attribut **'length** gibt es nur für **Reihungsuntertypen** und **Reihungsvariablen**, aber **nicht** für **skalare** Untertypen. Die Ausdrücke **TEILNEHMER'length** und **MATHE1'length** haben beide den Wert **8** (weil Reihungen vom Untertyp **TEILNEHMER** aus 8 Komponenten bestehen), der Ausdruck **SEMESTER'length** ist dagegen verboten.

Da **MATHE1** eine Reihung ist, bezeichnet **MATHE1'first** den ersten **Index** dieser Reihung und mit dem Ausdruck **MATHE1(MATHE1'first)** kann man auf die erste **Komponente** von MATHE1 zugreifen. Den Index **MATHE1'first** kann man aber nicht nur für die Reihung MATHE1 benützen, sondern überall da, wo man den ersten Wert des Untertyps SEMESTER braucht. Z.B. bezeichnet **RUSSISCH1(MATHE1'first)** die erste Komponente der Reihung RUSSISCH1.

Ein **Typ** besteht aus einer **Menge von Werten** und einer **Menge von Operationen**, die man auf diese Werte anwenden kann. Diese grundlegende Erklärung aus dem Glossar des ARM soll hier durch eine zusätzliche **Metapher** (einen Vergleich oder ein Bild) ergänzt werden: Ein **Untertyp** ist ein **Bauplan** für Variablen. Wenn man dem Ausführer befehlen will, eine Variable zu erzeugen, muß man ihm einen **Bauplan** nennen, nach dem er diesen Wertebehälter "bauen" soll. Z.B. ist ein **Ganzzahluntertyp** ein Bauplan, der dem Ausführer sagt, wie "groß" er den Wertebehälter bauen muß, damit auch die größten Werte des Untertyps hineinpassen. Ein (eingeschränkter) **Reihungsuntertyp** ist ein Bauplan, der dem Ausführer sagt, aus **wieviel** Komponenten eine Reihung besteht, welche **Indizes** die einzelnen Komponenten haben und nach welchem **Bauplan** er die einzelnen Komponenten bauen soll (der "Komponentenbauplan" ist also ein wichtiger Teil des "Reihungsbauplans").

Es kann leicht passieren, daß man einen **Untertyp** und **Variablen** dieses Untertyps begrifflich nicht scharf genug trennt und sie ab und zu verwechselt. Die Bauplan-Metapher soll dabei helfen, solche Verwechslungen möglichst zu vermeiden. Der Unterschied zwischen einem **Untertyp** und einer **Variablen** hat Ähnlichkeit mit dem Unterschied zwischen einem **Bauplan für Häuser** und einem **Haus**, welches nach diesem Bauplan gebaut wurde. Ein solcher Bauplan ist typischerweise aus Papier und wiegt höchstens ein paar Kilo, ein Haus besteht eher aus Holzbalken und Steinen und wiegt viele Tonnen. Ähnlich krass ist auch der Unterschied zwischen einem **Untertyp** ("einem Bauplan für Wertebehälter") und einer **Variablen** (einem Wertebehälter) **dieses Untertyps**.

Für **eindimensionale** Reihungen mit Komponenten vom Typ **!boolean** gibt es in Ada noch eine kleine Besonderheit: boolesche Operationen **and**, **or**, **xor** und **not**. Wendet man diese Operationen

auf zwei entsprechende Reihungen (gleicher Länge) an, wird jede **Komponente** der einen Reihung mit der **entsprechenden Komponenten** der anderen Reihung verknüpft, z.B. so:

Beispiel 14.1.12.: Boolesche Operationen für boolesche **Reihungen**:

```
01 declare
02    type ZEICHEN_MENGE is array(character) of boolean;
03    M1 : ZEICHEN_MENGE     := ('A'..'Z' => true,  others => false);
04    M2 : ZEICHEN_MENGE     := ('a'..'z' => true,  others => false);
05    M3 : ZEICHEN_MENGE     := ('0'|'2'|'4'|'6'|'8 ' => true, others => false);
06    M4 : ZEICHEN_MENGE     := ('0'..'9' => false, others => true);
07    M5, M6, M7, M8 : ZEICHEN_MENGE;
08 begin
09    M5 := M1 or ('0'..'9' => true, others => false);
10    M6 := M4 and M5;
11    M7 := not M4;
12    M8 := M7 xor M3;
```

Jede Reihung vom Typ **!ZEICHEN_MENGE** besteht aus 256 Komponenten vom Typ **!boolean** (weil zum Indexuntertyp **character** genau 256 Werte gehören). Weil in der Reihung **M1** die Komponente mit dem Index **'A'** den Wert **true** hat, sagt man auch: Die Menge **M1** enthält das Zeichen **'A'**. Insgesamt enthält die Mange **M1** alle großen Buchstaben, **M2** enthält alle kleinen Buchstaben, **M3** alle geraden Ziffern und **M4** alle Zeichen, die keine Ziffern sind. Nach Ausführung der Zuweisung in Zeile 09 enthält **M5** alle großen Buchstaben und alle Ziffern. Den Wert des Ausdrucks **M4 and M5** (in Zeile 10) berechnet der Ausführer, indem er jede Komponente der Reihung **M4** mit der entsprechenden Komponente der Reihung **M5** mit **and** verknüpft. Für die Ausdrücke in den übrigen Zeilen gilt entsprechendes. **o**

Aufgabe 14.1.13.: Welche Zeichen enthalten die Mengen **M6**, **M7** und **M8** (im vorigen Beispiel) nach Ausführung der Zuweisungen in den Zeilen 10 bis 12? **o**

Aufgabe 14.1.14.: Ein einzelnes Zeichen kann man zu einer Zeichenmenge wie M1 **hinzufügen**, indem man der entsprechenden Komponenten von M1 den Wert **true** zuweist. Geben Sie einen Befehl an, mit dem man das Zeichen **'7'** in die Menge M1 einfügen kann. Geben Sie einen Befehl an, mit dem man das Zeichen **'X'** aus der Menge M1 **entfernen** kann. **o**

Reihungen mit Komponenten vom Typ **!boolean** sind auch sehr nützlich, um **maschinennah** zu programmieren und "mit einzelnen **Bits herumzufummeln**" (siehe Abschnitt XXX).

Vieles, was in diesem Abschnitt über **Variablen** (eines Reihungsuntertyps) gesagt wurde, gilt ganz entsprechend auch für **Konstanten**, wie das folgende Beispiel zumindest andeuten soll:

Beispiel 14.1.13.: Mit **Konstanten** eines Reihungsuntertyps umgehen:

```
01 declare
02    MINIMUM : constant TEILNEHMER := (      SS90 | SS91 | SS92 | SS93 => 4,
03                                            WS90 | WS91 | WS92 | WS93 => 7);
04    MAXIMUM : constant TEILNEHMER := (others => 40);
05 begin
06    RUSSISCH1             := MINIMUM;
07    ENGLISCH1(SS90..WS91) := MAXIMUM(WS90..SS92);
08    if SPANISCH1 = MAXIMUM then ... end if;
09    for SEM in MINIMUM'range loop
10       if ENGLISCH(SEM) < MINIMUM(SEM) then
11          ada.text_io.put(item => "Zu wenig Teilnehmer!");
12       end if;
13    end loop;
```

o

Zusammenfassung 14.1.:

- Ein Reihungsuntertyp ist **eingeschränkt**, wenn alle seine Reihungen (Variablen, Konstanten und Werte) die **gleichen Indizes** (insbesondere also die **gleiche Länge**) haben.
- Indem man **eine** Reihungsvariable vereinbart, vereinbart man im allgemeinen **mehrere** Komponentenvariablen (z.B. 5 oder 50_000 etc.).
- Mit **Reihungsaggregaten** ("Reihungsliteralen") kann man im Prinzip jeden Wert eines Reihungstyps beschreiben.
- Aggregate kann man unter anderem zum **Initialisieren** und **Vergleichen von** sowie in **Zuweisungen an** Reihungsvariablen verwenden.
- Alle (oder einige) Komponenten einer Reihung kann man elegant mit **Schleifen** bearbeiten (was bei "selbständigen Einzelvariablen" nicht möglich ist).
- In Ada gibt es nicht nur Befehle zum Bearbeiten einzelner **Komponenten** einer Reihung, sondern auch Befehle zum Bearbeiten von **Teilreihungen** und von **ganzen Reihungen**.
- Als Extremfall kann eine Teilreihung aus **null** oder aus **allen** Komponenten einer Reihung bestehen (**leere Teilreihung**, **unechte Teilreihung**).
- Einen **Untertyp** (insbesondere einen Reihungsuntertyp) als einen **Bauplan für Variablen** zu verstehen, kann dabei helfen, Untertypen und Variablen klar und anschaulich zu unterscheiden.
- Zu jeder **Reihungsvariablen** und zu jedem **eingeschränkten Reihungsuntertyp** gibt es die Attribute **'first**, **'last**, **'range** und **'length**.
- Zu jedem **Reihungstyp** gehört ein **Indexuntertyp** und ein **Komponentenuntertyp**. Der Indexuntertyp muß **diskret** sein.
- Für jeden **eindimensionalen** Reihungstyp mit dem Komponententyp **boolean** gibt es die logischen Verknüpfungsoperationen **and**, **or**, **xor** und **not**.
- Zur Klasse der **zusammengesetzten** Typen gehören alle Typen, deren Werte und Variablen aus **Komponenten** zusammengesetzt sind. Zur Klasse der **einfachen** Typen gehören alle Typen, deren Werte und Variablen **einfach** (d.h. **nicht** aus Komponenten **zusammengesetzt**) sind.
- Alle **Reihungstypen** sind **zusammengesetzte** Typen. Alle **diskreten** Typen sind **einfache** Typen.

14.2. Uneingeschränkte Reihungsuntertypen

Die Beispiele in diesem Abschnitt gehen von den folgenden Vereinbarungen aus:

```
01 type    GROESSE   is range 34..48; -- 15 Schuhgroessen
02 type    ANZAHL    is range  0..100;
```

Im vorigen Abschnitt ging es um **eingeschränkte** Reihungsuntertypen und in diesem Abschnitt werden **uneingeschränkte** Reihungsuntertypen vorgestellt. Uneingeschränkte Reihungsuntertypen sind besonders nützlich im Zusammenhang mit **Unterprogrammen**, wie im Abschnitt 24. erläutert wird.

Ein **uneingeschränkter Reihungsuntertyp** legt nicht fest, wie **lang** seine Reihungen sind. Er legt zwar einen **Komponentenuntertyp** und einen **Indexuntertyp** fest, aber die einzelnen **Reihungen** des Reihungsuntertyps müssen den Indexuntertyp **nicht** "voll **ausschöpfen**". Jede Reihung kann einen beliebig großen oder kleinen Teilbereich des Indexuntertyps als **ihren** Indexbereich benutzen. Zeit für ein

Beispiel 14.2.1.: Ein Reihungstyp mit einem **uneingeschränkten** ersten **Untertyp**:

```
01 declare
02    type SENDUNG is array(GROESSE range <>) of ANZAHL;
03    DAMEN1  : SENDUNG(34..43);
04    HERREN1 : SENDUNG(38..48);
05    SOMMER1 : SENDUNG(GROESSE'first..GROESSE'last);
06    EILIG   : SENDUNG(42..42);
```

In Zeile 02 wird ein Reihungstyp !SENDUNG und sein erster Untertyp **SENDUNG** vereinbart. Wichtig ist dabei vor allem die Angabe **range** <> (lies: **Reinsch Boks** beziehungsweise/respectively read: **range box**). Diese Angabe macht **SENDUNG** zu einem **uneingeschränkten** Untertyp des Reihungstyps !SENDUNG. Technisch genauer: Der **Untertyp** SENDUNG besteht aus dem **Typ** !SENDUNG und der **leeren Indexeinschränkung**. Die leere Indexeinschränkung "schränkt nicht ein", d.h. sie läßt alle Teilbereiche des Indexuntertyps GROESSE als Indexbereich einer Reihung zu.

Die Reihungsvariable **DAMEN1** hat den Indexbereich **34..43**, sie ist also aus **zehn** Komponenten zusammengesetzt. Die Variable **HERREN1** hat den Indexbereich **38..48** und besteht somit aus **elf** Komponenten. **SOMMER1** enthält **fünfzehn** Komponenten und **EILIG** nur **eine**. **o**

Aufgabe 14.2.1.: Stellen Sie die Reihungen DAMEN1, HERREN1, SOMMER1 und EILIG **bildlich** dar, als "Kästchen in einem Kasten" (zumindest in Ihrem Kopf, besser noch auf einem Blatt Papier). **o**

Eselsbrücke: Eine Box "<>" besteht aus einem **Kleiner**zeichen "<" und einem **Größer**zeichen ">". Die wichtige Angabe **range** <> in Zeile 02 drückt aus, daß die Reihungen des Untertyps SENDUNG nicht alle gleich groß sein müssen, sondern "kleiner oder grösser" sein können.

Das Wort **uneingeschränkt** darf man hier nicht zu wörtlich nehmen. Als Fachbegriff hat es eine etwas andere Bedeutung als "im Alltag". Ein **uneingeschränkter Reihungsuntertyp** ist **nicht** "grenzenlos und frei von jeglichen Einschränkungen". Aber beim Vereinbaren einer Reihung (eines uneingeschränkten Untertyps) ist der Programmierer **nicht** auf einen ganz bestimmten Indexbereich **eingeschränkt**, sondern darf (innerhalb des festgelegten Indexuntertyps) einen Indexbereich frei ("uneingeschränkt") wählen.

Als **Bauplan** aufgefaßt ist ein uneingeschränkter Reihungsuntertyp **nicht ganz vollständig**, denn er legt die **Länge** (und den genauen **Indexbereich**) seiner Reihungen **nicht** fest. Als Analogie kann man sich einen Bauplan für eine Reihe von Reihenhäusern vorstellen. Dieser Bauplan legt zwar genau fest, wie jedes einzelne Haus zu bauen ist (das entspricht dem Komponentenuntertyp) und wie die Hausnummern aussehen werden (das entspricht dem Indexuntertyp), läßt aber offen, aus **wieviel** Häusern eine Reihe dieses Typs besteht und von wo bis wo genau die Hausnummern laufen. Ehe man einem Bauausführer den Auftrag gibt, nach diesem (nicht ganz vollständigen) Plan eine Reihe von Häusern zu bauen, muß man die fehlenden Informationen nachliefern (indem man z.B. die erste und letzte Hausnummer angibt, die Anzahl der Häuser kann der Ausführer dann selbst ausrechnen).

Entsprechend muß man in **jeder** Vereinbarung einer Reihung eines uneingeschränkten Reihungsuntertyps die fehlenden Informationen nachliefern, entweder **explizit** (indem man den ersten und

letzten Index ausdrücklich angibt) oder **implizit** (indem man einen **Anfangswert** für die Reihung angibt, aus dem der Ausführer den genauen Indexbereich **schließen** kann), z.B. so:

Beispiel 14.2.2.: Falsche und richtige Bauaufträge:

```
01 declare
02    HERREN2 : SENDUNG;                                -- Falsch! Beliebter Fehler!
03    DAMEN2  : SENDUNG(36..41);                        -- Richtig! Explizit.
04    DAMEN3  : SENDUNG := DAMEN2;                      -- Richtig! Implizit.
05    DAMEN4  : SENDUNG := (36..40=>5, 41=>3);          -- Richtig! Explizit.
06    DAMEN5  : SENDUNG := (36..40=>5, others=>3);      -- Falsch! Unklares others!
07    DAMEN6  : SENDUNG(36..41) := (others=>3);         -- Richtig! Explizit.
08    DAMEN7  : SENDUNG := (5, 7, 0, 3);                -- Richtig! Implizit.
```

Von der Reihung **HERREN2** kann der Ausführer nicht erkennen, "wie groß er sie bauen soll". Statt eigenmächtig und willkürlich einen bestimmten Indexbereich festzulegen, lehnt er die Vereinbarung in Zeile 02 ab und verlangt vom Programmierer einen genaueren Bauauftrag (d.h. eine eindeutige Vereinbarung).

Die Variablen **DAMEN2** bis **DAMEN4** kann der Ausführer problemlos erzeugen (jede mit dem Indexbereich **36..41**, d.h. aus **sechs** Komponenten zusammengesetzt).

Die Vereinbarung von **DAMEN5** wird abgelehnt, weil nicht klar ist, welche Indizes hier mit **others** gemeint sind. Beim **others** in Zeile 07 hat der Ausführer dagegen **keine** Verständnisprobleme.

Die letzte Vereinbarung (in Zeile 08) ist ein interessanter **Grenzfall**: Der Ausführer kann klar erkennen, daß er die Sendung **DAMEN7** aus **vier** Komponenten zusammensetzen soll. Aber welche **Indizes** sollen die vier Komponenten bekommen? In diesem Grenzfall trifft der Ausführer eine (eigenmächtige, aber "natürliche") Entscheidung und nimmt "die **kleinstmöglichen** Indizes", d.h. den Indexbereich **34..37**. **o**

Aufgabe 14.2.2.: Warum sind die folgenden **Variablenvereinbarungen** falsch?

```
01 HERREN3 : SENDUNG(5);
02 HERREN4 : SENDUNG(34..50);
03 HERREN5 : SENDUNG := (38..40 => 10, 42 => 5);
04 HERREN6 : SENDUNG := (38..40 => 10, 39..42 => 5);
05 HERREN7 : SENDUNG := (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
06 HERREN8 : SENDUNG;
```

o

Mit der **Typvereinbarung** oben im Beispiel 14.2.1. wird ein uneingeschränkter Reihungsuntertyp namens **SENDUNG** als **erster Untertyp** eines Typs **!SENDUNG** vereinbart. Danach kann der Programmierer dann **weitere Untertypen** des selben Typs vereinbaren, z.B. so:

Beispiel 14.2.3.: Mit einem uneingeschränkten Untertyp **weitere Untertypen** vereinbaren:

```
01 declare
02    subtype SHIPMENT           is SENDUNG;               -- Uneingeschraenkt!
03    subtype DAMEN_SCHUHE       is SENDUNG(34..43);       -- Eingeschraenkt!
04    subtype HERREN_SCHUHE      is SENDUNG(38..48);       -- Eingeschraenkt!
05    subtype MAEDCHEN_SCHUHE    is DAMEN_SCHUHE(34..40);  -- Nicht erlaubt!
06    subtype LADIES_SHOES       is DAMEN_SCHUHE;          -- Erlaubt!
07    subtype MAEDCHEN_SCHUHE    is SENDUNG(34..40);       -- Erlaubt!
08    subtype KNABEN_SCHUHE      is SHIPMENT(34..40);      -- Erlaubt!
```

Die Untertypen **SHIPMENT** und **SENDUNG** unterscheiden sich nur durch ihre **Namen** und sind ansonsten **gleich**. Beide bestehen aus dem **Typ** !SENDUNG und der **leeren Indexeinschränkung**. Das bedeutet vor allem: Auch der Untertyp SHIPMENT ist **uneingeschränkt**. Dagegen ist **DAMEN_SCHUHE** ein **eingeschränkter** Untertyp. Er besteht aus dem Typ !SENDUNG und der Indexeinschränkung **34..43**. Jede Reihung des Untertyps **DAMEN_SCHUHE** muß den Indexbereich **34..43** haben (und damit aus **zehn** Komponenten zusammengesetzt sein). Der Untertyp **HERREN_SCHUHE** ist entsprechend eingeschränkt auf den Indexbereich **38..48**. Die Vereinbarung in Zeile 05 ist **nicht** erlaubt, weil man einen eingeschränkten Reihungsuntertyp **nicht** "noch mal einschränken" darf. Dagegen ist die Vereinbarung von **LADIES_SHOES** in Ordnung. Allerdings unterscheiden sich auch die Untertypen **DAMEN_SCHUHE** und **LADIES_SHOES** nur durch ihre Namen (und sind beide gleich eingeschränkt auf den Indexbereich 34..43). Die letzten beiden Vereinbarung (in Zeile 07 und 08) sind erlaubt, weil die Untertypen **SENDUNG** und **SHIPMENT** beide uneingeschränkt sind. Auch die beiden Untertypen **MAEDCHE_SCHUHE** und **KNABEN_SCHUHE** unterscheiden sich nur durch ihre Namen und sind ansonsten gleich. **o**

Allgemein gilt: Jeder **Reihungsuntertyp** besteht aus einem **Reihungstyp** und einer **Indexeinschränkung**. Wenn die Indexeinschränkung **leer** ist, dann ist der Untertyp **uneingeschränkt**, sonst ist er **eingeschränkt**. Jeder Untertyp U ist Untertyp eines **Typs**, und nicht eines Untertyps, egal, wie U vereinbart wurde. Ausgehend von einem **uneingeschränkten Reihungsuntertyp** (z.B. SENDUNG) kann man weitere **uneingeschränkte** Untertypen (wie SHIPMENT) und **eingeschränkte** Untertypen (wie DAMEN_SCHUHE, HERREN_SCHUHE und MAEDCHEN_-SCHUHE) vereinbaren. Ausgehend von einem **eingeschränkten Reihungsuntertyp** (z.B. DAMEN_SCHUHE) kann man nur "weitere Namen für den gleichen Untertyp" vereinbaren (z.B. den Namen LADIES_SHOES), man kann die Einschränkung aber nicht mehr verändern.

Nach so viel "theoretischen Feinheiten" noch etwas ganz Praktisches zum Thema dieses Abschnitts. Es gibt in Ada einen **vordefinierten** uneingeschränkten Reihungsuntertyp, der sehr häufig benutzt wird. Er wurde vom Ausführer aufgrund der folgenden Vereinbarung erzeugt (siehe dazu auch (ARM A.1(37))):

```
01 type string is array(positive range <>) of character;
```

Eine Reihung vom Untertyp **string** wird im Deutschen (bzw. im "Denglischen") häufig einfach als **ein String** bezeichnet. Da **string** ein **uneingeschränkter** Reihungstyp ist, können Strings **verschieden lang** sein.

Zur Erinnerung: Zum vordefinierten Untertyp **positive** gehören alle positiven Werte des (ebenfalls vordefinierten) Untertyps **integer**. In diesem Zusammenhang gilt die Zahl **Null** als **nicht-positiv**. Die Werte von **positive** beginnen also mit **1**. Aus den im Abschnitt 9.4. behandelten Eigenschaften der Untertypen **integer** und **positive** folgt:

1. Wie **lang** Strings **maximal** sein können, wird vom ARM **nicht** festgelegt. Jeder **Ada-Ausführer** darf die Grenzen des Untertyps **positive** und damit die **maximale Stringlänge** selbst festlegen.
2. Ein Ada-Ausführer darf die maximale Stringlänge aber **nicht unterhalb** von 2**15 - 1 (also unterhalb von 32_767) festlegen.
3. Viele Ada-Ausführer legen die maximale Stringlänge wesentlich höher fest, häufig bei 2**31 - 1. Theoretisch können solche Ada-Ausführer Strings erzeugen und bearbeiten, die aus etwas mehr als **zwei Milliarden Komponente**n (Zeichen vom Untertyp **character**) bestehen. Ein solcher String würde auf einem heute üblichen Computer etwa **2 Gigabyte Hauptspeicher** belegen. Da heute viele Computer mit wesentlich weniger als 2 Gigabyte Hauptspeicher ausgerüstet werden, sind für

die maximale Stringlänge häufig nicht die Indexgrenzen des Untertyps **string** maßgebend, sondern der verfügbare Hauptspeicher.

Für den vordefinierten uneingeschränkten Reihungsuntertyp **string** gilt zunächst einmal alles, was für andere uneingeschränkte Reihungsuntertypen auch gilt. Variablen vom Untertyp **string** kann man also z.B. so vereinbaren:

Beispiel 14.2.4.: Variablen vom Untertyp **string** vereinbaren (auf die harte Tour):

```
01 declare
02    TEXT1 : string(1..10);
03    ROMAN : string(1..800_000);
04    HALLO : string := ('H', 'a', 'l', 'l', 'o', '!');
05    XXXXX : string(101..200) := (others => 'X');
06    ERROR : string; -- Wird vom Ausfuehrer abgelehnt!!!
```

Die Reihung **TEXT1** ist aus zehn Komponenten vom Untertyp **character** zusammengesetzt. In der Reihung **ROMAN** ist Platz für genau 800_000 Zeichen (das entspricht ungefähr 400 Schreibmaschinenseiten). Die Reihung **HALLO** hat den Indexbereich **1..6** ("die kleinstmöglichen Indizes"), die erste Komponente hat den Wert **'H'** und die letzte den Wert **'!'**. Wenn der Ausführer den String mit dem schwer aussprechbaren Namen **XXXXX** erzeugt, initialisiert er alle 100 Komponenten mit dem **character**-Wert 'X'. Da der Ausführer nicht klar erkennen kann, aus wieviel Komponenten er die Variable **ERROR** zusammensetzen soll (Aus einer? Aus 1_000? Oder aus zwei Milliarden Komponenten?), lehnt er die Vereinbarung in Zeile 06 ab. **o**

Die Vereinbarung der Variablen HALLO sollte zeigen, daß man mit üblichen Reihungsaggregaten **theoretisch** auch "Texte und Meldungen" beschreiben kann, daß das aber ziemlich **unpraktisch** ist. Deshalb gibt es in Ada hauptsächlich für den Untertyp **string** eine **besondere Notation** zur Beschreibung von Reihungswerten: **Zeichenkettenliterale**.

Beispiel 14.2.5.: Zeichenkettenliterale

```
01 declare
02    HALLO1 : string            := "Hallo!";
03    HALLO2 : string(77..82)    := "Hallo!";
04    TEXT2  : string            := "Dies ist ein laengerer Text, der "  &
05                                  "durch mehrere konkatenierte "       &
06                                  "Zeichenkettenliterale beschrieben wird!";
```

Die Variablen **HALLO1** und **HALLO2** haben beide den gleichen Anfangswert wie die Variable **HALLO** im vorigen Beispiel. **HALLO2** hat allerdings einen anderen Indexbereich als **HALLO** und **HALLO1**. Die Reihung **TEXT2** besteht aus genau ..., also aus ziemlich vielen Komponenten vom Untertyp **character**. Das Attribut **TEXT2'first** hat den Wert 1. **o**

Wenn der Programmierer vorhat, bestimmte Texte **nie zu ändern**, dann sollte er sie nicht als **Variablen** sondern als **Konstanten** vereinbaren. Dadurch macht er **versehentliche** Änderungsversuche zu formalen Fehlern (die der Ausführer entdeckt) und macht seinen Kollegen deutlich, daß die Texte sich im Verlauf einer Programmausführung garantiert nicht verändern.

Beispiel 14.2.6.: Konstanten vom Untertyp **string**:

```
01 declare
02    TEXT3 : constant string := "You are leaving the American sector!";
03    TEXT4 : constant string := "Die Erde ist eine flache Scheibe!";
o
```

Einlesen und **ausgeben** kann man Werte vom Untertyp **string** mit Prozeduren namens **get** und **put** aus dem Standardpaket **ada.text_io**, z.B. so:

Beispiel 14.2.7.: Strings **ausgeben** und **einlesen**:

```
01 begin
02    ada.text_io.put(item => TEXT3 & " Right now!";
03    ada.text_io.get(item => TEXT1);
```

Im **put**-Befehl darf man als **item**-Parameter einen **beliebigen Ausdruck** angeben. Der Wert dieses Ausdrucks wird ausgegeben. Im **get**-Befehl muß man als **item**-Parameter eine **Variable** angeben. In diese Variable wird ein **string**-Wert "hineingelesen". Der **get**-Befehl hat allerdings einen kleinen Haken: Er entspricht der Anweisung "Einmal voll!" an einer Tankstelle, wobei der **item**-Parameter **TEXT1** den **Tank** darstellt. Der Ausführer liest genau **TEXT1'length** viele Zeichen in die Reihung **TEXT1**. Solange der Benutzer noch nicht genug Zeichen eingegeben hat, wartet der Ausführer geduldig (und läßt sich auch durch wiederholtes und hartes Drücken der Returntaste nicht aus der Ruhe bringen). Für den **get**-Befehl in Zeile 03 muß der Benutzer genau **zehn** Zeichen eingeben (weil **TEXT1** mit dem Indexbereich **1..10** vereinbart wurde). Falls der Benutzer **mehr** eingibt, gehen die überzähligen Zeichen **nicht** verloren, sondern bleiben im **Eingabepuffer des Betriebssystems** und können mit weiteren **get**-Befehlen dort abgeholt werden. **o**

Damit der **Benutzer** die Länge eines Eingabestrings bestimmen kann, sollte der Programmierer anstelle der **starren** Prozedur **get** die **flexiblere** Prozedur **get_line** aufrufen, z.B. so:

Beispiel 14.2.8.: Mit **get_line** einen String (fast) beliebiger Länge einlesen:

```
01 declare
02    EIN : string(101..600);     -- Indexbereich 101..600, wird unten erlaeutert!
03    LBI : natural;              -- Letzter belegter Index von EINGABE
04 begin
05    ada.text_io.put_line(item => "Bitte geben Sie Ihren Namen ein: ");
06    ada.text:io.get_line(item =>  EINGABE, last => LBI);
```

Die Prozedur **get_line** hat außer dem üblichen **item**-Parameter noch einen **last**-Parameter, für den man eine Variable vom Untertyp **natural** angeben muß. Im Beispiel liest der Ausführer solange Zeichen nach **EIN**, bis der Benutzer auf die Returntaste drückt. Dann weist der Ausführer der Variablen **LBI** noch den Index **der** Komponenten von **EIN** zu, in die er das letzte Zeichen eingelesen hat. Nach Ausführung des **get_line**-Befehls bezeichnet der Ausdruck **EIN(EIN'first..LBI)** also genau **die** Teilreihung von **EIN**, in die der Ausführer Zeichen eingelesen hat. Die übrigen Komponenten von **EIN** werden durch die **get_line**-Prozedur nicht verändert.

Sonderfälle: 1. Wenn der Benutzer **mehr** Zeichen eingibt, als in **EIN** Platz haben, "füllt der Ausführer die Variable **EIN** voll", läßt die überzähligen Zeichen im Puffer des Betriebssystems stehen und weist der Variablen **LBI** den Wert **EIN'last** zu. 2. Wenn der Benutzer **null** Zeichen eingibt und gleich auf die Returntaste drückt, wird der Variablen **LBI** der Wert **EIN'first - 1** zugewiesen. Falls der erste Index von **EIN** gleich 1 wäre, bekäme **LBI** also den Wert 0. Deshalb sollte eine solche Variable im allgemeinen zum Untertyp **natural** gehören ("der bei 0 anfängt") und nicht zum Untertyp **positive** ("der erst bei 1 anfängt").

Man beachte, daß der Ausführer nicht die **Länge** der eingelesenen Zeichenkette nach **LBI** bringt, sondern den "**letzten belegten Index**" von **EIN**. Wenn der Benutzer z.B. die **drei** Zeichen "ABC" eingibt, steht anschließend in **LBI** die Zahl **103** (nicht die Zahl **3**!). Um diesen Punkt deutlich zu machen, wurde die Variable **EIN** mit einem Indexbereich von **101..600** vereinbart (und nicht mit dem Indexbereich **1..500**). Der **letzte Index** eines Strings und die **Länge** des Strings stimmen zwar manchmal "zufällig" überein, sind aber konzeptionell ("von der Idee her") immer **völlig verschieden**. **o**

Die **get_line**-Prozedur im vorigen Beispiel ist ein erstes Beispiel dafür, wie nützlich die **leere Zeichenkette** und **leere** (Index-) **Bereiche** sind. Wenn der Benutzer **keine** Zeichen eingibt, sondern gleich auf die Returntaste drückt, ist **EIN'first..LBI** ein leerer Bereich (weil **LBI** dann kleiner ist als **EIN'first**) und der Ausdruck **EIN(EIN'first..LBI)** bezeichnet die leere Zeichenkette. Genau die hat der Benutzer ja auch eingegeben.

Wenn der Ausführer eine Variable vom Typ **string** erzeugt hat, dann liegen die Indexgrenzen dieses Strings (und damit seine Länge) fest und können nicht mehr geändert werden. Deshalb werden **string**-Variablen auch als **Strings fester Länge** (fixed-length strings) bezeichnet. Zur Bearbeitung solcher Strings findet man im Paket **ada.strings.fixed** eine Reihe nützlicher Unterprogramme (siehe (ARM A.4.3)).

Unter einem **flexiblen String begrenzter Länge** (bounded length string) versteht man einen String, dessen Länge innerhalb gewisser Grenzen (z.B. bis 500 oder bis 1000 Zeichen) **veränderbar** ist. Realisiert wird ein solche String in Ada meistens durch einen String fester und **maximaler** Länge und eine zusätzliche Ganzzahlvariable, in der die **aktuelle** Länge des Strings steht. Im Paket **ada.strings.bounded** findet man alles Nötige, um solche Strings zu vereinbaren und zu bearbeiten. Siehe dazu (ARM A.4.4).

Anmerkung: Die **aktuelle Länge** eines flexiblen Stings begrenzter Länge kann (statt durch eine zusätzliche Ganzzahlvariable) auch dadurch dargestellt werden, daß der aktuell belegte Teil des Strings durch ein **bestimmtes Zeichen abgeschlossen** wird. Alle Zeichen rechts von diesem Abschlusszeichen gehören dann nicht mehr zum String. Als Abschlusszeichen wird meistens das sogenannte **Nullzeichen** verwendet und die Strings werden auch als **nullbegrenzte Strings** (null terminated strings) bezeichnet. In der Programmiersprache C werden Strings auf diese Weise realisiert.

Aufgabe 14.2.3.: Diskutieren Sie die **Vor**- und **Nachteile** der beiden hier skizzierten Realisierungen eines **flexiblen Strings fester Länge** (mit zusätzlicher Ganzzahlvariablen bzw. als nullbegrenzter String). **o**

Als **flexible Strings unbegrenzter Länge** (unbounded-length strings) bezeichnet man solche Strings, deren Länge **veränderbar** und nur durch den Wert des Attributs **positive'last** (größter Wert des Indexuntertyps) begrenzt ist. Typischerweise können solche Strings zumindest theoretisch bis zu 2 Gigabyte lang sein (falls der Hauptspeicher des verwendeten Rechners entsprechend groß ist). Im Paket **ada.strings.unbounded** findet man alles Nötige, um solche Strings zu vereinbaren und bearbeiten. Siehe dazu (ARM A.4.5).

Wenn nicht ausdrücklich anders gesagt ist im folgenden mit einem **String** immer ein String des vordefinierten Typs **!string** (d.h. ein String **fester** Länge) gemeint und nicht ein flexibler String begrenzter oder unbegrenzter Länge.

Aufgabe 14.2.4.: Schreiben Sie ein Programm namens **HALLO_30**, welches den Benutzer zum Eingeben seines Namens auffordert, einen String beliebiger Länge (maximal 30 Zeichen) einliest (z.B. den String "Carola" oder "R2D2") und schließlich eine freundliche Begrüßung ausgibt, in der der eingelesene String vorkommt (z.B. "Hallo Carola, wie geht es?"). **o**

Aufgabe 14.2.5.: Schreiben Sie ein Programm namens **REIHE_05**, welches **drei** Zeichenketten beliebiger Länge (bis maximal 20 Zeichen) einliest und sie durch senkrechte Striche '|' getrennt wieder ausgibt. Wenn der Benutzer also z.B. die **drei** Zeichenketten "ABC", "??" und "123456" eingibt, soll die **eine** Zeichenkette "ABC|??|123456" ausgegeben werden. **o**

Aufgabe 14.2.6.: Schreiben Sie ein Programm namens **REIHE_06**, welches eine Zeichenkette beliebiger Länge (maximal 60 Zeichen) einliest, die Anzahl der Buchstaben in dieser Zeichenkette ermittelt und diese Anzahl ausgibt. **o**

Der vordefinierte uneingeschränkte Reihungsuntertyp **wide_string** wurde vom Ausführer aufgrund der folgenden Vereinbarung erzeugt (siehe (ARM A.1(41))) :

02 type **wide_string** is array(**positive** range <>) of **wide_character**;

Zum Untertyp **wide_character** gehören **2******16** Zeichenwerte (das sind 65_536 Werte). Prozeduren zum Einlesen und Ausgeben von Zeichenketten des Untertyps **wide_string** befinden sich im Standardpaket **ada.wide_text_io** (siehe dazu auch (ARM 11.2)). Nähere Einzelheiten zum Untertyp **wide_character** findet man im (ARM 3.5.2(3) und A.1(36)) und zum Untertyp **wide_string** im (ARM 3.6.3(4) und A.1(41)).

Für die vordefinierten Reihungsuntertypen **string** und **wide_string** gelten in Ada die **gleichen Regeln**, wie für jeden anderen (vom Programmierer vereinbarten) uneingeschränkten Reihungsuntertyp auch. Das hat Vorteile in **zwei Richtungen**: Alles, was man über uneingeschränkte Reihungsuntertypen im allgemeinen lernt, gilt auch für die speziellen Untertypen **string** und **wide_string**, und alle Erfahrungen, die man mit **Strings** sammelt, kann man auch auf Reihungen anderer uneingeschränkter Reihungsuntertypen übertragen.

Um ehrlich zu sein: Die Untertypen **string** und **wide_string** haben ein paar "kleine Privilegien", die andere Reihungsuntertypen **nicht** haben. Zu diesen **string-Privilegien** gehören **Zeichenkettenliterale** (wie z.B. "Hallo!") und die Prozeduren **get**, **put** und **put_line** im Paket **ada.text_io** bzw. **ada.wide_text_io** (für andere Reihungsuntertypen muß der Programmierer entsprechende Ein-/Ausgabeprozeduren selbst programmieren, siehe dazu Abschnitt 20). Das Privileg der **Zeichenkettenliterale** teilen sich die Untertypen **string** und **wide_string** allerdings mit allen vom Programmierer vereinbarten **Zeichenkettentypen** (siehe dazu Abschnitt 14.3.). **o**

Für jede **Reihung** (Variable oder Konstante) gibt es die Attribute '**first**, '**last**, '**range** und '**length** mit den Bedeutungen **erster Index**, **letzter Index**, **Indexbereich** bzw. **Länge**. Das gilt auch für Reihungen, die mit Hilfe eines **uneingeschränkten** Reihungsuntertyps wie **SENDUNG** oder **string** vereinbart wurden (siehe dazu auch das Beispiel 14.2.1.).

Beispiel 14.2.9.: Reihungsattribute:

Die Reihung **EIN** wurde im vorigen Beispiel vereinbart. Der Ausdruck **EIN'first** hat den Wert **101**, **EIN'last** ist gleich **600**, **EIN'range** bezeichnet den Indexbereich **101..600** und **EIN'length** ist gleich **500**. **o**

Die Attribute **'first**, **'last**, **'range** und **'length** stehen aber für **uneingeschränkte** Reihungsuntertypen **nicht** zur Verfügung. Ausdrücke wie **SENDUNG'first**, **SENDUNG'last**, **string'range**, **wide_string'length** etc. sind in Ada-Programmen **nicht** erlaubt.

Man beachte, daß nur (Reihungs-) **Untertypen** uneingeschränkt sein könne, konkrete **Reihungen** dagegen ausnahmslos **eingeschränkt** sind. Praktisch bedeutet das: Nachdem der Ausführer eine Reihung erzeugt hat, kann er ihre Länge nicht mehr ändern. Man kann also auch Reihungen wie **DAMEN1** und **HERREN1** (siehe Zeile 11 bis 12 am Anfang dieses Abschnitts), die mit Hilfe eines uneingeschränkten Reihungstyps wie **SENDUNG** vereinbart wurden, nach ihrer Erzeugung nicht mehr verlängern oder verkürzen.

Andererseits sind Reihungen in Ada keine völlig **statischen** Gebilde, deren Längen schon beim Schreiben eines Programms festgelegt werden müssen. Man kann z.B. die Länge einer Reihung einlesen, muß das allerdings erledigen, bevor die Reihung **erzeugt** wird, z.B. so:

Beispiel 14.2.10.: Die Länge einer Reihung einlesen und dann die Reihung erzeugen lassen:

```
01 B1: declare
02    LAENGE : natural;
03    package NATURAL_EA is new ada.text_io.integer_io(num => natural);
04 begin -- B1
05    ada.text_io.put(item => "Wie lang?");
06    NATURAL_EA.get(item => LAENGE);
07    B2: declare
08       TEXT: string(1..LAENGE) := (others => '?');
09    begin -- B2
10       ada.text_io.put_line(item => TEXT);
11       ...
12    end B2;
13 end B1;
```

Im äußeren Block **B1** wird die Länge eingelesen (Zeile 06) und im inneren Block **B2** wird die Reihungsvariable **TEXT** mit einem entsprechenden Indexbereich **vereinbart** (Zeile 08). Die Rolle des äußeren Blocks B1 könnte auch durch eine umgebende **Prozedur** oder andere Programmeinheit übernommen werden. **o**

Wenn man einer **string**-Variablen einen Wert **zuweist**, dann muß der Wert **gleich lang** sein wie die Variable, d.h. der **Wert** muß aus **gleich vielen** Komponenten zusammengesetzt sein, wie die **Variable**. Längenunterschiede kann man mit Hilfe der Teilreihungsnotation beseitigen, z.B. so:

Beispiel 14.2.11.: Zuweisungen zwischen Reihungen unterschiedlicher Länge:

```
01 declare
02    T1 : string(1..100) := (others => 'X');
03    T2 : string(1..3)   := (others => 'Y');
04 begin
05    T1                                    := T2;          -- falsch!
06    T2                                    := T1;          -- falsch!
07    T1(98..100)                           := T2;          -- richtig!
08    T1(T1'first..T1'first+T2'length-1)    := T2;          -- richtig schoen!
09    T2                                    := T1(17..19);  -- richtig!
10    T2(2..3)                              := T1(33..34);  -- richtig!
11    ...
```

Die Zuweisung in Zeile 05 wird **abgelehnt**, weil es **viele Möglichkeiten** gibt, den kurzen String **T2** in den langen String **T1** zu kopieren. Es würde dem Stil der Sprache Ada widersprechen, wenn der **Ausführer** eine dieser Möglichkeiten auswählen würde. Statt dessen wird vom Programmierer verlangt, daß er klar und eindeutig ausdrückt, was er will. Ganz entsprechend wird die Zuweisung in Zeile 06 vom Ausführer abgelehnt. Die Zuweisung in Zeile 07 bewirkt, daß **T2** an das Ende von **T1** kopiert wird. Die Zuweisung in Zeile 08 mag auf den ersten Blick kompliziert erscheinen, drückt aber klar aus, daß **T2** an den Anfang von **T1** kopiert werden soll. Wenn die Länge von **T2** eines Tages geändert wird (z.B. von einem Kollegen des Programmierers), dann muß die Zuweisung in Zeile 07 von Hand angepaßt werden, die in Zeile 08 aber nicht. **o**

Aufgabe 14.2.7.: Wie könnte man die Zuweisung in Zeile 07 des vorigen Beispiels **änderungsfreundlicher** formulieren? **o**

Was im vorigen Beispiel (14.2.11.) über **Strings** gesagt wurde, gilt ganz genau so auch für **andere Reihungen**.

Häufig möchte man von zwei Strings nicht nur herausfinden, ob sie **gleich** sind oder nicht, sondern ob der eine **kleiner** ist als der andere (oder kleinergleich, größer oder größergleich). Die Vergleichsoperationen "<", "<=", ">" und ">=" für Werte vom Typ **!string** funktionieren in Ada entsprechend der **lexikografischen Ordnung** von Strings. Z.B. ist der String "ABC" **kleiner** als der String "BC", weil in einem Lexikon "ABC" **vor** "BA" stehen müßte. Viele Menschen beherrschen die lexikografische Ordnung (und wenden sie jedesmal an, wenn sie z.B. ein Wort in einem Lexikon nachschlagen), ohne ihren Namen zu kennen. Zwischen der lexikografischen Ordnung von Strings und der ebenso grundlegenden Ordnung von Ganzzahlen (0 ist kleiner als 1 und 1 ist kleiner als 2 etc.) gibt es grundlegende und interessante Unterschiede. Z.B. liegen zwischen den Ganzzahlen 1 und 5 genau **drei** weitere Ganzzahlen (2, 3 und 4). Zwischen dem String **"A"** und dem String **"B"** liegen dagegen theoretisch **unendliche viele** und praktisch immerhin noch **sehr, sehr viele** weitere Strings, z.B. "AA", "AAA", "AAAA", ... etc., aber auch "AB", "ABA", "ABAA", "ABAAA", ... etc. und sehr, sehr viele mehr.

Aufgabe 14.2.8.: In Ada gehören zum Untertyp **character** genau 256 Zeichen. **Wieviele** verschiedene Strings der Länge 10_000 gibt es ungefähr? Auf ein paar Millionen kommt es dabei nicht an. **o**

Aufgabe 14.2.9.: In welcher **Reihenfolge** würden die folgenden Strings in einem Lexikon stehen: "BBB", "AAA", "B", "BA", "AAAAAAAABA", "AAAAAAAAAA", "A", "BAB" ? **o**

Aufgabe 14.2.10.: Angenommen, Sie kennen die Reihenfolge aller **character**-Werte (d.h. von zwei beliebigen verschiedenen **character**-Werten wissen Sie, welcher größer und welcher kleiner ist). Wie können Sie dann von zwei beliebigen **Strings** S1 und S2 feststellen, welcher von beiden in einem Lexikon vor dem anderen stehen müßte? Geben Sie "Schritt für Schritt" genau an, wie Sie vorgehen würden. **o**

Aufgabe 14.2.11.: Schreiben Sie ein Programm namens **REIHU_03**, welches 1. zwei Strings S1 und S2 (beliebiger Länge, maximal je 20 Zeichen) einliest, 2. feststellt, ob S1 kleiner, gleich oder größer S2 ist und 3. eine entsprechende Meldung ausgibt. **o**

Alle bisher vorgestellten Reihungstypen und Reihungen sind **eindimensional**. Mehrdimensionale Reihungen werden im folgenden Abschnitt 16 behandelt. **Vergleichsoperationen** namens "=" und

"/=" ("gleich" und "ungleich") gibt es in Ada für (fast) **jeden Typ**, insbesondere für jeden **Reihungstyp**. Vergleichsoperationen namens "<", "<=", ">" und ">=" gibt es für jeden **diskreten Typ**, für jeden **Bruchzahltyp** (siehe Abschnitt 21.) und für jeden **diskreten Reihungstyp**. Ein **Reihungstyp** ist **diskret**, wenn er **eindimensional** ist und einen **diskreten Komponentenuntertyp** besitzt. Alle bisher vorgestellten Reihungstypen waren diskrete Reihungstypen. Erste Beispiele für nicht-diskrete Reihungstypen werden im nächsten Abschnitt 16 beschrieben.

Der **Konkatenationsoperator** "**&**" für **Strings** kam schon im ersten Beispielprogramm HALLO_01 vor. Er gehört aber nicht zu den "Privilegien" des Typs **!string**. Vielmehr gibt es für jeden **eindimensionalen Reihungstyp** !T mit Komponentenuntertyp K **vier** Konkatenationsoperationen namens "**&**", mit denen man 1. zwei !T-Werte oder 2. einen !T-Wert und einen K-Wert oder 3. einen K-Wert und einen !T-Wert oder 4. zwei K-Werte jeweils zu **einem** !T-Wert verknüpfen kann.

Beispiel 14.2.12.: Verschiedene **Konkatenationsoperatoren** namens "**&**" anwenden:

```
01 declare
02    C1 : character          := 'B';
03    T1 : string(1..2)       :=  C1 & 'C';    -- character, character -> string
04    T2 : string(1..3)       := 'A' &  T1;    -- character, string    -> string
05    T3 : string(5..7)    :=  T1 & 'A';       -- string,    character -> string
06    T4 : string(1..5)       :=  T2 & "DE";   -- string,    string    -> string
07    A1 : ANZAHL             :=  7;
08    S1 : SENDUNG(34..35)    :=  A1 & 15;     -- ANZAHL,    ANZAHL    -> SENDUNG
09    S2 : SENDUNG(37..39)    :=  S1 & A1;     -- SENDUNG,   ANZAHL    -> SENDUNG
10    S3 : SENDUNG(37..39) :=  A1 & S1;        -- ANZAHL,    SENDUNG   -> SENDUNG
11    S4 : SENDUNG(41..45)    :=  S1 & S2;     -- SENDUNG,   SENDUNG   -> SENDUNG
```

Die Kommentare geben jeweils die Untertypen der verknüpften Werte und den Untertyp des Verknüpfungsergebnisses an. Z.B. werden in Zeile 03 zwei **character**-Werte zu einem **string**-Wert verknüpft und in Zeile 10 ein **ANZAHL**-Wert und ein **SENDUNG**-Wert zu einem **SENDUNG**-Wert. Die **string**-Variable **T4** bekommt den Anfangswert **"ABCDE"** und die **SENDUNG**-Variable **S4** den Anfangswert **(7, 15, 7, 15, 7)**. **o**

Ganz ähnlich wie die Rechenoperationen namens "**+**", "**-**", "*****", "/" etc. jeweils zu einem bestimmten **Typ** gehören (und nicht nur zu irgendeinem **Untertyp** dieses Typs), gehören auch die Konkatenationsoperationen namens "**&**" und die Vergleichsoperationen namens "=", "/=", "<", "<=", ">" und ">=" jeweils zu einem **Reihungstyp**, und nicht nur zu irgendeinem **Untertyp** dieses Typs.

Ein Untertyp ist **definit** bzw. **indefinit**, wenn er als **Bauplan** aufgefaßt, **vollständig** bzw. **unvollständig** ist. Vereinbart man Variablen eines **indefiniten** Untertyps, muß man außer dem Namen des Untertyps noch weitere Informationen angeben, um damit "den unvollständigen Bauplan zu vervollständigen". Bei der Vereinbarung von Variablen eines **definiten** Untertyps genügt der Name des Untertyps, da "der Bauplan bereits vollständig ist". Alle **diskreten Untertypen** sind **definit** und alle **uneingeschränkten Reihungsuntertypen** (wie z.B. **SENDUNG** oder **string**) sind **indefinit**.

Beispiel 14.2.13.: Vereinbarungen von Variablen **definiter** und **indefiniter** Untertypen:

```
01 declare
02    V1 : string;               -- falsch!  Bauplan string  ist unvollstaendig!
03    V2 : SENDUNG;              -- falsch!  Bauplan SENDUNG ist unvollstaendig!
```

```
04    V3 : string(1..3);          -- richtig! Bauplan explizit vervollstaendigt!
05    V4 : SENDUNG(36..41);       -- richtig! Bauplan explizit vervollstaendigt!
06    V5 : string  := "ABC";      -- richtig! Bauplan implizit vervollstaendigt!
07    V6 : SENDUNG := (7, 60);    -- richtig! Bauplan implizit vervollstaendigt!
```

Man kann einen uneingeschränkten Reihungsuntertyp **explizit** vervollständigen, indem man nach dem Namen des Untertyps ausdrücklich **Indexgrenzen** angibt, oder man kann ihn **implizit** vervollständigen, indem man für die vereinbarte Variable einen **Anfangswert** angibt, aus dem der Ausführer die Indexgrenzen **schließen** kann (wie schon im Beispiel 14.2.2. erläutert wurde). **o**

Der **Komponentenuntertyp** eines Reihungstyps muß **definit**, d.h. "als Bauplan vollständig" sein. Von allen bisher behandelten Untertypen sind nur die **uneingeschränkten Reihungsuntertypen** wie SENDUNG und string **indefinit**, alle anderen dagegen **definit**. Hier ein paar Reihungstypvereinbarungen mit **falschen** und **richtigen** Komponentenuntertypen:

Beispiel 14.2.14.: Falsche und richtige Reihungstypvereinbarungen:

```
01 declare
02    type    INDEX    is range 1..25;
03    subtype STRING10 is string(1..10);
04    type    STRINGS1 is array(INDEX) of string;               -- falsch!
05    type    STRINGS2 is array(INDEX) of STRING10;             -- richtig!
06    type    AUFTRAG1 is array(INDEX) of SENDUNG;              -- falsch!
07    type    AUFTRAG2 is array(INDEX) of SENDUNG(34..48);      -- richtig!
```

Zum Untertyp **STRING10** (vereinbart in Zeile 03) gehören alle Strings mit dem Indexbereich **1..10**. Offenbar ist **STRING10** ein **eingeschränkter** Reihungsuntertyp und damit **definit** ("als Bauplan vollständig"). Die Typvereinbarungen in Zeile 04 und 06 sind falsch, weil als Komponentenuntertyp jeweils ein **indefiniter** Untertyp angegeben wurde. Die Komponenten der Reihungen des Untertyps **AUFTRAG2** gehören zu einem **namenlosen Untertyp**, der aus dem Typ **!SENDUNG** und der Indexeinschränkung **34..48** besteht.

Zusammenfassung 14.2.:

- Ein **uneingeschränkter Reihungsuntertyp** läßt offen, wie **lang** seine Reihungen sind und welche genauen **Indexgrenzen** sie haben.
- Als **Bauplan** aufgefaßt ist ein uneingeschränkter Reihungsuntertyp also **unvollständig** (Fachbegriff: **indefinit**).
- Vereinbart man **Variablen** eines uneingeschränkten Reihungsuntertyps, muß man den Bauplan durch die Angabe von Indexgrenzen **vervollständigen**.
- Die Angabe der Indexgrenzen kann **explizit** oder **implizit** (durch Angabe eines Anfangswertes für die vereinbarte Variable) erfolgen.
- Der **Komponentenuntertyp** eines Reihungstyps muß **definit** ("als Bauplan vollständig") sein.
- Der **Indexbereich** (und damit auch die **Länge**) einer **Reihung** ist **nicht veränderbar**. Vor der Vereinbarung einer Reihung kann man die Indexgrenzen aber variieren, z.B. einlesen. Dies gilt nur für Reihungen, die mit Hilfe eines **uneingeschränkten** Reihungsuntertyps vereinbart werden.
- Für die **vordefinierten** uneingeschränkten Reihungsuntertypen **string** und **wide_string** gelten weitgehend die gleichen Regeln, wie für andere vom Programmierer vereinbarte uneingeschränkte Reihungsuntertypen auch.
- Ausgehend von einem **uneingeschränkten** Reihungsuntertyp kann man weitere **uneingeschränkte** und **eingeschränkte** Untertypen des selben Reihungstyps vereinbaren.

- Ausgehend von einem **eingeschränkten** Reihungsuntertyp E kann man nur weitere Namen für den gleichen Untertyp vereinbaren, man kann die Indexeinschränkung von E aber nicht noch weiter einschränken.
- Für jeden **diskreten Reihungstyp** (d.h. für jeden eindimensionalen Reihungstyp mit diskretem Komponentenuntertyp) gibt es alle **sechs Vergleichsoperationen** namens "=", "/=", "<", "<=", ">" und ">=". Für andere Reihungstpyen gibt es nur "=" und "/=".
- Für jeden eindimensionale Reihungstyp gibt es vier **Konkatenationsoperationen** namens "**&**".
- **Uneingeschränkte Reihungsuntertypen** sind vor allem im Zusammenhang mit **Unterprogrammen** wichtig und nützlich.
- Außer **Strings fester Länge** (vom Typ **!string** bzw. **!wide_string**) gibt es in Ada auch **flexible Strings begrenzter Länge** und **flexible Strings unbegrenzter Länge**.

14.3. Zeichenkettentypen

Ein **Zeichentyp** ist ein **Aufzählungstyp**, in dessen Vereinbarung mindestens ein **Zeichenliteral** (wie z.B. 'A' oder 'x' oder '$' etc.) vorkommt. Siehe dazu auch Abschnitt 10.2..

Beispiel 14.3.1.: Zeichentypen (character types)

```
01 declare
02    type VOKAL            is ('a', 'e', 'i', 'o', 'u');
03    type ROEMISCHE_ZIFFER is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
04    type EXTREM           is ('A', AUS, KAPUTT);
```

Alle Aufzählungsliterale der Untertypen **VOKAL** und **ROEMISCHE_ZIFFER** sind **Zeichenliterale**. Zum Untertyp **EXTREM** gehört nur ein einziges Zeichenliteral. **o**

Man beachte, daß der **Vokal 'a'** nichts mit dem **character**-Wert **'a'** zu tun hat. Diese beiden Werte gehören zu verschiedenen Typen und werden "nur zufällig" mit dem gleichen Zeichenliteral bezeichnet. Entsprechend ist die römische Ziffer **'X'** ein ganz anderer Wert als der **character**-Wert **'X'**. Ein Zeichenliteral wie **'a'** oder **'X'** etc. kann also mehrere verschiedene Bedeutungen haben und wird dann als **überladen** bezeichnet. **Überladen zu sein** ist für ein Zeichenliteral keine "schlechte oder unangenehme" Eigenschaft, sondern ein ganz normaler Zustand (obwohl das Wort "überladen" ein bißchen negativ klingt).

Ein **Zeichenkettentyp** (string type) ist ein **Reihungstyp**, mit einem **Zeichentyp** als **Komponententyp** und einem beliebigen (diskreten) Indexuntertyp. Hier ein paar Beispiele:

Beispiel 14.3.2.: Zeichenkettentypen (string types)

```
01 declare
02    type V_WORT is array(positive  range <>) of VOKAL;
03    type V_TAB  is array(VOKAL)              of VOKAL;
04    type R_ZAHL is array(positive  range <>) of ROEMISCHE_ZIFFER;
05    type E_TAB  is array(character range <>) of EXTREM;
```

Die Untertypen **V_WORT**, **R_ZAHL** und **E_TAB** sind **uneingeschränkt**, **V_TAB** ist **eingeschränkt**. **o**

Es gibt in Ada zwei **vordefinierte** Zeichenkettentypen, die vom Ausführer aufgrund der folgenden beiden Vereinbarungen erzeugt wurden:

```
01 type      string is array(positive range <>) of      character;
02 type wide_string is array(positive range <>) of wide_character;
```

Diese Typen existieren also "schon immer". Sie müssen (und können) **nicht** noch mal vom Programmierer vereinbart werden. Siehe dazu auch (ARM 3.6.3.) und (ARM A.1(37) und (41)).

Für jeden **Reihungstyp** gibt es in Ada **Aggregate**, mit denen man die Werte des Typs direkt bezeichnen kann. Für jeden **Zeichenkettentyp** gibt es außerdem **Zeichenkettenliterale**, die man als bequeme und übersichtliche **Abkürzungen** für Reihungsaggregate verstehen kann.

Beispiel 14.3.3.: Zeichenkettenliterale für verschiedene **Zeichenkettentypen**

```
01 declare
02    HELLE_VOKALE_1 : V_WORT := (1 => 'i', 2 => 'e', 3 => 'a');
03    HELLE_VOKALE_2 : V_WORT := "iea";
04    VORGANEGER     : V_TAB  := "uaeio";
05    NACHFOLGER     : V_TAB  := "eioua";
06    ROEMISCH_1807  : R_ZAHL := "MDCCCVII";
07    ET1            : E_TAG  := "AA" & KAPUTT & AUS & "AAA";
```

Die beiden Variablen **HELLE_VOKALE_1** und **HELLE_VOKALE_2** haben gleiche Anfangswerte. In Zeile 02 wird der Anfangswert durch ein normales **Reihungsaggregat** beschrieben, in Zeile 03 dagegen durch ein **Zeichenkettenliteral**. Die Reihung **ET1** hat sieben Komponenten, von denen die ersten beiden und die letzten drei mit dem **EXTREM**-Wert **'A'** initialisiert werden. **o**

Die vom Programmierer vereinbarten Zeichenkettentypen (z.B. **!V_WORT**, **!V_TAB**, **!R_ZAHL** etc.) haben **nichts** mit den vordefinierten Zeichenkettentypen **!string** und **!wide_string** zu tun, sondern sind eigenständige Typen mit ihren eigenen **Zeichenkettenliteralen** und ihren eigenen **Konkatenationsoperationen** "&". Zeichenkettenliterale wie **"iea"** oder **"MDCCCVII"** etc. sind **überladen**. Der Ausführer versucht, aus dem jeweiligen Zusammenhang zu schließen, zu welchem Typ diese Literale gehören. Im vorigen Beispiel gehört die Zeichenkette **"MDCCCVII"** (in Zeile 06) eindeutig zum Typ **!R_ZAHL** und hat nichts mit dem Typ **!string** zu tun.

Um naheliegende Mißverständnisse zu vermeiden, sollte man eine Zeichenkette im Deutschen nur dann als **String** bezeichnen, wenn sie zum Untertyp **string** gehört. Im Beispiel 14.3.3. kommt **kein einziger String** vor.

Ein praktisches Problem taucht auf, wenn man Werte eines selbst vereinbarten **Zeichentyps** oder Werte eines selbst vereinbarten **Zeichenkettentyps** einlesen oder ausgeben will, z.B. Werte des Zeichentyps **!ROEMSICHE_ZIFFER** oder Werte des Zeichenkettentyps **!R_ZAHL**. Angenommen, das Paket **RZ_EA** ist wie folgt vereinbart:

```
01 package RZ_EA is new ada.text_io.enumeration_io(enum => ROEMISCHE_ZIFFER);
```

Wenn man mit diesem Paket römische Ziffern ausgibt, erscheinen sie in einfache **Anführungszeichen** eingefaßt, z.B. so: 'I', 'V', 'X' etc., und beim Eingeben von römischen Ziffern muß der Benutzer die einfachen Anführungszeichen mit eingeben. Solange man nur **einzelne Ziffern** einlesen oder ausgeben will, kann man die Anführungszeichen eventuell noch dulden, aber wenn man römische **Zahlen** (Werte des Untertyps **R_ZAHL**, Reihungen von römischen Ziffern) einlesen oder

ausgeben will, sollte man die Anführungszeichen möglichst **vermeiden**. Dazu empfiehlt es sich, beim Ausgeben die Zeichen vom Untertyp **ROEMISCHE_ZIFFER** in die "gleich aussehenden" Zeichen vom Untertyp **character** umzuwandeln und die **character**-Werte mit der Prozedur **ada.text_io.put** auszugeben, etwa so:

Beispiel 14.3.4.: Ausgeben von **römischen Zahlen**:

```
01 declare
02    type TRANS_TAB_1 is array(ROEMISCHE_ZIFFER) of character;
03    ROEM_TO_CHAR : TRANS_TAB_1 := ('I', 'V', 'X', 'L', 'C', 'D', 'M');
04    R1 : R_ZAHL := "MDCCCVII"; -- Diese Zahl wird ausgegeben
05    C  : character;
06 begin
07    for I in positive range R1'range loop
08       C := ROEM_TO_CHAR(R1(I));
09       ada.text_io.put(item => C);
10    end loop;
```

Die Reihung **ROEM_TO_CHAR** hat **ROEMISCHE_ZIFFERn** als Indizes und Zeichen vom Untertyp **character** als Komponenten. Mit dieser Reihung kann man z.B. die römische Ziffer **'X'** in den entsprechenden ("gleich aussehenden") **character**-Wert **'X'** umwandeln: Der Ausdruck **ROEM_TO_CHAR('X')** bezeichnet den **character**-Wert **'X'**. In Zeile 08 gehört **I** zum Untertyp **positive**, **R1(I)** zum Untertyp **ROEMISCHE_ZIFFER** und **ROEM_TO_CHAR(R1(I))** zum Untertyp **character**. **o**

Aufgabe 14.3.1.: Zeichnen Sie die Reihungen **ROEM_TO_CHAR** und **R1** (vereinbart im vorigen Beispiel) als "Kästchendiagramme". **o**

Aufgabe 14.3.2.: Schreiben Sie ein Programm namens **REIHU_02**, welches eine Folge von römischen Ziffern einliest, in einer Variablen vom Untertyp **R_ZAHL** speichert und dann wieder ausgibt. Der Untertyp **R_ZAHL** soll so wie oben im Beispiel 14.3.2. vereinbart werden. Für die Übersetzung von römischen Ziffern in entsprechende **character**-Werte sollen Sie wie im vorigen Beispiel einen Untertyp **TRANS_TAB_1** und eine Reihung **ROEM_TO_CHAR** vereinbaren. Für Umwandlungen in der umgekehrten Richtung (**character**-Werte in römische Ziffern) sollen Sie ganz entsprechend einen Untertyp **TRANS_TAB_2** und eine Reihung **CHAR_TO_ROEM** vereinbaren. **o**

Zusammenfassung 14.3.:

- Ein **Zeichentyp** ist ein spezieller Aufzählungstyp.
- Ein **Zeichenkettentyp** ist ein spezieller Reihungstyp.
- Die Zeichenkettentypen **!string** und **!wide_string** sind **vordefiniert**.
- Der Programmierer kann beliebig viele weitere Zeichenkettentypen vereinbaren.
- Die vordefinierten und die vom Programmierer vereinbarten Zeichenkettentypen sind weitgehend **gleichbereichtigt**, haben aber ansonsten **nichts miteinander zu tun**.
- Für jeden Zeichenkettentyp stehen entsprechende Zeichenkettenliterale zur Verfügung.
- Beim **Ausgeben** von Zeichenketten eines **selbst vereinbarten** Zeichenkettetyps sollte man die Komponenten in entsprechende **character**-Werte umwandeln. Entsprechend sollte man **character**-Werte **einlesen** und in Werte des betreffenden Zeichentyps umwandeln.

14.4. Mehrdimensionale Reihungstypen

Der **Indexuntertyp** eines Reihungstyps muß **diskret** sein. aber als **Komponentenuntertyp** ist jeder **definite** ("als Bauplan vollständige") Untertyp erlaubt. Da (unter anderem) alle **eingeschränkten** Reihungsuntertypen **definit** sind, kann man einen Reihungstyp vereinbaren, dessen Komponentenuntertyp ein (eingeschränkter) Reihungsuntertyp ist, z.B. so:

Beispiel 14.4.1.: Stellungen auf einem Schachbrett als Reihungen von Reihungen:

```
01 B1: declare
02    type BUCHSTABE is (     A, B, C, D, E, F, G, H);
03    type NUMMER    is range 1..8;
04    type FIGUR     is (     WD, WK, WT, WL, WP, WB,
05                            SD, SK, ST, SL, SP, SB, O);
06    type REIHE     is array(NUMMER)    of FIGUR;
07    type STELLUNG  is array(BUCHSTABE) of REIHE;
08    ANFANG : constant STELLUNG :=
09         (A => (WT, WB, O, O, O, O, SB, ST),
10         (B => (WP, WB, O, O, O, O, SB, SP),
11         (C => (WL, WB, O, O, O, O, SB, SL),
12         (D => (WD, WB, 0, 0, 0, 0, SB, SD),
13         (E => (WK, WB, 0, 0, 0, 0, SB, SK),
14         (F => (WL, WB, 0, 0, 0, 0, SB, SL),
15         (G => (WP, WB, 0, 0, 0, 0, SB, SP),
16         (H => (WT, WB, 0, 0, 0, 0, SB, ST));
17    SU : STELLUNG  := ANFANG;
18    B1 : BUCHSTABE := E;
19    N1 : NUMMER    := 2;
```

Die Literale des Untertyps **FIGUR** (Zeile 04) sind Abkürzungen mit den folgenden Bedeutungen: **W**: weiß, **D**: Dame, **K**: König, **T**: Turm, **L**: Läufer, **P**: Pferd, **B**: Bauer, **S**: schwarz, **O**: leeres Feld ("oh wie leer"). **WD** bezeichnet also die weiße Dame und **SB** einen schwarzen Bauern etc.

Der Reihungsuntertyp **STELLUNG** hat den eingeschränkten Reihungsuntertyp **REIHE** als Komponententyp. **STELLUNG** ist der Untertyp, um den es hier hauptsächliche geht. Seine Reihungen bezeichnet man auch als "**Reihungen von Reihungen**".

Weil die Konstante **ANFANG** (vereinbart in den Zeilen 08 bis 16) eine **Reihung von Reihungen** ist, wird ihr Wert durch ein **Aggregat von Aggregaten** beschrieben. Z.B. steht in Zeile 09 hinter **A =>** ein Aggregat, welches einen Wert des Untertyps **REIHE** beschreibt. **Acht** solcher **REIHE**-Aggregate bilden zusammen ein **STELLUNG**-Aggregat.

Jedes der acht **REIHE**-Aggregate ist einfach als Liste von **FIGUR**-Werten notiert, **ohne** Indizes und Pfeile (z.B. **(WT, WB, O, O, O, O, SB, ST)**). Das **STELLUNG**-Aggregat ist dagegen **mit** Indizes und Pfeilen (**A =>** ..., **B =>** ..., etc.) notiert. Gerade diese Notationen zu wählen war eine willkürliche Entscheidung des Programmierers. Alternativ hätte er alle Indizes und Pfeile weglassen oder andersherum auch die **REIHE**-Aggregate **mit** Pfeilen und Indizes notieren können. Der Programmierer hatte aber das Gefühl, daß diese alternativen Notationen nicht so übersichtlich sind, wie die im Beispiel gewählte. **o**

Aufgabe 14.4.1.: Stellen Sie eine Reihung des Untertyps **REIHE** bildlich dar ("als Kästchendiagramm"). **o**

Aufgabe 14.4.2.: Stellen Sie die Reihung **SU** bildlich dar ("als Kästchendiagramm"). Jede Komponente der Reihung **SU** sollte ganz ähnlich aussehen, wie Ihre Lösung der vorigen Aufgabe. **o**

Um auf eine **Komponente** der Reihung **SU** zuzugreifen, muß man hinter ihren Namen **SU** einen geeigneten Indexausdruck (in runden Klammern eingeschlossen) schreiben, z.B. **SU(A)** oder **SU(B1)** oder **SU(BUCHSTABE'succ(B1))** etc..

Der Ausdruck **SU(B1)** bezeichnet eine **Komponente** von **SU**. Diese Komponente ist eine **Reihung** vom Untertyp REIHE. Um auf eine Komponente der Reihung **SU(B1)** zuzugreifen, muß man hinter ihren "Namen" **SU(B1)** einen geeigneten Indexausdruck (in runden Klammern eingeschlossen) schreiben, z.B. **SU(B1)(3)** oder **SU(B1)(N1)** oder **SU(B1)(N1+1)** etc..

Beispiel 14.4.2.: Den klassischen Eröffnungszug "Weißer Königsbauer rückt zwei Felder vor" (kurz: e2 nach e4), kann man jetzt so programmieren:

```
20 begin -- B1
21    SU(E)(4) := SU(E)(2);     -- Bauern von e2 nach e4 kopieren
22    SU(E)(2) := O;            -- Den Bauern auf e2 "loeschen"
```
o

Aufgabe 14.4.3.: Angenommen, Weiß will nicht mit dem **Königsbauern** eröffnen, sondern mit einem **Pferd**. Wie könnten entsprechende Ada-Befehle aussehen? Wählen Sie eines der beiden weißen Pferde und eines der beiden Ziele, zu denen es springen kann. **o**

Aufgabe 14.4.4.: Angenommen, ein Schuhladen hat in jedem Semester (zwischen SS90 und WS93) seinen Bestand an Schuhen (Größen 34 bis 48, maximal 100 Paare pro Grösse) festgestellt. Vereinbaren Sie einen geeigneten Reihungsuntertyp und eine Reihung dieses Untertyps, in der man die festgestellten Bestände speichern kann. **o**

Die Stellungen auf einem Schachbrett sind **zweidimensionale** Gebilde. Der Typ **!STELLUNG** gibt solche Stellungen wieder und hat damit auch "so was Zweidimensionales". **!STELLUNG** ist technisch gesehen aber kein **echt zweidimensionaler** Reihungstyp (sondern nur "unecht zweidimensional"). Das folgende Beispiel zeigt **echte Mehrdimensionalität**.

Beispiel 14.4.3.: Ein **echt zweidimensionaler** Reihungstyp !STELLUNG2:

```
01 B2: declare
02    type BUCHSTABE is (     A, B, C, D, E, F, G, H);
03    type NUMMER    is range 1..8;
04    type FIGUR     is (     WD, WK, WT, WL, WP, WB,
05                            SD, SK, ST, SL, SP, SB, O);
07    type STELLUNG2 is array(BUCHSTABE, NUMMER) of FIGUR;
```

In der Typvereinbarung in Zeile 07 wurden **zwei** Indexuntertypen (**BUCHSTABE** und **NUMMER**) angegeben. Damit ist **STELLUNG2** ein **zweidimensionaler** Reihungsuntertyp (und sein **Typ** !STELLUNG2 ein zweidimensionaler Reihungstyp). Man beachte, daß **FIGUR** hier **direkt** die Rolle des Komponentenuntertyp von **STELLUNG2** spielt (und nicht indirekt über einen Untertyp **REIHE** wie im Beispiel 14.4.1.). **o**

Reihungen (Konstanten und Variablen) des Untertyps **STELLUNG2** vereinbart man **genauso** wie Reihungen des Untertyps **STELLUNG** (siehe Beispiel 14.4.1., Konstante **ANFANG** und Variable **SU**). Obwohl der Untertyp STELLUNG2 **echt** zweidimensional ist, beschreibt man seine Werte genauso durch **Aggregate von Aggregaten** wie beim **unecht** zweidimensionalen Untertyp **STELLUNG**. Die Notation für den **Zugriff auf Komponenten** ist bei **echt** zweidimensionalen Reihungen allerdings ein bißchen anders als bei **unecht** zweidimensionalen, wie das folgende Beispiel deutlich machen soll:

Beispiel 14.4.4.: Auf die **Komponenten** einer echt zweidimensionalen Reihung **zugreifen**:

```
08    ENDE: constant STELLUNG2 := (others => (others => O));
09    SE  :          STELLUNG2 := ANFANG2;
10    B1  :          BUCHSTABE := E;
11    N1  :          NUMMER    := 2;
12 begin
13    SE(E, 4) := SE(E, 2);
14    SE(E, 2) := O;
```

In der Stellung **ANFANG2** sind alle 64 Felder des Schachbretts mit der Pseudofigur **O** ("oh wie leer") belegt. In Zeile 08 steht das erste **others** für die Indizes der **ersten** Dimension (**1..8**) und das zweite **others** für die Indizes der **zweiten** Dimension (**A..H**).

Auf die Komponenten einer echt zweidimenisonalen Reihung wie **SE** greift man zu, indem man hinter den Namen der Reihung **zwei** geeignete Indexausdrücke (gemeinsam in **ein** Paar runder Klammern eingefaßt und durch ein Komma voneinander getrennt) schreibt, z.B. **SE(E, 4)** oder **SE(B1, N1)** oder **SE(BUCHSTABE'succ(B1), N1+1)** etc.. **o**

Auf **Teilreihungen** von echt mehrdimensionalen Reihungen kann man **nicht** zugreifen.

Für **unecht** mehrdimensionale Reihungen (die ja technisch gesehen **eindimensional** sind), gibt es (nur) die normalen Attribute '**first**, '**last**, '**range** und '**length**, wie für andere eindimensionale Reihungen auch.

Für **echt** mehrdimensionale Reihungen gibt es diese Attribute "für jede Dimension extra".

Beispiel 14.4.5.: Attribute der **echt mehrdimensionalen** Reihung SE:

Attribut	**Bedeutung**	**Attribut**	**Bedeutung**
SE'first(1)	1	SE'first(2)	A
SE'last(1)	8	SE'last(2)	H
SE'range(1)	1..8	SE'range(2)	A..H
SE'length(1)	8	SE'length(2)	8

SE'first, **SE'last** etc. (ohne Index in Klammern) bedeutet das gleiche wie **SE'first(1)**, **SE'last(1)** etc. **o**

Ein **unecht** mehrdimensionaler Reihungsuntertyp kann **eingeschränkt** oder **uneingeschränkt** sein, aber wenn er uneingeschränkt ist, dann nur in seiner "ersten" Dimension. Der Reihungsuntertyp der Komponenten muß ja **definit** und damit **eingeschränkt** sein.

Beispiel 14.4.6.: Ein unecht mehrdimensionaler uneingeschränkter Reihungsuntertyp:

```
01 B3: declare
02    type INDEX  is range 1..10;
03    type ERUT   is array(INDEX)          of character;
04    type UMURUT is array(INDEX range <>) of ERUT;
05    U13: UMURUT(1..3);
06    U26: UMURUT(2..6);
```

Die Reihung **U13** hat drei Komponenten vom Untertyp **ERUT** und somit insgesamt Platz für 30 Zeichen vom Untertyp character. **U26** hat fünf Komponenten vom Untertyp **ERUT** und somit insgesamt Platz für 50 Zeichen. "ERUT" soll "eingeschränkter Reihungsuntertyp" und "UMURUT" "unecht mehrdimensionaler uneingeschränkter RUT" heißen. **o**

Echt mehrdimensionale Reihungsuntertypen können auch **eingeschränkt** oder **uneingeschränkt** sein. Es gilt aber die einfache Regel: Entweder ist der Reihungsuntertyp in **allen** Dimensionen **eingeschränkt** oder er ist in **allen** Dimensionen **uneingeschränkt**. "Mischungen" sind nicht erlaubt (sie würden begriffliche Probleme aufwerfen und die Konstruktion von Compilern erschweren).

Beispiel 14.4.7.: Ein echt mehrdimensionaler uneingeschränkter Reihungsuntertyp:

```
01 B4: declare
02    type BUCHSTABE      is (A, B, C, D, E, F, G, H);
03    type NUMMER         is range 1..8;
04    type FIGUR          is (WD, WK, ..., O) -- wie oben in Beispiel 14.4.1.
05    type STELLUNG2U     is array(BUCHSTABE range <>, NUMMER range <>) of FIGUR;
06    MITTE : STELLUNG2U(D..E, 4..5) := ( D..E  => (WB, SB));
07    WEISS : STELLUNG2U(A..H, 1..2) := (others => (others => O));
08    CENTER: STELLUNG2U             := MITTE;
```

Der Reihungsuntertyp STELLUNG2U ist **zweidimenional-uneingeschränkt**. Als Bauplan ist er sozusagen "zweifach unvollständig". Entsprechend müssen in jeder Variablenvereinbarung zwei Indexbereiche angegeben werden, entweder explizit (wie in Zeile 06 und 07) oder implizit (wie in Zeile 08). Die Reihung **MITTE** umfaßt die vier zentralen Felder eines Schachbretts. Zur Reihung **WEISS** gehören die 16 Felder, auf denen anfangs weiße Figuren stehen. **o**

Reihungen von Reihungen ("**unecht** mehrdimensionale Reihungen") haben gegenüber **echt** mehrdimensionale Reihungen den **Vorteil**, daß man von ihnen **Teilreihungen** bilden und sie **konkatenieren** kann (mit "**&**"). Andererseits kann ein **echt** mehrdimensionaler Reihungsuntertyp in all seinen Dimensionen **uneingeschränkt** sein, was bei einem **unecht** mehrdimensionalen Reihungsuntertyp nicht möglich ist.

Zusammenfassung 14.4.:

- Eine Reihung, deren Komponenten Reihungen sind, bezeichnet man auch als **Reihung von Reihungen**.
- Außer Reihungen von Reihungen gibt es in Ada auch **echt mehrdimensionale** Reihungen.
- Einen echt n-dimensionalen Reihungstyp vereinbart man, indem man in einer Reihungstypvereinbarung (hinter **array** in Klammern) n **Indexuntertypen** angibt.
- Die **Werte** von echt mehrdimensionalen Reihungen und die **Werte** von Reihungen von Reihungen kann man durch die gleichen **Aggregate von Aggregaten** beschreiben.

- Der **Zugriff auf Komponenten** sieht bei echt mehrdimensionalen Reihungen etwa anders aus als bei Reihungen von Reihungen (z.B. **SE(H, 7)** statt **SU(H)(7)**).
- Für echt mehrdimensionale Reihungen gibt es die Attribute **'first**, **'last**, **'range** und **'length** für jede Dimension extra (**'first(1)**, **'first(2)**, ... etc.).
- Ein **echt** mehrdimensionaler Reihungsuntertyp kann in all seinen Dimensionen **eingeschränkt** oder in all seinen Dimensionen **uneingeschränkt** (aber nicht "gemischt") sein.
- Ein **unecht** mehrdimensionaler Reihungsuntertyp kann höchstens in seiner "ersten" (technisch: in seiner einzigen) Dimension uneingeschränkt sein.
- Für einen **unecht** mehrdimensionalen Reihungsuntertyp gibt es (wie für jeden eindimensionalen Reihungsuntertyp) die Möglichkeit, auf **Teilreihungen** zuzugreifen und vier **Konkatenationsoperationen** namens "**&**". Für **echt** mehrdimensionale Reihungsuntertypen gibt es keine Teilreihungen und keine Konkatenationsoperationen.
- Alle Reihungen, die zu einem **eingeschränkten Reihungsuntertyp** gehören, haben (in jeder Dimension) **die gleichen Indizes** und damit (in jeder Dimension) **die gleiche Länge**.
- Ein **uneingeschränkter Reihungsuntertyp** legt für seine Reihungen nur die **Anzahl der Dimensionen**, für jede Dimension einen **Indexuntertyp** und insgesamt einen **Komponentenuntertyp** fest. Zwei Reihungen des selben uneingeschränkten Reihungsuntertyps können sich (in jeder Dimension) durch ihre **Indexbereiche** und damit (in jeder Dimension) durch ihre **Längen** voneinander unterscheiden.

Zentraldokument: Nach Filialdokument A95-13-14

15. Verbundtypen und -untertypen

Eine **Reihung** (array) besteht aus **Komponenten**, die alle zum **gleichen Untertyp** gehören müssen, und auf die man mit Hilfe von **Indizes** zugreifen kann. Ein **Verbund** (record) besteht aus **Komponenten**, die zu **verschiedenen Untertypen** gehören dürfen, und auf die man mit Hilfe von sogenannten **Selektornamen** ("Auswählernamen") zugreifen kann.

15.1. Eingeschränkte Verbunduntertypen

Beispiel 15.1.1.: Einen **Verbundtyp**, seinen **ersten Untertyp** und ein paar Variablen und Konstanten des Untertyps vereinbaren:

```
01 declare
02    ------------------------------------------------
03    type    ANZAHL   is        range 0..100_000;
04    subtype ANZAHL_A is ANZAHL range 0..20;
05    subtype STRING10 is string(1..10);
06    ------------------------------------------------
07    type LAST_WAGEN is record
08       GEWICHT : ANZAHL   := 0;
09       ACHSEN  : ANZAHL_A := 0;
10       KENNZ   : STRING10 := "??????????";
11       HAENGER : boolean  := false;
12    end record;
13    ------------------------------------------------
14    LW1 :          LAST_WAGEN;
15    LW2 :          LAST_WAGEN := (7_500, 2, "BRB-XY1234", false);
16    LW3 : constant LAST_WAGEN := (KENNZ => "OHV-AB5678", ACHSEN => 3,
17                                  GEWICHT => 25_000, HAENGER => true);
18 begin
19    LW1             :=  LW2;
20    LW2             := (ACHSEN => 4, GEWICHT => 30_000, HAENGER => true,
21                        KENNZ => "ABC-DE9876");
22    LW1.ACHSEN      :=  5;
23    LW2.GEWICHT     :=  LW3.ACHSEN * 10_000;
24    LW2.KENNZ(5..6) := "YX";
25    if    LW1  = LW3 then ... end if;
26    while LW2 /= LW3 loop ... end loop;
...   ...
```

In den Zeilen 07 bis 12 wird ein **Verbundtyp** (record type) **!LAST_WAGEN** und sein erster Untertyp namens **LAST_WAGEN** vereinbart. Jeder Verbund dieses Untertyps besteht aus **vier Komponenten** mit den Selektornamen **GEWICHT**, **ACHSEN**, **KENNZ** und **HAENGER**. Die **GEWICHT**-Komponente gehört zum Untertyp **ANZAHL** und hat den **Vorbesetzungsausdruck 0**. Die **ACHSEN**-Komponente gehört zum Untertyp **ANZAHL_A** und hat ebenfalls den Vorbesetzungsausdruck **0**. Die **KENNZ**-Komponente gehört zum Untertyp **STRING10** und hat den Vorbesetzungsausdruck **"??????????"**. Die **HANEGER**-Komponente gehört zum Untertyp **boolean** und hat den Vorbesetzungsausdruck **false**.

In Zeile 14 wird eine Variable namens **LW1** vom Untertyp LAST_WAGEN vereinbart. Ihre vier Komponenten haben alle die Werte der entsprechenden Vorbesetzungsausdrücke (**0**, **0**, **"??????????"** und **false**).

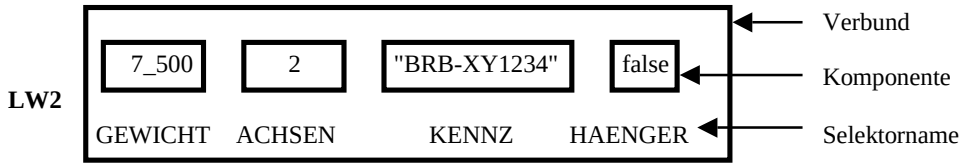
Die Variable **LW2** (vereinbart in Zeile 15) hat den durch das Verbundaggregat **(7_500, 2, "BRB-XY1234", false)** beschriebenen Anfangswert. Für die Konstante **LW3** (siehe Zeile 16) gilt entsprechendes.

Auf eine **Komponente** des Verbundes **LW1** kann man zugreifen, indem man hinter den Namen **LW1** einen Punkt "**.**" und den **Selektornamen** der betreffenden Komponenten schreibt, z.B. **LW1.GEWICHT** oder **LW1.KENNZ** etc..

Die Anweisungen in Zeile 19 bis 26 sollen einen Eindruck davon vermitteln, wie man einerseits **ganze Verbunde** auf einmal und andererseits **einzelne Komponenten** eines Verbundes verarbeiten kann. **o**

Vorbesetzungen darf der Programmierer in der Vereinbarung eines Verbundtyps für **beliebig viele** oder **wenige** der Komponenten angeben, z.B. für **alle** Komponenten oder nur für **einige** oder auch für **keine** Komponente.

Hier eine **bildliche Darstellung** der Variablen **LW2** ("ein Kästchendiagramm") mit dem Wert, den sie unmittelbar nach ihrer Erzeugung enthält:



Der große ("umfassende") Kasten stellt die **Verbundvariable** namens **LW2** dar, und die kleinen Kästchen die **Komponentenvariablen** oder einfach **die Komponenten von LW2**. Die unterschiedlichen Größen der kleinen Kästchen soll deutlich machen, daß die Komponenten zu verschiedenen Untertypen gehören. **In** den kleinen Kästchen stehen die **Werte** der Komponenten und **unter** jeder Komponenten steht ihr **Selektorname**. In einem**Verbund** sind im allgemeinen **Komponenten** verschiedener Untertypen zu **einer** Variablen **verbunden**.

Die Komponenten eines Verbundes können zu beliebigen Untertypen gehören. Die Komponentenuntertypen müssen nur **definit** ("als Bauplan vollständig") sein. Das schließt z.B. Komponenten vom Untertyp **string** aus (weil **string** ein **uneingeschränkter** und damit **indefiniter** Reihungsuntertyp ist), erlaubt aber Komponenten wie **KENNZ**, die zu einem eingeschränkten Untertyp des Typs **!string** gehören. Insbesondere können Verbundkomponenten zu einem **Verbunduntertyp** gehören. Dabei gibt es nur eine **Einschränkung**: Ein Verbund eines Verbundtyps **!V** darf weder direkt noch indirekt eine Komponente vom **selben** Typ **!V** enthalten ("Verbundtypen dürfen nicht rekursiv sein").

Verbundaggregate müssen ganz ähnlichen Regeln genügen wie **Reihungsaggregate**. Vor allem muß auch ein Verbundaggregat immer **vollständig** sein und für **jede** Komponente des betreffenden Verbundes einen geeigneten Wert enthalten. Von dieser einfachen Regel gibt es **keine Ausnahmen**. Auch dann nicht, wenn in der Vereinbarung eines Verbundtyps für **jede** Komponente eine **Vorbesetzung** angegeben wurde (wie das z.B. beim Typ **!LAST_WAGEN** im vorigen Beispiel der Fall ist).

Zwei **Verbundtypen** können sich viel stärker voneinander **unterscheiden** als z.B. zwei Ganzzahltypen oder zwei Aufzählungstypen. Die Verbunde eines Verbundtyps können z.B. aus "wenigen kleinen Komponenten" oder aus "vielen kleinen Komponenten" oder aus "wenigen großen Komponenten" oder aus "vielen riesigen Komponenten" etc. bestehen. Wie man Werte eines Verbundtyps z.B. von einer Tastatur **einlesen** oder zu einem Bildschirm **ausgeben** kann (d.h. "in

einer von Menschen lesbaren Form einlesen oder ausgeben kann"), hängt sehr stark von der Anzahl und Art der Komponenten ab. Deshalb gibt es in Ada **keine** spezielle Unterstützung für die Ein-/Ausgabe von Verbundwerten ("**keine** Paketschablone **record_io**", entsprechend den Schablonen **integer_io**, **enumeration_io** etc.) und der Programmierer muß GET- und PUT-Prozeduren für einen Verbunduntertyp "von Hand entwerfen und programmieren", wenn er sie braucht.

Aufgabe 15.1.1.: Schreiben Sie ein Programm namens **VERBE_02**, welches einen Wert des Verbunduntertyps **LAST_WAGEN** (siehe oben Beispiel 15.1.1.) von der aktuellen Eingabe (Tastatur) einliest und zur aktuellen Ausgabe (Bildschirm) ausgibt. **o**

Aufgabe 15.1.2.: Erweitern Sie das Programm **VERBE_02** (die Lösung der vorigen Aufgabe) zu einem Programm namens **VERBE_03**, in dem eine **Reihung** mit drei Komponenten vom Untertyp **LAST_WAGEN** vereinbart wird und **drei** Werte des Untertyps **LAST_WAGEN** in die Reihung eingelesen und wieder ausgegeben werden. **o**

Verbunde haben mit **Reihungen** die Eigenschaft gemeinsam, "aus Komponenten zusammengesetzt" zu sein. Im Grunde genommen sind Verbunde aber deutlich "**weniger aufregend und interessant**" als Reihungen. Das liegt vor allem daran, daß die **Indizes** einer Reihung **Werte** sind, die man **in Variablen speichern**, **einlesen**, **ausgeben** und vor allem (mit Hilfe von Ausdrücken) **berechnen** kann. Selbst wenn die Indizes zu einem Aufzählungstyp gehören, stehen immerhin noch die Operationen **'pred** und **'succ** für das Rechnen mit ihnen zur Verfügung. Für die **Selektornamen** eines **Verbundes** gibt es nichts dergleichen. Selektornamen kann man **nicht** in Variablen speichern, man kann sie **nicht** einlesen oder ausgeben und man kann **nicht** mit ihnen rechnen. Es gibt noch nicht einmal eine **pred**- oder **succ**-Operation für Selektornamen. Etwas konkreter: Ist **ANZ** eine Reihung und **G1** eine Variable ihres Indexuntertyps, dann kann man mit dem einen Ausdruck **ANZ(G1)** auf **jede** Komponente von **ANZ** zugreifen (indem man der Variablen **G1** jeweils den Index der gewünschten Komponenten zuweist). Dagegen bezeichnet der Ausdruck **LW1.GEWICHT** immer nur die **GEWICHT**-Komponente von **LW1** und es gibt **keinen** Ausdruck, mit dem man wahlweise auf verschiedene Komponenten von **LW1** zugreifen könnte.

In diesem Abschnitt wurden **eingeschränkte** Verbunduntertypen eingeführt. Im nächsten Abschnitt werden **uneingeschränkte** Verbunduntertypen behandelt und es wird gezeigt, daß Verbunde in Ada **eine** wichtige Eigenschaft haben können, die Reihungen **nicht** haben (bzw. nur als Komponenten eines Verbundes "erwerben" können): **Flexibilität**, d.h. die Möglichkeit, im Verlauf einer Programmausführung größer und wieder kleiner zu werden (und das beliebig oft).

Zusammenfassung 15.1.:

- Ein **Verbund** (record) besteht aus **Komponenten**
- Die Komponenten eines Verbundes können zu **verschiedenen** Typen gehören
- Auf die einzelnen Komponenten eines Verbundes greift man mit Hilfe von **Selektornamen** zu.
- Ein **Verbunduntertyp** (record subtype) ist "ein **Bauplan** für Verbunde".
- Zur "**Art**" einer **Verbundkomponenten** gehören ihr **Untertyp**, ihr **Selektorname** und ihre **Größe** (falls die Komponente selbst z.B. eine Reihung oder ein Verbund ist)
- Ein **eingeschränkter** Verbunduntertyp legt die **Anzahl** und **Art** der Komponenten seiner Verbunde genau fest.
- Damit ist ein **eingeschränkter Verbunduntertyp** immer **definit** ("als Bauplan vollständig").
- Ein Verbunduntertyp kann für einige oder alle Komponenten einen **Vorbesetzungsausdruck** festlegen.
- Ein **Verbundaggregat** beschreibt einen **Verbundwert**.

- Jedes Verbundaggregat muß **vollständig** und **eindeutig** sein.

15.2. Uneingeschränkte Verbunduntertypen

Der Verbunduntertyp **LAST_WAGEN** (vereinbart im Beispiel 15.1.1. des vorigen Abschnitts) ist ein **eingeschränkter** Verbunduntertyp (a constrained record subtype). Das bedeutet, daß er als **Bauplan** aufgefaßt **vollständig** ist. Jeder Verbund, der nach diesem Plan gebaut wird, hat genau **vier** Komponenten mit den im Bauplan festgelegten **Selektornamen** (GEWICHT, ACHSEN, ...) und **Untertypen** (ANZAHL, ANZAHL_A, ...).

In diesem Abschnitt werden **uneingeschränkte Verbunduntertypen** (unconstrained record subtypes) vorgestellt. Uneingeschränkte Verbunduntertypen werden manchmal auch als **Verbunduntertypen mit Diskriminanten** oder als **diskriminierte Verbunduntertypen** oder als **variante Verbunduntertypen** bezeichnet. Sie lösen folgendes Problem:

Manchmal haben gewisse Verbunde unterschiedlich viele Komponenten und gehören damit eigentlich zu **verschiedenen** Typen. Der Programmierer möchte aber unbedingt, daß die Verbunde trotz ihrer Unterschiede zu **einem** einzigen Typ gehören, damit er Operationen programmieren kann, die auf all diese Verbunde anwendbar sind. Zu einem **uneingeschränkten Verbunduntertyp** können Verbunde mit **verschieden vielen Komponenten** gehören (so, wie zu einem uneingeschränkten Reihungsuntertyp Reihungen verschiedener Länge gehören). Z.B. können zu einem Verbunduntertyp namens **KFZ** sowohl Verbunde zur Beschreibung von Lastwagen also auch Verbunde zur Beschreibung von Personenwagen gehören, obwohl diese Verbunde unterschiedlich viele Komponenten haben.

Den tieferen Sinn von **uneingeschränkte Verbunduntertypen** kann man auch so verstehen: Das starke Typensystem von Ada hat viele Vorteile, ist aber in einigen Situationen **stärker**, als dem Programmierer lieb ist. Mit uneingeschränkten Verbunduntertypen kann er das starke Typensystem an einigen Stellen gezielt und kontrolliert "**ein bißchen weicher machen**", ohne die Stärke des Systems und seine Vorteile zu zerstören.

Beispiel 15.2.1.: Ein Verbundtyp mit einem **uneingeschränkten ersten Untertyp**:

```
01 declare
02    ------------------------------------------------
03    type    WAGEN_ART is (LAST, PERSONEN);
04    type    ANZAHL    is range 0..100_000;
05    subtype ANZAHL_S  is ANZAHL range 0..10;
06    subtype ANZAHL_A  is ANZAHL range 0..20;
07    subtype STRING10  is string(1..10);
08    ------------------------------------------------
09    type KFZ(ART: WAGEN_ART) is record
10       GEWICHT : ANZAHL := 0;
11       KENNZ   : STRING10 := (others => '?');
12       case ART is
13          when PERSONEN =>
14             SITZE   : ANZAHL_S := 0;
15          when LAST =>
16             ACHSEN  : ANZAHL_A := 0;
17             HAENGER : boolean  := false;
18       end case;
19    end record;
20    ------------------------------------------------
21    PW1 : KFZ(ART => PERSONEN);
```

```
22    PW2 : KFZ(ART => PERSONEN) := (PERSONEN, 850, "BRB-AB1234", 5);
23    PW3 : KFZ                  := (ART => PERSONEN, SITZE => 5,
24                                     GEWICHT => 850, KENNZ => "OHV-XY4321");
25    LW1 : KFZ(ART => LAST)     := (ART=>LAST, GEWICHT=>25_000, ACHSEN=>3,
26                                     KENNZ=>"HVL-PQ9876", HAENGER=>false);
27    LW2 : constant KFZ         := (LAST, 30_000, "HVL-RS6789", 5, true);
28    PLW : KFZ; -- Verboten, weil der Bauplan KFZ unvollstaendig ist!
```

Der Untertyp **KFZ** (vereinbart in Zeile 09 bis 20) ist ein **uneingeschränkter Verbunduntertyp**. Das erkennt man an der **Diskriminantenvereinbarung (ART: WAGEN_ART)** in Zeile 09. Die **Diskriminante** namens **ART** vom Untertyp **WAGEN_ART** ist eine Komponente, die zum diskriminieren ("unterscheiden") verschiedener **Varianten** des Untertyps **KFZ** dient. Da die Diskriminante **ART** nur die beiden Werte **LAST** und **PERSONEN** annehmen kann, werden in diesem Beispiel nur **zwei Varianten** von **KFZ** unterschieden ("diskriminiert"). Zu jeder Varianten gehört eine eigene **Liste von Komponenten**, wie die folgende Tabelle verdeutlichen soll:

Jeder **KFZ**-Verbund der Varianten **LAST** hat folgende 5 Komponenten:	Jeder **KFZ**-Verbund der Varianten **PERSONEN** hat folgende 4 Komponenten:
`ART : WAGEN_ART := LAST;`	`ART : WAGEN_ART := PERSONEN;`
`GEWICHT : ANZAHL := 0;`	`GEWICHT : ANZAHL := 0;`
`KENNZ : STRING10 := (others => '?');`	`KENNZ : STRING10 := (others => '?');`
	`SITZE : ANZAHL_S := 0;`
`ACHSEN : ANZAHL_A := 0;`	
`HAENGER : boolean := false;`	

Man beachte, daß die **Diskriminante** ART auch eine **Komponente** ist. Die ersten drei Komponenten (**ART**, **GEWICHT** und **KENNZ**) sind in beiden Varianten des Untertyps **KFZ** gleich. In der **LAST**-Varianten kommen dazu die beiden Komponenten **ACHSEN** und **HAENGER** hinzu, in der **PERSONEN**-Varianten dagegeben nur die eine Komponente **SITZE**. Die Anfangswerte der Diskriminanten **ART** (die Werte **LAST** bzw. **PERSONEN**) sind in obiger Tabelle fett hervorgehoben und können **nicht** verändert werden.

Das Konstrukt in Zeile 12 bis 19 der obigen Typvereinbarung sieht einer **case-Anweisung** sehr ähnlich, ist aber keine **Anweisung** sondern der **Variantenteil** der Typ**vereinbarung**. Er beschreibt **die** Komponenten, die bei den verschiedenen Varianten verschieden sind. **o**

Der Verbunduntertyp **KFZ** ist **uneingeschränkt** und **indefinit**, d.h. "als Bauplan unvollständig". Er legt **nicht** eindeutig fest, ob ein Wertebehälter mit 4 oder mit 5 Komponenten gebaut werden soll. Deshalb muß man in jeder Vereinbarung einer Variablen (oder Konstanten) "den Bauplan vervollständigen", indem man einen Wert für die Diskriminante **ART** festlegt, entweder **explizit** oder **implizit**.

In der Vereinbarung der Variablen **PW1** (siehe oben im vorigen Beispiel Zeile 22) wird der Diskriminantenwert **explizit** festgelegt, d.h. unmittelbar hinter dem Untertypnamen **KFZ** in Klammern: **KFZ(ART => PERSONEN)**.

In der Vereinbarung der Konstanten **LW2** (Zeile 28) wird der Bauplan KFZ **implizit** durch die Angabe des Wertes **(LAST, 30_000, "HVL-RS6789", 5, true)** vervollständigt.

Bei den Variablen **PW2** und **LW1** wird der Wert der Diskriminanten **explizit** und bei **PW3 implizit** festgelegt. Die Vereinbarung in Zeile 29 ist **verboten**, weil der unvollständige Bauplan **KFZ** dort weder explizit noch implizit vervollständigt wird.

Die Vereinbarung von **PW2** soll deutlich machen: Wenn man den Wert der Diskriminanten **explizit** festlegt (**KFZ(ART => PERSON)**) und die Variable außerdem mit einem **Anfangswert** ausstattet (**:= (PERSONEN, 850, "BRB-AB1234", 5)**), dann muß man den Diskriminantenwert **zweimal** angeben. Das folgt aus der Regel, daß Aggregate **immer** und ohne Ausnahme vollständig sein müssen.

In einer Verbundtypvereinbarung darf kein **Selektorname** mehr als einmal vorkommen, auch wenn die Typvereinbarung einen **Variantenteil** enthält. Praktisch bedeutet das für den Typ **!KFZ**: Weil die Verbunde der **PERSONEN**-Variante eine Komponente mit dem Selektornamen **SITZE** haben, dürfen die Verbunde der LAST-Varianten **keine** Komponente mit diesem Selektornamen besitzen, und umgekehrt dürfen **PERSONEN**-Verbunde keine Komponenten namens **HAENGER** oder **ACHSE** haben. Verstöße gegen diese Regel meldet der Ausführer natürlich schon bei der Übergabe eines Programms ("zur Compilezeit") und nicht erst bei der Ausführung.

Aufgabe 15.2.1.: Stellen Sie die Variable **LW1** und die Konstante **PW1**, die im vorigen Beispiel vereinbart wurden, **bildlich** dar (als "Kästchendiagramm"), mindestens in Ihrem Kopf, noch besser auf einem Blatt Papier. **o**

Aufgabe 15.2.2.: Schreiben Sie ein Programm namens **VERBU_02**, welches Daten für zwei Verbunde vom Untertyp **KFZ** einliest und dann wieder ausgibt. Der Benutzer soll frei entscheiden können, zu welcher Varianten die Verbunde, die er eingibt, gehören sollen. Das Programm sollte also "darauf gefaßt sein", daß zwei Personenwagen, zwei Lastwagen oder ein Personen- und ein Lastwagen eingegeben werden. **o**

Der **Untertyp** KFZ ist **uneingeschränkt**, d.h. er legt den Wert der Diskriminanten **ART** und alles, was davon abhängt, **nicht** eindeutig fest. Jede **Variable** des Untertyps **KFZ** ist dagegen **eingeschränkt**, denn ihre **ART**-Komponente hat einen (explizit oder implizit) festgelegten Wert. Dieser Wert und alles, was davon abhängt, kann nicht verändert werden. Die Variable **PW1** kann also immer nur Daten eines Personenwagens enthalten und **LW1** nur Daten eines Lastwagens und man kann Personenwagen wie **PW1** nicht in Lastwagen und Lastwagen wie **LW1** nicht in Personenwagen "umbauen". Hier ein paar erlaubte und verbotene Zuweisungen:

Beispiel 15.2.2.: Erlaubte und verbotene Zuweisungen an Verbundvariablen und ihre Komponenten:

```
30 begin
31    PW1 := PW2;                                          -- erlaubt
32    LW1 := LW2;                                          -- erlaubt
33    PW1 := LW2;                                          -- verboten
34    LW1 := PW2;                                          -- verboten
35    PW1 := (PERSONEN, 0, "ABC-XY1234", 0);               -- erlaubt
36    PW1 := (LAST,     0, "ABC-XY1234", 0, false);        -- verboten
37    PW1.ACHSEN := 6;                                     -- erlaubt
38    PW1.ART    := LAST;                                  -- verboten
```

Alle Zuweisungen, die eine Diskriminante verändern würden, sind verboten. Für die Variablen des Untertyps **KFZ** gilt: "Einmal **LAST**, immer **LAST**, einmal **PERSONEN**, immer **PERSONEN**".
o

Der uneingeschränkte Untertyp **KFZ** besteht aus dem Typ **!KFZ** und der **leeren Diskriminanteneinschränkung**, die "nicht einschränkt". Praktisch bedeutet das: Alle Werte des **Typs** !KFZ gehören auch zu seinem ersten **Untertyp** KFZ.

Ausgehend von dem uneingeschränkten Untertyp **KFZ** kann man weitere **uneingeschränkte** oder **eingeschränkte** Untertypen des **Typs** !KFZ vereinbaren, d.h. vom Ausführer erzeugen lassen, z.B. so:

Beispiel 15.2.3.: Weitere uneingeschränkte und eingeschränkte Untertypen des Typs **!KFZ**:

```
01 declare
02    subtype WAGEN                is KFZ;
03    subtype LAST_WAGEN           is KFZ  (ART => LAST);
04    subtype PERSONEN_WAGEN       is WAGEN(ART => PERSONEN);
05    subtype PKW                  is PERSONEN_WAGEN;
06    subtype CAR                  is LAST_WAGEN(ART => PERSONEN); -- verboten!
```

Hier eine tabellarische Beschreibung der vereinbarten Untertypen:

Der Untertyp	besteht aus dem Typ	und der Diskriminanteneinschränkung	und ist damit
KFZ	!KFZ	leere Einschränkung	uneingeschränkt
WAGEN	!KFZ	leere Einschränkung	uneingeschränkt
LAST_WAGEN	!KFZ	ART => LAST	eingeschränkt
PERSONEN_WAGEN	!KFZ	ART => PERSONEN	eingeschränkt
PKW	!KFZ	ART => PERSONEN	eingeschränkt

Die **leere Diskriminanteneinschränkung** schränkt **nicht** ein. Deshalb sind die Untertypen **KFZ** und **WAGEN** uneingeschränkt und unterscheiden sich nur durch ihre Namen. In der Vereinbarung des Untertyps **PERSONEN_WAGEN** (in Zeile 04) könnte man statt "**WAGEN**" genauso gut "**KFZ**" schreiben, da ein Untertyp immer **Untertyp eines Typs** und nicht **Untertyp eines Untertyps** ist. Die Vereinbarung des Untertyps **CAR** (in Zeile 05) ist verboten, weil man einen eingeschränkten Verbunduntertyp wie **LAST_WAGEN** nicht noch mal einschränken darf. **o**

Alle bisher vereinbarten Verbund**variablen** waren **eingeschränkt**, d.h. die Anzahl und Art ihrer Komponenten war **unveränderbar**. Es ist in Ada aber auch möglich, **uneingeschränkte** Verbundvariablen zu vereinbaren, die z.B. "den Umbau eines Personenwagens in einen Lastwagen" erlauben. Als Voraussetzung dazu muß man zuerst einen Verbunduntertyp vereinbaren, der **uneingeschränkt** und trotzdem **definit** ist. Das erreicht man, indem man für die **Diskriminante** einen **Vorbesetzungsausdruck** angibt, z.B. so:

Beispiel 15.2.4.: Ein **uneingeschränkter** und trotzdem **definiter** Verbunduntertyp sowie **eingeschränkte** und **uneingeschränkte** Variablen dieses Untertyps:

Alle hier nur durch Ellipsen "..." angedeuteten Zeilen stimmen exakt mit den entsprechenden Zeilen im Beispiel 15.2.1. überein. "**FF**" soll "Flexibles Fahrzeug" heißen.

```
01 declare
...   ...
08    ---------------------------------------------
09    type KFZF(ART: WAGEN_ART := PERSONEN) is record
...   ...
20    end record;
```

```
21    -------------------------------------------
22    PW4 : KFZF(ART => PERSONEN);
23    PW5 : KFZF(ART => PERSONEN)     := (PERSONEN, 850, "BRB-AB1234", 5);
24    LW4 : KFZF(ART => LAST);
25    LW5 : KFZF(ART => LAST);        := (LAST,  30_000, "HVL-RS6789", 5, true);
26    FF1 : KFZF;
27    FF2 : KFZF                      := (LAST,  35_000, "OHV-PQ2468", 3, false);
28    FF3 : KFZF                      := (PERSONEN, 800, "BRB-MN1357", 4);
```

In Zeile 09 wird für die Diskriminantenkomponente **ART** der **Vorbesetzungsausdruck** PERSONEN angegeben.

Die Variablen **PW4** bis **LW5** (vereinbart in den Zeilen 22 bis 25) sind **eingeschränkte** ("unflexibel") Variablen, deren Diskriminante nicht verändert werden kann. Die Variablen **FF1** bis **FF3** (vereinbart in den Zeilen 26 bis 28) sind dagegen **uneingeschränkt** ("flexibel"), und ihre Diskriminante kann, zusammen mit allem, was davon abhängt, beliebig oft verändert werden.

Die Variablen **PW4** bis **LW5** sind deshalb **eingeschränkt**, weil der Wert ihrer Diskriminanten **explizit** festgelegt wurde. Für die Variablen **FF1** bis **FF3** wurde der Diskriminantenwert **nicht explizit** festgelegt, sondern zuerst einmal durch den **Vorbesetzungsausdruck** PERSONEN in der Vereinbarung des Untertyps **KFZF** (in Zeile 09) bestimmt. Bei der Variablen **FF2** wird diese Vorbesetzung der Diskriminanten durch den Anfangswert der Variablen **(LAST, 35_000, "OHV-PQ2468", 3, false)** gleich wieder **verändert**, bei der Variablen **FF3** wird die Vorbesetzung durch den Anfangswert "bestätigt", aber das macht keinen wesentlichen Unterschied. Bei allen drei Variablen **FF1**, **FF2** und **FF3** stammt der **erste** Wert der Diskriminanten vom Vorbesetzungsausdruck in der Typvereinbarung und darf deshalb nach den Regeln von Ada beliebig oft verändert werden.

Beispiel 15.2.5.: Erlaubte und verbotene Zuweisungen an uneingeschränkte Verbundvariablen:

```
29 begin
30    FF1               := FF3;
31    FF1               := PW4;
32    FF1               := (PERSONEN,   980, "BRB-AB9876", 2);
33    FF1               := (LAST,     7_500, "BRB-AB6789", 2, false);
34    FF1               := (PERSONEN,   980, "BRB-AB9876", 2);
35    FF1.GEWICHT       := 10_000;
36    FF1.KENNZ(1..3)   := "OHV";
37    FF1.ART           := PERSONEN; -- verboten!
```

Die Variable **FF1** ist "von Geburt" ein Personenwagen und hat damit **vier** Komponenten. Die ersten drei Zuweisungen (Zeile 30 bis 32) sind harmlos und erlaubt, weil sie dem Personenwagen **FF1** nur andere Personenwagen (ebenfalls mit vier Komponenten) zuweisen.

Besonders interessant sind die beiden Zuweisungen in Zeile 33 und 34. In Zeile 33 wird der 4-Komponenten-**Personenwagen** namens **FF1** in einen 5-Komponenten-**Lastwagen** namens **FF1** umgebaut. In Zeile 34 wird **FF1** in der umgekehrten Richtung umgebaut. Die Möglichkeit solcher "Umbauten" machen die Flexibilität von uneingeschränkten Verbundvariable aus.

Die **normalen Komponenten** von **FF1** darf man beliebig verändern, wie z.B. die GEWICHT-Komponente und die KENNZ-Komponente von **FF1** in Zeile 35 und 36. Der **Diskriminantenkomponenten** darf man dagegen nur dann einen Wert zuweisen, wenn man gleichzeitig **allen anderen** Komponenten des Verbundes auch einen Wert zuweist. Somit sind die Zuweisungen in Zeile 30 bis 34 erlaubt, die Zuweisung eines Werte nur an die Diskriminante in Zeile 37 ist dagegen **verboten**.

Mit dieser Regel wird sichergestellt, daß eine uneingeschränkte Verbundvariable wie **FF1** sich immer in einem **konsistenten Zustand** befindet: Entweder hat die Diskriminante **FF1.ART** den Wert **PERSON** und der ganze Verbund die dazu passenden **vier** Komponenten, oder die Diskriminante hat den Wert **LAST** und der ganze Verbund entsprechend **fünf** Komponenten. Der Ausführer erlaubt keine Zuweisungen, durch die die Variable **FF1** in einen **inkonsistenten Zustand** überführt würde.

Beim Untertyp **KFZ** hängt von der Diskriminanten die **Anzahl** und **Art** der weiteren Komponenten ab. Von der Diskriminanten kann aber auch die **Größe** einer Komponenten abhängen, wie das folgende Beispiel zeigen soll:

Beispiel 15.2.6.: Zeichenketten mit flexibler Länge:

```
01 declare
02    type FLEX(L: natural := 0) is
03       S : string(1..L);
04    end record;
05    F1: FLEX(L => 6) := (6, "Hallo!");      -- F1 ist "unflexibel"
06    F2: FLEX         := (6, "Hallo!");      -- F2 ist   "flexibel"
07 begin
08    F1 := (6, "Alloh!);                     -- Erlaubt, obwohl F1 unflexibel ist
09    F1 := (5, "Hello");                     -- Verboten,  weil F1 unflexibel ist.
10    F2 := (1_000, (1..1_000 => 'X'));       -- Erlaubt,   weil F2   flexibel ist.
11    F2 := (3, "ABC");                       -- Erlaubt,   weil F2   flexibel ist.
```

Jeder Verbund des Untertyps **FLEX** (vereinbart in Zeile 02 bis 04) hat genau **zwei** Komponenten namens **L** und **S**. Die **S**-Komponente ist vom Untertyp **string** und ihre Länge hängt von der Diskriminantenkomponenten **L** ab.

In der Vereinbarung der Variablen **F1** wird der Wert der Diskriminanten L **explizit** festgelegt und ist somit **nicht** änderbar. Bei **F2** wird der Diskriminantenwert zunächsteinmal durch die Vorbesetzung **:= 0** (siehe Zeile 02) bestimmt und kann beliebig oft geändert werden. Schon bei der Initialisierung der Variablen **F2** mit dem Wert **(6, "Hallo!")** bekommt die Diskriminante **F2.L** den Wert **6** und die normale Komponente **F2.S** den dazu passenden Wert **"Hallo!"**. Durch die Zuweisung in Zeile 10 bekommt die Diskriminante **F2.S** den Wert **1_000** und die Komponente **F2.S** wird entsprechend zu einem String der Länge **1_000** vergrößert. In Zeile 11 bekommt **F2.L** den Wert 3 und die Komponente **F2.S** "schrumpft" entsprechend zu einem String der Länge 3.

Aufgabe 15.2.3.: Wieviele Varianten gibt es von dem Untertyp **FLEX**, d.h. wieviele verschiedene Werte kann die Diskriminante **L** annehmen? Wie "groß" ist der größte Verbund des Untertyps FLEX? Genauer: Wieviele Zeichen vom Untertyp **character** müssen in die **S**-Komponente des größten Verbundes des Untertyps **FLEX** hineinpassen? **Achtung**: Die Antworten auf diese Fragen sind nicht bei allen Ada-Ausführern gleich. **o**

Für einen **menschlichen** Ausführer, der Ada-Programme mit Papier und Bleistift ausführt, sind **uneingeschränkte Verbundvariablen** wie **F2** nicht besonders schwer zu handhaben. Für einen heute üblichen **maschinellen** Ausführer ist es dagegen gar nicht leicht zu entscheiden, wieviel Speicherplatz er beim Erzeugen einer Variablen wie **F2** reservieren soll. Einige Ada-Ausführer reservieren für **jede** uneingeschränkte Variable des Untertyps **FLEX** soviel Speicherplatz, wie die **maximale Variante** einer solchen Variablen belegen würde. Praktisch bedeutet das: Einige Ada-Ausführer können die Variable **F2** nicht erzeugen, weil sie mehr als zwei Gigabyte Hauptspeicher belegen würde und der betreffende Computer nicht so viel Hauptspeicher besitzt. Die zwei Gigabyte haben

damit zu tun, daß die größte Ganzzahl des Untertyps **positive** auf vielen heute üblichen Rechnern gleich **2******31-1** (also ungefähr gleich 2_000_000_000) ist und der längste String theoretisch soviele Komponenten haben kann. Das folgende Beispiel zeigt, wie der **Programmierer** einem solchen Ada-**Ausführer** durch einen diskreten Hinweis **helfen** und ihn von unnötiger Speicherplatzverschwendung abhalten kann.

Beispiel 15.2.7.: Realistische Zeichenketten mit flexibler Länge:

```
01 declare
02    subtype NAT100 is natural range 1..100;
03    type FLEXR(L: NAT100 := 0) is
04       S : string(1..L);
05    end record;
06    F3: FLEXR(L => 6) := (6, "Hallo!");      -- F3 ist "unflexibel"
07    F4: FLEXR         := (6, "Hallo!");      -- F4 ist   "flexibel"
08 begin
09    F3 := (6, "Alloh!);                      -- Erlaubt, obwohl F3 unflexibel ist
10    F3 := (5, "Hello");                      -- Verboten,  weil F3 unflexibel ist.
11    F4 := (1_00, (1..1_00 => 'X'));          -- Erlaubt,   weil F4   flexibel ist.
12    F4 := (3, "ABC");                        -- Erlaubt,   weil F4   flexibel ist.
```

Zum Untertyp **NAT100** (vereinbart in Zeile 02) gehören nur die Ganzzahlen von 1 bis 100. Weil die Diskriminante **L** des Untertyps **FLEXR** (Zeile 03) vom Untertyp **NAT100** ist, kann jeder Ada-Ausführer erkennen, daß die **string**-Komponente **S** höchstens "100 Zeichen lang sein kann". Für die **uneingeschränkte** ("flexible") Variable **F4** muß selbst ein einfacher und vorsichtiger Ada-Ausführer nicht mehr als ungefähr **104** Byte reservieren (4 Byte für die L-Komponenten und 100 Byte für die S-Komponente). Für die **eingeschränkte** ("unflexible") Variable **F3** sind sogar nur etwa **10** Byte nötig (4 Byte für die L-Komponente und 6 Byte für die S-Komponente, deren Länge sich nicht ändern kann). **o**

Für die Realisierung von **uneingeschränkten Verbundvariablen** auf einem heute üblichen Computer gibt es nicht **eine beste Lösung**, sondern verschiedene Lösungen mit Vor- und Nachteilen. Meistens sind die Lösungen, die **weniger Speicherplatz** kosten, tendenziell etwas **langsamer** und die **schnelleren Lösungen** kosten etwas **mehr Speicherplatz**.

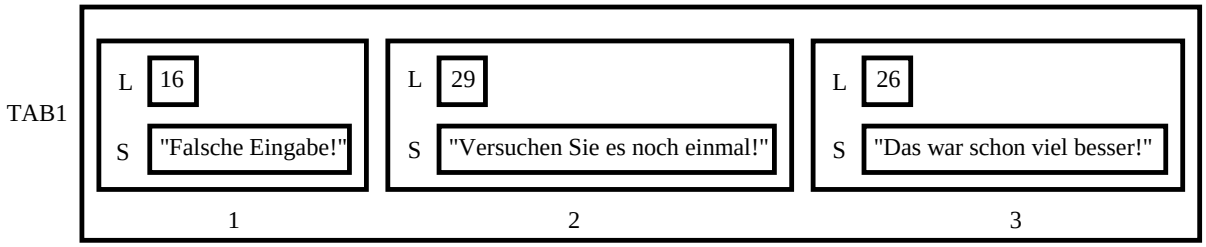
Angenommen, der Programmierer will mehrere Texte **unterschiedlicher Länge** in einer Reihung speichern, damit er leicht und schnell auf die einzelnen Texte zugreifen kann. Eine Reihung, deren Komponenten einfach zum Untertyp **string** gehören, ist nicht erlaubt, weil **string** ein uneingeschränkter und damit **indefiniter** ("als Bauplan unvollständiger") Untertyp ist. Der Verbundtyp **FLEXR** (vereinbart im Beispiel 15.2.7.) ist zwar auch **uneingeschränkt**, aber trotzdem **definit** und somit als Komponentenuntertyp eines Reihungstyps erlaubt, z.B. so:

Beispiel 15.2.8.: Eine nützliche Anwendung des Untertyps **FLEXR**:

```
01 declare
02    type TEXT_TABELLE is array(positive range <>) of FLEXR;
03    TAB1 : constant TEXT_TABELLE(1..3) :=
04       ((1 => (L => 16, S => "Falsche Eingabe!"),
05        (2 => (L => 29, S => "Versuchen Sie es noch einmal!"),
06        (3 => (L => 26, S => "Das war schon viel besser!")
07       );
08 begin
09    ...
10    ada.text_io.put_line(item => TAB1(1).S);
11    ...
12    ada.text_io.put_line(item => TAB1(3).S);
13    ...
14    for I in TAB1'range loop
```

```
15       ada.text_io.put_line(item => TAB1(I).S);
16    end loop;
17    ...
```

Weil der Untertyp FLEXR **definit** ("als Bauplan vollständig") ist, darf man ihn als **Komponentenuntertyp** des Reihungsuntertyps **TEXT_TABELLE** verwenden (Zeile 02). Die Reihung **TAB1** wird hier als **Konstante** vereinbart, weil die Texte, die sie enthält, nicht verändert werden sollen. Die Reihungskonstante **TAB1** hat den Indexbereich **1..3** und enthält somit 3 Verbunde des Untertyps FLEXR als Komponenten (Zeile 03 bis 07). In Zeile 10 bezeichnet der Teilausdruck **TAB1(1)** die erste Komponente von **TAB1** und **TAB1(1).S** die **S**-Komponente dieser ersten Komponente, d.h. den String "Falsche Eingabe!". Hier eine bildliche Darstellung der Reihung **TAB1**:



Genau genommen handelt es sich hier um eine **Reihung** TAB1 von **Verbunden** TAB1(I), von denen jeder unter anderem eine Komponente TAB1(I).S besitzt, die wiederum eine **Reihung** (vom Untertyp **string**) ist. **o**

Ein uneingeschränkter Verbunduntertyp kann mehr als eine Diskriminante haben. Von den Diskriminanten können weitere Komponenten abhängen, aber das muß nicht sein sein, wie das folgende Beispiel zeigt:

Beispiel 15.2.9.: Ein Verbunduntertyp mit **zwei Diskriminanten**:

```
01 declare
02    type GROESSE is range 34..48;
03    type FARBE   is (SCHWARZ, BRAUN, WEISS, BUNT, GESTREIFT);
04    type ART     is (DAMEN, HERREN, KINDER, TURN);
05    type ANZAHL  is range 0..100;
06    type SCHUH(F: FARBE := FARBE'first; A: ART := ART'first) is record
07       GR  : GROESSE := GROESSE'first;
08       ANZ : ANZAHL  := 0;
09    end record;
10    subtype SCHWARZER_HERREN_SCHUH is SCHUH(F => SCHWARZ, A => HERREN);
11    subtype BUNTER_TURN_SCHUH      is SCHUH(F => BUNT,    A => TURN);
12    subtype BUNTER_SCHUH           is SCHUH(F => BUNT); -- verboten!
13    S1, S2 : SCHUH;
14 begin
15    ... -- Werte nach S1 und S3 bringen
16    if S1     in SCHWARZER_HERREN_SCHUH then ... end if;
17    is S2 not in BUNTER_TURN_SCHUH      then ... end if;
```

Der Untertyp **SCHUH** (vereinbart in Zeile 06 bis 09) hat **zwei** Diskriminanten **F** und **A** und zwei normale Komponenten **GR** und **ANZ**. Die Komponenten hängen nicht von den Diskriminanten ab. Der Untertyp **SCHWARZER_HERREN_SCHUH** (vereinbart in Zeile 10) besteht aus dem Typ **!SCHUH** und der Diskriminanteneinschränkung **(F => SCHWARZ, A => HERREN)** und ist somit eingeschränkt. Die Variablen **S1** und **S2** (vereinbart in Zeile 13) sind **uneingeschränkt** (d.h. "flexibel"). In Zeile 16 wird geprüft, ob der momentane Wert der Variablen **S1** zum Untertyp

SCHWARZER_HERREN_SCHUH gehört, d.h. ob **S1.F** gleich **SCHWARZ** und **S1.F** gleich **HERREN** ist. **o**

Eine Diskriminante kann man nur durch **einen** bestimmten Wert einschränken, nicht durch einen **Bereich** von Werten. Wenn man, ausgehend von einem uneingeschränkten Verbunduntertyp, weitere Untertypen vereinbart, muß man entweder **alle** Diskriminanten einschränken oder **keine**. Nur einen Teil der Diskriminanten einzuschränken (wie das in Zeile 12 versucht wird) ist nicht erlaubt.

Aufgabe 15.2.4.: Wieviele Varianten des Untertyps SCHUH gibt es, d.h. auf wieviele Weisen kann man den Typ **!SCHUH** einschränken?

Im Beispielprogramm VERBU_04 wird gezeigt, wie man einen **Ganzzahluntertyp** und einen **Bruchzahluntertyp** zu **einem** Verbunduntertyp mit **zwei Varianten** zusammenfassen kann. Bruchzahltypen werden im Abschnitt 21 behandelt.

Im Beispielprogramm VERBU_03 wird gezeigt, wie man die **Maßeinheiten physikalischer Größen** (z.B. Meter pro Sekunde, Kilopond pro Quadratmeter, Kilopond mal Meter etc.) besondrs elegant durch drei Diskriminanten eines uneingeschränkten Verbunduntertyps realisieren kann.

Zusammenfassung 15.2.:

- Zur "**Art**" einer **Verbundkomponenten** gehören ihr **Untertyp**, ihr **Selektorname** und ihre **Größe** (falls die Komponente selbst z.B. eine Reihung oder ein Verbund ist).
- Ein **eingeschränkter** Verbunduntertyp legt die **Anzahl** und **Art** der Komponenten seiner Verbunde **genau fest**.
- Zu einem **uneingeschränkten** Verbunduntertyp können Verbunde mit **unterschiedlich vielen** oder **unterschiedlich gearteten** Komponenten gehören.
- Die Vereinbarung eines uneingeschränkten Verbunduntertyps erkennt man daran, daß sie nicht nur **normale** Komponenten sondern auch **Diskriminanten** beschreibt.
- Aus den Werten der Diskriminanten eines Verbundes folgt eindeutig, **wieviele** Komponenten der Verbund hat und **von welcher Art** sie sind.
- Ein **eingeschränkter** Verbunduntertyp beschreibt keine Diskriminanten und ist immer **definit** ("als Bauplan vollständig").
- Ein **uneingeschränkter** Verbunduntertyp ist **definit**, wenn für jede Diskriminante eine **Vorbesetzung** festgelegt wurde, andernfalls ist er **indefinit** ("als Bauplan unvollständig").
- In der Vereinbarung eines (uneingeschränkten) Verbunduntertyps müssen alle Selektornamen **verschieden voneinander** sein, auch wenn sie zu **verschiedenen Varianten** gehören.

16. Typen und Untertypen, ein paar begriffliche Feinheiten

In Ada ist es manchmal wichtig, **Typen** und **Untertypen** zu unterscheiden. Durch eine **Typvereinbarung** wie z.B.

```
01    type GANZ is range -5_000..+10_000;
```

wird ein neuer **Typ** !GANZ und sein **erster Untertyp** namens GANZ erzeugt. Durch eine **Untertypvereinbarung** wie z.B.

```
02    subtype KLEIN is GANZ range -10..+5;
```

wird nur ein neuer **Untertyp** (vom Typ !GANZ) erzeugt. Ein **Untertyp** besteht aus einem **Typ** und einer **Einschränkung**. Z.B. besteht der Untertyp GANZ aus dem Typ !GANZ und der Bereichseinschränkung -5_000..+10_000. Der Untertyp KLEIN besteht aus dem Typ !GANZ und der Bereichseinschränkung -10..+5. Ein Untertyp kann einen Namen haben, muß aber nicht, wie die folgenden Vereinbarungen verdeutlichen sollen:

```
03    G1:  GANZ range -17..+35:
04    type TAB is array(GANZ range 0..20) of GANZ range 100..200;
```

Die Variable **G1** gehört zu einem **anonymen Untertyp**, der aus dem Typ !GANZ und der Einschränkung -17..35 besteht. Der Indexuntertyp und der Komponentenuntertyp des Reihungstyps **!TAB** sind ebenfalls anonyme Untertypen (die aus dem Typ !GANZ und der Einschränkun 0..20 bzw. 100..200 bestehen).

Typen haben in Ada grundsätzlich keine Namen. Ein Typ gilt als **anonym**, wenn auch keiner seiner **Untertypen** einen Namen hat, wie im folgenden Beispiel:

```
05    TAB1, TAB2: array(KLEIN) of GANZ;
```

Die Reihungsvariable **TAB1** gehört zu einem **anonymen Reihungstyp** (und zu einem anonymen Untertyp dieses Reihungstyps). Die Reihung TAB2 gehört ebenfalls zu einem anonymen Reihungstyp, aber zu einem anderen. Praktisch bedeutet das: Die folgende Vereinbarung enthält einen Typfehler und ist somit verboten:

```
06 begin
07    TAB1 := TAB2; -- Typfehler, verboten!
```

Die Unterscheidung von **Typen** und **Untertypen** hat viel mit Effizienz zu tun: Fast alle **Typprüfungen** können schon bei der **Übergabe** eines Programmtextes an den Ausführer ("zur Compilezeit") ein für allemal erledigt werden. Während der **Ausführung** eines Programms ("zur Laufzeit") müssen dann nur noch **Untertypprüfungen** durchgeführt werden. **Typen** sind demnach weitgehend **statische** Gebilde, deren Eigenschaften vollständig aus dem Programmtext folgen müssen und nicht von dynamischen Größen (z.B. von eingelesenen Werten) abhängen können. Dagegen können **Untertypen** in Ada durchaus **dynamisch** sein, wie das folgende Beispiel deutlich machen soll:

Beispiel 16.1.: Ein Untertyp mit **dynamischen** ("eingelesenen") **Grenzen**:

```
01 with ada.text_io;
02 procedure TYPEN_01 is
03    VON, BIS, CHAR: character;
04 begin -- TYPEN_01
05    ada.text_io.put(item => "Drei Zeichen VON, BIS und CHAR? ");
06    ada.text_io.get(item => VON);
07    ada.text_io.get(item => BIS);
08    ada.text_io.get(item => CHAR);
09    BLOCK1: declare
10       subtype DYNAM is character range VON..BIS;
11       DYN1 :  DYNAM;
12    begin -- BLOCK1
13       ada.text_io.put(item => "Liegt CHAR im Bereich VON..BIS? ");
13       if CHAR in DYNAM then
14          DYN1 := CHAR;
15          ada.text_io.put_line(item => "Ja!");
16       else
17          ada.text_io.put_line(item => "Nein!");
18       end if;
19    end BLOCK1;
20 end TYPEN_01;
```

Die Grenzen des Untertyps **DYNAM** (vereinbart in Zeile 10) werden **eingelesen** (in Zeile 06 und 07). Wichtige Eigenschaften des Untertyps **DYNAM** werden also erst während jeder Ausführung des Programms **TYPEN_01** festgelegt und können von Ausführung zu Ausführung verschieden sein. Die Grenzen eines **Typs** sind dagegen immer statisch und werden schon bei der Übergabe eines Programms "für alle Ausführungen" festgelegt. **o**

Ein **Ganzzahltyp** besteht aus einer Menge von Werten (Ganzzahlen) und einer Menge von Operationen wie "+", "-", "*", "/" etc.. Diese **vordefinierten** Operationen gehören zum **Typ** und nicht nur zu irgendeinem seiner **Untertypen**. Z.B. löst die Operation "+" für den Typ !GANZ keine Ausnahme aus, wenn ihr Ergebnis nicht mehr zum **Untertyp** GANZ gehört. Erst wenn die Grenzen des **Typs** !GANZ überschritten werden, tritt ein **constraint_error** auf.

Anders verhält es sich mit Unterprogrammen, die der Programmierer vereinbart hat:

Beispiel 16.2.: Unterprogramme für den Untertyp **GANZ**:

```
01 with ada.text_io;
02 procedure TYPEN_02 is
03    type     GANZ    is -5_000..+10_000;
04    package  GANZ_EA is new ada.text_io.integer_io(num => GANZ);
05    function PLUS3(G: GANZ) return GANZ is
06    begin
07       return G + 3;
08    end PLUS3;
```

Mit den Prozeduren **GANZ_EA.get** und **GANZ_EA.put** kann man nur Werte des **Untertyps** GANZ einlesen bzw. ausgeben. Ganz entsprechend löst die Funktion **PLUS3** die Ausnahme **constraint_error** aus, wenn ihr Ergebnis nicht mehr zum **Untertyp** GANZ gehört. Unterprogramme wie GANZ_EA.get, GANZ_EA.put und PLUS3 gehören also zum **Untertyp** GANZ, nicht zum **Typ** !GANZ. **o**

Das nächste Beispiel zeigt, wie der Programmierer Unterprogramme erstellen kann, die alle Werte des **Typs** !GANZ "ausnützen", und nicht nur die Werte des **Untertyps** GANZ:

Beispiel 16.2.: Unterprogramme für den Basisuntertyp **GANZ'base**:

```
01 with ada.text_io;
02 procedure TYPEN_03 is
03    type     GANZ      is -5_000..+10_000;
04    package  GANZ_EA_B is new ada.text_io.integer_io(num => GANZ'base);
05    function PLUS3_B(G: GANZ'base) return GANZ'base is
06    begin
07       return G + 3;
08    end PLUS3;
```

Zum Basisuntertyp **GANZ'base** gehören alle Werte, die zum **Typ** !GANZ gehören. Mit den Prozeduren **GANZ_EA_B.get** und **GANZ_EA_B.put** kann man also alle Werte des **Typs** !GANZ einlesen bzw. ausgeben, nicht nur Werte des **Untertyps** GANZ. Entsprechend löst die Funktion **PLUS3_B** keine Ausnahme aus, solange ihr Ergebnis noch zum Basisuntertyp **GANZ'base** (d.h. zum **Typ** !GANZ) gehört. Unterprogramme wie GANZ_EA_B.get, GANZ_EA_B.put und PLUS3_B gehören also zum **Typ** !GANZ, nicht nur zum **Untertyp** GANZ. **o**

Ein Reihungs**typ** legt die **Anzahl der Dimensionen**, für jede Dimension einen **Indexuntertyp** und schließlich den **Komponentenuntertyp** seiner Reihungen fest, aber keine konkreten **Indexgrenzen**. Zu einem Reihungs**typ** gehören also immer Reihungen **unterschiedlicher** Länge. Das ist notwendig, damit alle **Teilreihungen** einer Reihung R zum selben **Typ** wie R gehören können. **Uneingeschränkt** oder **eingeschränkt** zu sein ist keine Eigenschaft eines Reihungs**typs**, sondern eine Eigenschaft eines **Untertyps**.

Ein Verbund**typ** legt fest, wieviele **Diskriminanten** (null oder mehr) seine Verbunde haben, zu welchen **Untertypen** diese Diskriminanten gehören und welche anderen "normalen" Komponenten zu den einzelnen Diskriminantenwerten gehören. Nur die **Untertypen** eines Verbundtyps mit Diskriminanten können **eingeschränkt** oder **uneingeschränkt** sein.

Auch bei **Ganzzahlen** und **Bruchzahlen** (Gleitpunktzahlen und Festpunktzahlen) unterscheidet man **eingeschränkte** und **uneingeschränkte** Untertypen. Ein uneingeschränkter Untertyp besteht aus einem Typ und einer **leeren Einschränkung**, ein eingeschränkter Untertyp besteht aus einem Typ und einer **nicht-leeren Einschränkung**. Der erste Untertyp eines **Ganzzahlttyps** ist immer **eingeschränkt**, der erste Untertyp eines **Gleitpunkttyps** ist **uneingeschränkt** (wenn der Programmierer in der Typvereinbarung nicht ausdrücklich eine Einschränkung angegeben hat). Was das genau bedeutet wird im (ARM 3) erklärt. Besonders wichtig sind die Stellen (ARM 3.5.4(9) und (10), 3.5.7(11) und (12) und 3.5.6(7)). Ist T ein Ganzzahluntertyp oder ein Bruchzahluntertyp, dann ist der Basisuntertyp **T'base** ein **uneingeschränkter** Untertyp.

Auch die Eigenschaft **definit** ("als Bauplan vollständig") oder **indefinit** ("als Bauplan unvollständig") zu sein, ist eine Eigenschaft von **Untertypen**, nicht von **Typen**. Z.B. ist der **Untertyp** string **indefinit**, man kann aber leicht auch **definite** Untertypen desselben Typs !string vereinbaren, z.B. so:

Beispiel 16.3.: Definite Untertypen des Typs !string:

```
01 subtype STRING10      is string(1..10);
02 subtype KURZER_STRING is string(positive'last-2..positive'last);
03 S1 : STRING10;
04 S2 : KURZER_STRING;
```

Jeder String des Untertyps **KURZER_STRING** besteht aus nur drei Komponenten. Diese drei Komponenten haben aber die drei größten Indizes, die ein String haben kann. Die Variablenverein-

barungen in den Zeilen 03 und 04 sollen deutlich machen, daß die Untertypen **STRING10** und **KURZER_STRING** beide **definit** ("als Bauplan vollständig") sind. **o**

Zusammenfassung 16.:

- **Typen** sind **statische** Gebilde. Fast alle **Typprüfungen** finden schon bei der **Übergabe** eines Programmtextes an den Ausführer statt.
- **Untertypen** können **dynamisch** sein. Die meisten **Untertypprüfungen** können erst während der **Ausführung** eines Programms stattfinden.
- Nur **Untertypen** können **eingeschränkt** oder **uneingeschränkt** sein.
- Nur **Untertypen** können **definit** oder **indefinit** sein.

Zentraldokument: Nach Filialdokument A95-15-16

17. Unterprogramme

Die einfache **Grundidee** eines **Unterprogramms** wurde schon im Abschnitt 3.3. skizziert und wird hier noch einmal kurz geschildert. Wenn der Programmierer eine bestimmte Folge von Befehlen mehr als einmal braucht, kann er die Folge mit einem **Namen** versehen und sie zu einem **Unterprogramm** machen. Der Name des Unterprogramms ist dann ein **neuer Befehl**, der für die ganze Befehlsfolge steht. Der Programmierer kann dem Ausführer diesen neuen Befehl beliebig oft geben (d.h. **das Unterprogramm aufrufen**), indem er im wesentlichen den Namen des Unterprogramms an die betreffenden Stellen seines Programms schreibt (statt die ganze Befehlsfolge des Unterprogramms an jede dieser Aufrufstellen zu kopieren).

Selbst wenn der Programmierer eine bestimmte Befehlsfolge nur **einmal** braucht, kann es vorteilhaft sein, daraus ein Unterprogramm zu machen. Indem er ein **großes** Programm in mehrere **überschaubare Unterprogramme** gliedert und die Unterprogramme mit **guten** ("sprechenden") **Namen** versieht, kann er den Kollegen und sich selbst das Lesen und **Verstehen** seines Programms erheblich **erleichtern**.

Es gibt (in Ada wie in den meisten anderen Programmiersprachen auch) **zwei Arten** von Unterprogrammen: **Funktionen** und **Prozeduren**. Der wesentliche Unterschied zwischen Funktionen und Prozeduren hat mit dem grundlegenden Unterschied zwischen **Ausdrücken** ("Berechne den Wert des Ausdrucks ... ") und **Anweisungen** ("Tue den Wert ... in den Behälter ...") zu tun: Jeder Aufruf einer **Funktion** ist ein **Ausdruck** und **liefert** einen Wert. Dagegen ist jeder Aufruf einer **Prozedur** eine **Anweisung** (und liefert keinen Wert). Mit einem **Funktionsaufruf** befiehlt der Programmierer dem Ausführer also immer, einen Wert zu berechnen (ohne dabei irgendwelche Behälterinhalte zu verändern) und mit einem **Prozeduraufruf** befiehlt er ihm, die Inhalte bestimmter Behälter zu verändern. "Einen Wert **liefern**" ist ein Fachausdruck, den man nur zur Beschreibung von **Funktionen** verwenden sollte.

Die zahlreichen Befehle namens **get** und **put** (in den Paketen **ada.text_io**, **OTTO_EA**, **BOOL_EA** etc.) sind Beispiele für **Prozeduren**. Operationen wie "+", "-", ... etc., "<", "=", "<=", ... etc., und Attribute wie **character'succ**, **character'pred** etc. sind Beispiele für **Funktionen**. Daß alle bisher erwähnten Beispielfunktionen so "ungewöhnliche Namen" wie "+", "-" etc. und keine "gewöhnlichen Namen" wie **SINUS**, **COSINUS**, **SUMME** etc. haben, ist eher ein Zufall und wird sich bald ändern.

Beispiel 17.1.: Aufrufe von Funktionen und Prozeduren:

```
01    Z1 : character := character'pred('A');
02 begin
03    ada.text_io.get(item => Z1);
04    if Z1 = 'A' then ... end if;
05    while Z1 < character'succ('Z') loop ... end loop;
```

Der **Aufruf** der **get**-Prozedur (in Zeile 03) verändert die Inhalte von **zwei** Behältern: Er entfernt einen **character**-Wert aus dem **Tastaturbehälter** und weist ihn der Variablen **Z1** zu.

Der (fett hervorgehobene) **Ausdruck** in Zeile 01 besteht aus einem **Aufruf** der Funktion **character'pred**. Daß der Ausführer den Wert dieses Ausdrucks als Anfangswert "in die Variable Z1 legt", wird ihm nicht durch den **Ausdruck** befohlen, sondern durch den **Initialisierungsteil** ":=" der Vereinbarung. Der Ausdruck **Z1 = 'A'** (in Zeile 04) besteht aus einem **Aufruf** der Gleichheits-

funktion "=" und verändert keine Behälterinhalte. Im Ausdruck **Z1 < character'succ('Z')** (in Zeile 05) werden **zwei** Funktionen aufgerufen: zuerst **character'succ** und dann die Kleinerfunktion "<". Wenn der Ausführer den Wert des gesamten Ausdrucks berechnet, "merkt er sich" gewisse Zwischenergebnisse (in seinem "Kurzzeitgedächtnis"), er legt diese Zwischenergebnisse aber **nicht** in irgendwelche dem Programmierer bekannte und zugängliche **Behälter**. Auch der Ausdruck in Zeile 05 bewirkt also nicht, daß irgendwelche Wertebehälter verändert werden. **o**

Der Programmierer kann selbst neue **Funktionen** und **Prozeduren vereinbaren** (d.h. vom Ausführer **erzeugen** lassen) und sie dann beliebig oft **aufrufen**. Z.B. bestehen sehr viele Ada-Programme aus einer vom Programmierer vereinbarten **Hauptprozedur** (und evtl. weiteren Unterprogrammen, Paketen und anderen Programmeinheiten).

Je nachdem **wie** der Programmierer ein Unterprogramm vereinbart, unterscheidet man zwei Arten: **enthaltene Unterprogramme** und **Bibliotheksunterprogramme**. Ein Bibliotheksunterprogramm ist "eine selbständige Einheit", die dem Ausführer unabhängig von anderen Einheiten übergeben und von ihm in seiner **Bibliothek** abgelegt wurde. Wird ein Unterprogramm U dagegen im **Vereinbarungsteil** einer Programmeinheit P vereinbart, dann ist U ein (in P) **enthaltenes** Unterprogramm.

Beispiel 17.2.: Ein **Bibliotheksunterprogramm** und zwei **enthaltene Unterprogramme**:

```
01 with ada.text_io;
02 procedure HALLO is
03    TEXT : constant string := "Hallo!";
04    procedure HALLO1 is
05       TEXT : constant string := "Hello one";
06    begin -- HALLO1
07       ada.text_io.put_line(item => TEXT);
08    end HALLO1;
09    procedure HALLO2 is
10       TEXT : constant string := "Alloh alloh!");
11    begin -- HALLO2
12       ada.text_io.put_line(item => TEXT);
13    end HALLO2;
14 begin -- HALLO
15    HALLO2;
16    ada.text_io.put_line(item => TEXT);
17    HALLO1;
18    HALLO2;
19 end HALLO;
```

HALLO ist eine **Bibliotheksprozedur** und **enthält** die beiden Prozeduren **HALLO1** (vereinbart in Zeile 04 bis 08) und **HALLO2** (vereinbart in Zeile 09 bis 13). **Vor** der Vereinbarung eines Bibliotheksunterprogramms wie **HALLO** dürfen **Kontextklauseln** stehen (siehe Zeile 01), vor der Vereinbarung eines **enthaltenen Unterprogramms** wie HALLO1 und HALLO2 **nicht**. **o**

Aufgabe 17.1.: Wieviele "Dinge" werden im Vereinbarungsteil der Prozedur **HALLO** (im vorigen Beispiel) vereinbart? Wie heißen diese Dinge? Um was für Dinge handelt es sich (um Typen oder um Variablen oder um ...)?. Welche Zeichenketten erscheinen auf dem Bildschirm, wenn der Ausführer das Programm **HALLO** ausführt? **o**

Aufgabe 17.2.: Übersetzen Sie zuerst die enthaltene Prozedur HALLO1 und dann die Bibliotheksprozedur HALLO ins Deutsche. **o**

Der wesentliche Unterschied zwischen einem **Bibliotheksunterprogramm** wie HALLO und einem **enthaltenen Unterprogramm** wie **HALLO1** hat mit der Frage zu tun, **von wo aus** man die Unterprogramme **aufrufen** kann. Das (in **HALLO**) enthaltene Unterprogramm **HALLO1** kann nur im Anweisungsteil von **HALLO** (der im Beispiel oben die Zeilen 15 bis 18 belegt) aufgerufen werden. Daß **HALLO** in irgendwelchen Programmteilen außerhalb von HALLO **nicht** aufgerufen werden kann, garantiert der Ada-Ausführer. Wenn eine Prozedur z.B. dazu dient, eine Wäscheschleuder oder einer Rakete zu starten, dann kann es sehr wichtig sein, daß die Prozedur nur "an wenigen Stellen eines Programms" aufgerufen werden kann.

Im Gegensatz zu **HALLO1** kann das Bibliotheksunterprogramm **HALLO** auch von jeder anderen Bibliothekseinheit aus aufgerufen werden, wenn diese andere Einheit mit einer entsprechenden Kontextklausel eingeleitet wird, z.B. so:

Beispiel 17.3.: Ein **Bibliotheksunterprogramm** ruft ein anderes auf:

```
01 with HALLO, ...;
02 procedure IRGEND_EINE is
03    ...
04 begin
05    ...
06    HALLO;
07    ...
08 end IRGEND_EINE;
```

Im Anweisungsteil der Bibliotheksprozedur **IRGEND_EINE** wird die Bibliotheksprozedur **HALLO** aufgerufen. **o**

Das Konzept eines Unterprogramms ist im Grunde genommen ganz einfach. Es wird aber häufig mit einer weiteren Idee kombiniert, nämlich mit der Parameter-Idee. **Unterprogramme mit Parametern** sind wesentlich **flexibler** und **nützlicher** als Unterprogramme **ohne Parameter**, aber auch ein bißchen schwieriger zu verstehen. Im nächsten Abschnitt werden **Prozeduren** (**ohne** Parameter und **mit** Parametern) behandelt und im übernächsten Abschnitt dann **Funktionen**.

Zusammenfassung 17.:

- Das Konzept eines **Unterprogramms** ist (nach dem Konzept einer beliebig oft veränderbaren Variablen) das **zweitwichtigste** Grundkonzept der meisten Programmiersprachen.
- Unterprogramme **mit Parametern** sind noch wesentlich flexibler und nützlicher als Unterprogramme **ohne** Parameter (aber auch ein bißchen schwieriger zu verstehen).
- Es gibt **zwei Arten** von Unterprogrammen: **Funktionen** und **Prozeduren**.
- Jeder Aufruf einer **Funktion** ist ein **Ausdruck** und **liefert einen Wert**. Normalerweise werden durch das Aufrufen einer Funktion **keine Behälterinhalte verändert**.
- Jeder Aufruf einer **Prozedur** ist eine **Anweisung** und dient dazu, bestimmte Behälterinhalte zu verändern (z.B. die Inhalte von Variablen oder die Inhalte von Geräten wie Bildschirmen, Tastaturen, Magnetplatten etc.).
- In Ada unterscheidet man **Bibliotheksunterprogramme** (die in der Bibliothek des Ausführers liegen und praktisch "von überall her aufgerufen werden können") und **enthaltene Unterprogramme** (die man nur in **der** Einheit aufrufen kann, in der sie **enthalten** sind).

17.1. Prozeduren

In früheren Abschnitten dieses Skripts kamen schon zahlreiche **Prozeduren ohne Parameter** vor. Trotzdem soll hier noch ein weiteres einfaches Beispiel angeführt werden:

Beispiel 17.1.1.: Eine enthaltene Prozedur **ohne Parameter**

```
01 with ada.text_io;
02 procedure UPROS_01 is
03    procedure DOPPEL_LINIE_1 is
04       LINIE : constant string(1..80) := (others => '-');
05    begin -- DOPPEL_LINIE_1
06       ada.text_io.put_line(item => LINIE);
07       ada.text_io.put_line(item => LINIE);
08    end DOPPEL_LINIE_1;
09    ...
10 begin -- UPROS_01
11    ...
12    DOPPEL_LINIE_1;
13    ...
```

Man beachte, daß in den Zeilen 03 bis 08 nur der Befehl steht, eine Prozedur namens **DOPPEL_LINIE_1** zu **erzeugen**, aber **nicht** der Befehl, diese Prozedur auch **auszuführen**. Den Befehl in Zeile 12 kann man dagegen so ins Deutsche übersetzen: Führe die Prozedur DOPPEL_LINIE_1 aus. **o**

Wenn der Programmierer das Problem hat, daß er zwei Minuszeichenlinien der Länge 80 auf dem Bildschirm benötigt, dann braucht er nur die Prozedur **DOPPEL_LINIE_1** aufzurufen, indem er ihren Namen gefolgt von einem Semikolon als Anweisung hinschreibt (siehe Zeile 12). Allerdings ist die Prozedur **DOPPEL_LINIE_1** sehr **unflexibel** und kann nur **ein einziges Problem** lösen. Wenn der Programmierer z.B. zwei Minuszeichenlinien der **Länge 60** (statt 80) braucht, oder wenn er zwei **Pluszeichenlinien** benötigt, kann er sein Problem **nicht** mit der Prozedur **DOPPEL_LINIE_1** lösen. Das folgende Beispiel zeigt eine Prozedur **mit Parametern**, die deutlich mehr Probleme lösen kann als **DOPPEL_LINIE_1**:

Beispiel 17.1.2.: Eine Prozedur **mit zwei Parametern:**

```
01 with ada.text_io;
02 procedure UPROS_02 is
03    procedure DOPPEL_LINIE_2(LAENGE: natural; ZEICHEN: character) is
04       LINIE : constant string(1..LAENGE) := (others => ZEICHEN);
05    begin -- DOPPEL_LINIE_2
06       ada.text_io.put_line(item => LINIE);
07       ada.text_io.put_line(item => LINIE);
08    end DOPPEL_LINIE_2;
09    N1 : natural   := 50;
10    Z1 : character := 'X';
11    ...
12 begin -- UPROS_02
13    ...
14    DOPPEL_LINIE_2(LAENGE => N1 + 10, ZEICHEN => '+');
15    ...
16    DOPPEL_LINIE_2(LAENGE => 80, ZEICHEN => character'succ(Z1));
17    ...
```

Die Prozedur **DOPPEL_LINIE_2**, vereinbart in den Zeilen 03 bis 08, hat zwei **formale Parameter**. Der erste heißt **LAENGE** und gehört zum Untertyp **natural**, der zweite heißt **ZEICHEN** und ist vom Typ **character** (siehe Zeile 03). Diese Parameter dürfen in der Prozedur **DOPPEL_LINIE_2** wie zwei **Konstanten** verwendet werden. In Zeile 04 wird mit dem Parameter **LAENGE** die Länge und mit dem Parameter **ZEICHEN** der Wert der Stringkonstanten **LINIE** festgelegt. Die

formalen Parameter (d.h. Konstanten) **LAENGE** und **ZEICHEN** werden bei jeder Ausführung der Prozedur DOPPEL_LINIE_2 **neu erzeugt** und ihre Werte werden durch Ausdrücke festgelegt, die in jedem Aufruf der Prozedur angegeben werden müssen.

Im **Prozeduraufruf** in Zeile 14 wird der Wert des formalen Parameters **LAENGE** durch den Ausdruck **N1 + 10** beschrieben, der Wert des Parameters **ZEICHEN** durch den Ausdruck **'+'**. Man bezeichnet den Ausdruck **N1 + 10** auch als den **aktuellen Parameter** für den **formalen Parameter LAENGE** und den Ausdruck **'+'** als den **aktuellen Parameter** für den **formalen Parameter ZEICHEN**.

Im **Prozeduraufruf** in Zeile 16 wird für den **formalen** Parameter LAENGE der **aktuelle** Parameter 80 und für den **formalen** Parameter ZEICHEN der **aktuelle** Parameter **character'succ(Z1)** angegeben.

Den Prozeduraufruf in Zeile 14 führt der Ausführer folgendermaßen aus:
1. Er berechnet die **Werte** der aktuellen Parameter **N1 + 10** und **'+'**. Als Ergebnis bekommt er die Werte **60** und **'+'** heraus.
2. Mit diesen Werten erzeugt er die **formalen Parameter** LAENGE und ZEICHEN, so als wären sie folgendermaßen vereinbart worden:

```
LAENGE  : constant natural   := 60;
ZEICHEN : constant character := '+';
```

3. Er führt die **Vereinbarungen** im Vereinbarungsteil der Prozedur **DOPPEL_LINIE_2** aus (in diesem Beispiel ist es nur **eine** Vereinbarung, in Zeile 04). Er erzeugt also eine Konstante namens **LINIE** vom Untertyp **string** mit den Indizes **1..60** und dem Wert **(others => '+')**.
4. Dann führt er die **Anweisungen** im Anweisungsteil der Prozedur **DOPPEL_LINIE_2** aus (zwei **put_line**-Anweisungen in Zeile 06 bis 07).
5. Schließlich **zerstört** er die formalen Parameter **LAENGE** und **ZEICHEN** und die Konstante **LINIE** wieder und
6. macht dann hinter der Aufrufstelle (also bei Zeile 15) weiter.

Die **Wirkung** dieses Prozeduraufrufs besteht darin, daß zwei Zeichenketten, jede aus **60 Pluszeichen** bestehend, zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgegeben werden. **o**

Aufgabe 17.1.1.: Beschreiben Sie in allen Einzelheiten, wie der Ausführer den Prozeduraufruf in Zeile 16 (im vorigen Beispiel) ausführt. Was wird durch diesen Prozeduraufruf zum Bildschirm ausgegeben? **o**

In diesem Beispiel gilt also:
Die **formalen Parameter** LAENGE und ZEICHEN sind **Konstanten**, die am Anfang der **Vereinbarung** der Prozedur DOPPEL_LINIE_2 (in Zeile 03) vereinbart sind. Die Werte dieser Konstanten stammen aber von Ausdrücken, die in jedem **Aufruf** der Prozedur als **aktuelle Parameter** angegeben werden müssen. Die Konstanten LAENGE und ZEICHEN werden am Anfang jeder Ausführung der Prozedur DOPPEL_LINIE_2 neu erzeugt und können jedesmal andere Werte haben. **Formale Parameter** "leben" **innerhalb** des Unterprogramms, **aktuelle Parameter** werden dagegen in jedem **Aufruf** des Unterprogramms (außerhalb des Unterprogramms) angegeben.

Aufgabe 17.1.2.: Schreiben Sie eine Prozedur namens **MULTI_LINIE** mit **drei** formalen Parametern. Mit dieser Prozedur soll man nicht nur **beliebig lange** Linien aus einem **beliebigen Zeichen** ausgeben können (wie schon bei **DOPPEL_LINIE_2**), sondern auch die **Anzahl der Linien** von Aufruf zu Aufruf anders bestimmen können. **o**

Bei der Prozedur **DOPPEL_LINIE_2** dienen beide Parameter (**LAENGE** und **ZEICHEN**) dazu, Werte von der jeweiligen Aufrufstelle **in** die Prozedur hinein zu transportieren. In der umgekehrten Richtung (von der Prozedur zur Aufrufstelle) findet kein Wertetransport statt. Deshalb bezeichnet man die Parameter LAENGE und ZEICHEN auch als **in-Parameter.** Wenn der Programmierer will, kann er die Parameter (in der ersten Zeile der Prozedur **DOPPEL_LINIE_2**, siehe oben Zeile 03) auch ausdrücklich als **in-Parameter** kennzeichnen, etwa so:

```
03a    procedure DOPPEL_LINIE_2(LAENGE: in natural; ZEICHEN: in character) is
```

Im folgenden Beispiel findet ein Wertetransport **aus** der Prozedur hinaus zu den aktuellen Parametern der jeweiligen Aufrufstelle statt. Damit ein solcher Transport überhaupt möglich ist, muß der Programmierer als aktuelle Parameter **Variablen** angeben, und nicht irgendwelche komplizierteren Ausdrücke wie **N1 + 10** oder **character'succ(Z1)**. Denn "zu einer Variable" kann man einen Wert "hintransportieren" (d.h. man kann den Wert der Variablen zuweisen), aber was sollte es bedeuten, einen Wert zu einem Ausdruck wie **N1 + 10** hinzutransportieren? Außerdem **muß** man die formalen Parameter ausdrücklich als **out-Parameter** kennzeichen, z.B. so:

Beispiel 17.1.3.: Eine Prozedur mit zwei **out-Parametern**:

```
01 with ada.text_io;
02 procedure UPROS_03 is
03    ZIFFER_GELESEN : boolean;
04    Z1, Z2         : character;
05    procedure GET_ZIFFER(ITEM : out character; ALLES_OK: out boolean) is
06       TEXT: constant string := "Bitte geben Sie eine Ziffer ein: ");
07    begin -- GET_ZIFFER
08       ada.text_io.put(item => TEXT);
09       ada.text_io.get(item => ITEM);
10       if ITEM in '0'..'9' then
11          ALLES_OK := true;
12       else
13          ALLES_OK := false;
14       end if;
15    end GET_ZIFFER;
16 begin -- UPROS_03
17    GET_ZIFFER(ALLES_OK => ZIFFER_GELESEN, ITEM => Z2);
18    if not ZIFFER_GELESEN then
19       ...
20    end if;
21    GET_ZIFFER(ITEM => Z1, ALLES_OK => ZIFFER_GELESEN);
22    if ZIFFER_GELESEN then ...
```

Die Prozedur **GET_ZIFFER** (vereinbart in Zeile 05 bis 15) hat zwei formale **out**-Parameter namens **ITEM** (vom Untertyp **character**) und **ALLES_OK** (vom Untertyp **boolean**). Diese **out**-Parameter dürfen in der Prozedur **GET_ZIFFER** wie (nicht-initialisierte) **Variablen** verwendet weden. In Zeile 09 wird in die formale-out-Parameter-Variable **ITEM** ein Zeichen "hineingelesen". Der formalen-out-Parameter-Variablen **ALLES_OK** wird in Zeile 11 bzw. 13 ein Wert ihres Untertyps **boolean** zugewiesen.

Den Prozeduraufruf in Zeile 17 führt der Ausführer folgendermaßen aus:
0. Er ignoriert die **aktuellen Parameter** ZIFFER_GELESEN und Z2 erstmal und macht **nichts** mit ihnen.
1. Er erzeugt die **formalen Parameter** ITEM und ALLES_OK so, als wären sie folgendermaßen vereinbart worden:

```
ITEM     : character;
ALLES_OK : boolean;
```

Man beachte, daß die Variablen **ITEM** und **ALLES_OK** vom Ausführer **nicht initialisiert** werden, d.h. sie enthalten **irgendeinen** sinnvollen oder unsinnigen Wert. Der Programmierer sollte auf **jeden** Wert gefaßt sein.
2. Der Ausführer führt die **Vereinbarungen** im Vereinbarungsteil der Prozedur **GET_ZIFFER** aus (es ist auch in diesem Beispiel nur **eine** Vereinbarung, in Zeile 06).
3. Dann führt der Ausführer die **Anweisungen** im Anweisungsteil der Prozedur **GET_ZIFFER** aus (**drei** Anweisungen in den Zeilen 08 bis 14, nämlich eine **put**-Anweisung in Zeile 08, eine **get**-Anweisung in Zeile 09 und eine **if-then-else**-Anweisung in den Zeilen 10 bis 14).
4. Dann weist er den aktuellen Parametern (**ZIFFER_GELESEN** und **Z2**) die Werte der formalen Parameter (**ALLES_OK** und **ITEM**) zu:

```
ZIFFER_GELESEN := ALLES_OK;
Z2             := ITEM;
```

Diese Zuweisungen finden statt, weil beide Parameter als **out**-Parameter gekennzeichnet sind. Die Zuweisungen stellen einen Wertetransport **aus** der Prozedur zu den aktuellen Parametern der Aufrufstelle (in Zeile 17) dar.
5. Schließlich zerstört der Ausführer die Variablen **ALLES_OK** und **ITEM** und die Konstante **TEXT** wieder und
6. macht dann hinter der Aufrufstelle (also bei Zeile 18) weiter.

Die Wirkung dieses Prozeduraufrufs besteht darin, daß ein Zeichen von der aktuellen Eingabe (d.h. von der Tastatur) letztlich in die Variable **Z2** eingelesen wird und der Variablen **ZIFFER_GELESEN** der Wert **true** bzw. **false** zugewiesen wird, je nachdem, ob das eingelesene Zeichen tatsächlich eine Ziffer ist oder nicht. **o**

Aufgabe 17.1.3.: Beschreiben Sie in allen Einzelheiten, wie der Ausführer den Prozeduraufruf in Zeile 21 (im vorigen Beispiel) ausführt. Wie könnte man die Wirkung dieses Prozeduraufrufs zusammenfassend beschreiben? **o**

Achtung: Die Pfeile "=>" in einem Prozeduraufruf zeigen immer von einem **formalen** zum entsprechenden **aktuellen** Parameter. Sie haben nichts mit der Richtung irgendeines **Wertetransports** zu tun und zeigen höchstens zufällig in die gleiche Richtung.

Der Sinn und Zweck eines **in**-Parameters ist es, den Wert des entsprechenden aktuellen Parameters (im **Aufruf** der Prozedur) **in** die Prozedur hinein zu transportieren, ein **out**-Parameter ist dazu da, einen Wert **aus** der Prozedur hinaus zum entsprechenden aktuellen Parameter (im Aufruf der Prozedur) zu transportieren. Was ist dann wohl der Sinn und Zweck eines **in-out**-Parameters? Das nächste Beispiel soll helfen, diese Frage zu beantworten:

Beispiel 17.1.4.: Eine Prozedur mit einem **in-out-Parameter**:

```
01 with ada.text_io;
02 procedure UPROS_04 is
03    type  GANZ is range -1_000..+1_000;
04    AG1 : GANZ := +17;
05    AG2 : GANZ := -35;
06    procedure PLUS3(FG: in out GANZ) is
07       DREI : constant GANZ := 3;
08    begin
09       FG := FG + DREI;
10    end PLUS3;
11 begin
12    ...
13    PLUS3(FG => AG1)
```

```
14    ...
15    PLUS3(FG => AG2);
16    ...
```

Die Prozedur **PLUS3** (vereinbart in Zeile 06 bis 10) hat einen **in-out**-Parameter namens **FG** vom Untertyp **GANZ**. Dieser **in-out**-Parameter darf in der Prozedur **PLUS3** wie eine (**initialisierte!**) **Variable** verwendet werden. In der Zeile 09 wird dem Ausführer befohlen, den Wert des Ausdrucks **FG + DREI** zu berechnen. Das geht nur deshalb gut, weil **FG** initialisiert ist. Außerdem wird dem Ausführer in der gleichen Zeile befohlen, den Wert des Ausdrucks **FG + DREI** der Variablen **FG** zuzuweisen. Das geht natürlich nur, weil **FG** eine **Variable** ist und keine Konstante.

Den Prozeduraufruf in Zeile 13 führt der Ausführer folgendermaßen aus:
1. Er ermittelt den Wert des aktuellen Parameters **AG1**. Wir nehmen hier an, daß dieser Wert gleich +17 ist.
2. Mit diesem Wert erzeugt der Ausführer den formalen Parameter **FG** so, als wäre er folgendermaßen vereinbart worden:

```
FG : GANZ := +17;
```

Man beachte, daß die Variable **FG** mit dem Wert des aktuellen Parameters (AG1) **initialisiert** wird. Dies geschieht, weil der Parameter **FG** (nicht nur als **out**-, sondern auch) als **in**-Parameter gekennzeichnet ist.
3. Er führt die **Vereinbarungen** im Vereinbarungsteil der Prozedur **PLUS3** aus (auch in diesem Beispiel ist es nur **eine** Vereinbarung, in Zeile 07).
4. Dann führt er die **Anweisungen** im Anweisungsteil der Prozedur **PLUS3** aus (eine Zuweisung in Zeile 09).
5. Dann weist er dem **aktuellen** Parameter **AG1** den Wert des **formalen** Parameters **FG** zu:

```
AG1 := FG;
```

Dies geschieht, weil der Parameter **FG** (nicht nur als **in**-, sondern auch) als **out**-Parameter gekennzeichnet ist.
6. Schließlich zerstört der Ausführer die Konstante **DREI** und die Variable **FG** wieder und
7. macht dann hinter der Aufrufstelle (also in Zeile 14) weiter.

Die Wirkung dieses Prozeduraufrufs besteht darin, daß der Wert der Variablen **AG1** um drei (genauer: um DREI) erhöht wird. **o**

Aufgabe 17.1.4.: Beschreiben Sie in allen Einzelheiten, wie der Ausführer den Prozeduraufruf in Zeile 15 (im vorigen Beispiel) ausführt. Wie könnte man die Wirkung dieses Prozeduraufrufs zusammenfassend beschreiben? **o**

Jeder formale Parameter einer Prozedur hat einen **Modus**. Es gibt in Ada insgesamt **drei** Parameter-**Modi** (oder: **Modusse**): **in**, **out** und **in out**. Wenn der Programmierer für einen Parameter **keinen** Modus angibt, dann hat der Parameter automatisch den Modus **in**. Hier eine Übersicht über die wichtigsten Regeln, die mit dem Modus eines Prozedurparameters zusammenhängen:

Modus des **formalen Parameters** FPAR einer Prozedur PROZ	In einem **Aufruf** der Prozedur PROZ ist als **aktueller Parameter** für FPAR erlaubt	Der **formale Parameter** FPAR hat in der Prozedur PROZ den **Charakter** einer
in	ein beliebiger **Ausdruck**	**Konstanten**
out	nur der Name einer **Variablen**	nicht-initialisierten **Variablen**
in out	nur der Name einer **Variablen**	initialisierten **Variablen**

Eine Prozedur kann im Prinzip **beliebig viele** formale Parameter haben, und der Programmierer kann den **Modus** von jedem einzelnen Parameter unabhängig von den anderen Parametern festlegen. Praktisch sollte man aber bedenken, daß Prozeduren mit **vielen** Parametern im allgemeinen schwer zu verstehen und aufzurufen sind, weil man von jedem Parameter genau wissen muß, "was man mit ihm anrichten kann". Als Faustregel mag gelten, daß **drei** Parameter ganz gut und **fünf** noch erträglich sind. Mehr Parameter sollte man höchstens in gut begründeten Sonderfällen vorsehen.

Analogie: Sehr gute und teure HiFi-Anlagen haben typischerweise nur sehr wenige Knöpfe und Schalter. Anlagen minderer Qualität verwirren den Benutzer dagegen häufig mit zu vielen Bedienelementen. Der Programmierer sollte möglichst hochwertige HiFi-Unterprogramme mit wenigen Knopfparametern schreiben.

Aufgabe 17.1.5.: Schreiben Sie eine Prozedur namens **UPROS_05**, in der ein Ganzzahltyp namens **!GANZ** und eine Prozedur namens **ADD_MULT** vereinbart wird. Die enthaltene Prozedur **ADD_MULT** soll **vier** Parameter vom Untertyp **GANZ** haben. Nach einem Aufruf der Prozedur soll im **dritten** Parameter die **Summe** und im **vierten** Parameter das **Produkt** der **ersten beiden** Parameter stehen. Welchen **Modus** sollten die einzelnen Parameter haben? Gestalten Sie die Prozedur **UPROS_05** so, daß der Benutzer zwei Ganzzahlen eingeben kann, daß dann die Prozedur **ADD_MULT** mit diesen Zahlen als Parameter aufgerufen wird und danach die Ergebnisse der Prozedur ausgegeben werden. **o**

Wenn eine Prozedur **keine** Parameter hat, ist sie im allgemeinen **unflexibel**. Wenn sie **viele** Parameter hat, dann ist es **mühsam**, bei jedem Aufruf für jeden **formalen** Parameter einen passenden **aktuellen** Parameter anzugeben. Speziell für **in**-Parameter bietet Ada deshalb eine Notation, mit der man die Angabe von aktuellen Parametern in einem Aufruf **optional** ("darf aber muß nicht") machen kann. Diese Notation ist unter dem Fachbegriff **Vorbesetzungen für in-Parameter** bekannt. Hier ein Beispiel:

Beispiel 17.1.5.: Vorbesetzungen für in-Parameter:

```
01 with ada.text_io;
02 procedure UPROS_09 is
03    procedure MULTI_LINIE(       LAENGE : in natural   := 80;
04                                 ZEICHEN: in character := '-';
05                                 ANZAHL : in natural   := 1)   is
06       LINIE : constant string(1..LAENGE) := (others => ZEICHEN);
07    begin -- MULTI_LINIE
08       for I in natural range 1..ANZAHL loop
09          ada.text_io.put_line(item => LINIE);
10    end MULTI_LINIE;
11 begin -- UPROS_09
12    MULTI_LINIE;
13    MULTI_LINIE(ANZAHL  =>  2 );
14    MULTI_LINIE(ZEICHEN => '+');
15    MULTI_LINIE(ZEICHEN => '#', ANZAHL => 3, LAENGE => 50);
16    ...
```

Die Prozedur **MULTI_LINIE** (vereinbart in den Zeilen 03 bis 10) hat **drei** formale **in**-Parameter namens **LAENGE**, **ZEICHEN** und **ANZAHL**. In den Zeilen 03 bis 05 ist für jeden formalen Parameter ein **Vorbesetzungsausdruck** angegeben worden (80, '-' bzw. 1). Damit entfällt für diese **formalen Parameter** die **Notwendigkeit**, in jedem Aufruf der Prozedur **MULTI_LINIE** einen entsprechenden **aktuellen Parameter** anzugeben. Wenn man für einen formalen Parameter mit Vorbesetzung **keinen** aktuellen Parameter angibt, wird die Vorbesetzung wirksam.

Im Prozeduraufruf in Zeile 12 wurde **kein einziger** aktueller Parameter angegeben. Der Aufruf bewirkt, daß **eine** Zeile bestehend aus **80 Minuszeichen** ausgegeben wird (entsprechend den Vorbesetzungen der drei Parameter). Der Aufruf in Zeile 13 bewirkt, daß **2** Zeilen (zu je 80 Minuszeichen) ausgegeben werden. Der Aufruf in Zeile 14 gibt eine Zeile bestehend aus 80 **Pluszeichen** aus etc.. **o**

Im vorigen Beispiel waren **alle** in-Parameter der Prozedur MULTI_LINIE mit Vorbesetzungen versehen worden. Es ist aber auch erlaubt, z.B. von **drei** in-Parametern nur **einen** oder **zwei** vorzubesetzen und die bzw. den anderen Parameter **nicht**.

In-Parameter mit Vorbesetzungen wurden (ohne Erklärung) schon in früheren Abschnitten dieses Skripts verwendet. Z.B. hat die Prozedur **OTTO_EA.put** zwei **in**-Parameter namens **width** und **base**, für die man aktuelle Werte angeben **kann**, aber **nicht muß**. Ähnliches gilt für Prozeduren wie **OTTO_EA.put**, **FARBE_EA.get** und **FARBE_EA.put** (siehe Abschnitte 9.7. und 10.4.).

Beendet wird die Ausführung einer Prozedur normalerweise, nachdem die letzte Anweisung im Anweisungsteil ausgeführt wurde ("wenn der Ausführer zum abschließenden **end** der Prozedur kommt"). Mit der **return**-Anweisung kann der Programmierer dem Ausführer aber auch an einer beliebigen Stelle des Anweisungsteils befehlen, die Ausführung der Prozedur zu beenden und "an die Stelle zurückzukehren, an der die Prozedur aufgerufen wurde". Eine **return**-Anweisung in der Hauptprozedur eines Programms beendet das gesamte Programm (die **return**-Anweisung bewirkt in diesem Fall sozusagen eine **Rückkehr in das Betriebssystem** des Rechners, von wo aus das Programm aufgerufen wurde).

Beispiel 17.1.6.: Prozeduren mit **return**-Anweisungen beenden:

```
01 with ada.text_io;
02 procedure UPROS_07 is
03    type    GANZ    is range 0..1_000;
04    package GANZ_EA is new ada.text_io.integer_io(num => GANZ);
05    G :     GANZ;
05    procedure PUT_PRIM(ZAHL: in GANZ) is
06    begin
07       for TEILER in GANZ range 2..ZAHL / 2 loop
08          if ZAHL mod TEILER = 0 then
09             ada.put_line(item => "G ist keine Primzahl!");
10             return; -- Beende die Prozedur PUT_PRIM!
11          end if;
12       end loop;
13       ada.text_io.put_line(item => "G is eine Primzahl!");
14    end PUT_PRIM;
15 begin -- UPROS_07
16    loop
17       ada.text_io.put(item => "Eine Ganzzahl G (0..1000, 0 zum Beenden)? ");
18       GANZ_EA.get(item => G);
19       if G = 0 then
20          return; -- Beende das Programm UPROS_07!
21       end if;
22       PUT_PRIM(ZAHL => G);
23    end loop;
24 end UPROS_07;
```

Die **return**-Anweisung in Zeile 10 befindet sich im Anweisungsteil der enthaltenen Prozedur namens **PUT_PRIM**. Also beendet sie nur diese Prozedur. Die **return**-Anweisung in Zeile 20 befindet sich im Anweisungsteil der Hauptprozedur **UPROS_07**. Also beendet sie das **gesamte**

Programm UPROS_07. Das Unterprogramm **PUT_PRIM** untersucht, ob sein Parameter ZAHL eine Primzahl ist oder nicht und gibt eine entsprechende Meldung aus. **o**

Im Prinzip darf eine Prozedur **beliebig viele return**-Anweisungen enthalten. Praktisch sollte man aber bedenken, daß viele **return**-Anweisungen eine Prozedur im allgemeinen **unübersichtlich** und **schwer verständlich** machen. Einige Firmen schreiben ihren Programmierern vor, daß ein Unterprogramm nur "einen Ausgang" haben darf, d.h. daß die Ausführung des Unterprogramms nur an **einer** Stelle seines Textes beendet werden darf. Damit darf eine Prozedur also höchstens eine **return**-Anweisung enthalten.

Das folgende Beispiel soll die Nützlichkeit **uneingeschränkter Reihungsuntertypen** demonstrieren und zeigen, daß es zwischen einem **out**- oder **in-out**-Parameter und einer "normalen **Variablen**" neben vielen Ähnlichkeiten auch einen wichtigen **Unterschied** gibt.

Beispiel 17.1.7.: Ein out-Parameter eines **uneingeschränkten Reihungsuntertyps**:

```
01 with ada.text_io;
02 procedure UPROS_08 is
03    AS1 : string(1..3); -- Ein String der Laenge 3
04    AS2 : string(4..9); -- Ein String der Laenge 6
05    ------------------------------------------------------------
06    procedure FUELLE_MIT_01(FS: out string) is
07    -- Fuellt FS mit den Zeichen '0' und '1' (immer abwechselnd).
08    begin
09       for I in natural range 0..FS'length - 1 loop
10          if I mod 2 = 0 then
11             FS(FS'first+I) := '0';
12          else
13             FS(FS'first+I) := '1';
14          end if;
15       end loop;
16    end FUELLE_MIT_01;
17    ------------------------------------------------------------
18 begin -- UPROS_08
19    FUELLE_MIT_01(FS => AS1);
20    FUELLE_MIT_01(FS => AS2)
21    ...
```

Die Prozedur **FUELLE_MIT_01** (vereinbart in Zeile 06 bis 16) hat einen **out**-Parameter namens **FS** vom (uneingeschränkten Reihungs-) Untertyp **string**. Man beachte, daß in Zeile 06 für den Parameter **FS** nur der Untertyp **string**, aber **keine konkreten Indexgrenzen** angegeben wurden (wie es in einer **Variablenvereinbarung** zwingend notwendig wäre). Ein (out- oder in-out-) **Parameter** ist also etwas ganz **Ähnliches** wie eine **Variable**, aber nicht genau das **Gleiche**.

Wenn er den Prozeduraufruf in Zeile 19 ausführt, erzeugt der Ausführer den formalen Parameter **FS** so, als wenn er folgendermaßen vereinbart worden wäre:

```
FS : string := AS1;
```

Dabei ist **AS1** der **aktuelle** Parameter, der im Aufruf in Zeile 19 angegeben wurde. Der **formale** Parameter **FS** erhält somit bei jeder Ausführung der Prozedur **FUELLE_MIT_01** seine **Indexgrenzen** von dem **aktuellen** Parameter. **o**

Man kann die Prozedur **FUELLE_MIT_01** also auf alle Strings anwenden, die zum (uneingeschränkten) Untertyp **string** gehören, egal, wie lang diese Strings sind oder welche konkreten

Indexgrenzen sie haben. Das wird durch den uneingeschränkten Reihungsuntertyp **string** des formalen Parameters **FS** genau ausgedrückt.

Zusammenfassung 17.1.:

- Der Programmierer kann Prozeduren **vereinbaren** ("vom Ausführer erzeugen lassen") und dann **aufrufen** ("ausführen lassen").
- Eine **Prozedur** kann im Prinzip **beliebig viele** (null oder mehr) **Parameter** haben.
- Jeder Prozedurparameter hat einen **Modus**: **in**, **out** oder **in out**.
- Mit einem **in-Parameter** wird ein Wert von der Aufrufstelle **in** die Prozedur transportiert, mit einem **out-Parameter** wird ein Wert **aus** der Prozedur zur Aufrufstelle transportiert und ein **in-out-Parameter** ist an zwei Wertetransporten (einem "**in**-Transport" und einem "**out**-Transport") beteiligt.
- Für einen (formalen) **in**-Parameter darf man in einem Prozeduraufruf als **aktuellen** Parameter einen **beliebigen Ausdruck** angeben, für einen **out**- oder **in-out**-Parameter nur den Namen einer **Variablen** (damit beim **out**-Transport ein Wert in diese Variable gebracht werden kann).
- Für einen **in**-Parameter kann man in der **Prozedurvereinbarung** eine **Vorbesetzung** angeben. In einem **Aufruf** der Prozedur muß man dann nicht unbedingt einen aktuellen Parameter angeben.
- Einem formalen Prozedurparameter kann man einen **uneingschränkten Reihungsuntertyp** ("einen unvollständigen Bauplan") zuordnen, **ohne** diesen Bauplan durch die Angabe konkreter Indexgrenzen zu vervollständigen (was in einer Variablenvereinbarung zwingend notwendig ist). Wenn der **formale Parameter** dann erzeugt wird, übernimmt er die Indexgrenzen des **aktuellen Parameters**.

17.2. Funktionen

Im folgenden Beispiel wird eine Funktion namens **ABSOLUT_KLEINER** zuerst **vereinbart** und dann mehrmals **aufgerufen**. Mit dieser Funktion kann man die **absoluten Werte** (die Beträge) zweier Ganzzahlen **vergleichen**. Die Funktion ABSOLUT_KLEINER ist in einer Prozedur namens UPROS_21 **enthalten**.

Beispiel 17.2.1.: Eine **Funktion** mit zwei Parametern:

```
01 with ada.text_io;
02 procedure UPROS_21 is
03    type GANZ is range -5_000..+10_000;
04    AG1, AG2: GANZ := 0;
05    B1      : boolean;
06    function ABSOLUT_KLEINER(FP1, FP2: GANZ) return boolean is
07       BETRAG1: constant GANZ := abs(FP1);
08       BETRAG2: constant GANZ := abs(FP2);
09    begin -- ABSOLUT_KLEINER
10       return BETRAG1 < BETRAG2;
11    end ABSOLUT_KLEINER;
12    ...
13 begin -- UPROS_21
14    ...
15    if ABSOLUT_KLEINER(FP1 => AG1, FP2 => 2 * AG2 + 1) then ... end if;
16    ...
17    B1 := ABSOLUT_KLEINER(FP2 => 17, FP1 => AG1 + AG2);
18    ...
```

In den Zeilen 06 bis 11 wird dem Ausführer befohlen, eine Funktion namens ABSOLUT_KLEINER zu **erzeugen**. In Zeile 15 und in Zeile 17 wird ihm jeweils befohlen, diese Funktion **auszuführen**.

Die Funktion **ABSOLUT_KLEINER** hat zwei **formale Parameter** namens **FP1** und **FP2** vom Untertyp **GANZ** und liefert ein Ergebnis vom Untertyp **boolean** (siehe Zeile 06).

Das Wort **return** kommt in der Vereinbarung der Funktion **zweimal** vor. Das **return** in Zeile 06 bedeutet etwa: "Diese Funktion liefert ein Ergebnis vom Untertyp ... ". Dadurch wird der **Ergebnisuntertyp** der Funktion (im Beispiel der Untertyp **boolean**) festgelegt. Dieses erste **return** ist **kein** eigenständiger Befehl an den Ausführer, sondern ein kleiner, aber wichtiger Bestandteil der Funktionsvereinbarung. In Zeile 10 steht dagegen ein eigenständiger Befehl, nämlich eine **return-Anweisung**, die man etwa so ins Deutsche übersetzen kann:

"Berechne den Wert des Ausdrucks **BETRAG1 < BETRAG2**, liefere diesen Wert als Ergebnis der Funktion an die Stelle, von der aus die Funktion aufgerufen wurde und beende die Ausführung der Funktion **ABSOLUT_KLEINER**".

In den Zeilen 07 und 08 wird die Funktion **abs** aufgerufen, die der Ausführer zusammen mit dem Ganzzahltyp **!GANZ** erzeugt. Diese Funktion hat **einen** Parameter vom Typ **!GANZ** und liefert den **abs**oluten Wert diese Parameters (z.B. ist **abs(-3)** gleich **+3** und **abs(+7)** gleich **+7**).

Eine Funktionsvereinbarung (wie die in Zeile 06 bis 11) muß **mindestens eine** und darf im Prinzip **beliebig viele** return-**Anweisungen** enthalten. Hinter dem **return** einer return-Anweisung muß ein Ausdruck stehen, der zum Ergebnisuntertyp der Funktion (im Beispiel: **boolean**) gehört. Wenn der Ausführer eine **return**-Anweisung ausführt, beendet er die Ausführung der Funktion und liefert den Wert des Ausdrucks hinter **return** als Ergebnis der Funktion. **o**

Angenommen, die Variablen **AG1** und **AG2** haben die Werte **-17** und **+5** und der Ausführer kommt im Verlauf einer Ausführung der Prozedur **UPROS_21** zur Zeile 15. Den Funktionsaufruf **ABSOLUT_KLEINER(FP1 => AG1, FP2 => 2 * AG2 + 1)** in dieser Zeile führt er dann folgendermaßen aus:

1. Er berechnet die Werte der aktuellen Parameter **AG1** und **2 * AG2 + 1**. Als Ergebnisse erhält er die Werte **-17** und **+11**.
2. Mit diesen Werten erzeugt er die formalen Parameter FP1 und FP2, so als wären sie wie folgt vereinbart worden:

```
FP1 : constant GANZ := -17;
FP2 : constant GANZ := +11;
```

3. Er führt die **Vereinbarungen** im **Vereinbarungsteil** der Funktion (Zeile 07 bis 08) aus. Er erzeugt also eine Konstante namens **BETRAG1** vom Untertyp GANZ mit dem Wert **+17** (das ist der **abs**olute Wert des Parameters **FP1**). Entsprechend wird die GANZ-Konstante **BETRAG2** mit dem Wert **+11** erzeugt.
4. Dann führt er die **Anweisungen** im **Anweisungsteil** der Funktion aus. In diesem einfachen Beispiel ist das nur die eine **return-Anweisung** in Zeile 10. Die wird ausgeführt, indem der Ausführer den Wert des Ausdrucks **BETRAG1 < BETRAG2** berechnet, sich diesen Wert **false** "als Ergebnis der Funktion **merkt**", die Parameter **FP1** und **FP2** sowie die Konstanten **BETRAG1** und **BETRAG2** zerstört, die Ausführung der Funktion beendet und mit dem Ergebnis **false** zur Aufrufstelle in Zeile 15 zurückkehrt. Dort steht jetzt sozusagen "if **false** then ... end if".

Aufgabe 17.2.1.: Beschreiben Sie in allen Einzelheiten, wie der Ausführer den Aufruf der Funktion **ABSOLUT_KLEINER** in Zeile 17 ausführt. Welchen Wert liefert dieser Funktionsaufruf als Ergebnis? **o**

In Ada kann eine **Funktion** nur Parameter mit dem Modus **in** haben. Man kann diesen Modus in der **Funktionsvereinbarung** ausdrücklich angeben (z.B. in Zeile 06: (FP1, FP2: **in** GANZ)) oder auch weglassen, weil er selbstverständlich ist. Die formalen Parameter einer Funktion haben also grundsätzlich den Charakter von **Konstanten**. Das paßt zu der Tatsache, daß jeder Aufruf einer Funktion ein **Ausdruck** ist und die Auswertung eines Ausdrucks normalerweise **nicht** den Effekt haben sollte, daß die Inhalte irgendwelcher Behälter (seien es Parameter, andere Variablen oder Geräte wie Bildschirme, Tastaturen etc.) verändert werden. In einem **Aufruf** einer Funktion darf man entsprechend für jeden formalen (**in**-) Parameter einen beliebig komplizierten oder einfachen **Ausdruck** des betreffenden Untertyps angeben, ganz wie bei **in**-Parametern einer Prozedur.

In den Aufrufen der Funktion **ABSOLUT_KLEINER** (siehe obiges Beispiel, Zeile 15 und 17) erfolgte die Zuordnung der aktuellen Parameter zu den formalen Parameter **über die Namen** der formalen Parameter (z.B. in Zeile 17: **(FP2 => 17, FP1 => AG1 + AG2)**). Statt dessen darf der Programmierer die Zuordnung auch **über die Position** der aktuellen Parameter vornehmen, z.B. so:

```
15a    if ABSOLUT_KLEINER(AG1, 2*AG2+1) then ... end if;
16a    ...
17a    B1 := ABSOLUT_KLEINER(AG1 + AG2, 17);
```

In den folgenden Abschnitten wird meistens diese kürzere Notation für Funktionsaufrufe verwendet.

Aufgabe 17.2.2.: Vereinbaren Sie in einer Prozedur namens **UPROS_22** eine Funktion namens **ADD5**, mit der man bis zu **fünf** Ganzzahlen des Untertyps **GANZ** auf einmal addieren kann. Richten Sie es durch geschicktes Hinschreiben bzw. Weglassen von **Vorbesetzungsausdrücken** so ein, daß man bei jedem Aufruf von **ADD** mindestens **zwei** aktuelle Parameter angeben **muß**, aber wahlweise auch **drei**, **vier** oder **fünf** aktuelle Parameter angeben **kann**. **o**

Aufgabe 17.2.3.: Vereinbaren Sie in einer Prozedur namens **UPROS_23** eine Funktion namens **MAX5**, mit der man aus bis zu **fünf** Ganzzahlen des Untertyps **GANZ** die größte herausfinden kann. Es soll aber möglich sein, die Funktion auf **null**, **ein**, **zwei**, **drei**, **vier** oder **fünf** Parameter anzuwenden. Zur Erinnerung: Die Attributfunktion **GANZ'max** hat zwei Parameter vom Typ **!GANZ** und liefert den größeren ihrer beiden Parameter als Ergebnis. **o**

Aufgabe 17.2.4.: Vereinbaren Sie in einer Prozedur namens **UPROS_24** eine Funktion namens **MAX_TEILER**, mit einem Parameter vom Untertyp **positive** und einem Ergebnis ebenfalls vom Untertyp **positive**. Die Funktion soll als Ergebnis den **größten echten Teiler** ihres Parameters liefern oder die Zahl 1 (falls der Parameter eine Primzahl ist). Beispiele: MAX_TEILER(12) ist gleich 6 und MAX_TEILER(17) ist gleich 1. **o**

Aufgabe 17.2.5.: Vereinbaren Sie in einer Prozedur namens **UPROS_25** eine Funktion namens **MIN_TEILER**, mit einem Parameter vom Untertyp **positive** und einem Ergebnis ebenfalls vom Untertyp **positive**. Die Funktion soll als Ergebnis den **kleinsten Teiler** (größer als 1) ihres Parameters liefern oder den Parameter selbst (falls er eine Primzahl ist). Beispiele: MIN_TEILER(12) ist gleich 2 und MIN_TEILER(17) ist gleich 17. **o**

Die bisher behandelten Funktionen (ABSOLUT_KLEINER, ADD5, MAX5 etc.) hatten "**normale Namen**", die mit einem Buchstaben beginnen und **nicht** zur Liste der reservierten Wörter gehören (siehe (ARM 2.9)). Funktionen mit normalen Namen darf man nur in der sogenannten **Präfixnotation** aufrufen. Bei dieser Notation schreibt man den Funktionsnamen **vor** die Parameter und schließt

die Parameter in Klammern ein. Die Zordnung der aktuellen zu den formalen Parametern kann wahlweise über die **Namen** der formalen Parameter erfolgen (z.B. **ABSOLUT_KLEINER(FP2 => 17, FP1 => AG1 + AG2)**) oder über die **Position** der aktuellen Parameter (z.B. **ABSOLUT_KLEINER(AG1 + AG2, 17)**).

Der Programmierer darf aber auch die folgenden 19 **Operatorsymbole** als Namen für seine Funktionen verwenden:

```
+   -   *   /   **   &   =   /=   <   <=   >   >=
MOD  REM  ABS  AND  OR  XOR  NOT
```

Man beachte, daß die sieben Operatorsymbole in der zweiten Zeile **reservierte Wörter** sind und deshalb **nicht** als Namen für Typen, Konstanten und Variablen etc. in Frage kommen.

Funktionen mit einem **Operatorsymbol** als Namen werden auch als **Operationen** bezeichnet. Das folgende Beispiel zeigt, wie der Programmierer eine Operation **vereinbaren** und **aufrufen** kann.

Beispiel 17.2.2.: Eine Divisionsoperation **"&"** für Ganzzahlen, die immer **abrundet**:

Die Ganzzahldivision "/" rundet immer **zur 0 hin**. Z.B. ist **-5 / 3** gleich **-1** (und nicht gleich **-2**). Die im folgenden vereinbarte Ganzzahldivision **"&"** rundet immer **nach links**. Z.B. ist **-5 & 3** gleich **-2** (und nicht gleich **-1**). Siehe dazu auch die Erläuterungen zu **rem** und **mod** im Abschnitt 9.6..

```
01 with ada.text_io;
02 procedure UPROS_26 is
...
10    type    GANZ    is range -999..+999;
11    package GANZ_EA is new ada.text_io.integer_io(num => GANZ);
12    G1, G2: GANZ;
...
15    ------------------------------------------------------------------------
16    function "&"(LINKS, RECHTS: GANZ) return GANZ is
17    -- Eine Ganzzahldivision, die immer abrundet, auch bei negativen
18    -- Ergebnissen, z.B. so:
19    -- +7 & +3 ist gleich 2, aber -7 & +3 ist gleich -3
20    ------------------------------------------------
21       QUOTIENT            : constant GANZ    := LINKS  /  RECHTS;
22       REST                : constant GANZ    := LINKS rem RECHTS;
23       UNGLEICHE_VORZEICHEN: constant boolean :=
24          (LINKS < 0) /= (RECHTS < 0);
25    begin
26       if UNGLEICHE_VORZEICHEN and then REST /= 0 then
27          return QUOTIENT - 1;
28       else
29          return QUOTIENT;
30       end if;
31    end "&";
32    ------------------------------------------------------------------------
33 begin -- UPROS_26
34    ... -- Werte nach G1 und G2 bringen
35    G3 := G1 & G2;
36    if G3 & 2 < 17 then ... end if;
37    G3 := "&"(G1, G2);
38    G3 := "&"(RECHTS => G2, LINKS => G1);
39    ...
```

In den Zeilen 16 bis 31 wird eine Funktion mit dem Operatorsymbol **"&"** als Name vereinbart. In Zeile 16 und in Zeile 31 muß das Operatorsymbol in **doppelte Anführungszeichen** eingefaßt werden wie ein Zeichenkettenliteral.

Beim Aufrufen der Operation **"&"** kann der Programmierer zwischen **zwei Notationen** wählen. In Zeile 35 und 36 wird die **Infixnotation** benützt, d.h. das Operatorsymbol steht **zwischen** den beiden Parametern ("**in**fix" wie "**in** der Mitte zwischen den beiden Parametern"). Dabei darf das Operatorsymbol **nicht** in Anführungszeichen eingefaßt werden.

In Zeile 37 und 38 wird die Funktion **"&"** in **Präfixnotation** aufgerufen. Das Operatorsymbol steht **vor** den Parametern und **muß** in doppelte Anführungszeichen eingefaßt werden wie ein Zeichenkettenliteral. In Zeile 37 erfolgt die Zuordnung der Parameter über die **Position** der **aktuellen** Parameter, in Zeile 38 dagegen über die **Namen** der **formalen** Parameter. **o**

Aufgabe 17.2.6.: Wenden Sie die Funktion **"&"** auf verschiedene Parameter an, berechnen Sie (mit Papier und Bleistift) das jeweilige Ergebnis der Funktion und füllen Sie die folgende Tabelle aus:

G1	G2	G1 / G2	G1 rem G2	G1 & G2
-6	3	-2	0	
-5	3	-1	-2	
-4	3	-1	-1	
-3	3	-1	0	
-2	3	0	-2	

Schauen Sie sich dann das Beispielprogramm **UPROS_26** an und überprüfen Sie damit Ihre Ergebnisse. **o**

Über **Operatorsymbole** wie +, -, / etc. lernen viele Menschen schon in früher Jungend viele Einzelheiten. Z.B. "weiß jeder":

- Die Operatoren **"*"** und **"/"** haben immer **zwei** Parameter
- Die Operatoren **"+"** und **"-"** können dagegen vor **einem** Parameter oder aber **zwischen zwei** Parametern stehen.
- Der Operator **not** hat immer nur **einen** Parameter.
- **Punktrechnung geht vor Strichrechnung**, z.B. ist der Ausdruck **A + B * C** gleichbedeutend mit **A + (B * C)** und nicht gleichbedeutend mit **(A + B) * C**.

Wenn der Programmierer eigene Funktionen mit Operatorsymbolen als Namen vereinbart, sollte er dieses **verbreitete Vorwissen** über Operatorsymbole beachten und es möglichst **nicht** außer Kraft setzen. Die Ada-Regeln für Operationen helfen ihm dabei.

Regel 1 für Operationen: Operationen mit den Namen, *****, /, ******, **&**, =, /=, <, <=, >, >=, **MOD**, **REM**, **AND**, **OR** und **XOR** müssen genau **zwei** Parameter haben. Operationen mit den Namen **+** und **-** dürfen wahlweise **einen** oder **zwei** Parameter haben. Operationen mit den Namen **ABS** und **NOT** müssen genau **einen** Parameter haben.

Regel 2 für Operationen: Die **Bindungsstärke** der Operatorsymbole bleibt immer so, wie im (ARM 4.5) erläutert, auch wenn der Programmierer eigene Funktionen mit diesen Operatorsymbolen als Namen vereinbart. Z.B. binden die Operatorsymbole **"*"** und **"/"** immer stärker, als **"+"** und **"-"** ("Punktrechnung geht vor Strichrechnung").

Regel 3 für Operationen: Wenn der Programmierer eine Operation **vereinbart**, muß er ihren Namen (d.h. das betreffende Operatorsymbol) in doppelte Anführungszeichen einfassen wie ein Zeichenkettenliteral, z.B. so: **"AND"**, **"+"**, **"**"**, **"NOT"** etc..

Regel 4 für Operationen: Operationen mit **zwei** Parametern darf der Programmierer wahlweise in **Präfixnotation** aufrufen (mit dem Funktionsnamen **vor** den Parameter) oder in **Infixnotation** (mit dem Funktionsnamen **zwischen** den beiden Parametern). Wenn er die **Präfixnotation** benützt, muß er den Funktionsnamen in doppelte Anführungszeichen einfassen wie ein Zeichenkettenliteral, z.B. **"AND"(AP1, AP2)** oder **"**"(AP3, AP4)** etc.. Wenn er die **Infixnotation** benützt, darf er den Funktionsnamen **nicht** in doppelte Anführungszeichen einfassen, z.B. so: **AP1 AND AP2** oder **AP3 ** AP4** etc..

Regel 5 für Operationen: Operationen mit **einem** Parameter (**+**, **-**, **ABS** und **NOT**) darf man nur in **Präfixnotation** aufrufen und dabei den Funktionsnamen wahlweise in doppelte Anführungszeichen einfassen oder **ohne** Anführungszeichen hinschreiben, z.B. so: **"+"(AP1)**, **+AP1**, "**NOT"(AP1)**, **NOT AP1** etc..

Allgemein dürfen in Ada die Namen von Unterprogrammen **überladen** sein. Das heißt, **ein** Name darf **mehrere** verschiedene Unterprogramme bezeichnen. Welches Unterprogramm jeweils gemeint ist, versucht der Ausführer anhand der **Typen der Parameter** und bei **Funktionen** auch anhand des **Ergebnistyps** herauszufinden. Wenn aufgrund dieser **Auflösung der Überladung** (overload resolution) genau **ein** Unterprogramm in Frage kommt, dann nimmt der Ausführer dieses Unterprogramm als Bedeutung des Namens. Andernfalls meldet er einen Fehler (schon bei der Übergabe des Programms, nicht erst bei der Ausführung). In diesem Fall muß der Programmierer den Unterprogrammaufruf "genauer formulieren", so daß der Ausführer **eindeutig** erkennen kann, was gemeint ist. **Operatorsymbole** werden besonders häufig und "intensiv" überladen.

In Ada gibt es vier vordefinierte Funktionen namens **"&"**, mit denen man Werte der Untertypen **character** und **string** jeweils zu einem String konkatenieren kann. Im folgenden Beispiel werden noch zwei weitere Funktion namens **"&"** vereinbart, die das Ausgeben von Ganzzahlwerten zusammen mit "verzierenden Texten" erleichtern.

Beispiel 17.2.3.: Das Operatorsymbol **"&"** wird noch weiter überladen:

```
01 with ada.text_io;
02 procedure UPROS_27 is
03 ------------------------------------------------------------------------
04 -- Zwei Konkatenationsoperationen namens "&" werden vereinbart und aufge-
05 -- rufen. Sie erleichtern es, GANZ-Zahlen und Texte gemischt auszugeben.
06 ------------------------------------------------------------------------
07    type  GANZ is range -5_000..+10_000;
08    DM  : GANZ := 17;
09    PFG : GANZ := 35;
10    ---------------------------------------------------------------------
11    function "&"(L: string; R: GANZ) return string is
12    -- Wandelt R in einen String RS um und liefert L & RS:
13       RS : constant string := GANZ'image(R);
14    begin
15       return L & RS;
16    end "&";
17    ---------------------------------------------------------------------
18    function "&"(L: GANZ; R: string) return string is
19    -- Wandelt L in einen String LS um und liefert R & LS:
20       LS : constant string := GANZ'image(L);
```

```
21    begin
22       return LS & R;
23    end "&";
24    ------------------------------------------------------------------------
25 begin
26    ada.text_io.put_line(item => "DM" & DM);
27    ada.text_io.put_line(item => PFG   & " Pfennige");
28    ada.text_io.put_line(item => DM & " Mark und" & PFG & " Pfennige");
29 end UPROS_27;
```

In den Zeilen 11 bis 16 wird eine Funktion namens **"&"** vereinbart, die einen **string**- und einen **GANZ**-Parameter hat und ein **string**-Ergebnis liefert. Die in Zeile 18 bis 23 vereinbarte Operation **"&"** hat dagegen einen **GANZ**- und einen **string**-Parameter und liefert ebenfalls ein **string**-Ergebnis. In den Zeilen 15 und 22 wird die vordefinierte Operation **"&"** aufgerufen, die zwei **string**-Parameter hat und ein **string**-Ergebnis liefert.

Die Attributfunktion **GANZ'image**, die in Zeile 13 und 20 aufgerufen wird, wandelt einen Wert des Typs **!GANZ** in eine entsprechende Zeichenkette vom Untertyp **string** um. Beispielsweise ist **GANZ'image(-17)** gleich **"-17"**. **o**

Aufgabe 17.2.7.: Welche Operation namens **"&"** wird in Zeile 26 aufgerufen? Welche **"&"**-Operation wird in Zeile 27 aufgerufen?. In Zeile 28 wird **dreimal** eine Operation namens **"&"** aufgerufen. Können Sie herausfinden, welche Operation jeweils gemeint ist? Gehen Sie bei Ihrer Analyse von links nach rechts vor. Wenn Sie herausgefunden haben, was der **erste** &-Operator bedeutet, dann hilft ihnen das bei der Analyse der **weiteren** &-Operatoren. **o**

Zusammenfassung 17.2.:

- Der Programmierer kann Funktionen **vereinbaren** ("vom Ausführer erzeugen lassen") und dann **aufrufen** ("ausführen lassen").
- Wenn eine Funktion aufgerufen wird, dann **liefert sie einen Wert**.
- Eine Funktion kann im Prinzip **beliebig viele** (null oder mehr) **Parameter** haben.
- Alle Parameter einer Funktion haben den Modus **in**. Deshalb kann man für jeden einen **Vorbesetzungsausdruck** angeben und die formalen Parameter haben den Charakter von **Konstanten**.
- Eine **Operation** ist eine Funktion mit einem **Operatorsymbol** als Namen.
- **Operationen** können nur **einen** oder **zwei** Parameter haben.
- Operationen mit **zwei Parametern** kann man wahlweise in **Infixnotation** aufrufen (mit dem Operatorsymbol **zwischen** den Parametern) oder in **Präfixnotation** (mit dem Operatorsymbol **vor** den Parametern).

17.3. Spezifikation und Rumpf eines Unterprogramms

Ein Unterprogramm muß man zuerst **vereinbaren** ("vom Ausführer erzeugen lassen") und kann es dann beliebig oft **aufrufen** ("ausführen lassen"). Insbesondere bei der Entwicklung **großer** Programme werden Unterprogramme häufig von **einer Person** vereinbart und von **anderen Personen** aufgerufen. Diese Arbeitsteilung wird im vorliegenden Skript folgendermaßen modelliert: Die Tätigkeit des **Vereinbarens** von Unterprogrammen ordnen wir der Rolle des **Programmierers** zu, die Tätigkeit des **Aufrufens** dagegen dem **Kollegen-2** (dem "Wiederverwender von Programmteilen"). In kleinen Projekten werden die **beiden** Rollen (des Programmierers und seines Kollegen-

2) häufig von **einer** Person übernommen, in großen Projekten dagegen auf **mehrere** Personen verteilt.

Um ein Unterprogramm aufrufen zu können, muß der Kollege-2 bestimmte Informationen haben bzw. bekommen. Im Einzelnen muß er folgendes wissen:

1. Wie **heißt** das Unterprogramm?
2. **Wieviele Parameter** hat es?

Falls das Unterprogramm mehr als null Parameter hat, sollte der Kollege-2 von jedem einzelnen Parameter folgendes wissen:

2.1. Zu welchem **Untertyp** gehört der Parameter?
2.2. Welchen **Modus** hat der Parameter (**in**, **out** oder **in out**)?
2.3. Nur bei **in**-Parametern: Gibt es eine **Vorbesetzung** für den Parameter? Welche?

Falls das Unterprogramm eine **Funktion** ist, muß der Aufrufer außerdem wissen:

3. Zu welchem **Untertyp** gehört das **Ergebnis** der Funktion?

Schließlich sollte der Kollege-2 unbedingt und möglichst genau wissen:

4. Was **macht** das Unterprogramm? Bei einer **Prozedur**: **Welche** Behälter verändert die Prozedur und **wie** verändert sie diese Behälter? Bei einer **Funktion**: Was für ein Ergebnis berechnet die Funktion aus ihren Parametern?

Manchmal ist es aus organisatorischen Gründen günstig, wenn der Kollege-2 die Antworten auf all diese Fragen schon bekommt, **bevor** der Programmierer das Unterprogramm **fertig geschrieben** hat. In Ada kann der Programmierer die Vereinbarung eines Unterprogramms deshalb in **zwei Teile** aufteilen. Der erste Teil, die **Spezifikation** des Unterprogramms, enthält die Antworten auf die oben aufgeführten Fragen seines Kollegen-2, die Antwort auf die Frage 4. ("Was macht das Unterprogramm?") allerdings nur in Form eines **Kommentars**. Der zweite Teil, der **Rumpf**, ist eine **vollständige** Beschreibung des Unterprogramms. D.h. der Rumpf beginnt mit einer **Wiederholung** der Spezifikation und enthält zusätzlich alle Befehle (Vereinbarungen und Anweisungen), aus denen das Unterprogramm bestehen soll.

In vielen Fällen ist das Angeben einer separaten **Spezifikation** eines Unterprogramms eine **freiwillige** organisatorische Maßnahme, in einigen Fällen ist sie unbedingt **erforderlich**. In den vorigen beiden Abschnitten wurde jedes Unterprogramm nur durch seinen **Rumpf** beschrieben, separate Spezifikationen wurden **nicht** angegeben. Praktisch sieht eine Unterprogramm**spezifikation** so aus, "wie die erste Zeile des **Rumpfes**". Die folgenden Beispiele sind **Spezifikationen** von Unterprogrammen, deren Rümpfe in den vorigen beiden Abschnitten angegeben wurden:

Beispiel 17.3.1.: Spezifikationen von Unterprogrammen:

```
01 declare
02    procedure DOPPEL_LINIE_1;
03    -- Gibt zwei Zeilen, bestehend aus je 80 Minuszeichen, zur
04    -- aktuellen Ausgabe aus.
05
```

```
06    procedure DOPPEL_LINIE_2(LAENGE: natural; ZEICHEN: character);
07    -- Gibt zwei Zeilen, bestehend aus je LAENGE vielen ZEICHEN, zur
08    -- aktuellen Ausgabe aus.
09
10    procedure GET_ZIFFER(ITEM: out character; ALLES_OK: out boolean);
11    -- Liest von der aktuellen Eingabe ein Zeichen nach ITEM und setzt
12    -- ALLES_OK auf true (bzw. false), wenn das Zeichen eine Ziffer
13    -- (bzw. keine Ziffer) ist.
14
15    type GANZ is range -5_000..+10_000;
16
17    function ABSOLUT_KLEINER(FP1, FP2: GANZ) return boolean;
18    -- Liefert true, wenn der absolute Wert von FP1 kleiner ist, als
19    -- der absolute Wert von FP2, und sonst false.
20
21    function "&"(L, R: GANZ) return GANZ;
22    -- Liefert den Quotienten "L durch R", rundet dabei aber immer ab.
23    -- Z.B. ist -5 & 3 gleich -2 (und nicht gleich -1). Diese Ganzzahl-
24    -- division "&" passt zur Restfunktion mod so, wie die Ganzzahl-
25    -- division "/" zur Restfunktion rem passt.
```

Eine solche Sammlung von **Spezifikationen** ist im allgemeinen **übersichtlicher** und **leichter zu lesen** als eine entsprechende Sammlung von **Rümpfen**, insbesondere dann, wenn die Rümpfe lang und kompliziert sind.

Man beachte, daß in den letzten beiden Spezifikationen (Zeile 17 bis 19 bzw. 21 bis 25) der Untertyp **GANZ** verwendet wird. Deshalb muß die entsprechende Typvereinbarung **vor** den Spezifikationen stehen (siehe Zeile 15). **o**

Wenn Unterprogramme in einer Programmeinheit P **enthalten** sind (z.B. in einer Prozedur **P**), dann kann man die **Spezifikationen** der Unterprogramme **irgendwo** im Vereinbarungsteil von **P** plazieren. Die dazugehörigen **Rümpfe** müssen dann irgendwo **danach** im selben Vereinbarungsteil aufgeführt werden. D.h. wenn man eine separate **Spezifikation** und einen **Rumpf** angibt, muß die Spezifikation **vor** dem Rumpf stehen.

Will man für ein **Bibliotheksunterprogramm** eine Spezifikation angeben, dann muß man sie dem Ausführer **vor** dem Rumpf übergeben. Praktisch bedeutet das bei vielen heute üblichen Ada-Ausführern: Man schreibt die Spezifikation in eine extra **Datei** (die dann typischerweise nur eine Zeile oder wenige Zeilen enthält) und übergibt diese Datei dem Ausführer (indem man sie "compiliert" oder "in ein bestimmtes Verzeichnis einträgt", je nach den konkreten Regeln des verwendeten Ada-Ausführers).

Spezifikationen für Bibliotheksunterprogramme haben vor allem den folgenden wichtigen Vorteil: Sobald man dem Ausführer die Spezifikation eines Bibliotheksunterprogramms **BU** übergeben hat, kann man ihm **weitere** Programmeinheiten übergeben, die das Unterprogramm **BU** mit einer **Kontextklausel** einbinden, z.B. so:

Beispiel 17.3.2.: Übergabe einer **Spezifikation** und einer davon **abhängigen Programmeinheit**:

1. Man übergibt dem Ausführer die **Spezifikation** eines Bibliotheksunterprogramms BU, z.B.:

```
01 function BU(P1: character; P2: boolean) return integer;
```

2. Danach (und noch bevor man dem Ausführer den **Rumpf** von **BU** übergeben hat), kann man ihm **weitere Programmeinheiten** übergeben, die **BU** einbinden, z.B.:

```
with BU, ...;
procedure WEITERE_EINHEIT is
...
end WEITERE_EINHEIT;
```

Würde man dem Ausführer eine solche **WEITERE_EINHEIT** übergeben, **bevor** er die Spezifikation oder den Rumpf des Unterprogramms **BU** gesehen hat, würde er einen Fehler melden ("Program unit BU is not visible here" oder so ähnlich) und die **WEITERE_EINHEIT** nicht akzeptieren. **o**

Insbesondere in **großen Projekten**, in denen **viele** ProgrammiererInnen **gleichzeitig** viele Unterprogramme entwickeln, die sich gegenseitig aufrufen, sind Spezifikationen von Bibliotheksunterprogrammen sehr hilfreich, weil sie **frühzeitige Prüfungen** der einzelnen Unterprogramme durch den Ausführer ermöglichen.

Die **Spezifikation** eines Unterprogramms endet mit einem Semikolon "**;**". An der entsprechenden Stelle im **Rumpf** des Unterprogramms steht das Wort **is**. Das ist aber fast der einzige Unterschied, der zwischen einer Spezifikation und dem Anfang des zugehörigen Rumpfes erlaubt ist. Hier ein paar nicht erlaubte und erlaubte Unterschiede:

Beispiel 17.3.3.: Eine **Unterprogrammspezifikation**, drei **nicht** erlaubte und ein **erlaubter** Anfang des zugehörigen Rumpfes:

```
01 function F(P1, P2: character) return character;
02
03 function F(P1, P2: in character) return character is ... end F;
04
05 function F(P1: character; P2: character) return character is ... end F;
06
07 function F(P2, P1: character) return character is ... end F;
08
09 function  F ( P1 ,  P2:  character  ) return   character is ... end F;
```

In Zeile 01 steht die **Spezifikation** einer Funktion namens **F**. In den nachfolgenden Zeilen (03, 05 07 und 09) ist jeweils ein **Rumpf** der Funktion **F** angedeutet. Die ersten drei Rümpfe (in Zeile 03, 05 und 07) sind **nicht** zulässig, weil ihr Anfang nicht genau genug mit der Spezifikation in Zeile 01 übereinstimmt. Die **fetten** Hervorhebungen sollen dabei helfen, die unerlaubten Abweichungen zu finden. Der in Zeile 09 angedeutete Rumpf ist **zulässig**. Sein Anfang unterscheidet sich nur durch ein paar Blankzeichen von der Spezifikation in Zeile 01. **o**

Praktisch sollte man die **Spezifikation** eines Unterprogramms möglichst mit einem Editorbefehl **kopieren**, wenn man damit beginnt, den **Rumpf** zu schreiben. So vermeidet man auf besonders einfache Weise unerlaubte Abweichungen.

Separate **Spezifikationen** sind vor allem bei der Entwicklung **großer** Programme nützlich. Hier ein **kleines** Beispiel, bei dem Spezifikationen unbedingt erforderlich sind:

Beispiel 17.3.4.: Unterprogramme, die sich **gegenseitig aufrufen** und deshalb unbedingt durch separate Spezifikationen beschrieben werden **müssen**:

```
01 procedure UPROS_51(WIE_OFT_51: natural);

02 procedure UPROS_52(WIE_OFT_52: natural);

03 with UPROS_52, ada.text_io;
04 procedure UPROS_51(WIE_OFT_51: natural) is
05 begin
06    ada.text_io.put_line(item => "UPROS_51: " & natural'image(WIE_OFT_51));
07    UPROS_52(WIE_OFT_52 => WIE_OFT_51 - 1);
08 end UPROS_51;

09 with UPROS_51, ada.text_io;
10 procedure UPROS_52(WIE_OFT_52: natural) is
11 begin
12    ada.text_io.put_line(item => "UPROS_52: " & natural'image(WIE_OFT_52));
13    UPROS_51(WIE_OFT_51 => WIE_OFT_52 - 1);
14 end UPROS_52;

15 with UPROS_51, ada.text_io, ada.integer_text_io;
16 procedure UPROS_50 is
17    ANZAHL: natural;
18 begin
19    ada.text_io.put(item => "Bitte geben Sie eine Anzahl ein: ");
20    ada.integer_text_io.get(item => ANZAHL);
21    UPROS_51(WIE_OFT_51 => ANZAHL);
22 end UPROS_50;
```

Dieses Beispiel besteht aus **fünf** Teilen. Die ersten beiden Teile bestehen nur aus je **einer** Zeile, die übrigen drei Teile aus je **sechs** bzw. **acht** Zeilen. Jeder Teil kann dem Ausführer **unabhängig** von den anderen Teilen übergeben werden. Zuerst sollte man dem Ausführer die **ersten beiden** Teile übergeben. Danach kann man ihm die restlichen drei Teile in **beliebiger Reihenfolge** überreichen.

Hier die Inhalte der einzelnen Teile:

Zeile 01:	Die **Spezifikation** einer Prozedur namens UPROS_71.
Zeile 02:	Die **Spezifikation** einer Prozedur namens UPROS_72.
Zeile 03 bis 08:	Der **Rumpf** der Prozedur UPROS_71.
Zeile 09 bis 14:	Der **Rumpf** der Prozedur UPROS_72.
Zeile 15 bis 22:	Der **Rumpf** einer Prozedur namens UPROS_70.

UPROS_70 ist das **Hauptunterprogramm** dieses Beispiels. Für **UPROS_70** ist **keine** separate Spezifikation erforderlich, deshalb wurde auch keine angegeben. Weil die beiden Unteprogramme **UPROS_71** und **UPROS_72** sich **gegenseitig** einbinden und aufrufen, sind für sie unbedingt **separate Spezifikationen erforderlich.** Ohne die Spezifikationen in Zeile 01 und 02 würde der Ausführer den Rumpf von **UPROS_71** nicht akzeptieren, weil er **UPROS_72** nicht kennt und den Rumpf von **UPROS_72** würde er genausowenig akzeptieren, weil er **UPROS_71** nicht kennt. Nach der Übergabe der Spezifikationen in Zeile 01 und 02 "**kennt**" der Ausführer die beiden Unterprogramme und akzeptiert ihre Rümpfe, sogar in beliebiger Reihenfolge.

Wenn der **Benutzer** das Programm **UPROS_70** aufruft wird die Aufforderung **"Bitte geben Sie eine Anzahl ein:"** ausgegeben. Gibt er daraufhin z.B. die Zahl **5** ein, werden weitere **5** Zeilen ausgegeben (**drei** vom Unterprogramm **UPROS_71** und **zwei** von **UPROS_72**). **o**

Zusammenfassung 17.3.:

- Ein Unterprogramm besteht mindestens aus einem **Rumpf**.
- In allen Fällen **darf** der Programmierer zusätzlich und vorab eine separate **Spezifikation** angeben. In einigen Fällen **muß** er das sogar (siehe Beispiel 17.3.4.).
- Die **Spezifikation** eines Unterprogramms enthält alle Informationen, die man braucht, um das Unterprogramm **aufzurufen**.

- Der **Rumpf** eines Unterprogramms enthält zusätzlich die **Befehle** (Vereinbarungen und Anweisungen), aus denen das Unterprogramm bestehen soll.
- Zu einer **guten Spezifikation** gehört auch ein **Kommentar**, aus dem hervorgeht, "was das Unterprogramm macht" (Zwei Gegenbeispiele findet man im Beispiel 17.3.4.).

17.4. Unterprogrammparameter eines uneingeschränkten Untertyps

Die formalen **Parameter** eines Unterprogramms haben große Ähnlichkeit mit **Konstanten** bzw. **Variablen**. Technisch gesehen **sind in**-Parameter **Konstanten** und **out**- und **in-out**-Parameter **Variablen**. Wenn ein formaler Parameter zu einem **einfachen** Untertyp gehört (z.B. zu einem Ganzzahluntertyp wie **integer** oder **GANZ** oder zu einem Aufzählungsuntertyp wie **character** oder **FARBE** etc.), dann kann man ihn ganz **genauso** behandeln wie eine "normale" Konstante bzw. Variable des betreffenden Untertyps. Entsprechendes gilt auch dann noch, wenn ein formaler Parameter zu einem **eingeschränkten** Reihungs- oder Verbunduntertyp gehört.

Ein formaler Parameter, der zu einem **uneingeschränkten Reihungsuntertyp** wie z.B. **string** gehört, erfordert dagegen "im Kopf des Programmierers" eine ganz **andere Behandlung**. Ein solcher Parameter ist "**abstrakter**" als eine normale Konstante bzw. Variable, weil er für Reihungen mit **verschiedenen Indexbereichen** steht. Wenn der Programmierer innerhalb eines Unterprogramms einen bestimmten Befehl auf einen formalen Reihungsparameter anwendet, dann muß er sich **all diese verschiedenen Reihungen** vorstellen, für die der Parameter steht, und muß sicherstellen, daß der Befehl bei **all** diesen Reihungen "richtig funktioniert".

Formale Parameter, die zu einem uneingeschränkten Reihungsuntertyp gehören, sind sehr **mächtige** Instrumente, deren korrekte Handhabung aber **keineswegs trivial** ist. In diesem Abschnitt sollen einige der Möglichkeiten und Gefahren deutlich gemacht werden, die mit solchen Parametern verbunden sind.

In allen Beispielen dieses Abschnitts wird der uneingeschränkte Reihungsuntertyp **string** benützt. Man kann die Beispiele aber auch auf jeden anderen **uneingeschränkten Reihungsuntertyp** übertragen, ohne daß sich dadurch an den aufgezeigten Möglichkeiten und Gefahren etwas wesentliches ändern würde.

Für uneingechränkte **Verbund**untertypen gilt ganz Entsprechendes wie für uneingeschränkte **Reihungs**untertypen. Da Verbundtypen aber im allgemeinen deutlich weniger wichtig sind als Reihungstypen (weil man mit Selektornamen nicht so rechnen kann wie mit Indizes), werden hier keine speziellen Beispiele mit formalen Parametern eines uneingeschränkten Verbunduntertyps vorgeführt. Außerdem gilt wohl: Wer gut mit formalen Parametern eines uneingeschränkten **Reihungs**untertyps umgehen kann, wird auch mit Parametern eines uneingeschränkten **Verbund**untertyps kaum Probleme haben.

Vorab und zur Einstimmung wird eine spezielle Eigenschaft der Konkatenationsoperation **"&"** (mit der man **zwei** Strings zu **einem** String konkatenieren kann) erläutert.

Beispiel 17.4.1.: Welchen **Indexbereich** hat ein konkatenierter String?

```
01 declare
02    TEXT1 : string(101..106)     := "Hallo!";
03    TEXT2 : string               :=  TEXT1     & " Wie geht es?";
04    TEXT3 : string               := "Sie da! " &  TEXT1;
```

Der Indexbereich **101..106** der Variablen **TEXT1** wurde vom Programmierer **explizit** festgelegt. Für die Variablen **TEXT2** und **TEXT3** wurde ein Indexbereich aber nur **implizit** durch Angabe eines **Anfangswertes** angegeben. Welcher konkrete Indexbereich wird diesen Variablen also vom Ausführer zugeordnet? Und welchen Indexbereich hat der **string**-Wert eines **Ausdrucks** wie z.B. **TEXT1 & " Wie geht es?"** ?

Die Regeln der Sprache Ada legen fest: Der Wert des Ausdrucks **TEXT1 & " Wie geht es?"** übernimmt den ersten Index von der Variablen **TEXT1**. Somit hat dieser **string**-Wert den Indexbereich **101..119**. Diesen Indexbereich übernimmt auch die Variable **TEXT2**.

Der Wert eines **Stringliterals** wie z.B. **"Sie da! "** hat immer einen Indexbereich, der mit **1** beginnt. Somit hat der Wert des Ausdrucks **"Sie da! " & TEXT1** den Indexbereich **1..14** und dieser Indexbereich wird auch von der Variablen **TEXT3** übernommen. **o**

Das folgende Beispiel zeigt eine Funktion, die ihren **string**-Parameter "verdoppelt" als Ergebnis liefern soll:

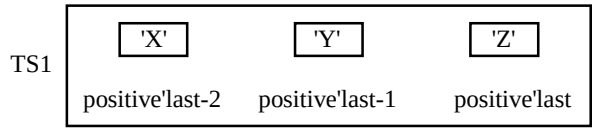
Beispiel 17.4.2.: Eine Funktion zum Verdoppeln von Strings:

```
01 declare
02    function VERDOPPLE_1(S: string) return string is
03    begin
04       return S & S;
05    end VERDOPPLE_1;
```

Mit dieser Funktion kann man sehr viele Strings ordnungsgemäß verdoppeln, aber leider nicht alle. Wenn man die Funktion auf bestimmte Strings mit "**extremen Indexbereichen**" anwendet, liefert sie kein Ergebnis, sondern löst die Ausnahme **constraint_error** aus. Hier die Vereinbarung eines solchen Strings mit einem extremen Indexbereich:

```
06    TS1 : string(positive'last-2..positive'last) := "XYZ";
```

Die folgende bildliche Darstellung der Variablen **TS1** soll deutlich machen, daß es sich eigentlich um einen "ganz kleinen und harmlosen" String handelt, dessen Verdopplung keine Ausnahme auslösen sollte:



Welchen konkreten Wert das Attribut **positive'last** hat, ist von Ada-Ausführer zu Ada-Ausführer verschieden und hier nicht wichtig. Wichtig ist dagegen, daß **kein** String einen Index haben kann, der größer ist als **positive'last**.

Der Funktionsaufruf **VERDOPPLE_1(S => TS1)** löst einen **constraint_error** aus, weil der formale Parameter **S** den Indexbereich des aktuellen Parameters **TS1** übernimmt, so daß also **S'first** gleich **positive'last-2** ist. Damit müßte der Wert des Ausdrucks **S & S** (siehe oben Zeile 04) den Indexbereich **positive'last-2..positive'last+3** haben, was unmöglich ist.

Die folgende Version der Funktion VERDOPPLE funktioniert in allen Fällen korrekt, in denen das überhaupt möglich ist, d.h. in denen der verdoppelte String nicht länger als der größtmögliche String ist:

```
01 declare
02    function VERDOPPLE_2(S: string) return string is
03       HALB : constant string(1..S'length) := S;
04    begin
05       return HALB & HALB;
06    end VERDOPPLE_2;
```

Auch wenn man die Funktion **VERDOPPLE_2** auf den String **TS1** mit dem extremen Indexbereich anwendet, hat die Konstante **HALB** doch den "ganz harmlosen" Indexbereich 1..3 und es tritt kein **constraint_error** auf. **o**

Dieses Beispiel sollte deutlich machen, daß ein formaler Parameter wie **S** für **alle möglichen Strings** steht, auch für solche mit extremen Indexbereichen, und daß man beim Programmieren eines Unterprogramms mit solch einem Parameter **alle** diese Strings berücksichtigen sollte.

Das folgende Beispiel soll deutlich machen, betonen und unterstreichen, daß nicht alle Strings einen Indexbereich haben, der mit **1** beginnt:

Beispiel 17.4.3.: Eine Funktion, die Buchstaben zählt:

```
01 declare
02    function ANZAHL_BUCHSTABEN_1(S : string) return natural is
03       ANZAHL : natural := 0;
04    begin
05       for I in positive range 1..S'length loop
06          if S(I) in 'A'..'Z' or S(I) in 'a'..'z' then
07             ANZAHL := ANZAHL + 1;
08          end if;
09       end loop;
10       return ANZAHL;
11    end ANZAHL_BUCHSTABEN_1;
12    TS2 : string       := "Hallo, wie geht es?";
13    TS3 : string(7..9) := "ABC";
```

Wenn man die Funktion **ANZAHL_BUCHSTABEN_1** z.B. auf den Teststring **TS2** (vereinbart in Zeile 12) anwendet, geht alles gut. Der Teststring **TS3** macht aber deutlich, daß die Funktion fehlerhaft ist und die Ausnahme **constraint_error** auslöst, statt das korrekte Ergebnis **3** zu liefern.

Aufgabe 17.4.1.: Führen Sie den Funktionsaufruf **ANZAHL_BUCHSTABEN_1(S => TS3)** mit Papier und Bleistift oder im Kopf aus. Was ist falsch an der Funktion? Programmieren Sie eine korrekte Funktion **ANZAHL_BUCHSTABEN_2**. **o**

Das nächste Beispiel soll auf die einfache Tatsache aufmerksam machen, daß zwei verschiedene Strings nicht notwendig den gleichen Indexbereich haben müssen:

Beispiel 17.4.4.: An wieviel Stellen stimmen zwei Strings überein?

```
01 declare
02    function ANZAHL_GLEICHER_STELLEN_1(S1, S2: string) return natural is
03    -- Liefert die Anzahl der Stellen, an denen S1 und S2 gleich sind.
04       ANZAHL : natural := 0;
05    begin
06       for I in positive range S1'range loop
07          if S1(I) = S2(I) then
08             ANZAHL := ANZAHL + 1;
09          end if;
10       end loop;
11       return ANZAHL;
12    end ANZAHL_GLEICHE_STELLEN_1;
13    TS11 : string(7..9) := "ABC";
14    TS12 : string(7..9) := "AbC";
15    TS21 : string(1..3) := "ABC";
16    TS22 : string(1..2) := "AB";
17    TS31 : string(1..3) := "ABC";
18    TS32 : string(7..9) := "ABC";
```

Der Funktionsaufruf **ANZAHL_GLEICHER_STELLEN_1(S1 => TS11, S2 => TS12)** kann ordnungsgemäß ausgeführt werden und liefert das korrekte Ergebnis **2**. Die Aufrufe **ANZAHL_GLEICHER_STELLEN_1(S1 => TS21, S2 => TS22)** und **ANZAHL_GLEICHER_STELLEN_1(S1 => TS31, S2 => TS32)** lösen dagegen die Ausnahme **constraint_error** aus (tun das aber aus etwas unterschiedlichen Gründen).

Aufgabe 17.4.2.: Was ist falsch an der Funktion **ANZAHL_GLEICHER_STELLEN_1**? Warum löst sie manchmal die Ausnahme **constraint_error** aus? Wie könnte eine verbesserte Version **ANZAHL_GLEICHER_STELLEN_2** aussehen? Probieren Sie selbst, ehe Sie sich die Lösung im nächsten Beispiel ansehen. **o**

Beispiel 17.4.4.: Fortsetzung: Eine bessere Funktion **ANZAHL_GLEICHER_STELLEN_2**:

```
01 declare
02    function ANZAHL_GLEICHER_STELLEN_2(S1, S2: string) return natural is
03       ANZAHL  :          natural := 0;
04       MINIMUM : constant natural := natural'min(S1'length, S2'length);
05    begin
06       for N in natural range 0..MINIMUM-1 loop
07          if S1(S1'first+N) = S2(S2'first+N) then
08             ANZAHL := ANZAHL + 1;
09          end if;
10       end loop;
11       return ANZAHL;
12    end ANZAHL_GLEICHER_STELLEN_2;
```

Die Attributfunktion **natural'min** (siehe Zeile 04) liefert das **Minimum** ihrer beiden Parameter. Dies ist wichtig, falls die beiden formalen Parameter S1 und S2 für **verschieden lange** Strings stehen. In Zeile 06 ist es günstiger, den Schleifenparameter N den Bereich **0..MINIMUM-1** durchlaufen zu lassen, statt den Bereich **1..MINIMUM**, so wird die Zeile 07 etwas einfacher. Entscheidend wichtig ist der Ausdruck **S1(S1'first+N)** (bzw. **S2(S2'first+N)**) in Zeile 07. Er bezeichnet garantiert eine Komponente der Reihung **S1** (bzw. **S2**) und löst keinen **constraint_error** aus. **o**

Im nächsten Beispiel spielt der **leere String** eine nützliche Rolle. Die Funktion ALLE_ZIFFERN hat einen string-Parameter namens VON und soll alle darin enthaltenen Ziffern (zu einem String zusammengefaßt) als Ergebnis liefern. Hier ein paar Beispiele:

```
ALLE_ZIFFERN(VON => "17 DM 25 PFG")        ist gleich "1725"
ALLE_ZIFFERN(VON => "12345")               ist gleich "12345"
ALLE_ZIFFERN(VON => "ABC")                 ist gleich ""    (der leere String)
```

Im letzten Beispiel wird die Funktion auf einen String angewendet, der keine Ziffern enthält. In diesem Fall muß sie den **leeren String** als Ergebnis liefern (jeder andere String wäre in diesem Fall als Ergebnis der Funktion unnatürlich).

Aufgabe 17.4.3.: Programmieren Sie die Funktion **ALLE_ZIFFERN**, und führen Sie sie mehrmals mit verschiedenen aktuellen Parametern aus (mit Papier und Bleistift). **o**

Die Ada-Regeln, die mit leeren Reihungen zu tun haben, sind so gewählt und aufeinander abgestimmt worden, daß man leere Reihungen in vielen Fällen **wie alle anderen Reihungen auch** bearbeiten kann und für sie keine **Sonderbehandlung** zu programmieren braucht. Z.B. kann man die Funktionen **VERDOPPLE_2**, **ANZAHL_BUCHSTABEN_2** und **ANZAHL_GLEICHER_STELLEN_2** (siehe die vorigen Beispiele in diesem Abschnitt) ohne weiteres auch auf leere Strings anwenden, und die Funktionen liefern auch dann ein vernünftiges Ergebnis.

Leere Reihungen haben andererseits ein paar merkwürdige Eigenschaften, die man insbesondere beim Programmieren von Unterprogrammen von vornherein berücksichtigen sollte, um Fehler zu vermeiden. Diese merkwürdigen Eigenschaften **leerer Reihungen** sollen am konkreten Beispiel von **leeren Strings** verdeutlicht werden.

Beispiel 17.4.5.: Verschiedene **leere Strings**:

```
01 declare
02    LS1 :             string(1..0);
03    LS2 :             string(+17..+11);
04    LS3 :             string(-11..-17);
05    LS4 :             string             := "";
06    LS5 : constant    string(1..0)       := "";
```

Zur Erinnerung: Ein **Bereich** ist **leer**, wenn seine Untergrenze **größer** ist als seine Obergrenze. Z.B. sind die Bereiche **1..0**, **+17..+11**, **-11..-17** und **'C'..'A'** leer.

Eine **Reihung** ist **leer**, wenn ihr **Indexbereich** leer ist. Somit sind die Reihungen **LS1** bis **LS3** (vereinbart in Zeile 02 bis 04) **leer**.

Alle **Indizes** eines Strings (d.h. einer Reihung vom Untertyp **string**) müssen zum Untertyp **positive** gehören, d.h. größer oder gleich **1** sein. Die Zahl **0** und alle **negative Zahlen** sind als Indizes eines Strings **nicht erlaubt**.

Scheinbar widerspricht schon die Vereinbarung der Variablen **LS1** dieser Regel, weil dort explizit der Indexbereich **1..0** festgelegt wird. Dieser Schein trügt aber. Denn da 1..0 ein **leerer** Bereich ist, gilt: **Alle** Werte des Bereichs **1..0** und damit alle Indizes der Reihung **LS1** gehören zum Untertyp **positive**. Oder andersherum: **Kein** Index der Reihung **LS1** gehört **nicht** zum Untertyp **positive** (oder können Sie ein Gegenbeispiel nennen?).

Der "logische Kniff" ist hier, daß **LS1** gar keine Indizes besitzt und deshalb sehr viele Aussagen auf **alle Indizes von LS1** zutreffen. Z.B. sind **alle Indizes von LS1** ganz offensichtlich aus grünem Käse (oder können Sie ein Gegenbeispiel nennen?).

Entsprechend gilt für den String **LS3**: Obwohl sein Indexbereich ausdrücklich durch die Angabe **-11..-17** festgelegt wurde (siehe oben Zeile 04), gehören **alle seine Indizes** (von denen er genau **null** Stück besitzt) zum Untertyp **positive**. Das mag anfänglich etwas merkwürdig klingen, beruht aber auf mathematisch-logischen Konventionen und Gesetzen, die deutlich älter sind als die Sprache Ada.

Obwohl der String **LS1** leer ist und somit genau **null** Indizes besitzt, sind die Attribute **LS1'first** und **LS1'last** definiert und haben die Werte **1** bzw. **0**, wie in Zeile 02 festgelegt. Entsprechend hat das Attribut **LS3'first** den Wert **-11** und **LS3'last** den Wert **-17**, wie in Zeile 04 festgelegt. Die Attribute **LS1'range** und **LS3'range** bezeichnen einen leeren Bereich von Werten des Untertyps **positive** und die Attribute **LS1'length** und **LS3'length** haben natürlich den Wert **0**. Für die Attribute der anderen Variablen (LS2 und LS4) und Konstanten (LS5) gilt entsprechendes.

Für die Vereinbarung der Variablen **LS4** in Zeile 05 gilt: Der **erste** Index des Stringliterals "" ist gleich **1** wie bei allen Stringliteralen. Also muß sein letzter Index gleich **0** sein und diesen (natürlich leeren) Indexbereich übernimmt die Variable **LS4**. Das Attribut **LS4'first** hat also den Wert **1** und **LS4'last** den Wert **0**.

Die Vereinbarung der Konstanten **LS5** in Zeile 06 soll deutlich machen: Obwohl der **Wert** dieser Konstanten (nämlich der leere String) eigentlich schon aus der Angabe des leeren Indexbereichs **1..0** folgt, muß man ihn doch noch mal ausdrücklich hinter "**:=**" angeben, wie bei jeder anderen Konstanten auch. **o**

Alle **leeren Strings** sind in folgendem Sinne **gleich**: Die Ausdrücke **LS1=LS2**, **LS2=LS3**, **LS3=LS4**, **LS4=LS5** und **LS5=""** haben alle den Wert **true**. Andererseits gibt es kleine Unterschiede zwischen den oben vereinbarten leeren Strings. Z.B. ist **LS1'first** gleich **1**, aber **LS3'first** ist gleich **-11** etc..

Das nächste Beispiel soll zwei beliebte Fehler deutlich machen, von denen der eine mit den merkwürdigen Eigenschaften leerer Stings zu tun hat.

Beispiel 17.4.6.: Die Prozedur **ERSTES_X_NACH_Y** hat einen **string**-Parameter namens **S** vom Modus **in out**, und soll darin das erste '**X**' durch ein '**Y**' ersetzen. Falls der String **S** kein '**X**' enthält, soll er unverändert bleiben. Hier eine erste Version dieser Prozedur:

```
01 declare
02    procedure ERSTES_X_NACH_Y_1(S: in out string) is
03       I: positive := S'first;
04    begin
05       while I <= S'last loop
06          if S(I)  = 'X' then
07             S(I) := 'Y';
08             return; -- Beende (die Schleife und) das Unterprogramm.
09          end if;
10          I := I + 1;
11       end loop;
12    end ERSTES_X_NACH_Y_1;
13    TS1 : string(positive'last-1..positive'last) := "AB";
14    TS2 : string(-11..-17); -- Ein leerer String.
```

Diese Prozedur kann viele Strings korrekt bearbeiten, aber bei den beiden Teststrings **TS1** und **TS2** (vereinbart in Zeile 12 und 13) versagt sie. Können Sie erkennen, was schief läuft?

Der Prozeduraufruf **ERSTES_X_NACH_Y_1(S => TS1)** bewirkt, daß in Zeile 10 die Ausnahme **constraint_error** ausgelöst wird. Denn der formale Parameter **S** übernimmt den Indexbereich vom aktuellen Parameter **TS1**, so daß **S'first** gleich **positive'last-1** gilt. Bei der zweiten und letzten Ausführung des Schleifenrumpfes wird dem Ausführer in Zeile 10 befohlen, der Variablen **I** den Wert **positive'last+1** zuzuweisen. Diesen Befehl muß er verweigern.

Der Prozeduraufruf **ERSTES_X_NACH_Y_1(S => TS2)** bewirkt, daß schon in Zeile 03 ein **constraint_error** ausgelöst wird. Denn **S'first** hat dann den Wert **-11**, und diesen Wert darf der Ausführer nicht in eine Variable des Untertyps **positive** legen.

Die folgende verbesserte Version der Prozedur ist nicht die schönste Lösung des Problems, aber sie ist korrekt und der ersten Version so ähnlich wie möglich:

```
01 declare
02 procedure ERSTES_X_NACH_Y_2(S: in ou string) is
03    I : integer := S'first - 1;
04 begin
05    while I+1 <= S'last loop
06       I := I + 1;
07       if S(I)  = 'X' then
08          S(I) := 'Y';
09          return; -- Beende (die Schleife und) das Unterprogramm.
10       end if;
11   end loop;
12 end ERSTER_X_NACH_Y_2;
```

In dieser Version gehört die Variable **I** zum Untertyp **integer** (statt zu **positive**), hat zuerst einen **um eins zu kleinen** Wert **S'first - 1** (Zeile 03) und wird erhöht, unmittelbar **bevor** sie in den Zeilen 07 und 08 benützt wird. In der ersten (falschen) Version hatte die Variable **I** zuerst den meistens richtigen Wert **S'first** und wurde **nach** ihrer Benützung in den Zeilen 06 und 07 erhöht. Dadurch wurde sie in bestimmten Extremfällen (TS1) "einmal zuviel erhöht". **o**

Wenn man ein Unterprogramm mit **string**-Parametern geschrieben hat, sollte man es (im Kopf oder mit Papier und Bleistift) daraufhin überprüfen, ob es auch Strings der folgenden Art korrekt verarbeiten kann:

```
01 declare
02    TS1 : string(+11..+17) := "ABC";
03    TS2 : string(-11..-17);
04    TS3 : string(positive'last-2..positive'last) := "XYZ";
```

TS1 ist ein String, dessen erster Index nicht gleich 1 ist. **TS2** ist ein **leerer String**, dessen Attribute **TS2'first** und **TS2'last** beide negative Werte haben. **TS3** ist ein String, dessen letzter Index gleich dem größtmöglichen Index überhaupt ist.

Natürlich muß man diese Teststrings für bestimmte Unterprogramme ein bißchen abwandeln und anpassen (je nachdem, was das Unterprogramm leisten soll) und durch weitere, spezifische Teststrings ergänzen.

Aufgabe 17.4.4.: Schreiben Sie drei Funktionen mit den folgenden Spezifikationen:

```
function  A_NACH_B  (S: string) return string;
function  A_NACH_BB (S: string) return string;
function AA_NACH_BBB(S: string) return string;
```

Jede dieser Funktionen soll als Ergebnis einen String liefern, der ihrem Parameterstring S "stark ähnelt", in dem aber folgende Ersetzungen vorgenommen wurden:

Die Funktion A_NACH_B	soll jedes Zeichen	'A'	durch das Zeichen 'B' ersetzen.
Die Funktion A_NACH_BB	soll jedes Zeichen	'A'	durch den String "BB" ersetzen.
Die Funktion AA_NACH_BBB	soll jeden Teilstring	"AA"	durch den String "BBB" ersetzen.

Falls der Parameter **S** kein Zeichen **'A'** bzw. keinen Teilstring **"AA"** enthält, soll der Wert von **S** unverändert als Ergebnis geliefert werden. Man beachte, daß **S** ein **in**-Parameter ist und somit nicht verändert werden kann. Das Ergebnis der ersten Funktion (A_NACH_B) ist immer **gleich lang** wie der Parameter **S**, aber die Ergebnisse der anderen beiden Funktionen können länger sein als **S**. Hier ein paar Beispiele für Aufrufe dieser Funktionen und was sie als Ergebnis liefern sollen:

Aktueller Parameter für S	Ergebnis der Funktion A_NACH_B	Ergebnis der Funktion A_NACH_BB	Ergebnis der Funk-tion AA_NACH_BBB
"AXAYA"	"BXBYB"	"BBXBBYBB"	"AXAYA"
"AAXAA"	"BBXBB"	"BBBBXBBBB"	"BBBXBBB"
"AAA"	"BBB"	"BBBBBB"	"BBBA"
"AAAA"	"BBBB"	"BBBBBBBB"	"BBBBBB"
"12345"	"12345"	"12345"	"12345"
""	""	""	""

Das dritte Beispiel (S => "AAA") soll deutlich machen, daß in der Zeichenkette **"AAA"** der Teilstring **"AA"** nur **einmal** vorkommt und nicht etwa **zweimal**. Dies ist für die Funktion AA_NACH_BBB wichtig. Das letzte Beispiel (S => "") soll daran erinnern, daß die Funktionen auch leere Strings ordnungsgemäß verarbeiten sollten. **o**

Zusammenfassung 17.4.:

- Der Indexbereich eines Stringliterals wie "Hallo!" beginnt immer mit **1**.
- Der Indexbereich eines konkatenierten Strings wie A & B & C beginnt immer mit dem ersten Index von A.
- Der Indexbereich eines **string**-Parameters beginnt nicht notwendig mit 1.
- Der Indexbereich eines **string**-Parameters kann bis **positiv**'**last** hinaufreichen, auch wenn der String relativ kurz ist.
- Zwei **string**-Parameter eines Unterprogramms sind nicht notwendig gleich lang und können ganz verschiedene Indexbereiche haben.
- Obwohl alle Indizes eines Strings zum Untertyp **positive** gehören müssen und gehören, können die Attribute '**first** und '**last** eines **leeren** Strings **negative** Werte haben.

17.5. Rekursive Unterprogramme

Ein **Unterprogramm**, welches sich selbst aufruft, nennt man **rekursiv** ("auf sich selbst zurückgreifend oder zurückgehend"). Die Idee einer **rekursiven Funktion** wurde schon vor Erfindung des Computers ab etwa 1930 von Mathematikern und Logikern entwickelt**.** Die ganz ähnliche Idee einer **rekursiven Prozedur** verbreitete sich erst mit der Programmiersprache Algol60.

Beispiel 17.5.1.: Eine rekursive Funktion:

Die sogenannte **Fakultätsfunktion** hat eine natürliche Zahl als Parameter und liefert eine natürliche Zahl als Ergebnis. Angewendet auf eine natürliche Zahl **n** liefert die Fakultätsfunktion das **Produkt** aller natürlichen Zahlen von **1** bis **n**. Angewendet auf die Zahl **0** oder auf die Zahl **1** liefert die Fakultätsfunktion als Ergebnis die Zahl **1**. Hier eine Vereinbarung dieser Funktion in Ada:

```
01 declare
02    type NAT is range 0..1_000_000_000;
03    function FAKULTAET(N: NAT) return NAT is
04    begin
05       if N < 2 then
06          return 1;                          -- kein rekursiver Aufruf!
07       else
08          return N * FAKULTAET(N-1);         -- rekursiver Aufruf!
09       end if;
10    end FAKULTAET;
11    N1 : NAT := FAKULTAET(3);                -- ein nicht-rekursiver Aufruf!
12    N2 : NAT;
13 begin
14    N2 := FAKULTAET(N1+2);                   -- ein nicht-rekursiver Aufruf!
15    ...
```

In der Vereinbarung der Funktion **FAKULTAET** (Zeile 03 bis 10) kommt ein Aufruf der Funktion **FAKULTAET** vor (in Zeile 08). Somit ist **FAKULTAET** eine **rekursive** Funktion. Den Aufruf der Funktion **FAKULTAET** innerhalb der Vereinbarung dieser Funktion (in Zeile 08, fett hervorgehoben) bezeichnet man als einen **rekursiven** (Funktions-) **Aufruf**.

Die Aufrufe der Funktion **FAKULTAET** in Zeile 11 und Zeile 14 sind **nicht-rekursive** Aufrufe, weil sie **außerhalb** der Vereinbarung der Funktion **FAKULTAET** stehen. **o**

Jedes rekursive Unterprogramm besteht im wesentlichen aus einer **Fallunterscheidung**, d.h. aus einer **if**-Anweisung, einer **case**-Anweisung oder einem ganz ähnlichen Befehl. Die Funktion **FAKULTAET** besteht im wesentlichen aus einer **if**-Anweisung (Zeile 05 bis 09).

Die Fallunterscheidung eines rekursiven Unterprogramms muß **rekursive Fälle** und **nicht-rekursive** ("normale") **Fälle** unterscheiden. Im obigen Beispiel liegt ein **normaler Fall** vor, wenn der Parameter **N** kleiner als **2** ist. In einem solchen Fall wird die Funktion FAKULTAET **nicht** rekursiv aufgerufen. Falls **N** gleich oder größer **2** ist, liegt ein **rekursiver Fall** vor, weil dann (in Zeile 08) die Funktion FAKULTAET **rekursiv aufgerufen** wird.

Aufgabe 17.5.1.: Führen Sie die Funktion **FAKULTAET** mehrmals mit Papier und Bleistift (oder "im Kopf") aus und füllen Sie dabei die folgende Wertetabelle aus:

N	0	1	2	3	4	5	6	
FAKULTAET(N)								

Wichtiger Hinweis: Wenden Sie die Funktion **FAKULTAET** zuerst auf die aktuellen Parameter **0** und **1** an, weil diese Parameter zu **nicht-rekursiven** Fällen führen. Tragen Sie das jeweilige Ergebnis in die Tabelle ein. Wenden Sie die Funktion erst danach auf den Parameter **2** an. Wenn Sie bei der Ausführung des Aufrufs **FAKULTAET(2)** zum rekursiven Aufruf **FAKULTAET(N-1)** in Zeile 08 kommen, dann sollten Sie diesen Aufruf **nicht** noch einmal ausführen, sondern **in der Wertetabelle nachsehen**, welches Ergebnis der Aufruf **FAKULTAET(1)** liefert. Ganz entsprechend sollten Sie vorgehen, wenn Sie dann die Aufrufe **FAKULTAET(3)**, **FAKULTAET(4)**, **FAKULTAET(5)**, ... etc. ausführen. **o**

Wenn man z.B. den Aufruf **FAKULTAET(4)** ausführen will, kommt man zum rekursiven Aufruf **FAKULTAET(3)**. Die Ausführung dieses Aufrufs führt zum rekursiven Aufruf **FAKULTAET(2)**. Die Ausführung dieses Aufrufs führt zum rekursiven Aufruf **FAKULTAET(1)**. Die Ausführung dieses Aufrufs liefert endlich ohne weitere rekursive Aufrufe ein Ergebnis (nämlich **1**). Die Verwaltung und Ausführung all dieser rekursiven Aufrufe ist nicht ganz einfach und häufig verwirrend. Ein **menschlicher** Ausführer kann diese Schwierigkeiten und Verwirrungen vermeiden, indem er eine **Wertetabelle** benützt und ausfüllt (wie im Hinweis zu Aufgabe 17.5.1. empfohlen). Heute übliche **maschinelle** Ausführer benützen im allgemeinen keine Wertetabelle, sondern beginnen und verwalten entsprechend viele **verschiedene Ausführungen** der Funktion **FAKULTAET**. Die folgende "dynamische" Darstellung soll deutlich machen, daß die **zuletzt** begonnene Ausführung der Funktion als **erste** beendet wird und die **zuerst** begonnene Ausführung **zuletzt**. Wenn ein maschineller Ausführer den Funktionsaufruf **FAKULTAET(4)** ausführt, dann führt er vor allem folgende Aktionen durch:

D01 Er beginnt die Ausführung **von FAKULTAET(4)**. In Zeile 08 wird klar, daß das
D02 Ergebnis dieses Aufrufs gleich **4 * FAKULTAET(3)** ist.
D03 Er beginnt die Ausführung von **FAKULTAET(3)**. In Zeile 08 wird klar, daß das
D04 Ergebnis dieses Aufrufs gleich **3 * FAKULTAET(2)** ist.
D05 Er beginnt die Ausführung von **FAKULTAET(2)**. In Zeile 08 wird klar, daß das
D06 Ergebnis dieses Aufrufs gleich **2 * FAKULTAET(1)** ist.
D07 Er beginnt die Ausführung von **FAKULTAET(1)**.
D08 Er beendet die Ausführung des Aufrufs **FAKULTAET(1)** (in Zeile 06)
D09 mit dem Ergebnis **1**.
D10 Er beendet die Ausführung des Aufrufs **FAKULTAET(2)** mit dem Ergebnis
D11 **2 * 1**, d.h. mit dem Ergebnis **2**.
D12 Er beendet die Ausführung des Aufrufs **FAKULTAET(3)** mit dem Ergebnis
D13 **3 * 2**, d.h. mit dem Ergebnis **6**.
D14 Er beendet die Ausführung des Aufrufs **FAKULTAET(4)** mit dem Ergebnis
D15 **4 * 6**, d.h. **24**.

Man sieht: Die Ausführung des Aufrufs **FAKULTAET(4)** wird als **erste** begonnen (in Zeile D01) und als **letzte** beendet (in Zeile D14). Die Ausführung des Aufrufs **FAKULTAET(1)** wird als **letzte** begonnen (in Zeile D07) und als **erste** beendet (in Zeile D08).

Der "Trick mit der Wertetabelle" beruht darauf, daß man die Werte der Ausdrücke **FAKULTAET(1)**, **FAKULTAET(2)**, **FAKULTAET(3)**, ... etc. "in einer geschickten Reihenfolge" berechnet und in die Wertetabelle einträgt. Dadurch braucht man nie einen rekursiven Aufruf der Funktion auszuführen, sondern kann seinen Wert in der Tabelle nachsehen.

Die Fakultätsfunktion kann man statt mit **Rekursion** auch mit einer **Schleife** programmieren und viele Menschen finden die Schleifen-Version einfacher als die rekursive Version. Im folgenden Beispiel geht es um eine Funktion, bei der viele Menschen die **rekursive Version** einfacher findern, als eine entsprechende **Schleifen-Version**.

Beispiel 17.5.2.: Wieviele Roboter können Roboter bauen?

Angenommen **ANZ_ELTERN** und **ANZ_TAGE** sind zwei natürliche Zahlen und **ANZ_ELTERN** viele Roboter können an **einem** Tag **einen** weiteren Roboter bauen. Angenommen, vor **ANZ_TAGEN** vielen Tagen haben wir mit **ANZ_ELTERN** vielen Robotern die Produktion begonnen. Wieviele Roboter konnten wir dann bis heute produzieren? Die folgende Funktion **FIBO**, deren Grundidee der Mathematiker Leonardo Fibonacci schon im Jahr 1202 in Pisa veröffentlichte, liefert die Antwort:

```
01 declare
02    type NAT is range 0..1_000_000_000;
03    function FIBO(ANZ_ELTERN, ANZ_TAGE: NAT) return NAT is
04       ANZ_ROBOTER_GESTERN: NAT;
05    begin
06       if ANZ_TAGE = 0 then
07          return ANZ_ELTERN;   -- keine Nachkommen
08       else
09          ANZ_ROBOTER_GESTERN := FIBO(ANZ_ELTERN, ANZ_TAGE-1);
10          return ANZ_ROBOTER_GESTERN + ANZ_ROBOTER_GESTERN / ANZ_ELTERN;
11       end if;
12    end FIBO;
13    N1 : NAT := FIBO(ANZ_ELTERN => 2, ANZ_TAGE => 5);
14    N2 : NAT;
15 begin
16    N2 := FIBO(ANZ_TAGE => 7, ANZ_ELTERN => N1-3);
17    ...
```

Die Funktion **FIBO** (vereinbart in Zeile 03 bis 12) ist **rekursiv**, weil sie sich in Zeile 09 selbst aufruft. Sie besteht im wesentlichen aus der **if**-Anweisung in Zeile 06 bis 11. Mit dieser Anweisung werden die **nicht-rekursiven Fälle** von den **rekursiven Fällen** unterschieden. Ein nicht-rekursiver Fall liegt vor, wenn der Parameter **ANZ_ELTERN** einen beliebigen Wert hat und **ANZ_TAGE** gleich **0** ist. Alle anderen Fälle sind rekursiv. Die Aufrufe der Funktion **FIBO** in den Zeilen 13 und 16 sind **nicht-rekursive** Aufrufe, da sie außerhalb der Vereinbarung der Funktion erfolgen.

In Zeile 09 wird dem Ausführer befohlen, die Anzahl der "gestern abend bei Arbeitsschluß" vorhandenen Roboter zu berechnen und in die Variable **ANZ_ROBOTER_GESTERN** zu tun. Der Ausdruck **ANZ_ROBOTER_GESTERN / ANZ_ELTERN** gibt dann an, wieviele Roboter "im Verlauf des heutigen Arbeitstages" produziert werden konnten. Die Summe **ANZ_ROBOTER_GESTERN + ANZ_ROBOTER_GESTERN / ANZ_ELTERN** (in Zeile 10) gibt dann an, wieviele Roboter "heute abend bei Arbeitsschluß" vorhanden sind. **o**

Der Parameter **ANZ_ELTERN** hat mit der Rekursivität der Funktion **FIBO** nichts zu tun, denn er wird im rekursiven Aufruf der Funktion (in Zeile 09) unverändert "weitergereicht". Deshalb sollte man zuerst für diesen Parameter einen bestimmten Wert festlegen und dann die Funktionswerte für diesen **ANZ_ELTERN**-Wert und verschiedene **ANZ_TAGE**-Werte berechnen. Insgesamt sieht eine **Wertetabelle für die Funktion FIBO** etwa so aus:

ANZ_TAGE =>	0	1	2	3	4	5	
FIBO(1, ANZ_TAGE)							
FIBO(2, ANZ_TAGE)							

FIBO(3, ANZ_TAGE)							
FIBO(4, ANZ_TAGE)							

Aufgabe 17.5.2.: Führen Sie die Funktion FIBO mehrmals mit Papier und Bleistift aus. Füllen Sie dabei in obiger Wertetabelle alle (oder zumindest einige) der leeren Kästchen aus. Wählen Sie die Reihenfolge, in der Sie die Kästchen ausfüllen, so geschickt, daß Sie nie einen rekursiven Funktionsaufruf ausführen müssen, sondern das Ergebnis eines solchen Aufrufs immer in der Wertetabelle nachsehen können. **o**

Aufgabe 17.5.3.: Was passiert, wenn ein (menschlicher oder maschineller) Ada-Ausführer den Funktionsaufruf **FIBO(ANZ_ELTERN => 0, ANZ_TAGE => 3)** ausführt? Dieser Fall und ähnliche Fälle wurden in obiger Wertetabelle absichtlich nicht vorgesehen. Wie könnte man die Funktion **FIBO** verbessern, so daß diese Fälle offensichtlich nicht erlaubt sind? **o**

Aufgabe 17.5.4.: Lassen Sie das Beispielprogramm **REKUP_02** mehrmals von einem maschinellen Ausführer ausführen und geben Sie dabei verschiedene Werte für die Anzahl der Eltern eines Roboters an. Beachten Sie dabei, daß auch bei Robotern eine Art "Bevölkerungsexplosion" stattfindet, wenn für die Produktion eines Roboters nur **wenige** Eltern (z.B. zwei oder ein Roboter) nötig sind und man die Roboter genügend **lange** Nachkommen produzieren läßt. Konkret bedeutet das: Das Programm **REKUR_02** wird in gewissen Fällen mit einem **constraint_error** abgebrochen, weil eine Roboterzahl zu groß wurde. **o**

Aufgabe 17.5.5.: Schreiben Sie eine Funktion namens **FIBO_S** ("S" wie "Schleife"), die genau das Gleiche leistet wie die Funktion **FIBO**, aber anstelle der Rekursion eine **Schleife** enthält. Beurteilen Sie dann, welche Version (**FIBO_S** oder **FIBO**) Sie persönlich **einfacher** finden. **o**

Die ersten beiden Beispiele für rekursive Unterprogramme waren **Funktionen**. Das folgende Beispiel stellt eine **rekursive Prozedur** vor, die mehrere Zeilen zum Bildschirm ausgibt. Die einzelnen Zeilen bestehen nur aus Sternchen **'*'**, sind aber so verschieden lang, daß auf dem Bildschirm ein "Sägezahnmuster" entsteht, z.B. das folgende 10 Zeilen lange Muster:

```
********
******
****
**
********
******
****
**
********
******
```

Beispiel 17.5.3: Eine rekursive Prozedur zum Ausgeben von Sägezahnmustern:

```
01 declare
03    procedure SAEGEZAHN(ZEILEN, AKT_L, MAX_L: natural) is
04       STERNCHEN : constant string(1..AKT_L) := (others => '*');
05    begin
06       if ZEILEN = 0 then
07          return;                                    -- kein rekursiver Aufruf!
08       else
09          ada.text_io.put_line(item => STERNCHEN);
10          if AKT_L > 2 then
11             SAEGEZAHN(ZEILEN-1, AKT_L-2, MAX_L);    -- rekursiver Aufruf!
12          else
```

```
13            SAEGEZAHN(ZEILEN-1, MAX_L, MAX_L);          -- rekursiver Aufruf!
14         end if;
15       end if;
16    end SAEGEZAHN;
17 begin
18    SAEGEZAHN(MAX_L => 8, AKT_L => 8, ZEILEN => 10);
19    ...
20    SAEGEZAHN(ZEILEN => 20, AKT_L => 3, MAX_L => 11);
21    ...
```

Die Prozedur **SAEGEZAHN** (vereinbart in Zeile 03 bis 16) ist **rekursiv**, denn sie enthält zwei **rekursive Prozeduraufrufe** in Zeile 11 und in Zeile 13. Die Aufrufe der Prozedur **SAEGEZAHN** in den Zeilen 18 und 20 sind **nicht-rekursiv**, weil sie **außerhalb** der Vereinbarung dieser Prozedur stehen.

Der Parameter **ZEILEN** gibt an, wie viele Zeilen noch auszugeben sind. Der Parameter **MAX_L** legt die maximale Länge einer Sternchenzeile und **AKT_L** die "aktuelle" Länge der nächsten Sternchenzeile fest. In Zeile 04 wird eine Stringkonstante vereinbart, die aus genau **AKT_L** vielen Sternchen besteht. In Zeile 06 wird zwischen **nicht-rekursiven Fällen** und **rekursiven Fällen** unterschieden. In den nicht-rekursiven Fällen, d.h. wenn **ZEILEN** gleich 0 ist und somit keine weiteren Zeilen mehr auszugeben sind, wird die Prozedur **SAEGEZAHN** ohne weitere Aktionen beendet (in Zeile 07). In den rekursiven Fällen, d.h. wenn **ZEILEN** noch größer als 0 ist, wird zunächst eine Sternchenzeile ausgegeben (in Zeile 09). Dann wird in Zeile 10 entschieden, ob als nächstes eine um **2** Sternchen kürzere Zeile ausgegeben werden soll (rekursiver Aufruf in Zeile 11) oder eine Sternchenzeile maximaler Länge (rekursiver Aufruf in Zeile 13). Der **ZEILEN**-Parameter ist in beiden rekursiven Aufrufen um **1** kleiner als der momentane Wert. **o**

Aufgabe 17.5.6.: Skizzieren Sie das Sägezahnmuster, welches durch den Prozeduraufruf in Zeile 18 zum Bildschirm ausgegeben wird. **o**

Aufgabe 17.5.7.: Schreiben Sie eine Prozedur namens **DREIECKE**, welche ganz ähnlich wie die Prozedur **SAEGEZAHN** ein Muster von Sternchenzeilen zum Bildschirm ausgibt. Die Sternchenzeilen sollen aber kein Sägezahnmuster bilden, sondern ein **Dreiecksmuster** wie z.B. das folgende 14 Zeilen lange Muster:

```
   *
  ***
 *****
*******
   *
  ***
 *****
*******
   *
  ***
 *****
*******
   *
  ***
```

o

Die Prozeduren **SAEGEZAHN** und **DREIECKE** sind "Fingerübungen", über die man sich mit rekursiven Prozeduren vertraut machen kann. Wirklich **nützlich** und **sinnvoll** sind rekursive Prozeduren aber vor allem dann, wenn man **rekursive Datenstrukturen** wie z.B. **verkettete Listen** oder **Bäume** zu bearbeiten hat. Solche rekursiven Datenstrukturen werden im Abschnitt XXX behandelt.

Die bisher in diesem Abschnitt besprochenen Unterprogramme (**FAKULTAET**, **FIBO**, **SAEGE-ZAHN**) sind **direkt rekursiv**, weil sie sich "direkt selbst aufrufen". Von **indirekter Rekursion** spricht man, wenn ein Unterprogramm U1 ein Unterprogramm U2 aufruft, und U2 wiederum U1 aufruft, oder wenn U17 von U16 aufgerufen wird, und U16 von U15 und U15 von U14 und ... und U2 von U1 und wenn schließlich U1 wieder von U17 aufgerufen wird.

Schon im Beispiel 17.3.4. (siehe oben im Abschnitt 17.3.3) wurden zwei indirekt rekursive Unterprogramme namens **UPROS_71** und **UPROS_72** vorgestellt.

Wenn man in Ada **indirekt rekursive Unterprogramme** schreibt, muß man für mindestens eines der Unterprogramme zusätzlich zum Rumpf eine **Spezifikation** angeben. Aus Gründen der Schönheit und Ausgewogenheit empfiehlt es sich, in solchen Fällen für **alle** an der indirekte Rekursion beteiligten Unterprogramme vorab Spezifikationen anzugeben.

Aufgabe 17.5.8.: Warum ist das folgende Unterprogramm kein richtiges rekursives Unterprogramm? Was macht ein Ada-Ausführer, wenn er den Prozeduraufruf in Zeile 08 ausführt?

```
01 declare
02    procedure NICHT_RICHTIG(N: natural) is
03    begin
04       ada.text_io.put(item => '*');
05       NICHT_RICHTIG(N+1);
06    end NICHT_RICHTIG;
07 begin
08    NICHT_RICHTIG(N => 5);
09    ...
```

o

Aufgabe 17.5.9.: Ein **Plusbaum** ist entweder eine **positive Ganzzahl** (z.B. **1**, **2**, **3**, etc.) oder eine "**voll geklammerte Addition**" von positiven Ganzzahlen. Beispiele für Plusbäume: **1**, **17**, **(1+1)**, **(2+5)**, **(7+136)**, ..., **((3+7)+5)**, **(3+(7+5))**, ..., **((5+6)+(16+15))**, ... etc.. Die Ausdrücke **1+1**, **(3+7+5)** und **(1+2+(3+4))** sind **keine** Plusbäume, weil sie **nicht voll geklammert** sind. Es gibt genau **zwei** Plusbäume, die den Wert **2** darstellen, nämlich den Plusbaum **2** und den Plusbaum **(1+1)**. Die Zahl **3** wird durch **fünf** verschiedene Plusbäume dargestellt, nämlich durch **3**, **(1+2)**, **(2+1)**, **(1+(1+1))** und **((1+1)+1)**. Ganz entsprechend wird die Zahl **4** durch **15** verschiedene Plusbäume dargestellt und die Zahl **5** durch **51** Plusbäume etc.. Schreiben Sie eine Funktion mit folgender Spezifikation:

```
function PLUSB(P: positive) return positive;
-- Liefert die Anzahl der Plusbaeume, die die Zahl P darstellen.
```

Beispiele:
Der Funktionsaufruf PLUSB(1) sollte das Ergebnis 1 liefern.
Der Funktionsaufruf PLUSB(2) sollte das Ergebnis 2 liefern.
Der Funktionsaufruf PLUSB(3) sollte das Ergebnis 5 liefern.
Der Funktionsaufruf PLUSB(4) sollte das Ergebnis 15 liefern.
Der Funktionsaufruf PLUSB(5) sollte das Ergebnis 51 liefern.
... ...

Selbst **mit** Rekursion ist es nicht ganz einfach, die Funktion **PLUSB** zu entwickeln. Aber **ohne** Rekursion wäre es vermutlich noch deutlich schwieriger. **o**

Allgemein und weitgehend unabhängig von der verwendeten Programmiersprache gilt: Alle Probleme, die man mit Hilfe von **Schleifen** lösen kann, sind im Prinzip auch mit **Rekursion** lösbar.

Und umgekehrt sind alle Probleme, die man mit Hilfe von **Rekursion** lösen kann, im Prinzip auch mit **Schleifen** lösbar. Diese "harte Tatsache", die man mathematisch beweisen kann, ändert nichts an der "weichen Erfahrung", daß sich bestimmte Probleme eleganter **mit Rekursion** und andere besser **mit Schleifen** lösen lassen.

Zusammenfassung 17.5.:

- Ein Unterprogramm U ist **direkt rekursiv**, wenn die **Vereinbarung** von U **Aufrufe** von U enthält.
- Ein Unterprogramm U1 ist **indirekt rekursiv**, wenn es ein Unterprogramm U2 direkt aufruft und U2 direkt oder indirekt wieder U1 aufruft.
- Jedes rekursive Unterprogramm besteht im wesentlichen aus einer **Fallunterscheidung** (**if**- oder **case**-Anweisung).
- Diese **if**- oder **case**-Anweisung muß zwischen **nicht-rekursiven Fällen** und **rekursiven Fällen** unterscheiden.
- Zahlreiche Probleme lassen sich besonders elegant und übersichtlich **mit Rekursion** lösen.
- Alle Probleme, die man mit **Schleifen** lösen kann, sind auch mit **Rekursion** lösbar.
- Alle Probleme, die man mit **Rekursion** lösen kann, sind auch mit **Schleifen** lösbar.

Zentraldokument: Nach Filialdokument A95-17-17

18. Lebensdauer und Sichtbarkeit

In einem Ada-Programm kann der Programmierer unter anderem folgende **Größen** vereinbaren: **Unterprogramme**, **Typen**, **Konstanten**, **Variablen** und **Ausnahmen**. Eine Größe **vereinbaren** bedeutet: sie vom Ausführer **erzeugen** und mit einem **Namen** versehen lassen. Man beachte, daß **Werte** nicht erzeugt, sondern (aufgrund eines Ausdrucks) **berechnet** werden. In diesem Abschnitt geht es vor allem um vereinbarte Größen und nicht um Werte.

Eine vereinbarte Größe existiert nicht "ewig", sondern wird vom Ausführer **erzeugt** und wieder **zerstört**. Sie hat also eine bestimmte **Lebensdauer**, die durch die Regeln der Sprache Ada genau festgelegt wird. Ziel des Programmierers sollte es sein, die Lebensdauer von Größen **möglichst kurz** zu halten. Denn solange eine Größe lebt, belegt sie möglicherweise Betriebsmittel (Speicher) und kann auch aus Versehen falsch benutzt werden.

Eng verbunden mit der **Lebensdauer** einer **Größen** ist die **Sichtbarkeit** ihrer **Vereinbarung**. In einem **großen Programm**, bei dessen Entwicklung ein ganzes Team die Rolle des Programmierers übernimmt, ist es praktisch schwierig bis unmöglich, für jede Größe **einen neuen Namen** zu erfinden. Insbesondere, wenn man schon vorhandende Programmteile wiederverwenden oder "dazugekaufte" Programmteile in ein großes Programm einbauen will, kann man es kaum vermeiden, daß bestimmte Namen mehrfach verwendet werden. Die **Sichtbarkeitsregeln** (visibility rules) einer höheren Sprache wie Ada machen es möglich, daß man in verschiedenen Programmteilen mehrere **verschiedene Größen** mit dem **gleichen Namen** wie z.B. **OTTO** vereinbaren kann.

Man kann eine Größe nur an den Stellen eines Programms **benützen**, an denen ihre Vereinbarung **sichtbar** ist. Z.B. kann man ein **Unterprogramm** nur da **aufrufen**, wo seine Vereinbarung sichtbar ist. Nur da, wo die Vereinbarung einer **Variablen** sichtbar ist, kann man den Wert der Variablen **verändern** oder einer anderen Variablen zuweisen etc.. Eine **Ausnahme** kann man nur da **auslösen**, wo ihre Vereinbarung sichtbar ist etc.. Die **Sichtbarkeitsregeln** von Ada legen genau fest, wo in einem Programmtext eine Vereinbarung sichtbar ist und wo sie nicht sichtbar ist.

In diesem Abschnitt werden die wichtigsten Lebensdauer- und Sichtbarkeitsregeln anhand von Beispielen erläutert. Dabei werden unter anderem die Fachbegriffe **Vereinbarungsbereich** (declarative region), **Gültigkeitsbereich** einer Vereinbarung (scope) und **Sichtbarkeitsbereich** einer Vereinbarung eingeführt.

Im Abschnitt 20 wurden **zwei Arten** von Unterprogrammen eingeführt: **Bibliotheksunterprogramme**, die dem Ausführer als eigenständige Einheiten übergeben und von ihm in seiner Bibliothek abgelegt werden, und **enthaltene Unterprogramme**, deren Vereinbarung in einer anderen Programmeinheit enthalten sind (z.B. in einem anderen Unterprogramm).

Die Lebensdauer- und Sichtbarkeitsregeln für **Bibliotheksunterprogramme** sind relativ einfach. Ein Bibliotheksunterprogramm **beginnt zu leben**, wenn der Ausführer die Vereinbarung des Unterprogramms akzeptiert, ein entsprechendes Unterprogramm erzeugt und in seiner Bibliothek ablegt. Das Bibliotheksunterprogramm lebt solange, bis es aus der Bibliothek des Ausführers **gelöscht** wird. Was man im Einzelnen tun muß, um ein Unterprogramm aus einer Bibliothek zu löschen, wird von der Sprache Ada nicht festgelegt und ist von Ausführer zu Ausführer verschieden.

Wenn ein Bibliotheksunterprogramm namens **OTTO** zu leben beginnt, ist es zunächst **nirgends** sichtbar, außer in seiner eigenen Vereinbarung (damit rekursive Aufrufe möglich sind). Man kann **OTTO** aber in jeder anderen Bibliothekseinheit sichtbar machen, indem man vor die Vereinbarung dieser anderen Einheit eine Kontextklausel **with OTTO;** schreibt. Siehe dazu das Beispielprogramm **HALLO_20** und die Beispiele 20.3. und 23.4..

Die wichtigsten Lebensdauer- und Sichtbarkeitsregeln für **enthaltene Größen**, deren Vereinbarungen in einem Unterprogramm **enthalten** sind, werden in den folgenden Beispielen erläutert. Der Name **LESIB** soll an "Lebensdauer und Sichtbarkeit" erinnern, der Name **EFUNK** an "enthaltene Funktion" und **EPROC** an "enthaltene Prozedur".

Beispiel 18.1.: Lebensdauer und Sichtbarkeit enthaltener Größen

```
01 with ada.text_io;
02 procedure LESIB_01 is
03    TEXT1: constant string := " Stdn ";
04    function EFUNK_01(ZAHL: integer) return string is
05       TEXT2: constant string := integer'image(ZAHL);
06    begin -- EFUNK_01
07       return TEXT2;
08    end EFUNK_01;
09    TEXT3: constant string := EFUNK_01(ZAHL => 35);
10 begin -- LESIB_01
11    ada.text_io.put(item => EFUNK_01(ZAHL => 17) &   TEXT1);
12    ada.text_io.put(item => TEXT3                & " Min ");
13    ada.text_io.put(item => EFUNK_01(ZAHL => 58) & " Sec ");
14 end LESIB_01;
```

Das Bibliotheksunterprogramm **LESIB_01** beginnt zu leben, wenn der Ausführer die Zeilen 01 bis 14 akzeptiert, ein Unterprogramm namens **LESIB_01** erzeugt und in seiner Bibliothek ablegt.

Im Vereinbarungsteil von **LESIB_01** werden **drei** Größen vereinbart: eine Konstante namens **TEXT1**, eine Funktion namens **EFUNK_01** und noch eine Konstante namens **TEXT3**. Man beachte aber, daß die Prozedur **LESIB_01** keine **Größen** namens **TEXT1**, **EFUNK_01** und **TEXT3** enthält, sondern nur die **Vereinbarungen** dieser Größen, d.h. **Befehle**, diese Größen zu **erzeugen**. Praktisch bedeutet das: Die Größen **TEXT1**, **EFUNK_01** und **TEXT3** beginnen noch **nicht** zu leben, wenn die Prozedur **LESIB_01** erzeugt wird, sondern erst "viele Nanosekunden später".

Wenn der Benutzer den Ausführer auffordert, das Programm **LESIB_01** auszuführen, dann führt der Ausführer zuerst die **Vereinbarungen** in den Zeilen 03 bis 09 aus und erzeugt die Konstante **TEXT1**, die Funktion **EFUNK_01** und die Konstante **TEXT3** (in dieser Reihenfolge). Erst in diesem Moment beginnen diese drei Größen zu leben.

Man beachte aber, daß die Funktion **EFUNK_01** keine **Größe** namens **TEXT2** enthält, sondern nur die **Vereinbarung** dieser Größe. Praktisch bedeutet das: Die Größe **TEXT2** beginnt nicht zu leben, wenn die Funktion **EFUNK_01** erzeugt wird, sondern erst "viele Nanosekunden später".

Als Teil des **put**-Befehls in Zeile 11 muß der Ausführer unter anderem den Funktionsaufruf **EFUNK_01(ZAHL => 17)** ausführen. Dazu erzeugt er eine Konstante namens **ZAHL** vom Typ **integer** mit dem Wert **17** und führt dann die Vereinbarung in Zeile 05 aus, d.h. er erzeugt eine Konstante namens **TEXT2**. Erst in diesem Moment beginnt die Größe **TEXT2** zu leben. Sie lebt nur, bis der Ausführer die Funktion **EFUNK_02** fertig ausgeführt hat.

In Zeile 13 wird die Funktion **EFUNK_01** erneut aufgerufen. Entsprechend muß der Ausführer erneut eine Konstante namens **ZAHL** und dann eine Konstante namens **TEXT2** erzeugen. Diese beiden Konstanten zerstört er wieder, wenn er die Funktion fertig ausgeführt hat.

Nachdem der Ausführer den Befehl in Zeile 13 fertig ausgeführt hat, zerstört er auch die Größen **TEXT1**, **EFUNK_01** und **TEXT_03** wieder und beendet damit diese Ausführung des Programms **LESIB_01**.

Mit diesem Beispiel soll deutlich gemacht werden:
1. Die Bibliotheksprozedur **LESIB_01** lebt beliebig lange in der Bibliothek des Ausführers (bis sie vom Programmierer wieder gelöscht wird).
2. Die Größen **TEXT1**, **EFUNK_02** und **TEXT3** leben nur, während der Ausführer die Prozedur **LESIB_01** ausführt.
3. Die Größe **TEXT2** lebt nur, während der Ausführer die Funktion **EFUNK_02** ausführt.

Soviel zur Lebensdauer der Größen **LESIB_01**, **TEXT1**, **EFUNK_01**, **TEXT3** und **TEXT2**.

Jede Vereinbarung hat einen bestimmten **Gültigkeitsbereich**. **Außerhalb** dieses Gültigkeitsbereichs ist die Vereinbarung auf jeden Fall **nicht sichtbar**, d.h. dort kann auf keinen Fall auf die vereinbarte Größe zugegriffen werden. **Innerhalb** ihres Gültigkeitsbereichs gibt es Stellen, an denen die Vereinbarung **sichtbar** ist und andere Stellen, an denen sie **nicht sichtbar** ist. Hier ein paar konkrete Beispiele:

In Zeile 03 wird eine Konstante namens **TEXT1** vereinbart. Der **Gültigkeitsbereich** dieser Vereinbarung reicht vom **Beginn der Vereinbarung** bis zum **Ende des umfassenden Unterprogramms**, d.h. von Zeile 03 bis 14. Der **Sichtbarkeitsbereich** dieser Vereinbarung beginnt unmittelbar **nach** der Vereinbarung und reicht (in diesem einfachen Beispiel) ununterbrochen bis zum Ende ihres Gültigkeitsbereichs, d.h. von Zeile 04 bis 14. Warum man zwischen einem **Gültigkeitsbereich** und einem **Sichtbarkeitsbereich** unterscheidet, wird in späteren (ein bißchen komplizierteren) Beispielen deutlich gemacht.

Allgemein reicht der **Gültigkeitsbereich** eines **Typs**, einer **Konstanten**, einer **Variablen** und einer **Ausnahme** vom Beginn der Vereinbarung bis zum Ende des unmittelbar umfassenden Vereinbarungsbereichs. In einfachen Fällen beginnt der **Sichtbarkeitsbereich** solcher Größen unmittelbar **nach** ihrer Vereinbarung und reicht bis zum Ende ihres Gültigkeitsbereichs.

Für eine **Unterprogrammvereinbarung** beginnt der Gültigkeitsbereich "ein bißchen später" und der Sichtbarkeitsbereich "ein bißchen früher". Der **Gültigkeitsbereich** einer Unterprogrammvereinbarung beginnt erst nach dem Wort **is** am Anfang des Rumpfes und reicht (wie bei anderen Größen auch) bis zum Ende des unmittelbar umfassenden Vereinbarungsbereichs. In einfachen Fällen ist ein Unterprogramm in seinem gesamten Gültigkeitsbereich auch **sichtbar** (und nicht erst **nach** seiner Vereinbarung). Diese Regelung macht es möglich, daß ein Unterprogramm sich rekursiv aufruft.

Falls ein enthaltenes Unterprogramm eine separate **Spezifikation** besitzt, beginnt sein Gültigkeitsbereich schon unmittelbar nach dieser Spezifikation, und nicht erst nach dem **is** am Anfang seines Rumpfes.

Die Vereinbarung der Funktion **EFUNK_01** (Zeile 04 bis 08 in obigem Beispiel) ist von Zeile 05 bis Zeile 14 **gültig** und **sichtbar**. Die Funktion könnte sich z.B. in Zeile 07 selbst aufrufen.

Der **Gültigkeitsbereich** der Konstantenvereinbarung in Zeile 09 reicht von Zeile 09 bis 14, ihr **Sichtbarkeitsbereich** von Zeile 10 bis 14.

Der **Gültigkeitsbereich** der Konstantenvereinbarung in Zeile 05 beginnt in Zeile 05, reicht aber nur bis Zeile 08, weil dort die umfassende Funktion **EFUNK_01** zu Ende ist. Der **Sichtbarkeitsbereich** der Vereinbarung reicht von Zeile 06 bis 08.

Hier ein paar praktische Konsequenzen der verschiedenen Sichtbarkeiten: Der Wert der Konstanten **TEXT3** kann mit Hilfe der Funktion **EFUNK_01** berechnet werden, weil die Funktion in Zeile 09 schon **sichtbar** ist. Der Wert der Konstanten **TEXT1** kann dagegen nicht auf diese Weise berechnet werden, weil die Funktion **EFUNK_01** in Zeile 03 noch **nicht sichtbar** ist.

Andersherum ist die Konstante **TEXT1** innerhalb der Funktion **EFUNK_01** sichtbar, die Konstante **TEXT3** aber nicht. Somit könnte man innerhalb der Funktion z.B. **TEXT2** mit **TEXT1** vergleichen, aber nicht mit **TEXT3**.

Auf die Konstante **TEXT2** kann man nur **innerhalb** der Funktion **EFUNK_01** zugreifen, **außerhalb** der Funktionsvereinbarung ist sie nicht sichtbar.

Aufgabe 18.1.: Ermitteln Sie mit Papier und Bleistift, was das Programm **LESIB_01** zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt. **o**

Allgemein gilt: Die Vereinbarung eines Unterprogramms ist gleichzeitig ein sogenannter **Vereinbarungsbereich**. Dies gilt gleichermaßen für **Bibliotheksunterprogramme** (wie z.B. für **LESIB_01**) und für **enthaltene Unterprogramme** (wie z.B. für **EFUNK_01**). Gibt man für ein Unterprogramm nicht nur einen **Rumpf**, sondern zusätzlich eine separate **Spezifikation** an, dann gehört die Spezifikation zum Vereinbarungsbereich des Unterprogramms dazu. Wird innerhalb eines Vereinbarungsbereichs eine Größe vereinbart, dann reicht deren **Gültigkeitsbereich** immer vom **Anfang der Vereinbarung** bis zum **Ende des unmittelbar umfassenden Vereinbarungsbereichs**.

Den **Vereinbarungsbereich** eines Unterprogramms sollt man nicht mit seinem **Vereinbarungsteil** verwechseln. Z.B. reicht der **Vereinbarungsbereich** der Prozedur **LESIB_01** von Zeile 02 bis 14, der **Vereinbarungsteil** dagegen nur von Zeile 03 bis 09.

Beispiel 18.2.: Gleichnamige Variablen in geschachtelten Vereinbarungsbereichen:

```
01 with ada.text_io;
02 procedure LESIB_02 is
03    ANKE: character := 'A';
04    BERT: character := 'B';
05    procedure EPROC_01 is
06       CARL: character := BERT;
07       BERT: character := ANKE;
08       ANKE: character := BERT;
09    begin -- EPROC_01
10       ada.text_io.put(item => CARL & BERT & ANKE);
11    end EPROC_01;
12 begin -- LESIB_02
```

```
13    EPROC_01;
14    ada.text_io.put(item => ANKE & BERT);
15 end LESIB_02;
```

Der Vereinbarungsbereich der Prozedur **EPROC_01** (Zeile 05 bis 11) ist im Vereinbarungsbereich der Prozedur **LESIB_02** (Zeile 02 bis 15) enthalten. Obwohl im **äußeren** Vereinbarungsbereich Variablen namens **ANKE** und **BERT** vereinbart werden (in Zeile 03 und 04), darf man im **inneren** Vereinbarungsbereich "neue Variablen mit gleichen Namen" vereinbaren (in Zeile 07 und 08).

Mit **ANKE-03** sei **die** Variable namens **ANKE** bezeichnet, die in Zeile 03 vereinbart wird. Entsprechend bezeichnet **ANKE-08** die Variable namens **ANKE**, die in Zeile 08 vereinbart wird, etc.. **ANKE-03** und **ANKE-08** sind zwei verschiedene Variablen, die nur den gleichen Namen haben.

Aus den mit dem Beispiel 18.1. eingeführten Sichtbarkeitsregeln folgt: In Zeile 06 ist die Variable **BERT-04** sichtbar. Die gleichnamige Variable **BERT-07** ist in Zeile 06 aber noch **nicht sichtbar**. Also wird die Variable **CARL** mit dem Wert von **BERT-04** (einem 'B') initialisiert.

Ganz entsprechend wird die Variable **BERT-07** mit dem Wert von **ANKE-03** (einem 'A') initialisiert, und nicht etwa mit dem Wert von **ANKE-08**.

In Zeile 08 ist die Variable **BERT-07** sichtbar und sie **verdeckt** dort die gleichnamige Variable **BERT-04**. Also wird **ANKE-08** mit dem Wert von **BERT-07** (einem 'A') initialisiert.

Man sieht: Der Name **BERT** ganz rechts in Zeile 06 bezeichnet die Variable **BERT-04** mit dem Wert 'B'. Dagegen bezeichnet der gleiche Name **BERT** in Zeile 08 die Variable **BERT-07** mit dem Wert 'A'.

Der **Sichtbarkeitsbereich** der Variablen **BERT-04** "hat ein Loch" und besteht somit aus zwei Teilen: Zeile 05 bis 06 und 12 bis 15. In den Zeilen 07 bis 11 wird **BERT-04** von **BERT-07** verdeckt und ist dort nicht (direkt) sichtbar.

Man beachte, daß das **Loch** im Sichtbarkeitsbereich von **BERT-04** aus dem **Gültigkeitsbereich** der Variablen **BERT-07** besteht, und nicht etwa aus ihrem **Sichtbarkeitsbereich**. Als Konsequenz ergibt sich: Die Zeile 07 liegt zwar im Gültigkeitsbereich von **BERT-04** und im Gültigkeitsbereich von **BERT-07**, trotzdem ist an dieser Stelle **keine** Variable namens BERT **sichtbar**. Denn der Sichtbarkeitsbereich von **BERT-07** beginnt erst in Zeile 08, trotzdem verdeckt **BERT-07** die Variable **BERT-04** schon in Zeile 07. Praktisch bedeutet das: In Zeile 07 dürfte **nicht** stehen:

```
08    BERT: character := BERT;
```

weil der Name BERT rechts vom Initialisierungszeichen ":=" **keine** vereinbarte Größe bezeichnet.

Aufgabe 18.2.: Ermitteln Sie mit Papier und Bleistift, was das Programm **LESIB_02** zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt. **o**

Beispiel 18.3.: Lebensdauer und Sichtbarkeit in Blöcken:

```
01 with ada.text_io;
```

```
02 procedure LESIB_03 is
03    OTTO: character := 'A';
04    procedure EPROC_01 is
05       OTTO: character := 'B';
06    begin -- EPROC_01
07       ada.text_io.put(item => OTTO);
08       B1: declare
09          OTTO: character := 'C';
10       begin -- B1;
11          ada.text_io.put(item => OTTO);
12       end B1;
13       ada.text_io.put(item => OTTO);
14    end EPROC_01;
15 begin -- LESIB_03
16    ada.text_io.put(item => OTTO);
17    EPROC_01;
18    B2: declare
19       OTTO: character := 'D';
20    begin -- B2
21       ada.text_io.put(item => OTTO);
22    end B2;
23    EPROC_01;
24    ada.text_io.put(item => OTTO);
25 end LESIB_03;
```

Eine **Blockanweisung** ist auch ein **Vereinbarungsbereich**. Der Gültigkeitsbereich einer Größe, die in einem Block vereinbart wird, endet mit dem Block.

Aufgabe 18.3.: Geben Sie für die Variablen **OTTO-03**, **OTTO-05**, **OTTO-09** und **OTTO-19** jeweils den **Gültigkeitsbereich** und den **Sichtbarkeitsbereich** an. Zur Erinnerung: **OTTO-03** ist **die** Variable namens **OTTO**, deren Vereinbarung in Zeile 03 steht. **o**

Aufgabe 18.4.: Ermitteln Sie mit Papier und Bleistift, was das Programm **LESIB_03** zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt. **o**

Eine Größe namens **OTTO** ist an einer bestimmten Stelle eines Programms **direkt sichtbar**, wenn man sie dort mit ihrem **einfachen Namen** OTTO bezeichnen kann. Manchmal ist eine Größe an einer bestimmten Stelle im Programm zwar nicht **direkt sichtbar**, aber immerhin noch **indirekt sichtbar** und kann mit einem **zusammengesetzten Namen** wie **LESIB_01.OTTO** ("die in **LESIB_01** vereinbarte Größe namens **OTTO**") oder **B1.OTTO** ("die in **B1** vereinbarte Größe namens **OTTO**") etc. bezeichnet werden. Im folgenden Beispiel sind verschiedene Variablen namens **OTTO** an verschiedenen Stellen nur **indirekt sichtbar**:

Beispiel 18.4.: Direkte und indirekte Sichtbarkeit:

```
01 with ada.text_io;
02 procedure LESIB_04 is
03    OTTO: character := 'A';
04    procedure EPROC_01 is
05       OTTO: character := 'B';
06    begin -- EPROC_01
07       ada.text_io.put(item =>          OTTO);
08       ada.text_io.put(item => LESIB_04.OTTO);
09       B1: declare
10          OTTO: character := 'C';
11       begin -- B1
12          ada.text_io.put(item =>          OTTO);
13          ada.text_io.put(item => EPROC_01.OTTO);
```

```
14          ada.text_io.put(item => LESIB_04.OTTO);
15          B2: declare
16             OTTO: character := 'D';
17          begin
18             ada.text_io.put(item =>          OTTO);
19             ada.text_io.put(item =>       B1.OTTO);
20             ada.text_io.put(item => EPROC_01.OTTO);
21             ada.text_io.put(item => LESIB_04.OTTO);
22          end B2;
23       end B1;
24    end EPROC_01;
25 begin -- LESIB_04
26    ada.text_io.put(item => OTTO);
27    EPROC_01;
28 end LESIB_04;
```

Die Variable **OTTO-05** ist in Zeile 08 **direkt sichtbar** und verdeckt dort die Variable **OTTO-03**. Trotzdem ist **OTTO-03** an dieser Stelle des Programms noch **indirekt sichtbar** und kann mit dem Namen **LESIB_04.OTTO** bezeichnet werden. Der zusammengesetzte Name **LESIB_04.OTTO** bezeichnet **die** Größe namens **OTTO**, deren Vereinbarung unmittelbar im Vereinbarungsbereich von **LESIB_04** steht. In diesem Beispiel ist das die Variable **OTTO-03**.

Ganz entsprechend bezeichnet der zusammengesetzte Name **EPROC_01.OTTO** in den Zeilen 13 und 20 **die** Größe namens **OTTO**, deren Vereinbarung unmittelbar im Vereinbarungsbereich namens **EPROC_01** steht. Das ist die Variable **OTTO-05**.

Ein zusammengesetzter Name wie z.B. **EPROC_01.OTTO** ist höchstens an **den** Stellen eines Programms erlaubt, an denen genau ein Vereinbarungsbereich namens EPROC_01 **direkt sichtbar** ist. Das ist z.B. **vor** Zeile 04 **nicht** der Fall.

Außerdem kann man mit der Punktnotation **EPROC_01.OTTO** eine Größe namens **OTTO** nur innerhalb ihres **Gültigkeitsvereichs** bezeichnen. Z.B. reicht der Gültigkeitsbereich der Variablen **OTTO-05** von Zeile 05 bis 24. **Außerhalb** dieses Bereichs kann man **OTTO-05** auch nicht mit einem zusammengesetzten Namen wie **EPROC_01.OTTO** bezeichnen.

Die **Lebensdauerregeln** und die **Sichtbarkeitsregeln** ergänzen und stützen sich gegenseitig, z.B. so: Wenn der Ausführer die **put**-Anweisung in Zeile 26 ausführt, lebt die Variable **OTTO-05** noch gar nicht. Also ist **OTTO-05** in Zeile 26 weder **direkt** noch **indirekt** sichtbar.

Aufgabe 18.5.: Ermitteln Sie mit Papier und Bleistift, was das Programm **LESIB_04** zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt. **o**

Die **Sichtbarkeitsregeln** für eine Vereinbarung sind unterschiedlich, je nachdem ob eine **aufrufbare** Größe vereinbart wird oder eine **nicht-aufrufbare** Größe. **Prozeduren** und **Funktionen** sind aufrufbare Größen, **Typen**, **Konstanten**, **Variablen**, **Ausnahmen** und **Blöcke** sind nicht-aufrufbare Größen.

Jede aufrufbare Größe hat ein **Profil**. Zum Profil eines **Unterprogramms** gehören

1. der **Name** des Unterprogramms
2. die **Anzahl** seiner formalen Parameter
3. die **Typen** der einzelnen Parameter (in der vom Programmierer festgelegten Reihenfolge)
4. und bei Funktionen der Typ des Funktionsergebnisses (der **Ergebnistyp**, the return type).

Nicht zum Profil eines Unterprogramms gehören die **Namen der formalen Parameter**, ihre **Modi** (**in**, **out** bzw. **in out**) und ihre **Vorbesetzungen**. Das **Profil** eines Unterprogramms ist also "ein bißchen abstrakter" als die **Spezifikation** des Unterprogramms. Unterprogramme mit **unterschiedlichen** Spezifikationen können das **gleiche** Profil haben, dagegen folgt aus der Gleichheit zweier Spezifikationen die Gleichheit der entsprechenden Profile.

Beispiel 18.5.: Unterprogramme mit gleichen und unterschiedlichen Profilen:

```
01 declare
02    procedure OTTO;
03    procedure EMIL;
04    procedure EMIL(C:        character);
05    procedure EMIL(X: in out character);
06    procedure EMIL(C: character; B: boolean);
07    procedure EMIL(C: boolean;   B: character);
08    procedure EMIL(X: integer;   Y: positive);
09    procedure EMIL(A, B: natural);
10    function  EMIL(A, B: natural) return boolean;
```

Da das **Profil** eines Unterprogramms vollständig durch seine Spezifikation bestimmt ist, wurden in diesem Beispiel nur Spezifikationen (und keine Rümpfe) von Unterprogrammen angegeben. Mit "**das Profil in Zeile 03**" ist im folgenden das Profil des Unterprogramms gemeint, dessen Spezifikation in Zeile 03 steht.

Die Profile in Zeile 04 und 05 sind **gleich**, weil die Spezifikationen sich nur durch den Namen und den Modus des (einzigen) Parameters unterscheiden. Die Profile in Zeile 08 und 09 sind gleich, weil **integer**, **positive** und **natural** Untertypen desselben **Typs !integer** sind (unterschiedliche **Untertypen** reichen nicht aus, um zwei Profile verschieden zu machen).

Alle anderen Profile sind paarweise **verschieden**. Insbesondere sind die Profile in Zeile 06 und 07 verschieden, weil die **Reihenfolge** der Parametertypen wichtig ist ("!character, !boolean" ist nicht gleich "!boolean, !character"). Das Profil einer **Funktion** ist grundsätzlich verschieden vom Profil einer **Prozedur**, weil eine Prozedur **keinen Ergebnistyp** hat wie ein Funktion. **o**

Innerhalb eines **Vereinbarungsbereiches** darf man höchstens **eine nicht-aufrufbare Größe** mit einem bestimmten Namen (z.B. OTTO) vereinbaren. Wenn man in verschiedenen, aber **geschachtelten** Vereinbarungsbereichen zwei nicht-aufrufbare Größen mit dem gleichen Namen vereinbart, dann **verdeckt** die weiter innen liegende Vereinbarung in ihrem Gültigkeitsbereich die weiter außen liegende Vereinbarung.

Innerhalb eines **Vereinbarungsbereichs** darf man **beliebig viele aufrufbare Größen** mit einem bestimmten Namen (z.B. EPROC_01) vereinbaren, solange diese Größen sich durch ihre **Profile** voneinander unterscheiden. Wenn man in verschiedenen, aber **geschachtelten** Vereinbarungsbereichen zwei aufrufbare Größen mit gleichen Namen vereinbart, dann **verdeckt** die weiter innen liegende Vereinbarung in ihrem Gültikeitsbereich die weiter außen liegende Vereinbarung **nur**, wenn die beiden aufrufbaren Größen das **gleiche Profil** haben.

Wenn an einer Stelle eines Programms mehrere aufrufbare Größen mit **gleichen Namen** und **verschiedenen Profilen** sichtbar sind, dann sagt man auch, daß der Name dieser Größen (mit mehreren Bedeutungen) **überladen** ist. Im folgenden Beispiel wird der Name **EPROC_01** mehrfach überladen:

Innerhalb eines Vereinbarungsbereichs darf man **nicht** eine **aufrufbare** Größe und eine **nicht-aufrufbare** Größe mit gleichem Namen vereinbaren. Z.B. schließt ein **Typ** namens **OTTO** eine Prozedur namens **OTTO** aus und eine Funktion namens **EMIL** verträgt sich nicht mit einer Variablen namens **EMIL** etc..

Beispiel 18.6.: Ein Prozedurname mit drei Bedeutungen:

```
01 with ada.text_io;
02 procedure LESIB_05 is
03    procedure EPROC_01(ZAHL: integer) is
04       TEXT: constant string := integer'image(ZAHL);
05    begin
06       ada.text_io.put_line(item => TEXT);
07    end EPROC_01;
08    procedure EPROC_01(ZAHL: integer; TXT: string) is
09       TEXT: constant string := integer'image(ZAHL) & TXT;
10    begin
11       ada.text_io.put_line(item => TEXT);
12    end EPROC_01;
13    procedure EPROC_01(TXT: string; ZAHL: integer) is
14       TEXT: constant string := TXT & integer'image(ZAHL);
15    begin
16       ada.text_io.put_line(item => TEXT);
17    end EPROC_01;
18 begin -- LESIB_05
19    EPROC_01(ZAHL => 123);
20    EPROC_01(35, " DM");
21    EPROC_01("ECU", 17);
22 end LESIB_05;
```

Mit **EPROC_01-03** ist im folgenden **das** Unterprogramm namens **EPROC_01** gemeint, dessen Vereinbarung in Zeile **03** beginnt. Entsprechend bezeichnet **EPROC_01-08** das Unterprogramm **EPROC_01**, dessen Vereinbarung in Zeile 08 beginnt etc..

Die Unterprogramme **EPROC_01-03**, **EPROC_01-08** und **EPROC_01-13** unterscheiden sich durch ihre **Profile** und dürfen deshalb innerhalb **eines** Vereinbarungsbereichs vereinbart werden. In den Zeilen 19 bis 21 bezeichnet der Name **EPROC_01** jedesmal eine andere Prozedur, aber der Ausführer kann bei jedem Aufruf eindeutig erkennen, welche der drei Prozeduren gemeint ist.

Ein Prozeduraufruf wie etwa

```
21a   EPROC_01(ZAHL => 35, TXT => " DM");
```

wäre **nicht** erlaubt, weil der Ausführer nicht erkennen könnte, ob die Prozedur **EPROC_01-08** oder **EPROC_01-13** gemeint ist.

Aufgabe 18.6.: Ermitteln Sie mit Papier und Bleistift, was das Programm **LESIB_05** zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt. **o**

Wenn an einer Stelle eines Programms mehrere verschiedene Vereinbarungsbereiche mit gleichem Namen (z.B. LESIB_06) sichtbar sind, dann kann es passieren, daß auch ein zusammengesetzter Name wie **LESIB_06.OTTO** nicht eindeutig **eine** Größe namens **OTTO** bezeichnet. In solchen Fällen hilft die folgende Regel:

Der Ausführer hat einen **Vereinbarungsbereich** namens **standard**, in dem alle vordefinierten Größen vereinbart sind, z.B. die Untertypen **boolean**, **character**, **string**, **integer**, **positive**, **natural** etc. und die Ausnahmen **constraint_error**, **storage_error** etc.. Wenn der Programmierer dem Ausführer die Vereinbarung einer Bibliothekseinheit **BE** übergibt, dann erzeugt der Ausführer die Einheit **BE** am Ende des Vereinbarungsbereichs **standard**. Damit sind in **BE** alle vordefinierten Größen sichtbar. Außerdem kann man die Einheit **BE** "in Notfällen" auch mit dem zusammengesetzten Namen **standard.BE** bezeichnen, wie das folgende Beispiel zeigt.

Beispiel 18.7.: Geschachtelte Vereinbarungsbereiche mit **gleichen Namen** und der Vereinbarungsbereich **standard**:

```
01 with ada.text_io;
02 procedure LESIB_06 is
03     OTTO: character := 'A';
04     procedure LESIB_06 is
05         OTTO: character := 'B';
06         procedure LESIB_06 is
07             OTTO: character := 'C';
08         begin
09             ada.text_io.put(item => OTTO);
10             ada.text_io.put(item => standard.LESIB_06.LESIB_06.OTTO);
11             ada.text_io.put(item => standard.LESIB_06.OTTO);
12         end LESIB_06;
13     begin
14         LESIB_06;
15         ada.text_io.put(item => OTTO);
16         ada.text_io.put(item => standard.LESIB_06.OTTO);
17     end LESIB_06;
18 begin
19     ada.text_io.put(item => OTTO);
20     LESIB_06;
21 end LESIB_06;
```

Der zusammengesetzte Name **standard.LESIB_06.OTTO** in Zeile 11 und in Zeile 16 bezeichnet die Größe namens **OTTO**, die in dem Vereinbarungsbereich namens LESIB_06 vereinbart ist, der im Vereinbarungsbereich **standard** vereinbart ist. Offenbar kann es sich dabei nur um die Variable **OTTO-03** handeln.

Ganz entsprechend bezeichnet der Name **standard.LESIB_06.LESIB_06.OTTO** in Zeile 10 eindeutig die Variable **OTTO-05**.

In den Zeilen 09 bis 11 ist ein zusammengesetzter Name wie **LESIB_06.LESIB_06** nicht erlaubt, weil an dieser Stelle **kein** Vereinbarungsbereich namens **LESIB_06** direkt sichtbar ist, in dem ein Vereinbarungsbereich namens **LESIB_06** vereinbart ist. Nur mit Hilfe des Vereinbarungsbereichs **standard** kann man an diesen Stellen des Programms **LESIB_06** auf die Variablen **OTTO-03** und **OTTO-05** zugreifen.

Aufgabe 18.7.: Geben Sie für jede der drei Prozeduren **LESIB_06-02**, **LESIB_06-04** und **LESIB_06-06** den **Gültigkeitsbereich** an. Beschreiben Sie für jede der drei Prozeduren, an welchen Stellen sie **direkt sichtbar** ist, d.h. in welchen Zeilen des Programms sie mit dem einfachen Namen **LESIB_06** bezeichnet werden kann. Beschreiben Sie für jede der drei Variablen **OTTO-03**, **OTTO-05** und **OTTO-07**, an welchen Stellen sie direkt sichtbar ist.

Aufgabe 18.8.: Welche Prozedur wird durch den einfachen Namen **LESIB_06** in Zeile **20** bezeichnet? **LESIB_06-02**, **LESIB_06-04** oder **LESIB_06-06**? Ebenso für Zeile **14**.

Aufgabe 18.9.: Ermitteln Sie mit Papier und Bleistift, was das Programm **LESIB_06** zur aktuellen Ausgabe (d.h. zum Bildschirm) ausgibt. **o**

Ein wichtiges Ziel des Programmierers ist es, die **Sichtbarkeitsbereiche** seiner Größen, insbesondere die Sichtbarkeitsbereiche seiner **Variablen**, möglichst **klein** zu gestalten. Denn an **den** Stellen, an denen eine Variable **nicht** sichtbar ist, kann sie auch nicht fälschlich verändert werden. Mit "engen Sichtbarkeitsbereichen" kann man also Fehlermöglichkeiten verringern.

Eine Größe, die in einem Vereinbarungsbereich VB **vereinbart** wurde, wird auch als **lokale Größe des Bereichs VB** bezeichnet. Eine Größe, die in VB **sichtbar** ist, aber **außerhalb** von VB **vereinbart** wurde, wird relativ zu VB auch als **globale Größe** bezeichnet. Haufig ist eine Größe **lokal** zu einem Vereinbarungsbereich VB und gleichzeitig **global** zu mehreren Vereinbarungsbereichen VB1, VB2, ... etc., die in VB geschachtelt sind.

Grundsätzlich sollte man alle Größen in einem Programm, insbesondere aber alle Variablen, "so lokal wie möglich" und nur "so global wie nötig" vereinbaren. Das folgende Beispiel soll andeuten, wie man es **nicht** machen soll:

Beispiel 18.8.: Unnötig globale Variablen: Es sei angenommen, daß in diesem Beispiel die Variable **OTTO** nur in der Prozedur **EPROC_01** benützt wird und die Variable **EMIL** nur in der Funktion **EFUNK_01**.

```
01 procedure LESIB_07 is
02    OTTO: character;
03    EMIL: integer;
04    procedure EPROC_01 is
05       ...
06    begin
07       OTTO := 'A';
08       ...
09    end EPROC_01;
10    function EFUNK_01(N: integer) return character is
11       ...
12    begin
13       EMIL := N+1;
14       ...
15    end EFUNK_01;
16 begin -- LESIB_07
17    ...
18 end LESIB_07;
```

Relativ zur Prozedur **LESIB_07** sind **OTTO** und **EMIL** offenbar **lokale** Variablen. Relativ zu den Unterprogrammen **EPROC_01** und **EFUNK_01** sind **OTTO** und **EMIL** dagegen **globale** Variablen.

In diesem Beispiel ist die Variable **OTTO** nicht nur im Unterprogramm **EPROC_01** sichtbar, wo sie gebraucht wird, sondern auch im Unterprogramm **EFUNK_01**, wo sie **nicht** gebraucht wird. Damit ist nicht ausgeschlossen, daß **OTTO** innerhalb von **EFUNK_01** aus Versehen verändert oder sonstwie "mißbraucht" wird. Wenn man die Variable **OTTO** stattdessen lokal in **EPROC_01** vereinbart (in Zeile 05), dann schränkt man damit ihren Sichtbarkeitsbereich ein und schließt bestimmte Flüchtigkeitsfehler mit Sicherheit aus. Allerdings lebt **OTTO** dann jeweils nur während die Prozedur **EPROC_01** ausgeführt wird und kann nicht dazu benützt werden, einen Wert zwischen zwei Ausführungen aufzubewahren. Ganz entsprechendes gilt auch für die Variable **EMIL** und die Funktion **EFUNK_01**. **o**

Im Gegensatz zu **Variablen** können **Typen**, **Konstanten**, **Ausnahmen** und **Unterprogramme** weder aus Versehen noch absichtlich **verändert** werden. Aber auch solche unveränderbaren Größen können aus Versehen **falsch benützt** werden: Eine **Ausnahme** kann man fälschlich **auslösen**, ein **Unterprogramm** kann man aus Versehen **aufrufen** (z.B. weil man es mit einem anderen Unterprogramm mit "ähnlichem Namen" verwechselt) etc.. Deshalb sollte man auch bei unveränderbaren Größen die Sichtbarkeitsbereiche **nicht unnötig groß** gestalten.

Indem der Programmierer eine Größe G **lokal** in einem Unterprogramm UP vereinbart, "beweist er" seinen Kollegen auf einfache aber wirkungsvolle Weise, daß G **außerhalb** von UP weder verändert noch sonstwie benützt wird. Insbesondere bei der Fehlersuche in sehr großen Programmen sind solche "Beweise, daß bestimmte Aktionen **nicht möglich** sind" für die Leser eines Programms von großem Nutzen. Erfahrene Programmierer gestalten die Sichtbarkeitsbereiche ihrer Vereinbarungen ganz bewußt und möglichst klein.

Zusammenfassung 18:

- Jeder **Block** und jede **Vereinbarung eines Unterprogramms** bildet einen **Vereinbarungsbereich**, in dem andere Größen (z.B. **Typen**, **Konstanten**, **Variablen**, **Ausnahmen** und **Unterprogramme**) vereinbart werden können.
- Eine **vereinbarte Größe** wird vom Ausführer **erzeugt** und wieder **zerstört**. Zwischen diesen beiden Zeitpunkten **lebt** sie.
- Die Vereinbarung einer Größe (und damit auch die vereinbarte Größe) hat einen bestimmten **Gültigkeitsbereich** (scope). **Außerhalb** dieses Gültigkeitsbereichs kann man auf keinen Fall auf die Größe zugreifen.
- Prozeduren und Funktionen sind **aufrufbare Größen**. Typen, Konstanten, Variablen und Ausnahmen sind **nicht-aufrufbare Größen**.
- Innerhalb eines Vereinbarungsbereichs darf man höchstens **eine nicht-aufrufbare Größe** mit einem bestimmten Namen (z.B. **OTTO**) vereinbaren, oder **beliebig viele aufrufbare Größen**, die sich aber durch ihre **Profile** unterscheiden müssen.
- Die Vereinbarung einer Größe (und damit auch die vereinbarte Größe) ist an bestimmten Stellen ihres Gültigkeitsbereichs **direkt sichtbar**. An diesen Stellen kann die Größe mit ihrem **einfachen Namen** bezeichnet werden.
- In bestimmten Fällen wird die Vereinbarung einer Größe (und damit auch die vereinbarte Größe) an bestimmten Stellen ihres Gültigkeitsbereichs von einer anderen Vereinbarung **verdeckt** und ist damit an diesen Stellen **nicht** direkt sichtbar.
- In bestimmten Fällen ist eine **verdeckte** Größe an bestimmten Stellen ihres Gültigkeitsbereichs zumindest noch **indirekt sichtbar** und kann an diesen Stellen mit einem **zusammengesetzten Namen** ("in Punktnotation") wie **A.B.C.OTTO** oder **standard.A.B.C.OTTO** bezeichnet werden. Dabei sind **standard**, **A**, **B** und **C** Vereinbarungsbereiche, die ineinander geschachtelt sind.
- In **geschachtelten Vereinbarungsbereichen** gilt: Eine **nicht-aufrufbare** Größe verdeckt alle gleichnamigen, "weiter außen" vereinbarten Größen. Eine **aufrufbare** Größe verdeckt alle gleichnamigen, "weiter außen" vereinbarten **nicht-aufrufbaren** Größen. Eine **aufrufbare** Größe verdeckt nur **die** "weiter außen" vereinbarten **aufrufbaren** Größen, die das gleiche **Profil** haben.
- Im Idealfall sollte eine Größe nur **solange leben**, wie sie wirklich **gebraucht** wird.
- Im Idealfall sollte eine Größe nur genau an den Stellen eines Programms **sichtbar** sein, an denen sie **gebraucht** wird ("so lokal wie möglich, so global wie nötig").

19. Pakete

Wenn ein Programm nur aus **Unterprogrammen** und **Blöcken** besteht, hängt die **Lebensdauer** der darin vereinbarten Größen eng mit der **Sichtbarkeit** ihrer Vereinbarungen zusammen. Ein Programm besteht im allgemeinen aus mehreren Vereinbarungsbereichen, die zum Teil "nebeneinander" liegen und zum Teil ineinander **geschachtelt** sind. Je weiter "außen" eine Größe vereinbart wird, desto **größer** ist im allgemeinen ihr Sichtbarkeitsbereich und desto **länger** lebt sie. Die "am weitesten innen" vereinbarten Größen haben die **kleinsten** Sichtbarkeitsbereiche ("small is beautiful") und leben am **kürzesten**.

Allein mit Unterprogrammen und Blöcken kann der Programmierer die Sichtbarkeit und die Lebensdauer seiner Typen, Variablen, Unterprogramme etc. nur **unzureichend** kontrollieren. Wenn er z.B. erreichen will, daß eine Variable **V** zwischen zwei Ausführungen eines Unterprogramms **U** am Leben bleibt und ihren Wert behält, dann kann er V nicht **innerhalb** von **U** als U-**lokale** Variable vereinbaren, sondern muß sie **außerhalb** von **U** als U-**globale** Variable plazieren. Dadurch wird die Variable **V** im allgemeinen aber auch an solchen Stellen sichtbar, an denen sie nicht gebraucht wird und besser unsichtbar wäre.

Ein verwandtes Problem liegt vor, wenn man drei Unterprogramme **U1**, **U2** und **U3** hat und drei Variable **V12**, **V23** und **V31**, von denen jede in genau zwei Unterprogrammen gebraucht wird: **V12** in **U1** und **U2**, **V23** in **U2** und **U3** und **V31** in **U3** und **U1**. Egal, wo und in welcher Reihenfolge man die Variablen und die Unterprogramme vereinbart, sind die Variablen immer an zu viel Stellen oder an zu wenig Stellen sichtbar.

Ein anderes Problem taucht auf, wenn der Kollege-2 bestimmte Programmteile in anderen Programmen **wiederverwenden** will. Häufig gehören mehrere Unterprogramme, Variable und andere Größen "inhaltlich eng zusammen", und können nur **gemeinsam** wiederverwendet werden. Als Beispiel sei angenommen, daß 5 Typen, 17 Unterprogramme, 23 Variablen und 3 Ausnahmen gegenseitig voneinander abhängen und wiederverwendet werden sollen. Wenn der Kollege-2 diese Größen in einem System von hunderten oder tausenden von Größen finden und kopieren muß, dann können ihm dabei leicht Fehler unterlaufen. Vergißt er eine Größe, dann macht ihn wahrscheinlich der Ausführer irgendwann auf seinen Fehler aufmerksam, aber wenn er ein paar nicht benötigte Größen mitkopiert, merkt er das vielleicht nie.

Um ein Teilsystem eines großen Systems **wiederverwendbar** zu machen, ist es wünschenswert bis notwendig, die einzelnen Größen des Teilsystems von Anfang an von anderen Größen **abzutrennen** und **zu einer Einheit zusammenzufassen**. Eine solche Einheit sollte man z.B. mit **einem** einfachen Befehl in ein anderes Programm einbauen können.

Um solche und noch ein paar weitere Probleme lösen zu können, hat man das Konzept eines **Moduls** entwickelt und in verschiedene Programmiersprachen aufgenommen. In Ada werden Module als **Pakete** (packages) bezeichnet, um sie von anderen Varianten des gleichen Konzepts (und von ganz anderen Bedeutungen des Wortes "Modul") zu unterscheiden.

19.1. Abstrakte Variablen

Pakete sind sehr **vielseitige** Werkzeuge, mit denen der Programmierer zahlreiche und sehr **verschiedene** Probleme lösen kann. In diesem ersten Abschnitt über Pakete soll gezeigt werden, wie man mit Hilfe eines Paketes Variablen "vor Mißbrauch" schützen und die Fehlersuche in einem Programm erheblich vereinfachen kann. Derart geschützte Variable werden auch als **abstrakte Variable** bezeichnet.

Als Beispiel wird gezeigt, wie man einen sogenannten **Stapel** (stack) in Ada realisieren kann. Ein Stapel ist ein Speicher, in dem man **Elemente** ablegen kann. Dabei darf man weitere Elemente nur **oben** auf den Stapel legen, und wenn man abgelegte Elemente wieder zurückholen will, darf man jeweils nur das **oberste** Element des Stapels ansehen oder entfernen (etwa so, wie bei einem Stapel von Spielkarten). Es ist ausdrücklich **verboten**, Elemente weiter **unten** aus dem Stapel herauszuziehen oder neue Elemente **unter** andere Elemente des Stapels zu schieben.

Legt man drei Elemente A, B und C in dieser Reihenfolge auf einen Stapel (zuerst A, zuletzt C), und holt sie dann wieder zurück, bekommt man sie "in umgekehrter Reihenfolge" (zuerst C, zuletzt A). Mit Hilfe eines Stapels kann man also die Reihenfolge von Elementen "umkehren" und erstaunlich viele nicht-triviale Probleme lösen (z.B. prüfen, ob in einem Text alle **Klammern** "richtig gepaart" sind, siehe unten Aufgabe 19.1.3).

Die minimale Ausstattung eines Stapels besteht aus **drei** Unterprogrammen, die traditionsgemäß **PUSH**, **TOP** und **POP** heißen. Mit **PUSH** kann man ein weiteres Element oben auf den Stapel legen, **TOP** liefert eine Kopie des obersten Elements und **POP** entfernt das oberste Element vom Stapel.

Das folgende Beispiel zeigt ein Paket namens **PAKET_01**, welches einen Stapel realisiert. Auf diesen Stapel kann man einzelne Zeichen vom Untertyp **character** ablegen. Der Stapel bietet Platz für maximal 100 Zeichen.

Beispiel 19.1.1.: Ein Stapel als abstrakte Variable realisiert:

```
01 package PAKET_01 is
02 -- Dient zum Verwalten eines Stapels von Zeichen (vom Typ character).
03 -- Der Stapel selbst ist im Rumpf des Paketes verborgen.
04 ------------------------------------------------------------------------
05    procedure PUSH(ZEICHEN: character);
06    -- Wenn der Stapel voll ist, wir die Ausnahme UEBERLAUF ausgeloest.
07    -- Sonst wird das ZEICHEN auf den Stapel gelegt.
08    ------------------------------------------------------------------------
09    procedure POP;
10    -- Wenn der Stapel leer ist, wird die Ausnahme UNTERLAUF ausgeloest.
11    -- Sonst wird das oberste Element vom Stapel entfernt.
12    ------------------------------------------------------------------------
13    function TOP return character;
14    -- Wenn der Stapel leer ist, wird die Ausnahme UNTERLAUF ausgeloest.
15    -- Sonst wird eine Kopie des obersten Stapelelements als Ergebnis
16    -- geliefert. Der Stapel bleibt unveraendert.
17    ------------------------------------------------------------------------
18    function ANZAHL return natural;
19    -- Liefert die Anzahl der Elemente, die sich momentan auf dem
20    -- Stapel befinden.
21    ------------------------------------------------------------------------
22    UEBERLAUF: exception; -- Wird evtl. von PUSH ausgeloest.
23    UNTERLAUF: exception; -- Wird evtl. von POP oder TOP ausgeloest.
24 end PAKET_01;
```

```
25 package body PAKET_01 is
26    STAPEL: string(1..100);
27    LBI   : natural := STAPEL'first-1; -- Letzter belegter Index des STAPELs.
28    -- Zeigt "vor den STAPEL", wenn kein Index belegt ist.
29    ------------------------------------------------------------------------
30    procedure PUSH(ZEICHEN: character) is
31    begin
32       if LBI = STAPEL'last then
33          raise UEBERLAUF;
34       else
35          LBI         := LBI + 1;
36          STAPEL(LBI) := ZEICHEN;
37       end if;
38    end PUSH;
39    ------------------------------------------------------------------------
40    procedure POP is
41    begin
42       if LBI < STAPEL'first then
43          raise UNTERLAUF;
44       else
45          LBI := LBI - 1;
46       end if;
47    end POP;
48    ------------------------------------------------------------------------
49    function TOP return character is
50    begin
51       if LBI < STAPEL'first then
52          raise UNTERLAUF;
53       else
54          return STAPEL(LBI);
55       end if;
56    end TOP;
57    ------------------------------------------------------------------------
58    function ANZAHL return natural is
59    begin
60       return LBI;
61    end ANZAHL;
62 end PAKET_01;
```

Das **PAKET_01** besteht aus **zwei** Teilen, der **Spezifikation** in Zeile 01 bis 24 und dem **Rumpf** in Zeile 25 bis 62. Diese beiden Teile kann man dem Ausführer **nacheinander** übergeben (erst die Spezifikation, später den Rumpf).

In der **Spezifikation** des Paketes werden insgesamt **sechs** Größen vereinbart, vier Unterprogramme namens **PUSH**, **POP**, **TOP** und **ANZAHL** und zwei Ausnahmen namens **UEBERLAUF** und **UNTERLAUF**. Die Unterprogramme werden allerdings nicht vollständig vereinbart. Die Spezifikation des Paketes enthält nur die **Spezifikationen** der Unterprogramme (in den Zeilen 05, 09, 13 und 18).

Im **Rumpf** des Paketes werden zwei Variablen namens **STAPEL** und **LBI** vereinbart. Außerdem enthält der Paketrumpf die **Rümpfe** der vier Unterprogramme, die in der Paketspezifikation spezifiziert wurden.

Die Sichtbarkeitsregeln für Pakete legen fest, daß die Variablen **STAPEL** und **LBI** nur **innerhalb** des Paketrumpfes **sichtbar** sind. Sollte der Programmierer beim Testen des Paketes feststellen, daß irgendwelche fehlerhaften Anweisungen falsche Werte in diese Variablen gebracht haben, dann kann er sicher sein, daß diese fehlerhaften Anweisungen sich **im Paketrumpf** befinden müssen. Diese Tatsache kann die Fehlersuche erheblich vereinfachen.

Indem der Programmierer die Variablen **STAPEL** und **LBI** im Rumpf eines Paketes vereinbart, "beweist er" sich und seinen Kollegen, daß durch Befehle außerhalb dieses Rumpfes nicht auf die Variablen zugegriffen werden kann. Der Paketrumpf macht hier also nichts möglich, was nicht auch

ohne Paket möglich wäre. Vielmehr garantiert er, daß bestimmte unerwünschte Aktionen **nicht möglich** sind. Das ist der tiefere Sinn von Paketen.

Das **PAKET_01** selbst und die in seiner Spezifikation vereinbarten Größen (die vier Unterprogramme und die zwei Ausnahmen) sind zunächst auch nur **innerhalb** des Paketes sichtbar. Man kann das Paket aber in beliebigen Bibliothekseinheiten **direkt sichtbar** machen, indem man vor diese Einheiten eine Kontextklausel **with PAKET_01;** schreibt. Und da, wo das Paket **PAKET_01** direkt sichtbar ist, sind die in seiner Spezifikation vereinbarten Größen zumindest **indirekt sichtbar** und können dort mit zusammengesetzten Namen wie **PAKET_01.PUSH**, **PAKET_01.POP**, ..., **PAKE_01.UEBERLAUF**, ... etc. bezeichnet werden.

Das folgende Beispiel zeigt eine kleine Prozedur, die das **PAKET_01** mit einer Kontextklausel einbindet und Größen aus dem Paket benutzt, d.h. die im Paket vereinbarten **Unterprogramme** aufruft. Die im Paket vereinbarten **Ausnahmen** kommen in diesem einfachem Beispiel **nicht** zum tragen.

Beispiel 19.1.2.: Ein kleines Programm, in dem das **PAKET_01** benutzt wird:

```
01 with PAKET_01, ada.text_io;
02 procedure TESTP_01 is
03 -- Minimales Testprogramm zum Testen des Pakets PAKET_01.
04    TEXT1: string := "?se theg eiw ,ollaH";
05    TEXT2: string(1..TEXT1'length);
06 begin
07    for I in positive range TEXT1'range loop
08       PAKET_01.PUSH(ZEICHEN => TEXT1(I));
09    end loop;
10    for I in positive range 1..PAKET_01.ANZAHL loop
11       TEXT2(I) := PAKET_01.TOP;
12       PAKET_01.POP;
13    end loop;
14    ada.text_io.put_line(item => TEXT2);
15 end TESTP_01;
```

Das Programm **TESTP_01** legt die Zeichen des Strings **TEXT1** (vereinbart in Zeile 04) auf den Stapel im **PAKET_01** (siehe Zeile 08), holt sie dann wieder zurück in den String **TEXT2** (siehe Zeile 11 und 12) und gibt den Wert von **TEXT2** zur aktuellen Ausgabe aus (in Zeile 14). Bei der Ausgabe stehen in **TEXT2** die gleichen Zeichen wie in **TEXT1**, aber in umgekehrter Reihenfolge. **o**

Aufgabe 19.1.1.: Schreiben Sie ein Programm namens **BEN1P_01** ("Benutzer 1 von **PAKET_01**"), welches das **PAKET_01** einbindet und benützt. Das Programm **BEN1P_01** soll mit der Prozedur **ada.text_io.get_line** eine beliebig lange Zeichenkette einlesen (maximal 100 Zeichen) und deren Zeichen in umgekehrter Reihenfolge wieder ausgeben. Wenn z.B. "ABC" eingelesen wird, soll also "CBA" wieder ausgegeben werden. **o**

Aufgabe 19.1.2.: Schreiben Sie ein Programm namens **BEN2P_01**, welches genau das Gleiche leistet, wie das Programm **BEN1P_01** (siehe vorige Aufgabe), aber "interessanter strukturiert" ist. Das Programm **BEN2P_01** soll aus **zwei** Prozeduren namens **BEN2P_01** und **BEN3P_01** bestehen, die beide das **PAKET_01** einbinden. Die Hauptprozedur **BEN2P_01** soll eine Zeichenkette einlesen, ihre Zeichen auf den Stapel in **PAKET_01** legen und dann **BEN3P_01** aufrufen. Die Prozedur **BEN3P_01** soll alle Zeichen vom Stapel in **PAKET_01** herunterholen und ausgeben. **o**

Aufgabe 19.1.3.: Schreiben Sie ein Programm namens **BEN4P_01**, welches eine beliebig lange Zeichenkette einliest (mit **get_line**, maximal 100 Zeichen), prüft, ob darin alle runden, eckigen und geschweiften Klammern **richtig gepaart** sind und eine entsprechende Meldung ausgibt. In jeder der folgenden Zeichenketten sind die Klammern **nicht** richtig gepaart: **"A(B"**, **"(A)B)"**, **"}ABC{", "(A[B)C]".** Zeichen, die keine Klammern sind, haben keinen Einfluß auf die Korrektheit oder Fehlerhaftigkeit einer Zeichenkette und sollen einfach "überlesen" werden. Möglicherweise ist ein **Stapel** zur Lösung dieser Aufgabe hilfreich. Statt selbst einen Stapel zu programmieren, dürfen Sie das **PAKET_01** einbinden und benutzen. **o**

Allgemein gilt für Pakete:

Ein **Paket** besteht normalerweise aus einer **Spezifikation** (specification) und einem **Rumpf** (body). Die Spezifikation **muß** vorhanden sein. Der Rumpf **kann** fehlen, wenn in der Spezifikation nur rumpflose Größen vereinbart werden wie Typen, Konstanten, Variablen und Ausnahmen.

Zum Vergleich: Auch ein **Unterprogramm** besteht aus einer **Spezifikation** und einem **Rumpf**. Aber bei einem Unterprogramm muß der **Rumpf** vorhanden sein und die **Spezifikation** kann fehlen.

In der **Spezifikation eines Paketes** dürfen **beliebige Vereinbarungen** von **rumpflosen** Größen wie Typen, Konstanten, Variablen und Ausnahmen stehen. Außerdem dürfen dort **Spezifikationen** von Programmeinheiten (von Unterprogrammen und Paketen etc.) stehen, aber **keine Rümpfe**.

Im **Rumpf eines Paketes** dürfen **beliebige Vereinbarungen** stehen, auch Spezifikationen und Rümpfe. Wenn in der Spezifikation eines Paketes **P** eine Spezifikation steht, dann **muß** im Rumpf von **P** der zugehörige Rumpf stehen. Z.B. stehen in der Spezifikation des Paketes **PAKET_01** die Spezifikationen der vier Unterprogramme **PUSH**, **POP**, **TOP** und **ANZAHL**. Deshalb müssen die Rümpfe dieser vier Unterprogramme im Rumpf des Paketes **PAKET_01** stehen.

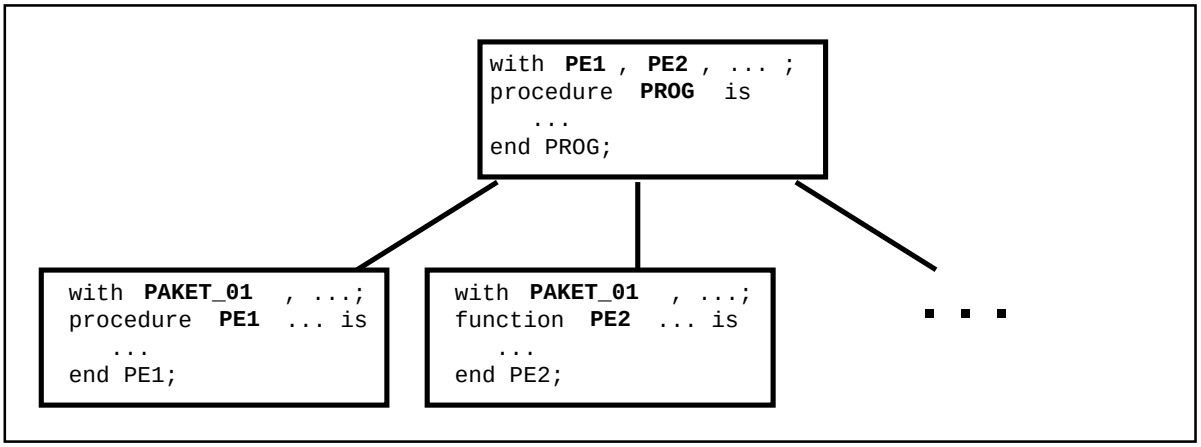
Die Spezifikation und der Rumpf eines Paketes bilden gemeinsam einen **Vereinbarungsbereich**. Wenn man in diesem Vereinbarungsbereich eine Größe vereinbart, dann beginnt der **Gültigkeitsbereich** dieser Vereinbarung "wie immer" (bei nicht-aufrufbaren Größen am Anfang der Vereinbarung, bei aufrufbaren Größen nach ihrer Spezifikation bzw. nach dem Wort **is** am Anfang ihres Rumpfes) und reicht bis zum **Ende des Paketrumpfes**. Z.B. reicht der Gültigkeitsbereich der Prozedur **PUSH** von Zeile 06 bis 62, der Gültigkeitsbereich der Ausnahme **UNTERLAUF** von Zeile 23 bis 62 und der Gültigkeitsbereich der Variablen **LBI** von Zeile 27 bis 62. Dies gilt auch dann, wenn die Paketspezifikation und der Paketrumpf in **zwei verschiedenen Dateien** stehen und dem Ausführer **getrennt** voneinander übergeben werden.

Eine Vereinbarung ist innerhalb ihres Gültigkeitsbereichs **sichtbar**, wenn sie nicht von einer Vereinbarung mit dem gleichen Namen **verdeckt** wird. Außerdem sind die in der Spezifikation eines Paketes **P** vereinbarten Größen **indirekt sichtbar** in allen Programmeinheiten, vor denen eine Kontextklausel **with P;** steht. Z.B. sind **PUSH**, **POP**, **TOP**, **ANZAHL**, **UNTERLAUF** und **UEBERLAUF** in der Prozedur **TESTP_01** indirekt sichtbar.

Es gibt **Bibliothekspakete** (deren Spezifikationen und Rümpfe dem Ausführer als selbständige Einheiten übergeben werden) und **enthaltene Pakete** (deren Vereinbarungen in einer anderen Programmeinheit enthalten sind). Das **PAKET_01** ist ein Bibliothekspaket.

Die **Lebensdauerregeln** für **Bibliothekspakete** sind ziemlich einfach. Die in einem Bibliothekspaket vereinbarten Größen werden jeweils zu **Beginn einer Programmausführung** erzeugt und **leben** bis zum **Ende der Programmausführung**.

Angenommen, das Bibliothekspaket **PAKET_01** wird von **mehreren** Programmeinheiten **PE1**, **PE2** ... etc. eingebunden (mit Kontextklauseln **with PAKET_01;**) und diese Programmeinheiten **PE1**, **PE2** ... etc. gehören gemeinsam zu einem Programm **PROG**, etwa so:



Wieviele Exemplare des Paketes **PAKET_01** werden dann in das Programm **PROG** eingebunden? Mit anderen Worten: Gibt es in dieser Situation für **PE1** einen Stapel und für **PE2** einen **zweiten** Stapel, oder beziehen sich die **PUSH**- und **POP**-Befehle in **PE1** auf denselben Stapel wie die **PUSH**- und **POP**-Befehle in **PE2**?

Antwort: In das Programm **PROG** wird nur **ein** Exemplar von **PAKET_01** eingebunden, unabhängig davon, wie viele Kontextklauseln **with PAKET_01;** vor Programmeinheiten des Programms **PROG** stehen. Alle **PUSH**- und **POP**-Befehle im ganzen Programm **PROG** beziehen sich damit auf **denselben** Stapel, und nicht auf mehrere "gleiche" Stapel. Siehe dazu auch oben die Aufgabe 19.1.2.

Wenn das **PAKET_01** dagegen in mehrere verschiedene Programme **PROG1**, **PROG2** ... etc. eingebunden wird, dann bekommt jedes Programm ein **eigenes Exemplar** des Paketes und damit **seinen eigenen Stapel**.

Aufgabe 19.1.4.: Vereinbaren Sie **drei Variablen** namens V12, V23 und V31 und **drei Bibliotheksprozeduren** namens U1, U2 und U3 und richten Sie es so ein, daß jede Variable in genau zwei Unterprogrammen **sichtbar** ist: V12 in U1 und U2, V23 in U2 und U3 und V31 in U3 und U1. "Verpacken" Sie dazu jede der drei Variablen in ein Paket. **o**

Aufgabe 19.1.5.: Programmieren Sie einen passenden **Paketrumpf** zu der folgenden **Paketspezifikation**:

```
01 package PAKET_04 is
02    type GANZ is range -5_000..+10_000;
03    function "&"(L: GANZ; R: string) return string;
04    -- Wandelt L in einen String SL um und liefert SL & R als Ergebnis.
05    function "&"(L: string; R: GANZ) return string;
06    -- Wandelt R in einen String SR um und liefert L & SR als Ergebnis.
07 end PAKET_04;
```

Schreiben Sie außerdem ein kleines Testprogramm namens **TESTP_04**, in dem das **PAKET_04** eingebunden und benützt wird. Im Programm **TESTP_04** sollen ein paar Variablen des Typs **PAKET_04.GANZ** vereinbart und die Funktionen **PAKET_04."&"** ein paar mal aufgerufen werden. Wenn Sie **vor** die Prozedur **TESTP_04**, aber **nach** der Kontextklausel die **use**-Klausel **use type PAKET_04.GANZ;** schreiben, können Sie die Funktionen namens **"&"** auch in **Infix-notation** aufrufen (**zwischen** ihren beiden Parametern, **ohne** Anführungszeichen und vor allem ohne **PAKET_04.** davor, z.B. so: **35 & "DM"** oder **"ECU" & 17**). Siehe dazu auch (ARM 8.4) und Abschnitt 30. **o**

Zusammenfassung 19.1.:

- Ein Paket besteht aus einer **Spezifikation** und einem **Rumpf**.
- Alle Vereinbarungen im **Rumpf** sind außerhalb des Paketes **nicht sichtbar**.
- In der **Spezifikation** eines Paketes dürfen **keine Rümpfe** stehen, ansonsten aber **beliebige Vereinbarungen**.
- Im **Rumpf** eines Paketes dürfen **beliebige Vereinbarungen** stehen, auch Spezifikationen und Rümpfe.
- Man unterscheidet ähnlich wie bei Unterprogrammen **Bibliothekspakete** und **enthaltene Pakete** (deren Vereinbarung in einer anderen Programmeinheit **enthalten** ist).
- Alle Vereinbarungen in der **Spezifikation** eines Bibliothekspaketes **P** können in anderen Bibliothekseinheiten **indirekt sichtbar** gemacht werden, indem man vor diese anderen Einheiten **Kontextklauseln** wie **with P;** schreibt.
- Die in einem **Bibliothekspaket** vereinbarten Größen werden zu **Beginn jeder Programmausführung** erzeugt und leben bis zum **Ende der Programmausführung**.
- Die Variablen **STAPEL** und **LBI** (vereinbart im Rumpf des Paketes **PAKET_01**) werden als **abstrakte Variablen** bezeichnet, weil die Paketbenutzer ihre **konkreten Eigenschaften** (z.B. ihre Typen, ihre Größe) nicht "sehen" und sie nur mit den durch das Paket zur Verfügung gestellten Unterprogrammen **PUSH**, **POP**, ... etc. bearbeiten können.

19.2. Ein konkreter Datentyp

Das **PAKET_01** im vorigen Abschnitt ist nützlich, wenn man genau **einen** Stapel mit Platz für **100** Zeichen braucht. Falls man einen **kleineren** oder einen **größeren** Stapel braucht oder wenn man **mehrere** verschiedene Stapel verschiedener Größen benötigt, ist das **PAKET_01** höchstens nach "mühsamen und fehlerträchtigen" Änderungen und Anpassungen einsetzbar.

Das **PAKET_02** im folgenden Beispiel stellt seinen Benutzern nicht **einen Stapel** zur Verfügung, sondern einen Typ **!S_TYP**. Jede **Variable** von diesem Typ ist ein **eigener Stapel**. Wenn man eine **!S_TYP**-Variable vereinbart, kann man außerdem die **Größe** des Stapels frei wählen. Somit ist es mit dem **PAKET_02** z.B. leicht möglich, in einem Programm gleichzeitig zwei Stapel mit Platz für je 5_000 Zeichen und drei Stapel mit Platz für je 20 Zeichen etc. zu verwalten.

Beispiel 19.2.1.: Ein konkreter Datentyp **!S_TYP**:

```
01 package PAKET_02 is
02 -- Dient zum Verwalten von Stapeln von Zeichen (vom Typ character). Der
03 -- Kollege-2 kann beliebig viele Stapel (d.h. Variablen vom Untertyp S_TYP)
04 -- vereinbaren und die Groesse eines jeden Stapels selbst festlegen.
05 ------------------------------------------------------------------------
```

```
06    type S_TYP(MAX_ANZ: natural) is record
07       LBI   : natural := 0;
08       STAPEL: string(1..MAX_ANZ);
09    end record;
10    ------------------------------------------------------------------------
11    procedure PUSH(ZEICHEN: character; STAPEL: in out S_TYP);
12    -- Wenn der STAPEL voll ist, wird die Ausnahme UEBERLAUF ausgeloest.
13    -- Sonst wird das ZEICHEN auf den Stapel gelegt.
14    ------------------------------------------------------------------------
15    procedure POP(STAPEL: in out S_TYP);
16    -- Wenn der STAPEL leer ist, wird die Ausnahme UNTERLAUF ausgeloest.
17    -- Sonst wird das oberste Element vom Stapel entfernt.
18    ------------------------------------------------------------------------
19    function TOP(STAPEL: S_TYP) return character;
20    -- Wenn der STAPEL leer ist, wird die Ausnahme UNTERLAUF ausgeloest.
21    -- Sonst wird eine Kopie des obersten Stapelelements als Ergebnis
22    -- geliefert. Der Stapel bleibt unveraendert.
23    ------------------------------------------------------------------------
24    function ANZAHL(STAPEL: S_TYP) return natural;
25    -- Liefert die Anzahl der Elemente, die sich momentan auf dem
26    -- STAPEL befinden.
27    ------------------------------------------------------------------------
28    UEBERLAUF: exception; -- Wird evtl. von PUSH ausgeloest.
29    UNTERLAUF: exception; -- Wird evtl. von POP oder TOP ausgeloest.
30 end PAKET_02;

31 package body PAKET_02 is
32    ------------------------------------------------------------------------
33    procedure PUSH(ZEICHEN: character; STAPEL: in out S_TYP) is
34    begin
35       if STAPEL.LBI = STAPEL.STAPEL'last then
36          raise UEBERLAUF;
37       else
38          STAPEL.LBI                := STAPEL.LBI + 1;
39          STAPEL.STAPEL(STAPEL.LBI) := ZEICHEN;
40       end if;
41    end PUSH;
42    ------------------------------------------------------------------------
43    procedure POP(STAPEL: in out S_TYP) is
44    begin
45       if STAPEL.LBI < STAPEL.STAPEL'first then
...    ...
60    ------------------------------------------------------------------------
61    function ANZAHL(STAPEL: S_TYP) return natural is
62    begin
63       return STAPEL.LBI;
64    end ANZAHL;
65 end PAKET_02;
```

Das **PAKET_02** besteht aus der **Spezifikation** in Zeile 01 bis 30 und dem **Rumpf** in Zeile 31 bis 65. Der Rumpf ist hier nicht vollständig wiedergegeben.

Der Untertyp **S_TYP** (vereinbart in Zeile 06 bis 09) ist ein **uneingeschränkter Verbundtyp** mit einer **Diskriminanten** namens **MAX_ANZ**. Jeder Verbund dieses Untertyps hat genau **drei** Komponenten mit den Selektornamen **MAX_ANZ**, **LBI** und **STAPEL**. Die Größe der **STAPEL**-Komponenten hängt vom Wert der Diskriminanten **MAX_ANZ** ab (siehe Zeile 08).

Weil man beliebig **viele** Stapel (d.h. Variablen des Untertyps **S_TYP**) vereinbaren kann, muß man beim Aufrufen der Unterprogramme **PUSH**, **POP** etc. jeweils angeben, **welcher** Stapel bearbeitet werden soll. Die Unterprogramme haben deshalb (im Vergleich zu ihren "Vorgängerversionen" im **PAKET_01**) einen zusätzlichen Parameter namens **STAPEL** vom Untertyp **S_TYP** (siehe Zeile 11, 15, 19 und 24). Da die Prozeduren **PUSH** und **POP** ihren **STAPEL**-Parameter verändern, muß er den Modus **in out** haben. Bei den Funktionen **TOP** und **ANZAHL**, die als "anständige Funktionen" keine Behälterinhalte verändern, hat der **STAPEL**-Parameter den Modus **in**.

In den Rümpfen der Unterprogramme **PUSH**, **POP** etc. kommen zusammengesetzte Namen wie **STAPEL.LBI** und **STAPEL.STAPEL** vor (siehe z.B. Zeile 35 bis 39 oder Zeile 45). Der Name **STAPEL.LBI** bezeichnet die **LBI**-Komponente des Verbundparameters **STAPEL**. Entsprechend ist **STAPEL.STAPEL** die **STAPEL**-Komponente des Verbundparameters **STAPEL**. **o**

Das folgende Beispiel zeigt eine kleine Prozedur, die das **PAKET_02** mit einer Kontextklausel einbindet und Größen aus dem Paket benutzt:

Beispiel 19.2.2.: Ein kleines Programm, in dem das **PAKET_02** benutzt wird:

```
01 with PAKET_02, ada.text_io;
02 procedure TESTP_02 is
03 -- Minimales Testprogramm zum Testen des Pakets PAKET_02.
04    TEXT1: string := "?se theg eiw ,ollaH";
05    TEXT2: string(1..TEXT1'length);
06    OTTO : PAKET_02.S_TYP(MAX_ANZ => 50); -- Ein Stapel fuer 50 Zeichen
07 begin
08    for I in positive range TEXT1'range loop
09       PAKET_02.PUSH(STAPEL => OTTO, ZEICHEN => TEXT1(I));
10    end loop;
11    for I in positive range 1..PAKET_02.ANZAHL(STAPEL => OTTO) loop
12       TEXT2(I) := PAKET_02.TOP(STAPEL => OTTO);
13       PAKET_02.POP(STAPEL => OTTO);
14    end loop;
15    ada.text_io.put_line(item => TEXT2);
16 end TESTP_02;
```

In Zeile 06 wird ein Stapel namens **OTTO** vereinbart, auf den man maximal 50 Zeichen legen kann. Ansonsten hat dieses Programm große Ähnlichkeit mit dem Programm **TESTP_01** im vorigen Abschnitt. **o**

Aufgabe 19.2.1.: Schreiben Sie ein Programm namens **BEN1P_02** ("Benutzer 1 von **PAKET_02**"), in dem **zwei** Stapel vom Untertyp **PAKET_02.S_TYP** vereinbart werden. Das Programm soll eine beliebig lange Zeichenkette **einlesen** (maximal 100 Zeichen), die einzelnen Zeichen der Zeichenkette auf den **ersten** Stapel legen, sie dann wieder herunterholen und auf den **zweiten** Stapel legen und sie dann auch von da wieder runterholen und **ausgeben**. Da die Reihenfolge der Zeichen **zweimal** umgedreht wird, müßten sie dadurch in der gleichen Reihenfolge auf dem Bildschirm erscheinen, in der sie eingegeben wurden. **o**

Das **PAKET_02** hat im Vergleich zum **PAKET_01** einige **Vorteile**. Mit dem **PAKET_02** kann der Kollege-2 **mehrere** Stapel verwalten und die **Größe** jedes einzelnen Stapels **selbst bestimmen**. Trotzdem hat das **PAKET_02** noch eine wesentliche **strukturelle Schwäche**, die hier diskutiert und im nächsten Abschnitt beseitigt werden soll.

Das **PAKET_02** stellt seinen Benutzern den Untertyp **S_TYP** zur Verfügung und zwei **Schnittstellen** zum Bearbeiten von **S_TYP**-Variablen: eine **offizielle Schnittstelle** und eine **inoffizielle**.

Zur **offiziellen Schnittstelle** gehören alle in der Spezifikation des Paketes vereinbarten Größen, d.h. die vier Unterprogramme **PUSH**, **POP**, **TOP** und **ANZAHL** und die beiden Ausnahmen **UEBERLAUF** und **UNTERLAUF**. Ist **OTTO** eine Variable vom Untertyp **S_TYP**, dann kann man die Unterprogramme der offiziellen Schnittstelle auf **OTTO** anwenden und in bestimmten Situationen wird dadurch die Ausnahme **UEBERLAUF** bzw. **UNTERLAUF** ausgelöst.

Zur **inofiziellen Schnittstelle** gehört die Möglichkeit, auf die Komponenten einer **S_TYP**-Variablen wie **OTTO** mit zusammengesetzten Namen wie **OTTO.LBI** und **OTTO.STAPEL** direkt zuzugreifen.

In einigen Fällen kann ein Paketbenutzer seine **S_TYP**-Variable **OTTO** wahlweise mit der offiziellen oder mit der inoffiziellen Schnittstelle bearbeiten. Um z.B. die Anzahl der Elemente auf dem Stapel **OTTO** herauszufinden, kann er den offiziellen Funktionsaufruf **PAKET_02.ANZAHL(STAPEL => OTTO)** benützen, oder aber inoffiziell direkt auf die Komponente **OTTO.LBI** zugreifen. Mit dem offiziellen Prozeduraufruf **PAKET_02.POP(STAPEL => OTTO)** kann er das oberste Element vom Stapel **OTTO** entfernen. Eine ganz ähnliche Wirkung hat der inoffizielle Befehl **OTTO.LBI := OTTO.LBI - 1**.

Mit der **inoffiziellen Schnittstelle** kann man außerdem ein paar "Tricks" programmieren, die offiziell nicht möglich sind. Z.B. kann man mit dem inoffiziellen Ausdruck **OTTO.STAPEL(3)** jederzeit auf das (von unten gerechnet) 3. Element des Stapels **OTTO** zugreifen. Das "geht" sogar dann, wenn gerade weniger als drei Elemente auf dem Stapel liegen.

Allgemein gilt: Wenn eine Schnittstelle **intensiv benutzt** wird (an vielen Stellen in vielen Programmen), dann ist es in aller Regel **teuer bis unmöglich**, sie zu verändern. Denn eine **Veränderung** der Schnittstelle würde eine entsprechende **Anpassung** aller Programmstellen nötig machen, an denen die Schnittstelle benützt wird. Solche Anpassungen vorzunehmen ist in aller Regel mühsam, fehlerträchtig und damit teuer. Das gilt gleichermaßen für **offizielle** und für **inoffizielle** Schnittstellen.

Für das **PAKET_02** gilt (bzw. soll hier angenommen werden), daß seine **offizielle Schnittstelle** ziemlich ausgereift ist und vermutlich **nie verändert** zu werden braucht. Die inoffizielle Schnittstelle beruht dagegen auf der Entscheidung, einen Stapel (d.h. eine **S_TYP**-Variable) als einen Verbund mit drei Komponenten namens **MAX_ANZ**, **LBI** und **STAPEL** zu realisieren und die **STAPEL**-Komponente als Reihung vom Typ **!string**. Es ist leicht vorstellbar, daß sich wenige Monate nach Fertigstellung des Paketes herausstellt, daß diese Implementierungsentscheidung **verbessert** werden kann und sollte.

Als konkretes Beispiel einer solchen Verbesserung sei hier einmal angenommen, daß es sich aus irgendwelchen Gründen als günstiger herausstellt, den Untertyp **S_TYP** wie folgt zu vereinbaren:

```
06    type S_TYP(MAX_ANZ: natural) is record
07       LBI   : natural := 99;
08       STAPEL: string(100..99+MAX_ANZ);
09    end record;
```

In dieser verbesserten Version beginnen die Indizes der **STAPEL**-Komponenten nicht bei **1** sondern bei **100**. Daß das eine **Verbesserung** ist, soll hier nicht begründet, sondern **angenommen** werden.

Die Verbesserung des Untertyps **S_TYP** macht es notwendig, die Funktion **ANZAHL** anzupassen. Der **return**-Befehl in Zeile 63 muß jetzt wie folgt aussehen:

```
63    return STAPEL.LBI - 99;
```

Damit ist die offizielle Schnittstelle des Paketes **PAKET_02** wieder konsistent ("in sich stimmig"), denn die Unterprogramme **PUSH**, **POP** und **TOP** funktionieren nach wie vor ordnungsgemäß. Und alle Programme, in denen nur die **offizielle Schnittstelle** des Paketes benützt wird, funktionieren

mit der verbesserten Version des Paketes genauso gut wie mit der alten Version (vielleicht sogar ein bißchen besser, z.B. schneller).

Die Verbesserung des Untertyps **S_TYP** stellt allerdings unvermeidbar eine **Veränderung** der **inoffiziellen Schnittstelle** dar. Wenn die inoffizielle Schnittstelle schon intensiv benutzt wird (an vielen Stellen in vielen Programmen) ist eine solche Veränderung teuer bis unmöglich.

Das Betriebssystem DOS ist ein besonders praxisrelevantes und eindrückliches Beispiel für die Problematik, die hier anhand des Paketes **PAKET_02** skizziert wurde. Das Betriebssystem DOS stellt den Programmen, die "unter ihm laufen", eine **offizielle Schnittstelle** zur Verfügung, mit der man unter anderem Daten ein- und ausgeben kann. In frühen DOS-Versionen war diese offizielle Schnittstelle (aus verschiedenen Gründen) relativ **langsam**. Findige Programmierer fanden aber bald heraus, wie man die **offizielle Schnittstelle** umgehen und Daten über eine **inoffizielle Schnittstelle** deutlich schneller ausgeben konnte. So entstanden unter anderem hunderte von Spielprogrammen mit beeindruckender Grafikausgabe, die auch auf relativ billigen Computern erstaunlich schnell liefen und sich gut verkauften. Als man das Betriebssystem DOS dann unter **Beibehaltung** seiner offiziellen Schnittstelle aber durch **Veränderungen** an seiner inoffiziellen Schnittstelle verbessern wollte, war das aus wirtschaftlichen Gründen **nicht mehr möglich**: Tausende von DOS-Programmen hätten angepaßt und an Millionen von Benutzern verteilt (verkauft? verschenkt?) werden müssen. Bis heute (1998) **muß** das Betriebssystem DOS (und als Folge davon auch Windows95) gewisse Schwachstellen der frühen DOS-Versionen unverbessert beibehalten.

In jedem Programm müssen bestimmte Größen (Typen, Variablen, Unterprogramme etc.) vereinbart und an bestimmten Stellen des Programms **sichtbar** gemacht werden, damit sie dort benützt werden können. Vor allem bei großen Programmen, die über einen längeren Zeitraum hin von mehreren Personen entwickelt und weiterentwickelt werden, ist es aber mindestens genauso wichtig, Informationen über bestimmte "Tatbestände" an bestimmten Stellen eines Programms "**unsichtbar**" zu halten, damit sie dort **nicht** benutzt werden können. Nur so ist es möglich, diese "Tatbestände" später noch zu verändern und zu verbessern. Im Englischen bezeichnet man dieses wichtige Ziel mit dem Schlagwort **information hiding**.

Im nächsten Abschnitt wird ein weiteres Paket namens **PAKET_03** vorgestellt, welches nicht nur eine **verbesserte** Version des Paketes **PAKET_02** ist, sondern vor allem eine später noch **verbesserbare** Version, die nur eine **offizielle Schnittstelle** zur Verfügung stellt und **keine** darüber hinausgehenden Informationen sichtbar macht.

Zusammenfassung 19.2.:

- Das **PAKET_02** stellt seinen Benutzern keine **abstrakte Variable** zur Verfügung (wie das **PAKET_01**), sondern einen **konkreten Typ !S_TYP**.
- Bentzer des Pakets **PAKET_02** können beliebig viele Stapel (Variablen des Untertyps **S_TYP**) vereinbaren und die Größe jedes einzelnen Stapels selbst bestimmen.
- **!S_TYP** wird als **konkreter Typ** bezeichnet, weil die Paketbenutzer alle konkreten Eigenschaften des Typs sehen können (z.B. können sie sehen, daß eine Variable dieses Typs ein Verbund mit **drei** Komponenten namens **MAX_ANZ**, **LBI** und **ANZAHL** ist etc.).
- Das **PAKET_02** stellt seinen Benutzern nicht nur eine **offizielle Schnittstelle** zum Bearbeiten von **S_TYP**-Variablen zur Verfügung, sondern zusätzlich auch noch eine **inoffizielle Schnittstelle**, die auf den konkreten Eigenschaften des Typs **!S_TYP** beruht.
- Es besteht die Gefahr, daß die **inoffizielle Schnittstelle** (intensiv) **benutzt** wird und deshalb später aus organisatorisch-wirtschaftlichen Gründen **kaum noch verbessert** werden kann.

19.3. Ein abstrakter Datentyp

In diesem Abschnitt wird das **PAKET_03** vorgestellt, welches seinen Benutzern die gleiche **offizielle Schnittstelle** zur Verfügung stellt wie das **PAKET_02** im vorigen Abschnitt, aber **keine** zusätzliche **inoffizielle Schnittstelle**. Der entscheidende Punkt ist, daß das **PAKET_03** seinen Benutzern zwar den Untertyp **S_TYP** sichtbar macht, aber gleichzeitig "alle konkreten Eigenschaften dieses Untertyps" unsichtbar hält.

Beispiel 19.3.1.: Ein abstrakter Datentyp **!S_TYP**:

```
01 package PAKET_03 is
02 -- Dient zum Verwalten von Stapeln von Zeichen (vom Typ character). Der
03 -- Kollege-2 kann beliebig viele Stapel (d.h. Variablen vom Untertyp S_TYP)
04 -- vereinbaren und die Groesse eines jeden Stapels selbst festlegen.
05 ------------------------------------------------------------------------
06    type S_TYP(MAX_ANZ: natural) is limited private;
07    ------------------------------------------------------------------------
...    -- Spezifikationen der Unterprogramme PUSH, POP, TOP und ANZAHL
...    -- und Vereinbarungen der Ausnahmen, UEBERLAUF und UNTERLAUF
...    -- ganz genau so wie im PAKET_02.
26    ------------------------------------------------------------------------
27 private
28    type S_TYP(MAX_ANZ: natural) is record
29       LBI   : natural := 0;
30       STAPEL: string(1..MAX_ANZ);
31    end record;
32 end PAKET_03;
```

Der Rumpf des Paketes **PAKET_03** unterscheidet sich nur durch seinen **Namen** vom Rumpf des Paketes **PAKET_02** und wird deshalb hier nicht noch einmal wiedergegeben.

Das **PAKET_03** stellt seinen Benutzern exakt die gleiche **offizielle Schnittstelle** zur Verfügung, wie das **PAKET_02** im vorigen Abschnitt (den Untertyp **S_TYP** und zum Bearbeiten von **S_TYP**-Variablen die vier Unterprogramme **PUSH**, **POP**, **TOP** und **ANZAHL** sowie die zwei Ausnahmen **UEBERLAUF** und **UNTERLAUF**). Das **PAKET_03** macht darüberhinaus aber keinerlei **inoffizielle Schnittstelle** sichtbar.

Das gesamte **PAKET_03** (Spezifikation und Rumpf) hat folgende **Struktur**:

```
01 | package  PAKET_03   is
   | ...                        --    Öffentlicher    Teil der Spezifikation
27 | private
   | ...                        --    Privater        Teil der Spezifikation
32 | end PAKET_03;
```

```
33 | package  body PAKET_03     is
   | ...                        -- Rumpf
67 | end PAKET_03;
```

Die **Spezifikation** des Paketes (Zeile 01 bis 32) wird durch das reservierte Wort **private** (in Zeile 27) in **zwei** Teile geteilt, nämlich in einen **öffentlichen Teil** (in Zeile 02 bis 26) und einen **privaten Teil** (in Zeile 28 bis 31).

Konzeptionell gehört der **private Teil** der Spezifikation schon zum **Rumpf** des Paketes: Vereinbarungen, die im **privaten Teil** oder im **Rumpf** stehen, sind außerhalb des Paketes garantiert **nicht sichtbar**. Nur die Vereinbarungen, die im **öffentlichen Teil** der Spezifikation stehen, können in anderen Programmeinheiten (mit einer Kontextklausel **with PAKET_03;** indirekt) **sichtbar** gemacht werden.

Im **öffentlichen Teil** der Spezifikation steht eine **unvollständige Typvereinbarung** (in Zeile 06). Sie muß durch eine **vollständige** Typvereinbarung im privaten Teil **vervollständigt** werden. Das geschieht tatsächlich in den Zeilen 28 bis 31. Die vollständige Typvereinbarung sieht genauso aus wie die "normale" Typvereinbarung im **PAKET_02**.

Die **unvollständige Typvereinbarung** in Zeile 06 macht **nur** sichtbar, daß
1. **S_TYP** ein limitiert privater **Untertyp** ist und daß
2. jede **S_TYP**-Variable eine Diskriminante namens **MAX_ANZ** vom Untertyp **natural** besitzt.

Für die Benutzer des Paketes bedeutet das praktisch, daß für sie nur **die offizielle Schnittstelle** des Paketes sichtbar ist, daß sie darüberhinaus aber **keine** "konkreten Eigenschaften" des Untertyps **S_TYP** "**sehen und ausnützen**" können.

Insbesondere können die Paketbenutzer **nicht** "sehen", daß **S_TYP** ein **Verbund**untertyp ist, daß jede **S_TYP**-Variable aus **drei** Komponenten namens **MAX_ANZ**, **LBI** und **STAPEL** besteht und daß die **STAPEL**-Komponente zum Typ **!string** gehört etc..

Die Benutzer des Paketes **PAKET_03** können in ihren Programmeinheiten beliebig viele Variablen des Untertyps **S_TYP** vereinbaren und müssen in jeder solchen Variablenvereinbarung einen Wert für die Diskriminante **MAX_ANZ** festlegen, z.B. so:

```
01 with PAKET_03, ...;
02 procedure IRGEND_EINES_BENUTZERS is
03    ...
04    OTTO: PAKET_03.S_TYP(MAX_ANZ => 100);   -- Platz fuer   100 Zeichen
05    ...
06    EMIL: PAKET_03.S_TYP(MAX_ANZ => 5_000); -- Platz fuer 5_000 Zeichen
07    ...
```

Variablen wie **OTTO** und **EMIL** dürfen die Paketbenutzer mit der **offiziellen Schnittstelle** des Pakets bearbeitet. Aber sie dürfen **nicht** auf die Komponenten von **OTTO** oder **EMIL** zugreifen, auch nicht auf die Diskriminanten **OTTO.MAX_ANZ** oder **EMIL.MAX_ANZ**. In diesem Sinne sind die Komponenten einer **S_TYP**-Variablen (insbesondere ihre Diskriminante) den Paketbenutzern **nicht sichtbar**.

Weil das **PAKET_03** seinen Benutzern exakt die gleiche **offizielle Schnittstelle** zur Verfügung stellt wie das **PAKET_02**, kann das Testprogramm **TESTP_02** (in dem nur die offizielle Schnittstelle des Paketes **PAKET_02** benützt wird) leicht in ein entsprechendes Testprogramm für das **PAKET_03** umgewandelt werden. Man muß nur die Kontextklausel **with PAKET_02, ...;** durch die Klausel **with PAKET_03, ...;** und den Namen **TESTP_02** etwa durch den Namen **TESTP_03** ersetzen.

Den Typ **!S_TYP**, der im öffentlichen Teil des Paketes **PAKET_03** nur **unvollständig** und erst im privaten Teil dann **vollständig** vereinbart wurde, bezeichnet man auch als einen **abstrakten (Daten-) Typ**. Damit ist gemeint, daß die Benutzer des Paketes **PAKET_03** die **konkreten**

Eigenschaften dieses Typs (bzw. seines Untertyps **S_TYP**) **nicht** sehen und **S_TYP**-Variablen nur mit Hilfe der **offiziellen Schnittstelle** des Paketes bearbeiten können.

In Ada kann man einen abstrakten Typ wahlweise als **privaten Typ** vereinbaren oder als **limitiert privaten Typ**. Der Unterschied zwischen diesen beiden Arten von abstrakten Typen ist "reltaiv klein und technisch" und nicht etwa "bedeutend und konzeptionell". Beide Arten von Typen müssen zuerst **unvollständig** im **öffentlichen** Teil eines Paketes und dann **vollständig** im **privaten** Teil desselben Paketes vereinbart werden. Zum Bearbeiten von Variablen (und Konstanten) eines nur **privaten** Typs stehen den Paketbenutzern zusätzlich zur offiziellen Schnittstelle des Paketes noch **drei** weitere Operationen zur Verfügung: die Vergleichsoperationen namens `"="` und `"/="` und die Zuweisungsoperation `":="`. Für einen **limitiert privaten** Typ stehen diese zusätzlichen Operationen **nicht** zur Verfügung.

Aufgabe 19.3.1.: Begründen Sie, daß es **sinnvoll** ist, den Typ **!S_TYP** im **PAKET_03** als **limitiert privaten** Typ (und nicht nur als **privaten** Typ) zu vereinbaren. Nehmen Sie dazu einmal an, daß **!S_TYP** als **privater** Typ vereinbart worden wäre und daß **OTTO** und **EMIL** zwei Variablen dieses privaten Typs sind. In welchen konkreten Fällen liefert dann ein Vergleich wie zum Beispiel **OTTO = EMIL** den Wert **false**, obwohl die Paketbenutzer **true** für "das eigentlich **richtige** Ergebnis" halten? **o**

Aufgabe 19.3.2.: Erweitern Sie die **Spezifikation** des Paketes **PAKET_03** um die folgende **Funktionsspezifikation**:

```
26a   function "="(L, R: S_TYP) return boolean;
26b   -- Liefert true, wenn auf den Stapeln L und R die momentan die gleichen
26c   -- Elemente in der gleichen Reihenfolge liegen, und sonst false.
```

und fügen Sie einen entsprechenden **Rumpf** für die Funktion `"="` in den Rumpf des Paketes **PAKET_03** ein. Diese Gleichheitsfunktion soll **S_TYP**-Variable "vernünftig" vergleichen und so, "wie man das eigentlich möchte" (wie man es **nicht** möchte haben Sie ja als Lösung zur vorigen Aufgabe beschrieben). **o**

Um ehrlich zu sein: Eigentlich wäre es folgerichtiger, wenn es **private Teile von Paketspezifikationen** überhaupt nicht geben würde und die **vollständige Vereinbarung** eines privaten Typs im **Rumpf** des betreffenden Paketes angegeben werden müßte. Die Ada-Entwickler haben diese einfache und konsequente Regelung auch erwogen, auf Anraten erfahrener Compilerbauer aber verworfen: Sie würde die Konstruktion von **effizienten Ada-Compilern** bis zur Unmöglichkeit erschweren. Um beim heutigen Stand der Compilerbautechnik und für heute übliche Computer effiziente Ada-Compiler möglich zu machen, hat man den **privaten Teil** einer Paketspezifikation als eine Art "Vorschau auf den Rumpf" erfunden. Diese Lösung ist nicht besonders "schön", gehört aber trotzdem zu den besten heute bekannten Lösungen dieses Problems (siehe dazu auch die Lösung bzw. Vermeidung des Problems in den Sprachen Modula und Java).

Zusammenfassung 19.3.:

- Ein Datentyp ist **abstrakt**, wenn nur seine **offzielle Schnittstelle** sichtbar ist, aber keine darüber hinausgehende **inoffizielle Schnittstelle**.
- In Ada kann man **abstrakte Typen** als **private** oder als **limitiert private** Typen in der **Spezifikation eines Paketes** vereinbaren.
- Ein abstrakter Typ muß **zweimal** vereinbart werden: Zuerst **unvollständig** im **öffentlichen** Teil der Paketspezifikation, später **vollständig** im **privaten** Teil derselben Paketspezifikation.

- Zur Bearbeitung von Variablen und Konstanten eines **limitiert privaten** Typs stehen den Paketbenutzern nur die Unterprogramme zur Verfügung, die im öffentlichen Teil der Paketspezifikation (in der auch der Typ vereinbart wurde) spezifiziert sind.
- Zur Bearbeitung von Variablen und Konstanten eines **privaten** Typs stehen den Paketbenutzern zusätzlich noch die Vergleichsoperationen **"="** und **"/="** und die Zuweisungsanweisung **":="** zur Verfügung.
- Der **private Teil einer Paketspezifikation** wurde "nur" erfunden, um den Bau **effizienter Compiler** zu ermöglichen. "Eigentlich" gehört dieser Teil schon zum **Rumpf** des Pakets.

19.4. Ein paar Feinheiten und Aufgaben

Einen **abstrakten Datentyp** muß man in Ada **zweimal** vereinbaren: Zuerst **unvollständig** als **privaten** oder **limitierten privaten** Typ im **öffentlichen** Teil einer Paketspezifikation und dann **vollständig** im **privaten** Teil derselben Paketspezifikation.

Soll das Paket seinen Benutzern mit dem abstrakten Typ auch ein paar **Konstanten** dieses Typs zur Verfügung stellen, dann muß man auch diese Konstanten **zweimal** vereinbaren: Zuerst **unvollständig** im **öffentlichen** Teil und dann **vollständig** im **privaten** Teil der Paketspezifikation, z.B. so:

Beispiel 19.4.1.: Konstanten eines abstrakten Typs vereinbaren:

```
01 package PAKET_04 is
02    type GANZ is private;
03    GK0 : constant GANZ;                -- GANZ-Konstante 0
04    GK1 : constant GANZ;                -- GANZ-Konstante 1
05    ...
06 private
07    type GANZ is range -1_000_000_000..+1_000_000_000;
08    GK0 : constant GANZ := 0;
09    GK1 : cosntant GANZ := 1;
10 end PAKET_04;
```

Im **öffentlichen** Teil der Paketspezifikation (**vor** dem Wort **private** in Zeile 06) ist **sichtbar**, daß **GANZ** (irgendein) **Untertyp** ist, aber es ist dort **nicht sichtbar**, ob **GANZ** z.B. ein Verbunduntertyp oder ein Ganzzahluntertyp etc. ist. Deshalb können die **GANZ**-Konstanten **GK0** und **GK1** in den Zeilen 03 und 04 nur **unvollständig** vereinbart werden, d.h. **ohne** Angabe ihrer **Werte**.

Im **privaten** Teil der Paketspezifikation (**nach** dem Wort **private** in Zeile 06) müssen die Konstanten **GK0** und **GK1** dann (**nach** dem Untertyp **GANZ**) vollständig vereinbart werden, d.h. **mit** Angabe ihrer Werte. Nach Zeile 07 ist sichtbar, daß **GANZ** ein Ganzzahluntertyp ist und die Werte **0** und **1** zu seinem Wertebereich gehören.

Würde man die unvollständigen Konstantenvereinbarungen in Zeile 03 und 04 **weglassen**, dann wären **GK0** und **GK1** nur **innerhalb** des Paketes sichtbar und **außerhalb** des Paketes garantiert **nicht sichtbar**.

Angenommen, in einem Programm namens **TESTP_04** sollen bestimmte Ganzzahlvariablen nur jeweils **um 1 erhöht** bzw. **um 1 vermindert** werden. Außerdem weiß der Programmierer, daß im

Verlauf einer Ausführung des Programms **TESTP_04** die **Anzahl der Erhöhungen** gleich der **Anzahl der Verminderungen** sein muß.

Das **PAKET_04** stellt seinen Benutzern einen abstrakten (Ganzzahl-) Untertyp **GANZ** zur Verfügung und die folgenden Prozeduren:

GET	zum **Einlesen** eines GANZ-Wertes von der aktuellen Eingabe
PUT	zum **Ausgeben** eines GANZ-Wertes zur aktuellen Ausgabe
INC	zum **Erhöhen** einer GANZ-Variablen um 1 ("increment")
DEC	zum **Vermindern** einer GANZ-Variablen um 1 ("decrement")
PUT_STATISTIK	gibt zur aktuellen Ausgabe aus, wie oft die Prozedur **INC** und wie oft die Prozedur **DEC** aufgerufen wurden

Außerdem stellt das **PAKET_04** seinen Benutzern zwei **GANZ**-Konstanten namens **GK0** und **GK1** mit den Werten **0** bzw. **1** zur Verfügung.

Beispiel 19.4.2.: Die Spezifikation des Paketes PAKET_05:

```
01 package PAKET_04 is
02 ------------------------------------------------------------------------
03 -- Erläutertungen zum Sinn und Zweck dieses Paketes:
   -- ...
16 ------------------------------------------------------------------------
17    type GANZ is private;
18    procedure   GET(ITEM:    out GANZ);
19    procedure   PUT(ITEM: in     GANZ);
20    procedure   INC(ITEM: in out GANZ);
21    procedure   DEC(ITEM: in out GANZ);
22    procedure   PUT_STATISTIK;
23    GK0: constant GANZ; -- GANZ-Konstante 0
24    GK1: constant GANZ; -- GANZ-Konstante 1
25 private
26    type GANZ is range -1_000_000_000..+1_000_000_000;
27    GK0: constant GANZ := 0;
28    GK1: constant GANZ := 1;
29 end PAKET_04;
```

Da **GANZ** als **privater** Untertyp vereinbart wurde, stehen den Paketbenutzern zur Bearbeitung von **GANZ**-Werten die folgenden Dinge **nicht zur Verfügung** bzw. **zur Verfügung**:

Nicht zur Verfügung stehen	**Zur Verfügung stehen**
Rechenopertionen wie "-", "+", "*", "/", **mod**, **rem**, **abs**	Die in der Paketspezifikation spezifizierten Prozeduren **GET**, **PUT**, **INC** etc.
Die vier **Vergleichsoperationen** "<", "<=", ">", ">="	Die beiden **Vergleichsoperationen** "=" und "/="
Ganzzahlliterale wie 0, 1, -17, +123 etc.	Die **Zuweisungsanweisung** ":="
Attribute wie **GANZ'first**, **GANZ'last** etc.	Die in der Paketspezifikation vereinbarten Konstanten **GK0** und **GK1**

Aufgabe 19.4.1.: Schreiben Sie einen zu der obigen Spezifikation passenden **Rumpf** für das **PAKET_04** und ein kleines Testprogramm namens **TESTP_04**, welches das **PAKET_04** einbindet und die darin vereinbarten Größen benützt. **o**

Für Pakete gilt die folgende **Sichtbarkeitsregel**: Indem man eine Kontextklausel wie etwa **with PAKET_05;** vor eine Programmeinheit wie etwa **TESTP_05** schreibt, macht man in **TESTP_05** nur **die** Größen indirekt **sichtbar**, die im öffentlichen Teil der Spezifikation von PAKET_05 **vereinbart** wurden, und **nicht** all die Größen, die dort **sichtbar** sind (aber anderswo vereinbart wurden). Im folgenden soll anhand eines praktischen Beispiels deutlich gemacht werden, was diese Regel bedeutet.

In einer Paketspezifikation darf man auch **Schablonen** (wie z.B. die Paketschablone **ada.text_io.-enumeration_io**) instanziieren, wie im folgenden Beispiel angedeutet:

Beispiel 19.4.2.: Eine Schablone in einer Paketspezifikation instanziieren:

```
01 with ada.text_io;
02 package PAKET_05 is
03    type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, WEISS);
04    package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
05    ...
06 end PAKET_05;
```

Man beachte, daß **FARBE** hier **nicht** als **abstrakter** Untertyp, sondern als ganz **konkreter** Aufzählungstyp vereinbart wurde. Mit der Vereinbarung des Paketes **FARBE_EA** in Zeile 04 soll es den Benutzern des Paketes **PAKET_05** möglich gemacht werden, **FARBE**-Werte ein- und auszugeben ohne selbst eine solche Instanz der Paketschablone **enumeration_io** vereinbaren zu müssen, z.B. so:

```
01 with PAKET_05;
02 procedure TESTP_05 is
03    F1 : PAKET_05.FARBE := PAKET_05.ROT;
04 begin
05    PAKET_05.FARBE_EA.put(item => F1);
06    PAKET_05.FARBE_EA.get(item => F1);
07    ...
```

In der Prozedur **TESTP_05** ist das **PAKET_05 direkt sichtbar**, das Paket **FARBE_EA** ist damit **indirekt sichtbar** und die Prozeduren **put** und **get** sind sozusagen "**doppelt indirekt**" sichtbar. Zusammengesetzte Namen wie **PAKET_05.FARBE_EA.put** sind in manchen Fällen informativer als nötig. Mit einer **Umbenennung** in der Paketspezifikation kann man den Paketbenutzern kürzere Bezeichnungen zur Verfügung stellen, z.B. so:

Beispiel 19.4.3.: Umbenennung einer **Prozedur**:

```
01 with ada.text_io;
02 package PAKET_05 is
03    type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, WEISS);
04    package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
05    procedure GET(ITEM: FARBE) renames FARBE_EA.get;
06    ...
07 end PAKET_05;
```

In Zeile 05 wird nicht eine **neue Prozedur** sondern nur eine **neue Sicht** (view) auf die Prozedur **FARBE_EA.get** vereinbart. Nach dieser Vereinbarung kann man die Prozedur innerhalb des Paketes wahlweise mit **FARBE_EA.get** oder einfach mit **GET** und außerhalb des Paketes mit **PAKET_05.FARBE_EA.get** oder einfacher mit **PAKET_05.GET** bezeichnen.

Daß die Vereinbarung in Zeile 05 als **Umbenennung** bezeichnet wird, erleichtert das Verständnis dieses Beispiels nicht besonders, denn hier wurde weder der Name **get** der Prozedur noch der Name

item ihres Parameters verändert. Die Großschreibung **GET** und **ITEM** soll ja nur deutlich machen, daß diese Namen vom Programmierer gewählt wurden, und nicht von der Sprache Ada vorgegeben sind. Für den Ada-Ausführer sind die Namen **get** und **GET** bzw. **item** und **ITEM** gleich. Der Programmierer hätte in Zeile 05 auch schreiben können:

```
05a   procedure ANNA(BERTA: FARBE) renames FARBE_EA.get;
```

Diese "**radikale Umbenennung**" würde allerdings die Benutzer des Paketes **PAKET_05** zwingen sich einzuprägen, daß die ihnen unter dem Namen **get** bekannte Prozedur mit dem gewohnten **item**-Parameter jetzt plötzlich **ANNA** heißt und einen Parameter namens **BERTA** hat. Um die Paketbenutzer nicht unnötig zu diesem "Umdenken" zu zwingen, wird hier die "**konservative Umbenennung**" bevorzugt, die den Prozedurnamen und den Parameternamen **unverändert** läßt.

Wichtig an der sogenannten **Umbennenung** in Zeile 05 ist, daß sie eine **Vereinbarung** ist und im öffentlichen Teil der Spezifikation des Paketes **PAKET_05** steht. Somit ist diese Vereinbarung z.B. in der Prozedur **TESTP_05** nur noch "**einfach indirekt**" sichtbar und nicht mehr "**doppelt indirekt**".

In einer **Umbenennung** eines Unterprogramms kann man den **Namen des Unterprogramms**, die **Namen der Parameter** und (falls vorhanden) sogar die **Vorbesetzungen der Parameter** beliebig verändern, z.B. konservativ oder radikal. **Nicht** verändern darf man dagegen die **Anzahl der Parameter**, ihre **Reihenfolge** und vor allem ihre **Untertypen**.

Im obigen Beispiel ist eine Umbenennung der Prozedur **FARBE_EA.put** deutlich schwieriger, als es die Umbenennung der Prozedur **FARBE_EA.get** war. Das hängt damit zusammen, daß die **put**-Prozedur **drei** Parameter besitzt, von denen zwei (der **width**- und der **set**-Parameter) zur **Formatkontrolle** dienen.

Die **Formatkontrolle** bei der Ausgabe von Aufzählungswerten wurde im Abschnitt 10.4. behandelt. Hier nur eine kurze Wiederholung: Bei der Ausgabe eines **FARBE**-Wertes mit der Prozedur **FARBE_EA.put** kann man die **Mindestbreite** (width) der Ausgabe angeben und festlegen, ob die Ausgabe mit **grossen** oder **kleinen** Buchstaben erfolgen soll. Diese Formatkontrolle kann man wahlweise über den **width**- und den **set**-Parameter der **put**-Prozedur ausüben, oder indem man den Variablen **FARBE_EA.default_width** und **FARBE_EA.default_setting** vor dem Aufrufen der put-Prozedur entsprechende Werte zuweist.

Der **width**-Parameter und die Variable **default_width** gehören zum Untertyp **field**. Der **set**-Parameter und die Variable **default_setting** gehören zum Untertyp **type_set**. Die Untertypen **field** und **type_set** sind im Paket **ada.text_io** vereinbart. Dabei ist **field** ein **Ganzzahluntertyp** und **type_set** ein **Aufzählungsuntertyp**, zu dem nur die beiden Werte **lower_case** und **upper_case** gehören. Siehe dazu (ARM A.10.1(6)-(7)).

Variablen und Werte der Untertypen **field** und **type_set** kann man nur **da** angeben und manipulieren, wo die Untertypen (zumindest indirekt) **sichtbar** sind. Es wäre aber keine schöne Lösung, von jedem Benutzer des Paketes **PAKET_05** zu verlangen, daß er auch das Paket **ada.text_io** einbinden muß, damit in seiner Programmeinheit die Untertypen **field** und **type_set** (indirekt) sichtbar sind. Das **PAKET_05** sollte benutzerfreundlich sein und **alle** Größen zur Verfügung stellen, die man zum Ein- und Ausgeben von **FARBE**-Werten benötigt. Außerdem ist es erstrebenswert, daß diese Größen für die Benutzer des Paketes **PAKET_05** nur **einfach** indirekt sichtbar sind und nicht

"**doppelt indirekt**" oder "noch indirekter". Das macht eine ganze Reihe von **Umbenennungen** in der Spezifikation des Paketes erforderlich.

Mit **renames** umbenennen kann man **Konstanten**, **Variablen**, **Unterprogramme**, **Ausnahmen** und **Aufzählungsliterale**. Ein Aufzählungsliteral wird dabei als eine **parameterlose Funktion** mit dem Aufzählungsuntertyp als Ergebnisuntertyp (return subtype) behandelt. Einen **Untertyp** kann man **nicht** mit **renames** umbenennen, weil man ihn schon mit einer "ganz normalen" **Untertypvereinbarung** umbenennen kann. Die vollständige Spezifikation des Paketes **PAKET_05** enthält Umbenenungen von **Variablen**, **Aufzählungsliteralen** und **Unterprogrammen** (mit renames) und von **Untertypen** (mit Untertypvereinbarungen). Mit einer kleinen Ausnahme sind alle Umbenennungen **konservativ**, d.h. sie ändern die **Namen** der Größen nicht wirklich.

Beispiel 19.4.4.: Die vollständige Spezifikation des Paketes **PAKET_05**:

```
01 with ada.text_io;
02 package PAKET_05 is
03 ------------------------------------------------------------------------
...   -- Erläuternde Kommentare
15 ------------------------------------------------------------------------
16    type    FARBE    is (SCHWARZ, ROT, GRUEN, BLAU, WEISS);
17    package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
18    ---------------------------------------------------------------------
19    -- Umbenennungen von Groessen aus den Paketen ada.text_io und FARBE_EA:
20    subtype    FIELD              is ada.text_io.field;
21    subtype    TYPE_SET           is ada.text_io.type_set;
22    subtype    POSITIVE_COUNT     is ada.text_io.positive_count;
23    function   LOWER_CASE return TYPE_SET renames ada.text_io.lower_case;
24    function   UPPER_CASE return TYPE_SET renames ada.text_io.upper_case;
25    DEFAULT_WIDTH          : FIELD      renames FARBE_EA.default_width;
26    DEFAULT_SETTING        : TYPE_SET   renames FARBE_EA.default_setting;
27    procedure GET(ITEM     : out FARBE) renames FARBE_EA.get;
28    procedure PUT(ITEM     : in  FARBE;
29                  WIDTH    : in  FIELD    := DEFAULT_WIDTH;
30                  SET      : in  TYPE_SET := DEFAULT_SETTING)
31       renames FARBE_EA.put;
32    procedure NEW_LINE(SPACING: in POSITIVE_COUNT := 2) -- Vorbesetzung 2!
33       renames ada.text_io.new_line;
34 end PAKET_05;
```

Zum Untertyp **FIELD** (siehe Zeile 20) gehören der **width**-Parameter der Prozedur FARBE_EA.**put** und die Variable **FARBE_EA.default_width**. Zum Untertyp **TYPE_SET** (sieh Zeile 21) gehören der **set**-Parameter der Prozedur FARBE_EA.**put** und die Variable **FARBE_EA.default_setting**. Zum Untertyp **POSITIVE_COUNT** (siehe Zeile 22) gehört der **spacing**-Parameter der Prozedur **ada.text_io.new_line**, die in Zeile 33 bis 34 umbenannt wird. Die Funktionen **LOWER_CASE** und **UPPER_CASE** (siehe Zeile 23 und 24) sind Umbenennungen der Aufzählungsliterale **lower_case** und **upper_case** des Untertyps **type_set**. In Zeile 25 und 26 werden zwei Variablen aus dem Paket FARBE_EA umbenannt. Die Umbenennung der Prozedur **FARBE_EA.get** (siehe Zeile 27) kam schon oben im Beispiel 19.4.3. vor und ist hoffentlich leicht zu verstehen. Die Prozedur FARBE_EA.put hat drei Parameter und ihre Umbennenung belegt die Zeilen 28 bis 31. In Zeile 29 könnte man statt **DEFAULT_WIDTH** auch **FARBE_EA.default_width** schreiben, die beiden Namen bezeichnen dieselbe Variable. Die Umbenennung der Prozedur **ada.text_io.new_line** in Zeile 32 bis 33 ist **nicht** völlig **konservativ**: "Im Original" ist der **spacing**-Parameter mit **1** vorbesetzt, in der Umbenennung mit **2**. Damit macht es einen Unterschied, ob ein Paketbenutzer den Aufruf **PAKET_05.NEW_LINE;** oder den Aufruf **ada.text_io.new_line;** verwendet, obwohl in beiden Fällen **dieselbe Prozedur** aufgerufen wird.

Die "Umbenennungen" der Untertypen **field** und **type_set** (in den Zeilen 20 bis 21) und die Umbenennungen der Aufzählungsliterale **lower_case** und **upper_case** (in Zeile 23 bis 24) sind **notwendig**, damit ein Benutzer des Paketes **PAKET_05** Werte des Untertyps **FARBE** einlesen und mit voller Formatkontrolle ausgeben kann, **ohne** das Paket **ada.text_io** in seine Programmeinheit einbinden zu **müssen** (wenn er es trotzdem einbindet, schadet das nicht). Ganz entsprechend ist die "Umbenennung" des Ganzzahluntertyps **positive_count** in Zeile 22 und die Umbenennung der Prozedur **new_line** in Zeile 32 bis 33 **notwendig**, damit die Paketbenutzer diese Prozedur aufrufen können, ohne **ada.text_io** einzubinden. Die übrigen Umbenennungen "dienen nur der **Bequemlichkeit** der Paketbenutzer", indem sie "**doppelt indirekte Sichtbarkeit**" in "**einfach indirekte Sichtbarkeit**" umwandeln. **o**

Beispiel 19.4.5.: Ein kleines Testprogramm:

```
01 with PAKET_05;
02 procedure TESTP_05 is
03 -- Minimales Testprogramm zum Testen des Paketes PAKET_05.
04 begin
05    ------------------------------------------------------------------------
06    for F in PAKET_05.FARBE loop
07       PAKET_05.PUT(ITEM => F);
08    end loop;
09    PAKET_05.NEW_LINE;  -- Cursor 2 Zeilen nach unten!
10    ------------------------------------------------------------------------
11    for F in PAKET_05.FARBE loop
12       PAKET_05.PUT(ITEM  => F, WIDTH => PAKET_05.FARBE'width + 1);
13    end loop;
14    PAKET_05.NEW_LINE;  -- Cursor 2 Zeilen nach unten!
15    ------------------------------------------------------------------------
16    PAKET_05.DEFAULT_WIDTH     := PAKET_05.FARBE'width + 2;
17    PAKET_05.DEFAULT_SETTING   := PAKET_05.LOWER_CASE;
18    for F in PAKET_05.FARBE loop
19       PAKET_05.PUT(ITEM => F);
20    end loop;
21    PAKET_05.NEW_LINE;  -- Cursor 2 Zeilen nach unten!
22    ------------------------------------------------------------------------
23 end TESTP_05;
```

Man beachte, daß diese Prozedur **FARBE**-Werte ausgibt, **ohne** das Paket **ada.text_io** einzubinden. Sogar die Prozedur **new_line** wird über ihre Umbenennung im **PAKET_05** aufgerufen, und nicht unter ihrer üblichen Bezeichnung **ada.text_io.new_line**. **o**

Mit den Umbenennungen im **PAKET_05** wurde folgendes Problem gelöst: Das Paket soll einen Benutzern unter anderem die Prozedur FARBE_EA.**put** zur Verfügung stellen. Dieses Prozedur hat drei Parameter (**item**, **width** und **set**), die zu den Untertypen FARBE, **field** und **type_set** gehören. Die Untertypen **field** und **type_set** sind im Paket **ada.text_io** vereinbart und unter anderem in der Spezifikation des Paketes **PAKET_05** indirekt **sichtbar**. Damit sind sie aber nicht automatisch auch für Benutzer des Paketes PAKET_05 sichtbar (weder direkt noch indirekt). Die Umbenennungen der Untertypen **field** und **type_set** (in Zeile 20 und 21 der Paketspezifikation) bewirken, daß diese Untertypen für die Paketbenutzer indirekt **sichtbar** sind und die Paketbenutzer die Prozedur FARBE_EA.**put** (auch unter dem Namen **PAKET_05.PUT** bekannt) mit all ihren Parametern aufrufen kann.

Deutlich gemacht werden sollte mit diesem Beispiel, daß die **Sichtbarkeit** in Ada-Paketen **nicht transitiv** ist: Nur weil eine Vereinbarung im öffentlichen Teil eines Paketes P **sichtbar** ist, ist sie damit nocht **nicht** für die Benutzer von P sichtbar. Der Programmierer kann solche Vereinbarungen

für die Benutzer von P sichtbar **machen**, wenn er das ausdrücklich wünscht (durch entsprechende Umbenennungen im öffentlichen Teil von P). **o**

Mit einer sogenannten **use-Klausel** kann der Ada-Programmierer **indirekte Sichtbarkeit** in **direkte Sichtbarkeit** überführen, wie die folgenden beiden Beispiele deutlich machen sollen:

Beispiel 19.4.6.: Ohne use-Klausel sind **zusammengesetzte Namen** erforderlich:

```
01 package PAKET_06 is
02    type RATIONAL is private;
03    function "+"(L, R: RATIONAL) return RATIONAL;
04    -- Liefert die Summe der rationalen Zahlen L und R.
05    ...
06 end PAKET_06;
```

Der **Rumpf** des Paketes **PAKET_06** ist hier nicht wesentlich und wird deshalb nicht angegeben. Wichtig ist aber eine Bibliothekseinheit, die das **PAKET_06** einbindet und benützt:

```
01 with PAKET_06;
02 procedure TESTP_06 is
03    R1, R2, R3 : PAKET_06.RATIONAL;
04    ...
05 begin
06    ... -- Geeignete Werte nach R2 und R3 bringen
07    R1 : PAKET_06."+"(L => R2, R => R3); -- Präfixnotation erforderlich!
08    ...
```

Die Kontextklausel **with PAKET_06;** macht das **PAKET_06** in der Bibliotheksprozedur TESTP_06 **direkt sichtbar** und die öffentlichen Größen des Paketes **indirekt sichtbar**. Weil die Funktion **"+"** in Zeile 07 nur **indirekt** sichtbar ist, muß sie dort mit dem zusammengesetzten Namen **PAKET_06."+"** bezeichnet und kann nur in **Präfixnotation** aufgerufen werden (siehe Zeile 07). **Infixausdrücke** mit einem **zusammengesetzten Operationsnamen** wie etwa **(R1 PAKET_06."+" R2)** sind in Ada **nicht** erlaubt.

Indem man die Kontextklausel vor **TESTP_06** um eine **use**-Klausel **use PAKET_06;** erweitert, macht man die öffentlichen Größen des Paketes in TESTP_06 **direkt sichtbar**, etwa so:

Beispiel 19.4.7.: Mit use-Klausel genügen **einfache** Namen:

```
01 with PAKET_06; use PAKET_06;
02 procedure TESTP_06 is
03    R1, R2, R3 : RATIONAL;      -- "PAKET_06." nicht erforderlich!
04    ...
05 begin
06    ... -- Geeignete Werte nach R2 und R3 bringen
07    R1 := R2 + R3;              -- Infixnotation möglich!
08    ...
```

Weil die Funktion **"+"** jetzt in Zeile 07 **direkt** sichtbar ist, kann sie dort auch in **Infixnotation** aufgerufen werden. **o**

Durch eine **use**-Klausel werden aber nicht immer **alle** öffentlichen Vereinbarungen eines Paketes **direkt sichtbar** gemacht. Die öffentlichen Vereinbarungen des Paketes, die von anderen Vereinbarungen **verdeckt** werden, bleiben trotz **use**-Klausel weiterhin nur **indirekt** sichtbar. Wenn z.B. im **PAKET_06** eine öffentliche Größe namens **R1** vereinbart wird, dann wird diese Vereinbarung von der Vereinbarung der Variablen **R1** in der Prozedur **TESTP_06** (Zeile 03) verdeckt. Die Größe

PAKET_06.R1 bleibt dann in **TESTP_06** trotz **use**-Klausel nur indirekt sichtbar und muß mit ihrem zusammengesetzten Namen **PAKET_06.R1** bezeichnet werden.

Ganz entsprechendes gilt, wenn eine Bibliothekseinheit **BE** zwei Pakete **P1** und **P2** einbindet und sowohl in **P1** also auch in **P2** z.B. je eine öffentliche Variable namens **OTTO** vereinbart wird. Trotz einer Kontextklausel mit **use**-Klausel **with P1, P2; use P1, P2;** vor der Bibliothekseinheit **BE** müssen in **BE** die Variablen namens **OTTO** mit den zusammengesetzten Namen **P1.OTTO** bzw. **P2.OTTO** bezeichnet und voneinander unterschieden werden. Man beachte aber, daß sich gleichnamige **Unterprogramme** nur dann **verdecken**, wenn sie das gleiche **Profil** haben (siehe Abschnitt 18, insbesondere das Beispiel 18.5.).

Mit **use**-Klauseln kann der Programmierer sich **Schreibarbeit sparen** und seinen Kollegen das **Lesen** seiner Programme **erschweren**. Um dieser Gefahr zu entgehen, **verbieten** einige Firmen **use**-Klauseln grundsätzlich. Ein erwägenswerter Kompromiß besteht darin, **use**-Klauseln nur für das Paket **ada.text_io** zu erlauben und eventuell für einige weitere Pakete, die "jeder Leser des Programms mit Sicherheit gut kennt".

Der Verzicht auf **use**-Klauseln ist für den Programmierer vor allem dann bitter, wenn ein Paket **Operationen** (d.h. Funktionen mit Namen wie **"+"**, **"*"**, **"&"** , ... etc.) zur Verfügung stellt, weil diese Operationen dann nur in der ungewohnten **Präfixnotation** aufgerufen werden können. Andererseits macht man mit einer normalen **use**-Klausel evtl. mehr Vereinbarungen direkt sichtbar, als für die Programmleser günstig ist. Mit Ada95 hat man deshalb eine spezielle Form der **use**-Klausel eingeführt, die nur die **Operationen** eines bestimmten Untertyps **direkt sichtbar** macht, aber alle anderen Größen **indirekt sichtbar** läßt, etwa so:

Beispiel 19.4.8.: Eine **use-type-Klausel** macht nur Operationen eines Untertyps direkt sichtbar:

```
01 with PAKET_06; use type PAKET_06.RATIONAL;
02 procedure TESTP_06 is
03    R1, R2, R3 : PAKET_06.RATIONAL; -- RATIONAL ist nur indirekt sichtbar!
04    ...
05 begin
06    ... -- Geeignete Werte nach R2 und R3 bringen
07    R1 := R2 + R3; -- "+" für RATIONAL-Werte ist direkt sichtbar
08    ...
```

Aufgabe 19.4.2.: Eine Zahl ist **rational**, wenn sie sich als **Bruch zweier Ganzzahlen** darstellen läßt. Jede rationale Zahl kann durch **unendlich viele** Brüche dargestellt werden, aber nur **einer** dieser Brüche ist **vollständig gekürzt** und hat einen **Nenner**, der **größer oder gleich 1** ist. Diesen Bruch bezeichnet man auch als die **kanonische Darstellung** der rationalen Zahl. Auf einem Rechner kann man rationale Zahlen z.B. als **Verbunde** oder als **Reihungen** mit zwei Komponenten eines Ganzzahltyps darstellen. Das **PAKET_06** stellt seinen Benutzern einen abstrakten Untertyp namens **RATIONAL** zur Darstellung von rationalen Zahlen und "eine Reihe dazu pasender Unterprogramme und Untertypen" zur Verfügung. Der Untertyp **RATIONAL** ist **abstrakt** (genauer: **limitiert privat**), um damit die Möglichkeit offen zu lassen, von einem **Verbundtyp** zu einem **Reihungstyp** zu wechseln oder umgekehrt. Hier die vollständige Spezifikation des Paketes:

```
01 package PAKET_06 is
02 ------------------------------------------------------------------------
... -- Kommentare zum Sinn und Zweck dieses Pakets
15 ------------------------------------------------------------------------
16    type RATIONAL is limited private;
17    function "+"(L, R: RATIONAL) return RATIONAL;
```

```
18    function "-"(L, R: RATIONAL) return RATIONAL;
19    function "*"(L, R: RATIONAL) return RATIONAL;
20    function "/"(L, R: RATIONAL) return RATIONAL;
21    ------------------------------------------------------------------------
22    type        GANZ           is range -1e9..+1e9;
23    subtype     ZAEHLER_TYP  is GANZ'base;
24    subtype     NENNER_TYP     is GANZ'base range 1..GANZ'base'last;
25    function    "/"(Z: ZAEHLER_TYP; N: NENNER_TYP) return RATIONAL;
26    function    ZAEHLER(R: RATIONAL) return ZAEHLER_TYP;
27    function    NENNER (R: RATIONAL) return NENNER_TYP;
28    procedure   ASSIGN(L: out RATIONAL; R: in RATIONAL);
29    ------------------------------------------------------------------------
30 private
31    type RATIONAL is record
32       Z: ZAEHLER_TYP := 0;
33       N: NENNER_TYP := 1;
34    end record;
35 end PAKET_06;
```

Zum Untertyp **GANZ'base** gehören alle Zahlen, die zum **Typ** !GANZ gehören. In diesem Beispiel sind das **mindestens** die Ganzzahlen von -1_000_000_000 bis +1_000_000_000 (siehe Zeile 22), möglicherweise aber auch wesentlich mehr (je nach Ada-Ausführer). Da der Untertyp **RATIONAL** limitiert privat ist (siehe Zeile 16), steht den Paketbenutzern die Zuweisungsanweisung **":=" nicht** zur Verfügung. Die Prozedur **ASSIGN** (siehe Zeile 28) soll ein Ersatz dafür sein.

Schreiben Sie einen zu dieser Spezifikation passenden **Rumpf** für das **PAKET_06** und ein kleines Testprogramm namens **TESTP_06**, welches das **PAKET_06** einbindet und benützt. Das Programm **TESTP_06** soll **vier** Ganzzahlen einlesen, daraus **zwei** rationale Zahlen **R1** und **R2** machen und die Werte der vier Ausdrücke **R1+R2**, **R1-R2**, **R1*R2** und **R1/R2** (evtl. mit einem kleinen erläuternden Kommentar versehen) ausgeben. **o**

Wenn in einem Programm **use**-Klauseln verwendet werden, dann häufig zusammen mit **with**-Klauseln. Das muß aber nicht unbedingt so sein. Hier die wichtigsten Regeln für **with**-Klauseln und use-Klauseln:

1. **with**-Klauseln dürfen nur **vor** einer **Bibliothekseinheit** stehen (z.B. vor einer Bibliotheksprozedur, einer Bibliotheksfunktion oder einem Bibliothekspaket), nicht aber vor **enthaltenen Einheiten**.
2. **with**-Klauseln dürfen sowohl vor der **Spezifikation** als auch vor dem **Rumpf** einer Bibliothekseinheit stehen. Vor eine Spezifikation sollte man eine **with**-Klausel aber nur schreiben, wenn sie dort auch unbedingt **gebraucht** wird (sonst vergrößert man den Sichtbarkeitsbereich der ge-with-ten Einheiten unnötig).
3. Mit einer **with**-Klausel darf man **beliebige Bibliothekseinheiten** einbinden ("withen"), z.B. Bibliotheks**prozeduren**, Bibliotheks**funktionen** und Bibliotheks**pakete**.
4. In einer **use**-Klausel darf man nur **Pakete** erwähnen, und zwar **Bibliothekspakete** und **enthaltene Pakete**. Die Pakete müssen an der Stelle, an der die **use**-Klausel steht, **sichtbar** sein. Eine **use**-Klausel **use P1, P3;** darf z.B. unmittelbar nach einer **with**-Klausel **with P1, P2, P3;** stehen.
5. Eine **use**-Klausel darf aber auch überall da stehen, wo **Vereinbarungen** erlaubt sind, z.B. im Vereinbarungsteil eines **Unterprogramms**, eines **Paketes** oder eines **Blocks**. Sie gilt dann ab der Stelle, wo sie steht, bis zum Ende des unmittelbar umschließenden **Vereinbarungsbereichs**.
6. Vor einer Bibliothekseinheit dürfen beliebig viele **with**- und **use**-Klauseln stehen.

Beispiel 19.4.8.: Eine Bibliotheksprozedur mit besonders vielen with- und use-Klauseln:

Hier wird angenommen, daß es sich bei den Bibliothekseinheiten **PAKET_01**, **PAKET_02** etc. um **Pakete** handelt und bei den Bibliothekseinheiten **UPROS_01**, **UPROS_02**, ... um Unterprogramme.

```
01 with PAKET_01, PAKET_02, UPROS_01, UPROS_02, PAKET_03, PAKET_04;
02 use  PAKET_01;
03 with ada.text_io, PAKET_05, UPROS_03, PAKET_06;
04 use  PAKET_04, ada.text_io;
05 procedure OTTO is
06    X : integer;
07    package INT_EA is new integer_io(num => integer);
08    Y : boolean;
09    use INT_EA;
10    Z : character;
11    use PAKET_06;
12 begin
13    ...
14 end OTTO;
```

Das Paket **INT_EA** (vereinbart in Zeile 07) ist ein **enthaltenes** Paket, kein Bibliothekspaket. In Zeile 09 werden seine öffentlichen Größen direkt sichtbar gemacht. **o**

Die wichtigste Regel für use-Klauseln: **Don't use them too much**.

Zusammenfassung 19.4.:

- **Konstanten** eines abstrakten Typs müssen (ähnlich wie der abstrakte Typ selbst) **zweimal** vereinbart werden: **unvollständig** im **öffentlichen** Teil einer Paketspezifikation und dann **vollständig** im **privaten** Teil derselben Paketspezifikation.
- Allein deshalb, weil eine Größe **G** im öffentlichen Teil eines Paketes **P sichtbar** ist, ist **G** für die Benutzer von **P** noch **nicht** sichtbar ("die Sichtbarkeit in Paketen ist **nicht transitiv**").
- Der Programmierer kann **G** aber für die Benutzer von **P** sichtbar machen, indem er **G** (im öffentlichen Teil von **P**) **umbenennt**.
- Mit einer **with**-Klausel vor einer Bibliothekseinheit **BE** kann man beliebige Bibliothekseinheiten **BE1**, **BE2**, ... etc. in **BE** einbinden ("withen").
- Mit einer **use**-Klausel vor einer Bibliothekseinheit **BE** oder in einem Vereinbarungsteil kann man die öffentlichen Größen eines sichtbaren Paketes **direkt sichtbar** machen.

Zentraldokument: Nach Filialdokument A95-19-19

20. Schablonen (generische Einheiten)

Viele **Algorithmen** sind weitgehend **unabhängig von den Typen** der bearbeiteten Daten. Um z.B. die Werte zweier Variablen zu vertauschen, sind im Grunde immer die gleichen Befehle notwendig, egal, ob es sich dabei um **character**-Variablen, **boolean**-Variablen oder **integer**-Variablen etc. handelt. Ganz ähnlich sind viele Sortierverfahren weitgehend unabhängig davon, ob man einfache Ganzzahlen, Textzeilen oder komplexe Verbunde damit sortiert.

Ada-**Unterprogramme** sind "konkreter" als solche Algorithmen, denn in einem Unterprogramm muß man die **Typen** der bearbeiteten Daten **genau festlegen**. So ist es z.B. nicht möglich, **eine** Prozedur zu schreiben, mit der man wahlweise die Werte zweier **character**-Variablen oder zweier **boolean**-Variablen vertauschen kann. Vielmehr muß man für jeden Typ eine eigene Vertausche-Prozedur vereinbaren. Ganz ähnlich kann man kein Unterprogramm schreiben, mit dem man beliebige Daten sortieren kann, sondern man muß für jeden Typ ein eigenes Sortier-Unterprogramm vereinbaren.

Wenn der Ada-Programmierer mehrere Programmeinheiten **PE1**, **PE2**, ... etc. vereinbaren muß, die alle "das Gleiche, aber mit Daten verschiedener Typen", machen, dann braucht er diese Programmeinheiten aber nicht "einzeln von Hand" zu schreiben. Statt dessen kann er eine **Schablone** S programmieren und die Programmeinheiten **PE1**, **PE2**, ... etc. als **Instanzen** von S vereinbaren. In der Schablone S kann er einen **formalen Typparameter** vorsehen, der dann bei jeder Instanziierung durch einen **aktuellen Typ** ersetzt wird. Somit kann dann jede Instanz der Schablone Daten eines anderen Typs bearbeiten.

In Ada unterscheidet man **Paketschablonen** und **Unterprogrammschablonen**, genauer: **Funktionsschablonen** und **Prozedurschablonen**. Jede Instanz einer Paketschablone ist ein **Paket**. Ganz entsprechend ist jede Instanz einer Funktionsschablone bzw. einer Prozedurschablone eine **Funktion** bzw. eine **Prozedur**.

Eine Schablone kann **Parameter** haben, die man als **generisch formale Parameter** bezeichnet, um sie deutlich von den formalen Parametern eines Unterprogramms zu unterscheiden. Wenn man eine Schablone instanziiert, muß man für jeden **generisch formalen Parameter** der Schablone einen entsprechenden **generisch aktuellen Parameter** angeben.

Man unterscheidet verschiedene **Arten** von generisch formalen Parametern:

- formale Typen (oder: Typparameter)
- formale Unterprogramme
- formale Objekte (oder: Objektparameter)
- formale Pakete (oder: Paketparameter)

Besonders wichtig sind die **Typparameter**. Sie werden im nächsten Abschnitt genauer behandelt. Formale Unterprogramme werden im Abschnitt 20.3. besprochen. Objektparameter haben große Ähnlichkeit mit den formalen Parametern eines Unterprogramms. Beispiele dazu findet man im Abschnitt 20.4..

Beim Instanziieren einer Schablone muß man für jeden **formalen Typ** der Schablone einen **aktuellen Untertyp**, für jedes **formale Objekt** ein **aktuelles Objekt** (einen Ausdruck bzw. eine

Variable), für jedes **formale Unterprogramm** ein **aktuelles Unterprogramm** und für jedes **formale Paket** ein **aktuelles Paket** angeben.

Einige Schablonen wurden in früheren Abschnitten dieses Skripts schon häufiger instanziiert. Hier zwei Beispiele, die dem Leser möglicherweise vertraut vorkommen:

```
01 package BOOL_EA is new ada.text_io.enumeration_io(enum => boolean);
02 package  INT_EA is new ada.text_io.integer_io    (num  => integer);
```

Diese Vereinbarungen kann man etwa so ins Deutsche übersetzen:
01: Erzeuge ein Paket namens **BOOL_EA** als Instanz der Schablone **enumeration_io** (die sich im Paket **ada.text_io** befindet). Dabei soll der generisch formale Typparameter **enum** durch den aktuellen Untertyp **boolean** ersetzt werden (damit man mit dem Paket **BOOL_EA** Werte des Untertyps **boolean** bearbeiten kann).
02: Erzeuge ein Paket namens **INT_EA** als Instanz der Schablone **integer_io** (die sich im Paket **ada.text_io** befindet). Dabei soll der formale Typ **num** durch den aktuellen Untertyp **integer** ersetzt werden (damit man mit dem Paket **INT_EA** Werte des Untertyps **integer** bearbeiten kann).

Anmerkung zur Terminologie: **Paketschablonen** werden im Englischen als **generic packages** und im deutschen Referenzmanual für Ada83 (Deutsche Norm DIN 66 268 vom Mai 1988) entsprechend als **generische Pakete** bezeichnet. Diese Bezeichnungen (im Englischen genauso wie im Deutschen) sind **unglücklich** gewählt, denn ein **generisches Paket** ist **kein** Paket, sondern eine ganz **andere Art** von Programmeinheit. Ganz ähnlich ist in einigen Gegenden Deutschlands ein **kalter Hund** kein **Hund**, sondern ein (hauptsächlich aus Schokolade und Keksen bestehender) **Kuchen**. Solche Bezeichnungen verstoßen gegen sehr "allgemeine und tiefliegende Regeln der Sprachlogik" und werden hier vermieden. Im folgenden werden auch solche Kuchen, die aus Schokolade und Keksen bestehen, als **Kuchen** bezeichnet, und sogenannte generische Pakete bzw. Unterprogramme als **Paketschablonen** bzw. **Unterprogrammschablonen**. **o**

Schablonen (generic units) sind sehr mächtige Konstrukte, mit denen man zahlreiche Probleme **ökonomisch** ("ein für allemal") und **sicher** lösen kann. Andererseits ist das Programmieren von Schablonen eine sehr anspruchsvolle Aufgabe. Denn während man eine Schablone entwirft und programmiert, muß man **alle Instanzen** "übersehen und berücksichtigen", die man später von dieser Schablone bilden kann. Das erfordert in aller Regel ziemlich viel Übung.

Zusammenfassung 20:
- Eine **Schablone** ist eine Programmeinheit, von der man **Instanzen** vereinbaren kann.
- Es gibt Paketschablonen, Prozedurschablonen und Funktionsschablonen.
- Eine Schablone kann generisch formale **Parameter** haben.
- Wenn man eine Schablone instanziiert, muß man für jeden generisch **formalen** Parameter einen entsprechenden generisch **aktuellen** Parameter angeben.

20.1. Schablonen mit formalen Typen

Die Werte von zwei Variablen zu vertauschen erfordert immer die gleichen Befehle, unabhängig vom Typ der beteiligten Variablen. Als erstes Beispiel in diesem Abschnitt wird eine entsprechende **Prozedurschablone** vorgestellt.

Beispiel 20.1.1.: Eine Prozedurschablone mit einem Typparameter:

```
01 generic
02    type IRGEND_EINER is private;
03 procedure SCHAB_01(DIESEN, MIT_DEM: in out IRGEND_EINER);

04 procedure SCHAB_01(DIESEN, MIT_DEM: in out IRGEND_EINER) is
05    TEMP: IRGEND_EINER := DIESEN;
06 begin
07    DIESEN  := MIT_DEM;
08    MIT_DEM := TEMP;
09 end SCHAB_01;
```

Jede Schablone besteht aus einer **Spezifikation** und einem **Rumpf**. Von dieser Regel gibt es keine Ausnahme. Hier im Beispiel steht die Spezifikation in Zeile 01 bis 03, der Rumpf in Zeile 04 bis 09.

Die Spezifikation einer Schablone beginnt immer mit dem Wort **generic** (siehe Zeile 01). Falls die Schablone generisch formale Parameter hat, müssen die unmittelbar danach vereinbart werden. Im Beispiel wird **ein** Typparameter namens **IRGEND_EINER** vereinbart (in Zeile 02). Die Angabe **is private** bedeutet (etwas vereinfacht gesagt), daß man beim Instanziieren der Schablone für den formalen Typparameter **IRGEND_EINER** fast **jeden** beliebigen Untertyp angeben darf. Genaueres hierzu weiter unten.

Die Zeile 03 sieht genauso aus, wie die Spezifikation einer Prozedur namens **SCHAB_01** mit zwei Parametern namens **DIESEN** und **MIT_DEM**. Allerdings wird diesen beiden Parametern kein "konkreter Untertyp" zugeordnet, sondern statt dessen der Typparameter **IRGEND_EINER**.

Der **Rumpf** der Prozedurschablone (in Zeile 04 bis 09) sieht genauso aus wie der Rumpf einer Prozedur. Auch in diesem Rumpf steht der Typparameter **IRGEND_EINER** da, wo sonst ein Untertyp stehen müßte (in Zeile 04 und in Zeile 05).

Die folgende Prozedur **TESTS_01** bindet die Schablone **SCHAB_01** ein und instanziiert sie dreimal:

```
01 with SCHAB_01, ada.text_io;
02 procedure TESTS_01 is
03 -- Kleines Testprogramm zum Testen der Prozedurschablone SCHAB_01.
04    procedure TAUSCHE is new SCHAB_01(IRGEND_EINER => boolean);
05    procedure SWAP    is new SCHAB_01(IRGEND_EINER => character);
06    procedure OTTO    is new SCHAB_01(IRGEND_EINER => integer);
07    B1 : boolean   := true; B2 : boolean   := false;
08    C1 : character := 'X';  C2 : character := 'Y';
09    N1 : natural   := 123;  N2 : natural   := 321;
10 begin
11    TAUSCHE(DIESEN  => B1, MIT_DEM => B2);
12    SWAP   (MIT_DEM => C1, DIESEN  => C2);
13    OTTO   (N1, N2);
...   ...
20 end TESTS_01;
```

Mit der Kontextklausel in Zeile 01 wird die Schablone **SCHAB_01** (und das Paket **ada.text_io**) eingebunden. Dadurch wird die Schablone in der Prozedur **TESTS_01** sichtbar und kann dort benützt (d.h. instanziiert) werden.

In den Zeilen 04 bis 06 werden **drei** verschiedene Prozeduren namens **TAUSCHE**, **SWAP** und **OTTO** vereinbart. Jede dieser Prozeduren ist eine **Instanz** der Schablone **SCHAB_01** und hat zwei formale Parameter namens **DIESEN** und **MIT_DEM**. Allerdings gehören diese Parameter bei jeder

Prozedur zu einem anderen (Unter-) Typ: bei der Prozedur **TAUSCHE** zum Untertyp **boolean**, bei der Prozedur **SWAP** zum Untertyp **character** und bei der Prozedur **OTTO** zum Untertyp **integer**.

In Zeile 22 werden die Werte der Variablen **B1** und **B2** vertauscht (mit Hilfe der Prozedur **TAUSCHE**). In Zeile 23 werden die Werte der Variablen **C1** und **C2** vertauscht (mit Hilfe der Prozedur **SWAP**). In Zeile 24 werden die Werte der Variablen **N1** und **N2** vertauscht (mit Hilfe der Prozedur **OTTO**). Man beachte, daß die Variablen **N1** und **N2** zum Untertyp **natural** gehören. Da **natural** und **integer** Untertypen desselben Typs **!integer** sind und alle Werte des Untertyps **natural** auch zum Untertyp **integer** gehören, geht auch diese letzte Vertauschung immer gut. **o**

Im folgenden Beispiel wird eine **Funktionsschablone** für Potenzierungsfunktionen vorgestellt. Zwar bekommt man zu jedem Ganzzahltyp, den man vereinbart, automatisch einen Potenzierungsoperator namens **"**"** dazu. Aber dessen **zweiter** Parameter (der Exponent) muß immer zum Untertyp **natural** gehören, unabhängig davon, zu welchem Untertyp der erste Parameter gehört. Dagegen haben die Instanzen der Funktionsschablone **SCHAB_02** zwei Parameter vom **gleichen** Untertyp.

Beispiel 20.1.2.: Eine **Funktionsschablone** mit einem **Typparameter**:

```
01 generic
02    type FOGAT is range <>; -- Formaler Ganzzahltyp
03 function SCHAB_02(ZAHL, HOCH: FOGAT) return FOGAT;
04 -- Jede Instanz dieser Schablone potenziert die ZAHL mit der Hochzahl HOCH.
05 -- Wenn HOCH kleiner als 1 ist, wird 1 als Ergebnis geliefert. Zum Untertyp
06 -- FOGAT muessen mindestens die beiden Zahlen 0 und 1 gehoeren.
```

Dies ist nur die **Spezifikation** der Funktionsschablone **SCHAB_02**. Die Schablone hat einen Typparameter namens **FOGAT** (wie "formaler Ganzzahltyp"). Aus der Angabe **is range** <> (in Zeile 02) folgt, daß beim Instanziieren dieser Schablone als aktueller Parameter für **FOGAT** nur ein **signierter Ganzzahluntertyp** angegeben werden darf. Aus dem Kommentar der Spezifikation geht hervor, daß zu diesem signierten Ganzzahluntertyp auf jeden Fall die Werte 0 und 1 gehören müssen.

Aufgabe 20.1.1.: Programmieren Sie einen passenden **Rumpf** für die Schablone **SCHAB_02**. Zur Erinnerung: Der Rumpf einer **Funktionsschablone** sieht genauso aus, wie der Rumpf einer **Funktion**. **o**

Das folgende Testprogramm **TESTS_02** bindet die Schablone **SCHAB_02** ein, instanziiert sie zweimal und ruft die Instanzfunktionen ein paarmal auf:

```
01 with SCHAB_02, ada.text_io;
02 procedure TESTS_02 is
03 -- Kleines Testprogramm zum Testen der Funktionsschablone SCHAB_02.
04    type GANZ is range -5_000..+10_000;
05    function POTENZ is new SCHAB_02(FOGAT => GANZ);
06    function "**"   is new SCHAB_02(FOGAT => integer);
07    GP : GANZ;    -- eine    GANZ-Potenz
08    IP : integer; -- eine integer-Potenz
09 begin
10    for G in GANZ range -2..+5 loop
11       GP := POTENZ(ZAHL => -5, HOCH => G);
12       ada.text_io.put_line(item => GANZ'image(GP));
```

```
13    end loop;
14    ada.text_io.new_line;
15    for H in integer range -2..+5 loop
16       IP := (-5) ** H;
17       ada.text_io.put_line(item => integer'image(IP));
18    end loop;
19 end TESTS_02;
```

In Zeile 05 wird eine Funktion namens **POTENZ** als Instanz der Schablone **SCHAB_02** vereinbart. Mit der Funktion **POTENZ** kann man Werte des Untertyps **GANZ** potenzieren. In Zeile 06 wird eine Funktion namens **"**"** als Instanz der Schablone **SCHAB_02** vereinbart. Mit dieser Funktion **"**"** kann man Werte des Untertyps **integer** potenzieren.

Die beiden Instanzfunktionen **POTENZ** bzw. **"**"** werden in Zeile 11 bzw. 16 aufgerufen. **o**

Aufgabe 20.1.2.: Führen Sie das Programm **TESTS_02** mit Papier und Bleistift aus und geben Sie an, welche Zahlen von diesem Programm zur aktuellen Ausgabe ausgegeben werden (es sind insgesamt 16 Zahlen). **o**

Im folgenden Beispiel wird eine **Paketschablone** mit einem Typparameter vorgestellt.

Beispiel 20.1.3.: Eine **Paketschablone** mit einem **Typparameter**:

```
01 generic
02    type FODISK is (<>);
03 package SCHAB_03 is
04 ------------------------------------------------------------------------
05 -- Eine Instanz dieser Schablone stellt ihren Benutzern zwei Operationen
06 -- zur Erleichterung der Ausgabe von FODISK-Werten zur Verfuegung.
07 ------------------------------------------------------------------------
08    function "&"(L: string; R: FODISK) return string;
09    -- Wandelt R in einen String RS um und liefert L & RS.
10    function "&"(L: FODISK; R: string) return string;
11    -- Wandelt L in einen String LS um und liefert LS & R.
12 end SCHAB_03;
```

Auch von der Paketschablone **SCHAB_03** wurde hier nur die **Spezifikation** wiedergegeben (den Rumpf soll der Leser selbst entwickeln). Die Schablone hat einen Typparameter namens **FODISK** (wie "formaler diskreter Typ"). Aus der Angabe **is (<>)** in Zeile 02 folgt, daß man beim Instanzieren der Schablone für **FODISK** nur einen **diskreten** Untertyp angeben darf. **o**

Aufgabe 20.1.3.: Programmieren Sie einen passenden **Rumpf** für die Paketschablone **SCHAB_03**. Zur Erinnerung: Der Rumpf einer **Paketschablone** sieht genauso aus wie der Rumpf eines **Paketes**. Der Rumpf der Paketschablone **SCHAB_03** muß die Rümpfe der beiden Funktionen namens **"&"** enthalten, die in der Spezifikation der Schablone spezifiziert wurden (in Zeile 08 bzw. 10). **Hinweis**: Diese Aufgabe hat große Ähnlichkeit mit der **Aufgabe 19.1.5. o**

Aufgabe 20.1.4.: Schreiben Sie ein Testprogramm namens **TESTS_03**, welches die Schablone **SCHAB_03** einbindet und benützt. In diesem Testprogramm soll die Schablone **SCHAB_03** min-

destens **zweimal** instanziiert werden, einmal mit einem **Aufzählungsuntertyp** für **FODISK** und einmal mit einem **Ganzzahluntertyp** für **FODISK**. Die Funktionen namens **"&"** in den Instanzpaketen sollen ein paarmal aufgerufen und ihre Ergebnisse zur aktuellen Ausgabe ausgegeben werden. **o**

Die Schablonen **SCHAB_01** bis **SCHAB_03** in den obigen Beispielen haben je einen **formalen Typ**. Für jeden dieser formalen Typen wird in seiner Vereinbarung (jeweils am Anfang der Schablone, nach dem Wort **generic**) eine bestimmte **Typklasse** festgelegt (durch eine Angabe wie **is private** oder **is range** <> oder **is (<>)**). Das Festlegen dieser Typklasse ist ein wichtiger Schritt, der Konsequenzen in zwei Richtungen hat:

Die **Typklasse** eines formalen Typs legt fest,
1. welche **aktuellen Untertypen** man beim Instanziieren der Schablone für ihn angeben darf
2. **was man** mit dem formalen Typ innerhalb der Schablone **machen darf**.

Für die formalen Typen der Schablonen **SCHAB_01** bis **SCHAB_03** gilt:

Name der Schablone	**Name des formalen Typs**	**Welche aktuellen Untertypen darf man beim Instanziieren der Schablone angeben?**
SCHAB_01	IRGEND_EINER	(Fast) jeden Untertyp
SCHAB_02	FOGAT	Jeden signierten Ganzzahluntertyp
SCHAB_03	FODISK	Jeden diskreten Untertyp

Innerhalb einer Schablone **repräsentiert** ein **formaler Typ** alle **aktuellen Untertypen**, die man (beim Instanziieren der Schablone) für ihn angeben darf. Mit dem formalen Typ darf man in der Schablone **nur das machen**, was man mit **jedem** dieser aktuellen Untertypen, die er repräsentiert, auch machen dürfte.

Z.B. repräsentiert der formalen Typ **FOGAT** in der Schablone **SCHAB_02** alle **signierten Ganzzahluntertypen**. Also darf man innerhalb von **SCHAB_02** mit dem formalen Typ **FOGAT** unter anderem Variablen vereinbaren und Operationen wie **"+"**, **"-"**, **"*"**, "/" etc. auf diese Variablen anwenden, denn diese Operationen stehen für alle signierten Ganzzahluntertypen zur Verfügung.

Der formale Typ **FODISK** repräsentiert (in der Schablone **SCHAB_03**) alle **diskreten Untertypen**. Also darf man innerhalb von **SCHAB_03** mit dem formalen Typ **FODISK** unter anderem Variablen vereinbaren, aber man darf **nicht** die Operationen **"+"**, **"-"**, **"*"**, "/" auf diese Variablen anwenden, denn diese Operationen stehen **nicht** für alle diskreten Untertypen zur Verfügung. Man darf auf **FODISK**-Variablen aber die Funktionen **FODISK'succ** und **FODISK'pred** anwenden, die für jeden diskreten Untertyp zur Verfügung stehen.

Der formale Typ **IRGEND_EINER** repräsentiert (in der Schablone **SCHAB_01**) alle Untertypen, für die mindestens die folgenden **drei Befehle** zur Verfügung stehen: Die Zuweisungsanweisung **":="** und die beiden Vergleichsoperationen **"="** und **"/="**. Das trifft auf sehr viele Untertypen zu, sogar auf solche, die in der Spezifikation eines Paketes als **private** Untertypen vereinbart wurden. Innerhalb der Schablone **SCHAB_01** darf man mit dem formalen Typ **IRGEND_EINER** Variablen vereinbaren, aber man darf auf diese Variablen nur die Befehle ":=", "=" und "/=" anwenden.

Mit diesen Beispielen sollte deutlich gemacht werden: Je **größer** ("umfangreicher") die Typklasse eines formalen Typs ist, desto **weniger** darf man in der betreffenden Schablone mit dem formalen

Typ (und seinen Variablen etc.) machen. Je **kleiner** und spezieller die Typklasse eines formalen Typs ist, desto **mehr** darf man mit dem formalen Typ (und seinen Variablen etc.) machen.

Um ehrlich zu sein: Die obige Diskussion des formalen Typs **IRGEND_EINER** ist ein bißchen vereinfacht und unvollständig. Tatsächlich repräsentiert **IRGEND_EINER** nur alle **definiten** ("als Baupläne vollständigen") Untertypen, für die mindestens die drei Befehle ":=", "=" und "/=" zur Verfügung stehen. Ein uneingeschränkter Reihungsuntertyp wie z.B. **string** ist **nicht** definit und darf deshalb (beim Instanziieren der Schablone **SCHAB_01**) nicht für **IRGEND_EINER** angegeben werden. Nur weil der formale Typ **IRGEND_EINER** ausdrücklich nur **definite** Untertypen repräsentiert, darf man in der Schablone **SCHAB_01** Variablen mit ihm vereinbaren, etwa so:

```
07    TEMP: IRGEND_EINER;
```

Zur Erinnerung: Eine Variablenvereinbarung wie etwa

```
08    TEMP: string;
```

ist **nicht** erlaubt, weil der Untertyp **string** als Bauplan unvollständig ist und (ohne ergänzende Indexangaben) die Größe der Variablen TEMP **nicht** festlegt. **o**

Die folgende Tabelle beschreibt **alle Typklassen**, die man für einen **formalen Typ** einer Schablone angeben kann. Die erste Spalte enthält die konkrete Syntax, mit der man die betreffende Typklasse bei der Vereinbarung eines formalen Typs festlegt. Die zweite Spalte enthält eine kurze Beschreibung der Typklasse. Einige der Typklassen werden erst später (im Abschnitt XXX) besprochen.

Vereinbarung eines generisch **formalen** Typparameters	Welche Untertypen darf man als **aktuellen** Parameter für den formalen Typparameter angeben?
`type T1(<>) is limited private;`	**Jeden** Untertypen
`type T2     is limited private;`	Jeden **definiten** Untertypen
`type T3(<>) is private;`	Jeden Untertyp, für den wenigstens die drei Befehle ":=", "=" und "/=" zur Verfügung stehen.
`type T4     is private;`	Jeden **definiten** Untertyp, für den wenigstens die Befehle ":=", "=" und "/=" zur Verfügung stehen.
`type T5  is range <>;`	Jeden **siginierten Ganzzahl**untertyp
`type T6  is mod <>;`	Jeden **modularen Ganzzahl**untertyp
`type T7  is (<>);`	Jeden **diskreten** Untertyp
`type T8  is digits <>;`	Jeden **Gleitpunkt**untertyp
`type T9  is delta <> digits <>;`	Jeden **dezimalen Festpunkt**untertyp
`type T10 is delta <>;`	Jeden **gewöhnlichen Festpunkt**untertyp
`type T11 is array(IT) of KT;`	Jeden **eingeschränkten Reihungsuntertyp** mit Indexuntertyp IT und Komponentenuntertyp KT
`type T12 is array(IT range <>) of KT;`	Jeden **uneingeschränkten Reihungsuntertyp** mit Indexuntertyp IT und Komponentenuntertyp KT
`type T13 is access ZT;`	Jeden **Zeigeruntertyp**, dessen Werte auf Objekte des Untertyps ZT zeigen

Aufgabe 20.1.5.: Schauen Sie sich im (ARM A.10.1(79)) die Spezifikation der Paketschablone **enumeration_io** an. Diese Schablone hat einen Typparameter namens **enum**. Welche Untertypen darf man für **enum** angeben, wenn man die Schablone instanziiert? Inwiefern ist der Name **enumeration_io** für diese Schablone irreführend? Schlagen Sie einen **besseren Namen** für die Schablone vor. **o**

Aufgabe 20.1.6.: Häufig braucht man für die Werte eines Aufzählungsuntertyps eine "zyklische Nachfolgerfunktion", die, wenn man sie auf den **größten** Wert des Untertyps anwendet, als Ergebnis den **kleinsten** Wert des Untertyps liefert (z.B. ist der zyklische Nachfolger von **MONTAG** der **SONNTAG**). Programmieren Sie eine entsprechende Funktionsschablone namens **SCHAB_04**. **o**

Das folgende Beispiel stellt eine Prozedurschablone mit **mehreren** formalen Typparametern vor. Einer der Parameter ist ein formaler **Reihungstyp**.

Beispiel 20.1.4.: Eine Prozedurschablone mit **drei** formalen Typen:

```
01 generic
02    type   INDX is (<>);
03    type   KOMP is (<>);
04    type   REIH is array(INDX range <>) of KOMP;
05 procedure SCHAB_05(R: in out REIH);
06 -- Jede Instanz dieser Schablone sortiert R aufsteigend.
```

Auch von dieser Schablone ist hier nur die **Spezifikation** wiedergegeben (siehe unten Aufgabe 20.1.6.). Die Spezifikation legt fest, daß man beim Instanziieren der Schablone **drei** aktuelle Untertypen angeben muß, und zwar zwei **diskrete** Untertypen (für **INDX** und **KOMP**) und einen **uneingeschränkten Reihungsuntertyp** (für **REIH**). Außerdem darf man für **REIH** nicht irgendeinen uneingeschränkten Reihungsuntertyp angeben, sondern nur einen solchen, dessen Index- bzw. Komponentenuntertyp mit dem für **INDX** bzw. **KOMP** angegebenen Untertyp übereinstimmt.

Hier ein kleines Testprogramm, in dem die Schablone **SCHAB_05** benützt wird:

```
01 with SCHAB_05, ada.text_io;
02 procedure TESTS_05 is
03 -- Kleines Testprogramm zum Testen der Prozedurschablone SCHAB_05.
04    type TABELLE is array(character range <>) of natural;
05    -- Erzeuge 2 Sortierprozeduren als Instanzen von SCHAB_05:
06    procedure SORT1 is new SCHAB_05(INDX => character, KOMP => natural,
07                                    REIH => TABELLE);
08    procedure SORT2 is new SCHAB_05(INDX => positive,  KOMP => character,
09                                    REIH => string);
10    TAB1: TABELLE('A'..'E') := (17, 35, 9, 12, 3); -- Wird sortiert.
11    STR1: string (123..130) := "BARBARA!";         -- Wird sortiert.
12 begin
13    SORT1(R => TAB1); -- TAB1 sortieren
14    SORT2(R => STR1); -- STR1 sortieren
...    ...
22 end TESTS_05;
```

In den Zeilen 06 bis 09 werden zwei Prozedur namens **SORT1 SORT2** als Instanzen der Schablone **SCHAB_05** vereinbart. Mit der Prozedur **SORT1** kann man Reihungen des Untertyps **TABELLE** sortieren. Mit der Prozedur **SORT2** kann man die Zeichen eines Strings sortieren. In Zeile 13 bzw. 14 wird die Reihung **TAB1** bzw. der String **STR1** sortiert. **o**

Aufgabe 20.1.7.: Programmieren Sie einen passenden Rumpf für die Schablone **SCHAB_05**. Legen Sie dabei ein möglichst leicht zu programmierendes Sortierverfahren zu Grunde, z.B. das Verfahren "Sortieren durch Vertauschen von benachbarten Komponenten" (bubblesort). **o**

Die Schablone **SCHAB_05** ist nicht so "abstrakt und allgemein", wie man sich das eigentlich wünschen würde. Mit Instanzen dieser Schablone kann man nur solche Reihungen sortieren, die einen **diskreten** Komponentenuntertyp haben. Das schließt z.B. Reihungen mit einem **Verbunduntertyp** als Komponentenuntertyp aus. Die Einschränkung des formalen Typs **KOMP** auf **diskrete** Untertypen wurde vorgenommen, weil im Rumpf der Schablone eine **Vergleichsfunktion** wie "<" benötigt wird und eine solche Funktion für jeden diskreten Untertyp zur Verfügung steht. Hätte man dem formalen Typ **KOMP** eine größere Typklasse zugeordnet, etwa so:

```
03 type KOMP is private;
```

würde **KOMP** unter anderem auch Verbunduntertypen repräsentieren, für die **keine** Vergleichsoperation "<" zur Verfügung steht. Entsprechend dürfte man im Rumpf der Schablone **KOMP**-Größen dann auch nicht mit "<" vergleichen. Eine "allgemeinere" Prozedurschablone, mit deren Instanzen man **beliebige Reihungen** sortieren kann, wird unten im **Beispiel 20.3.2.** im Zusammenhang mit **formalen Unterprogrammen** vorgestellt.

Zusammenfassung 20.1.:

- Am Anfang einer Schablone kann man unter anderem **formale Typen** (oder: **Typparameter**) vereinbaren.
- Jedem formalen Typ muß in seiner Vereinbarung eine **Typklasse** zugeordnet werden.
- Der formale Typ **repräsentiert** innerhalb der Schablone alle Untertypen dieser Typklasse.
- Innerhalb der Schablone darf man mit einem formalen Typ nur das machen, was man auch mit jedem Untertyp machen dürfte, den er repräsentiert.
- Je **größer** (bzw. kleiner) die Typklasse eines formalen Typs ist, desto **weniger** (bzw. mehr) darf man innerhalb der Schablone mit dem formalen Typ machen.

20.2. Schablonen ohne formale Parameter

Eine **Unterprogramm**schablone **ohne** generisch formale Parameter ist kaum sinnvoll, weil mehrere Instanzen nicht mehr leisten würden als eine. Statt eine **Schablone** sollte man in einem solchen Fall eher ein **Unterprogramm** schreiben. Eine **Paket**schablone ohne generisch formale Parameter kann dagegeben sinnvoll sein, wenn die Instanzpakete **Variablen** enthalten: **Zwei** Instanzen enthalten dann "doppelt soviel Variablen" wie **eine** Instanz.

Das folgende Beispiel baut auf dem **Beispiel 19.1.1.** auf. Dort wurde ein Paket namens **PAKET_01** vorgestellt, welches einen **Stapel** (für 100 Zeichen vom Untertp **character**) als **abstrakte Variable** realisiert. Aus dem **Paket** namens **PAKET_01** kann man mit ganz geringfügigen Veränderungen eine **Schablone** machen, wie das folgende Beispiel deutlich machen soll:

Beispiel 20.2.1.: Eine Paketschablone **ohne** generisch formale Parameter:

```
01 generic
02 package SCHAB_06 is
...   ------------------------------------------------------------------------
...   -- Alles ganz genau so wie im PAKET_01
```

```
...   ------------------------------------------------------------------
26 end SCHAB_06;
```

Die Schablone **SCHAB_06** und das Paket **PAKET_01** unterscheiden sich nur durch das Wort **generic** (am Anfang der Spezifikation der Schablone) und durch ihre Namen voneinander. Insbesondere stimmen der Rumpf der **Schablone** und der Rumpf des **Paketes** (bis auf die Namen) vollständig überein.

Das folgende Testprogramm bindet die Schablone **SCHAB_06** ein, instanziiert sie einmal und benützt das Instanzpaket zum Bearbeiten eines kleinen Textes:

```
01 with SCHAB_06, ada.text_io;
02 procedure TESTS_06 is
03 -- Minimales Testprogramm zum Testen der Paketschablone SCHAB_06.
04    package PAKET_06 is new SCHAB_06;
05    TEXT1: string := "?se theg eiw ,ollaH";
06    TEXT2: string(1..TEXT1'length);
07 begin
08    for I in positive range TEXT1'range loop
09       PAKET_06.PUSH(ZEICHEN => TEXT1(I));
10    end loop;
11    for I in positive range 1..PAKET_06.ANZAHL loop
12       TEXT2(I) := PAKET_06.TOP;
13       PAKET_06.POP;
14    end loop;
15    ada.text_io.put_line(item => TEXT2);
16 end TESTS_06;
```

In Zeile 04 wird ein Paket namens **PAKET_01** als Instanz der Schablone **SCHAB_06** vereinbart. Dieses Paket **PAKET_01** hat weitgehend die gleichen Eigenschaften wie das gleichnamige Paket im Beispiel 19.1.1. (einziger Unterschied: dort ist **PAKET_01** ein **Bibliothekspaket**, hier ist **PAKET_01** ein in der Prozedur TESTS_05 **enthaltenes** Paket). **o**

Es kann sinnvoll sein, in einem Programm **mehrere** Instanzen der Schablone **SCHAB_06** zu vereinbaren, weil jede Instanz eine eigene Stapelvariable mit Platz für 100 Zeichen hat.

Aufgabe 20.2.1.: Schreiben Sie ein Programm namens **BEN1S_06** ("Benutzer 1 der Schablone **SCHAB_06**"), in dem **zwei** Instanzen der Schablone **SCHAB_06** vereinbart werden. Das Programm soll eine (beliebig lange) Zeichenkette **einlesen** (maximal 100 Zeichen), die einzelnen Zeichen der Zeichenkette auf den **ersten** Stapel legen, sie dann wieder herunterholen und auf den **zweiten** Stapel legen und sie dann auch von da wieder runterholen und **ausgeben**. Da die Reihenfolge der Zeichen **zweimal** umgedreht wird, müßten sie dadurch in der gleichen Reihenfolge auf dem Bildschirm erscheinen, in der sie eingegeben wurden. **o**

20.3. Schablonen mit formalen Unterprogrammen

Eine Schablone kann nicht nur formale Typen (Typparameter) besitzen, sondern auch **formale Unterprogramme**, d.h. **formale Funktionen** und **formale Prozeduren**. Beim Instanziieren einer Schablone muß man für jedes **formale** Unterprogramm ein entsprechendes **aktuelles** Unterprogramm angeben.

Wenn bestimmte "Anschlußbedingungen" erfüllt sind, kann man zwei Funktionen zu einer neuen Funktion **komponieren**. Im folgenden Beispiel wird eine **Funktionsschablone** vorgestellt, mit der man bestimmte Funktionen komponieren kann.

Beispiel 20.3.1.: Eine Funktionsschablone mit zwei formalen Funktionen:

```
01 generic
02    with function F1(S1: string) return string;
03    with function F2(S2: string) return string;
04 function SCHAB_07(S: string) return string;
05 -- Eine Instanz dieser Schablone ist die Komposition der Parameter-
06 -- funktionen F1 und F2. Angewendet auf einen Wert S liefert diese
07 -- Kompositionsfunktion den Wert F2(F1(S)).

08 function SCHAB_07(S: string) return string is
09 begin
10    return F2(F1(S));
11 end SCHAB_07;
```

Die **Spezifikation** dieser Funktionsschablone steht in Zeile 01 bis 06, ihr **Rumpf** in Zeile 07 bis 10. Die Schablone hat zwei **formale Funktionen** namens **F1** und **F2** (siehe Zeile 02 und 03). Wenn man eine Instanz dieser Schablone vereinbart, muß man entsprechend zwei Funktionen angeben, die je einen Parameter vom Untertyp **string** haben und ein Ergebnis von diesem Untertyp liefern. Das Wort **with** in Zeile 02 (bzw. in Zeile 03) ist notwendig, damit der Ausführer die Zeile 02 (bzw. die Zeile 03) von der Zeile 04 unterscheiden kann. Dieses **with** kennzeichnet also ein **formales Unterprogramm** und hat nichts mit dem **with** einer **Kontextklausel** zu tun.

Das folgende Testprogramm bindet die Schablone **SCHAB_07** ein und instanziiert sie mehrmals:

```
01 with SCHAB_07, ada.text_io;
02 procedure TESTS_07 is
03 -- Kleines Testprogramm zum Testen der Schablone SCHAB_07:
04    ------------------------------------------------------------------------
05    function A_TO_B(TEXT: string) return string is
06    -- Liefert eine Kopie von TEXT, in der alle 'A's durch 'B's ersetzt sind.
...    ...
15    end A_TO_B;
16    ------------------------------------------------------------------------
17    function GROSS(TEXT: string) return string is
18    -- Liefert eine Kopie von TEXT, in der jeder kleine Buchstabe durch den
19    -- entsprechenden grossen Buchstaben ersetzt wurde.
...    ...
29    end GROSS;
30    ------------------------------------------------------------------------
31    function FILET(TEXT: string) return string is
32    -- Liefert TEXT ohne das erste und ohne das letzte Zeichen.
...    ...
35    end FILET;
36    ------------------------------------------------------------------------
37    function K1 is new SCHAB_07(F1 => GROSS, F2 => A_TO_B);
38    function K2 is new SCHAB_07(F1 => FILET, F2 => K1);
39 begin -- TESTS_07
40    ada.text_io.put_line(item => K1(S => "rababbal"));
41    ada.text_io.put_line(item => K2(S => "rababbal"));
42 end TESTS_07;
```

Die drei Funktionen **A_TO_B**, **GROSS** und **FILET** dienen dazu, Strings zu bearbeiten. Jede dieser Funktionen hat einen Parameter vom Untertyp **string** und liefert ein Ergebnis vom gleichen Untertyp.

In Zeile 37 wird eine Funktion namens **K1** als Instanz der Schablone **SCHAB_07** vereinbart. Die Funktion **K1** ist die **Komposition** der Funktionen **GROSS** und **A_TO_B**, d.h. sie "kombiniert die Effekte der Funktionen **GROSS** und **A_TO_B**". **K1** hat einen Parameter namens **S** vom Untertyp **string** und liefert ein Ergebnis vom Untertyp **string**.

Die in Zeile 38 vereinbarte Funktion **K2** ist eine Komposition der Funktionen **FILET** und **K1**, d.h. sie "kombiniert die Effekte der Funktionen **GROSS**, **A_TO_B** und **FILET**".

Aufgabe 20.3.1.: Führen Sie das Programm **TESTS_07** mit Papier und Bleistift aus. Welche Zeichenketten werden durch die **put_line**-Befehle in Zeile 40 und 41 zur aktuellen Ausgabe ausgegeben? **o**

Das folgende Beispiel stellt eine Prozedurschablone **SCHAB_08** vor, mit deren Instanzen man **beliebige Reihungen** wahweise auf- oder absteigend **sortieren** kann. **SCHAB_08** ist eine "verallgemeinerte Version" der Schablone **SCHAB_05** (siehe oben **Beispiel 20.1.4.**) und hat als zusätzlichen Parameter eine **formale Funktion** namens **SIND_OK**. Mit der entsprechenden aktuellen Funktion sollte man von zwei der zu sortierenden Komponenten feststellen können, ob sie schon in der richtigen Reihenfolge liegen oder nicht. Gibt man beim Instanziieren der Schablone für **SIND_OK** eine "kleiner-oder-gleich-Funktion" wie z.B. "<=" an, wird **aufsteigend** sortiert. Wenn man für **SIND_OK** eine "größer-oder-gleich-Funktion" wie z.B. ">=" angibt, wird **absteigend** sortiert.

Beispiel 20.3.2.: Eine Prozedurschablone zum Sortieren **beliebiger Reihungen**:

```
01 generic
02    type INDX is (<>);
03    type KOMP is private;
04    type REIH is array(INDX range <>) of KOMP;
05    with function SIND_OK(K1, K2: KOMP) return boolean;
06 procedure SCHAB_08(R: in out REIH);
07 -- Jede Instanz dieser Schablone sortiert die Komponenten der Reihung R so,
08 -- dass die Funktion SIND_OK fuer je zwei nebeneinanderliegende Komponenten
09 -- den Wert true liefert.

10 procedure SCHAB_08(R: in out REIH) is
11    ALLE_OK : boolean := false; -- Bleibt true, wenn alle sortiert sind.
12    TEMP    : KOMP;             -- Zum vertauschen zweier Komponenten.
13    I2      : INDX;             -- Fuer den Nachfolger von I1 (s.u.)
14 begin
15    while not ALLE_OK loop
16       ALLE_OK := true; -- Optimistische Annahme.
17       for I1 in INDX range R'first..INDX'pred(R'last) loop
18          I2 := INDX'succ(I1);  -- "+1" wuerde nicht immer gehen!
19          if not SIND_OK(R(I1), R(I2)) then
20             TEMP    := R(I1);   -- R(I1) und R(I2) vertauschen
21             R(I1)   := R(I2);
22             R(I2)   := TEMP;
23             ALLE_OK := false;   -- Die Optimistische Annahme war falsch!
24          end if;
25       end loop;  -- Ende der   for-Schleife
26    end loop;     -- Ende der while-Schleife
27 end SCHAB_08;
```

Hier wurde sowohl die **Spezifikation** der Schablone (in Zeile 01 bis 08) also auch ihr **Rumpf** (in Zeile 09 bis 28) vollständig angegeben.

Wenn die Variable **ALLE_OK** nach einer Ausführung der **for**-Schleife in Zeile 16 bis 24 noch den Wert **true** hat, dann ist die Reihung **R** fertig sortiert.

In Zeile 16 kann man vor dem Wort **loop** nicht einfach **R'last-1** schreiben, weil **R'last** zu **irgend-einem** diskreten Untertyp **INDX** gehören kann und für einige dieser Untertypen (nämlich für die **Aufzählungsuntertypen**) keine Operation namens **"-"** zur Verfügung steht. Dagegen gibt es die Funktion **INDX'pred** für **jeden** diskreten Untertyp **INDX**. Entsprechend kann man in Zeile 17 nicht einfach **I+1**, sondern muß **INDX'succ(I)** schreiben.

Das folgende kleine Testprogramm bindet die Schablone **SCHAB_08** ein und instanziiert sie zweimal:

```
01 with SCHAB_08, ada.text_io;
02 procedure TESTS_08 is
03 -- Kleines Testprogramm zum Testen der Prozedurschablone SCHAB_04.
04    type ANSCHRIFT is record
05       NAME : string(1..4) := (others => '?');
06       NR   : integer      := 0;
07    end record;
08    type TABELLE is array(natural range <>) of ANSCHRIFT;
09    ------------------------------------------------------------------------
10    function AUF_NACH_NAME(AN1, AN2: ANSCHRIFT) return boolean is
11    begin
12       return AN1.NAME <= AN2.NAME;
13    end AUF_NACH_NAME;
14    ------------------------------------------------------------------------
15    function AB_NACH_NR(AN1, AN2: ANSCHRIFT) return boolean is
16    begin
17       return AN1.NR >= AN2.NR;
18    end AB_NACH_NR;
19    ------------------------------------------------------------------------
20    procedure SORT_AUF_NAME is new SCHAB_08(        INDX    => natural,
21                                                    KOMP    => ANSCHRIFT,
22                                                    REIH    => TABELLE,
23                                                    SIND_OK => AUF_NACH_NAME);
24    procedure SORT_AB_NR    is new SCHAB_08(        INDX    => natural,
25                                                    KOMP    => ANSCHRIFT,
26                                                    REIH    => TABELLE,
27                                                    SIND_OK => AB_NACH_NR);
28    ------------------------------------------------------------------------
...    ...
40    TAB1 : TABELLE(1..3) := (("OTTO", 17), ("EMIL", 35), ("KARL", -23));
41 begin -- TESTS_08
...    ...
43    SORT_AUF_NAME(R    => TAB1);
...    ...
45    SORT_AB_NR   (R    => TAB1);
...    ...
47 end TESTS_08;
```

Reihungen des Untertyps **TABELLE** (vereinbart in Zeile 08) haben Indizes vom Untertyp **natural** und Komponenten vom Verbunduntertyp **ANSCHRIFT**. Die Reihung **TAB1** (siehe Zeile 40) besteht aus drei **ANSCHRIFT**-Komponenten.

Die Funktion **AUF_NACH_NAME** (siehe Zeile 10 bis 13) ist eine "kleiner-oder-gleich-Funktion". Sie prüft, ob zwei Anschriften entsprechend ihrer **NAME**-Komponente **aufsteigend** sortiert sind. Die Funktion **AB_NACH_NR** (siehe 15 bis 18) ist eine "größer-oder-gleich-Funktion". Sie prüft, ob zwei Anschriften entsprechend ihrer **NR**-Komponente **absteigend** sortiert sind.

In Zeile 20 bis 27 werden **zwei** Instanzen der Prozedurschablone **SCHAB_08** vereinbart. Die beiden Instanziierungen unterscheiden sich nur durch die **aktuelle Funktion**, die jeweils für die formale Funktion **SIND_OK** angegeben wird. Mit der Prozedur **SORT_AUF_NAME** kann man Reihungen des Untertyps **TABELLE** sortieren, und zwar **aufsteigend** nach **Namen** (NAME). Mit der Prozedur **SORT_AB_NR** kann man solche Reihungen **absteigend** nach **Nummern** (NR) sortieren.

In Zeile 35 wird die Reihung **TAB1** aufsteigend nach Namen sortiert. In Zeile 37 wird dieselbe Reihung absteigend nach Nummern sortiert. **o**

Um ehrlich zu sein: Mithilfe der Schablone **SCHAB_08** kann man nicht **alle beliebigen** Reihungen sortieren, sondern nur **eindimensionale** Reihungen, die zu einem **uneingeschränkten** Reihungsuntertyp gehören. Man könnte aber leicht ähnliche Schablonen für eingeschränkte Reihungsuntertypen schreiben oder für zweidimensionale oder für dreidimensionale etc.. Man kann mit Schablonen allerdings nicht von der Eigenschaften **eingeschränkt/uneingeschränkt** bzw. von der Dimensionalität der Reihungsuntertypen abstrahieren, d.h.. man kann nicht **eine** Schablone programmieren, die sowohl für **eingeschränkte** wie auch für **uneingeschränkte** Reihungsuntertypen funktioniert oder für Reihungsuntertypen mit verschieden vielen Dimensionen. **o**

Aufgabe 20.3.2.: Erweitern Sie die Prozedur **TESTS_08** zu einer Prozedur **BEN1S_08** so, daß die Reihung **TAB1** auch **absteigend** nach **Namen** und dann **aufsteigend** nach **Nummern** sortiert wird. Vereinbaren Sie dazu zwei geeignete Vergleichsfunktionen namens **AB_NACH_NAME** und **AUF_NACH_NR** (ähnlich wie **AUF_NACH_NAME** und **AB_NACH_NR**) und instanziieren Sie damit die Schablone **SCHAB_08** (ähnlich wie in Zeile 20 bis 27). **o**

Aufgabe 20.3.3.: Die Paketschablonen **integer_io** und **enumeration_io** unterstützen die Ein-/Ausgabe beliebiger Ganzzahlwerte bzw. beliebiger diskreter Werte. Es gibt aber keine Schablone **array_io**, die die Ein-/Ausgabe **beliebiger** Reihungswerte unterstützt. Schreiben Sie selbst eine solche Schablone. Statt **array_io** soll sie hier **SCHAB_09** genannt werden und folgende Spezifikation haben:

```
01 generic
02    type INDX is (<>);
03    type KOMP is private;
04    type REIH is array(INDX range <>) of KOMP;
05    with procedure GET(ITEM: out KOMP);
06    with procedure PUT(ITEM: in  KOMP);
07 package SCHAB_09 is
08    procedure GET(ITEM: out REIH);
09    -- Liest von der aktuellen Eingabe fuer jede Komponente der Reihung ITEM
10    -- einen entsprechenden Wert ein.
11    procedure PUT(ITEM: in  REIH);
12    -- Gibt die Komponenten der Reihung ITEM in einer von Menschen lesbaren
13    -- Form zur aktuellen Ausgabe aus.
14 end SCHAB_09;
```

Diese Schablone erwartet als aktuelle Parameter unter anderem zwei Prozeduren namens **GET** und **PUT** (siehe Zeile 05 bis 06), mit denen man **einzelne Komponenten** einer Reihung einlesen bzw. ausgeben kann. Sie macht daraus zwei Prozeduren namens **GET** und **PUT** (siehe Zeile 08 bzw. 11), mit denen man jeweils eine ganze **Reihung** einlesen bzw. ausgeben kann. Programmieren Sie den **Rumpf** der Schablone **SCHAB_09** und ein kleines Testprogramm namens **TESTS_09**, welches die Schablone **SCHAB_09** einbindet und benützt. **o**

Zwei Funktionen mit je einem Parameter kann man schon dann **komponieren**, wenn der **Ergebnisuntertyp** der einen Funktion mit dem **Parameteruntertyp** der anderen Funktion übereinstimmt.

Mit der folgenden Funktionsschablone kann man alle Funktionen komponieren, bei denen das der Fall ist:

Beispiel 20.3.3.: Eine Funktionsschablone mit drei formalen Typen und zwei formalen Funktionen:

```
01 generic
02    type ANY1(<>) is limited private;
03    type ANY2(<>) is limited private;
04    type ANY3(<>) is limited private;
05    with function F1(P1: ANY1) return ANY2;
06    with function F2(P2: ANY2) return ANY3;
07 function SCHAB_10(P: ANY1) return ANY3;
08 -- Jede Instanz dieser Schablone ist die Komposition der Funktionen F1 und
09 -- F2 und liefert, angewendet auf P, den Wert F2(F1(P)).

10 function SCHAB_10(P: ANY1) return ANY3 is
11 begin
12    return F2(F1(P));
13 end SCHAB_10;
```

Für jeden der formalen Typen **ANY1**, **ANY2** und **ANY3** darf man (beim Instanziieren der Schablone) **jeden beliebigen Untertyp** angeben, auch einen **indefiniten** Untertyp wie z.B. **string** oder einen Untertyp, für den noch nicht einmal die Befehle **":="**, **"="** und **"/="** zur Verfügung stehen (d.h. einen Untertyp, der in der Spezifikation eines Paketes als **limitiert privater** Untertyp vereinbart wurde). **o**

Aufgabe 20.3.4.: Schreiben Sie ein kleines Testprogramm namens **TESTS_10**, welches die Schablone **SCHAB_10** einbindet und mindestens zweimal mit verschiedenen (generisch aktuellen) Parametern instanziiert. Die Instanzfunktionen sollen ein paar Mal aufgerufen und ihre Ergebnisse zur aktuellen Ausgabe ausgegeben werden. **o**

Zusammenfassung

- Eine Schablone kann auch **formale Unterprogramme** als (generisch formale) Parameter haben.
- Beim Instanziieren einer Schablone muß man für jedes **formale** Unterprogramm ein entsprechendes **aktuelles** Unterprogramm angeben.

20.4. Schablonen mit formalen Objekten

Außer formalen Typen und formalen Unterprogrammen kann eine Schablone auch formale Objekte als Parameter haben. **Generisch formale Objektparameter** einer Schablone haben große Ähnlichkeit mit den formalen Parametern eines Unterprogramms. Insbesondere hat jeder generisch formale Objektparameter einen **Untertyp** und einen **Modus** (**in**, **out** oder **in out**). Beim Instanziieren einer Schablone darf man für jeden **in**-Parameter einen beliebigen **Ausdruck** des betreffenden Untertyps angeben. Für einen **out**- oder **in-out**-Parameter darf man dagegen nur eine **Variable** angeben.

Das folgende Beispiel beruht, ähnlich wie das **Beispiel 20.2.1.**, auf dem Paket **PAKET_01** aus dem **Beispiel 19.1.1.**, welches einen **Stapel** realisiert. Die Schablone **SCHAB_10** erlaubt es dem Instanziierer, sowohl die Größe des Stapels als auch den Untertyp der Stapelelemente zu bestimmen.

Beispiel 20.4.1.: Eine Schablone für Stapelpakete, bei der man die **Stapelgröße** und den **Untertyp der Stapelelemente** wählen kann:

```
01 generic
02    MAX_ANZ : natural;            -- Groesse des Stapels
03    type E_TYP is private;        -- Untertyp der Stapelelemente
04 package SCHAB_11 is
05 -- Jede Instanz dieser Paketschablone dient zum Verwalten eines Stapels
06 -- von Elementen (vom Typ E_TYP). Der Stapel selbst ist im Rumpf
07 -- des Instanz-Paketes verborgen und bietet Platz fuer MAX_ANZ Elemente.
08 ------------------------------------------------------------------------
09    procedure PUSH(ELEMENT: E_TYP);
...    ...
13    procedure POP;
...    ...
17    function TOP return E_TYP;
...    ...
22    function ANZAHL return natural;
...    ...
26    UEBERLAUF: exception; -- Wird evtl. von PUSH ausgeloest.
27    UNTERLAUF: exception; -- Wird evtl. von POP oder TOP ausgeloest.
28 end SCHAB_11;

29 package body SCHAB_11 is
30    type TABELLE is array(positive range <>) of E_TYP;
31    STAPEL: TABELLE(1..MAX_ANZ);
32    LBI   : natural := STAPEL'first-1; -- Letzter belegter Index des STAPELs.
33                                       -- Zeigt anfangs "vor den STAPEL".
34    ------------------------------------------------------------------------
35    procedure PUSH(ELEMENT: E_TYP) is
36    begin
37       if LBI = STAPEL'last then
38          raise UEBERLAUF;
39       else
40          LBI         := LBI + 1;
41          STAPEL(LBI) := ELEMENT;
42       end if;
43    end PUSH;
44    ------------------------------------------------------------------------
...   -- Ganz entsprechend die Rümpfe der uebrigen Unterprogramme
..    -- POP, TOP und ANZAHL.
66    ------------------------------------------------------------------------
67 end SCHAB_11;
```

Neu ist hier eigentlich nur der **formale Objektparameter** namens **MAX_ANZ** vom Untertyp **natural**, der in Zeile 02 vereinbart wird. Da nicht ausdrücklich ein anderer Modus angegeben wurde, hat dieser Objektparamter den Modus **in**. Beim Instanziieren der Schablone muß man für den Objektparameter **MAX_ANZ** einen **Ausdruck** angeben, der einen Wert des Untertyps **natural** beschreibt, etwa so:

```
01 with SCHAB_11, ada.text_io;
02 procedure TESTS_11 is
03 -- Minimales Testprogramm zum Testen der Paketschablone SCHAB_11.
04    package PAKET_01 is new SCHAB_11(E_TYP => character, MAX_ANZ => 100);
05    TEXT1: string := "?se theg eiw ,ollaH";
06    TEXT2: string(1..TEXT1'length);
07 begin
08    for I in positive range TEXT1'range loop
09       PAKET_01.PUSH(ELEMENT => TEXT1(I));
10    end loop;
11    for I in positive range 1..PAKET_01.ANZAHL loop
12       TEXT2(I) := PAKET_01.TOP;
13       PAKET_01.POP;
14    end loop;
15    ada.text_io.put_line(item => TEXT2);
16 end TESTS_11;
```

In Zeile 04 wird ein **PAKET_01** als Instanz der Schablone **SCHAB_11** vereinbart. Dabei wird für den formalen Typ **E_TYP** der aktuelle Untertyp **character** und für den formalen Objektparameter **MAX_ANZ** der aktuelle Wert **100** angegeben. Das **PAKET_01** enthält somit einen Stapel mit Platz für **100** Elemente vom Untertyp **character**. Das Programm **TESTS_11** gibt die Meldung **"Hallo, wie geht es?"** zur aktuellen Ausgabe aus. **o**

Aufgabe 20.4.1.: Angenommen, Sie brauchen einen Stapel für **50** Elemente des Untertyps **boolean** und einen Stapel für **2000** Elemente des Untertyps **integer**. Vereinbaren Sie entsprechende Pakete namens **BOOLEAN_STAPEL** und **INTEGER_STAPEL** als Instanzen der Schablone **SCHAB_11**. **o**

Die folgende Schablone **SCHAB_12** hilft einem dabei, aus einer Funktion mit **zwei** Parametern und einem konkreten Wert für den einen Parameter eine Funktion mit nur **einem** Parameter zu erzeugen.

Beispiel 20.4.2.: Eine Schablone mit einem **formalen Typ**, einer **formalen Funktion** und einem **formalen Objektparameter**:

```
01 generic
02    type ANY(<>) is private;                       -- Irgend ein Typ
03    with function F(P1, P2: ANY) return ANY;       -- Eine zweistellige Funktion
04    K : ANY;                                       -- Ein ANY-Wert
05 function SCHAB_12(P: ANY) return ANY;
06 -- Eine Instanz dieser Funktionsschablone angewendet auf einen Wert P
07 -- liefert als Ergebnis den Wert F(P, K).

08 function SCHAB_12(P: ANY) return ANY is
09 begin
10    return F(P, K);
11 end SCHAB_12;
```

Der formale Typ **ANY** ist notwendig, um auszudrücken, daß der Objektparameter **K** zum gleichen Untertyp gehören muß, wie die beiden Parameter der Funktion **F** und das Ergebnis dieser Funktion. Man beachte, daß für den Objektparamter **K** in diesem Beispiel kein "konkreter Untertyp" wie etwa **integer**, **boolean** oder **string** etc. festgelegt wird, sondern nur der formale Typ **ANY**. Hier ein kleines Testprogramm:

```
01 with SCHAB_12, ada.text_io;
02 procedure TESTS_12 is
03 -- Minimales Testprogramm zum Testen der Schablone SCHAB_12
04    function KONES         is new SCHAB_12(ANY=>string,  F=>"&", K=>"es");
05    function PLUS3         is new SCHAB_12(ANY=>integer, F=>"+", K=>3);
06    function MAL17         is new SCHAB_12(ANY=>integer, F=>"*", K=>17);
07    package  INTEGER_EA    is new ada.text_io.integer_io(num => integer);
08    T1 : constant string := "Hallo, wie geht ";
09 begin
10    ada.text_io.put(item => "KONES(T1) ist gleich ");
11    ada.text_io.put(item =>  KONES(T1));
12    ada.text_io.new_line;
13    ada.text_io.put(item => "PLUS3(15) ist gleich ");
14    INTEGER_EA .put(item =>  PLUS3(15), width => 0);
15    ada.text_io.new_line;
16    ada.text_io.put(item => "MAL17(-3) ist gleich ");
17    INTEGER_EA .put(item =>  MAL17(-3), width => 0);
18    ada.text_io.new_line;
19 end TESTS_12;
```

Die Funktionen **KONES** ("konkateniere mit **es**"), **PLUS3** und **MAL17** sind drei Instanzen der Schablone **SCHAB_12**. Die einstellige Funktion **KONES** wird dabei aus der zweistelligen Kon-

katenationsfunktion **"&"** und dem Wert **"es"** erzeugt (siehe Zeile 04). Ist **S** irgendein **String**, dann liefert der Funktionsaufruf **KONES(S)** den gleichen Wert, wie der Ausdruck **S & "es"**.

Ganz entsprechend wird die einstellige Funktion **PLUS3** aus der zweistelligen Funktion **"+"** und dem Wert **3** erzeugt (siehe Zeile 05). Ist **N** irgendein Wert des Untertyps **integer**, dann liefert der Funktionsaufruf **PLUS3(N)** den gleichen Wert wie der Ausdruck **N + 3**.

Die einstellige Funktion **MAL17** wird aus der zweistelligen Funktion "*" und dem Wert **17** erzeugt (siehe Zeile 06). Ist **N** irgendein Wert des Untertyps **integer**, dann liefert der Funktionsaufruf **MAL17(N)** den gleichen Wert wie der Ausdruck **N * 17**. **o**

Aufgabe 20.4.2.: Führen Sie das Programm **TESTS_12** mit Papier und Bleistift aus, und geben Sie an, welche Zeichenketten das Programm zur aktuellen Ausgabe ausgibt. **o**

Aufgabe 20.4.3.: Am Anfang der Schablone **SCHAB_12** wird ein formaler Typ namens **ANY** vereinbart (siehe oben Zeile 02). Seine Typklasse wird durch die Angabe **is private** festgelegt. Schreibt man statt dessen **is limited private**, lehnt der Ausführer die Schablone ab. Mit welcher Begründung? Fragen Sie Ihren maschinellen Ada-Ausführer. **o**

Zusammenfassung 20.4.:

- Eine Schablone kann auch **formale Objekte** als Parameter haben.
- Jedes formale Objekt gehört zu einem **Untertyp** und hat einen **Modus** (**in**, **out** oder **in out**)
- Für einen **in**-Parameter darf man beim Instanziieren einen beliebigen **Ausdruck** des betreffenden Untertyps angeben, für einen **out**- oder **in-out**-Parameter muß man eine **Variable** angeben.

20.5. Schablonen mit formalen Paketen

In speziellen Fällen braucht man innerhalb einer Schablone **S** eine Instanz einer bestimmten Paketschablone **PS**. Mithilfe eines **formalen Paketes** (vereinbart am Anfang der Schablone **S**) kann man verlangen, daß beim Instanziieren von **S** eine Instanz von **PS** als aktueller Parameter angegeben werden muß.

Beispiel 20.5.1.: Eine Prozedurschablone mit einem formalen Typ und zwei formalen Paketen:

```
01 with ada.text_io;
02 generic
03    type FODISK is (<>);
04    with package FOPAK1 is new ada.text_io.enumeration_io(enum => FODISK);
05    with package FOPAK2 is new ada.text_io.integer_io(<>);
06 procedure SCHAB_13(ITEM: FODISK);

07 procedure SCHAB_13(ITEM: FODISK) is
08 begin
09    FOPAK1.put     (item => ITEM);
10    ada.text_io.put(item => " Position: ");
11    FOPAK2.put     (item => FODISK'pos(ITEM), width => 0);
12    ada.text_io.new_line;
13 end SCHAB_13;
```

Das formale Paket **FOPAK1** (siehe Zeile 04) repräsentiert alle Instanzen der Paketschablone **enumeration_io**, die mit dem für **FODISK** angegebenen Untertyp als **enum**-Parameter erzeugt wurden. Das formale Paket **FOPAK2** (siehe Zeile 05) **alle** Instanzen der Paketschablone **integer_io**, unabhängig davon, mit welchem Untertyp als **num**-Parameter sie erzeugt wurden. Das

folgt aus der Angabe **(<>)** am Ende der Zeile 05. Beim Instanziieren der Schablone **SCHAB_13** muß man als **generisch aktuelle Parameter** also folgende Größen angeben:

1. Für den formalen Typ **FODISK** irgendeinen diskreten Untertyp **DT** .
2. Für das formale Paket **FOPAK1** eine Instanz der Paketschablone **enumeration_io**, die mit dem Untertyp **DT** als **enum**-Parmater erzeugt wurde.
3. Für das formale Paket **FOPAK2** eine Instanz der Paketschablone **integer_io**, die mit irgendeinem beliebigen (Ganzzahl-) Untertyp als **num**-Parameter erzeugt wurde.

Was das konkret bedeutet, soll das folgende Testprogramm deutlich machen:

```
01 with SCHAB_13, ada.text_io;
02 procedure TESTS_13 is
03 -- Minimales Testprogramm zum Testen der Schablone SCHAB_13.
04    package CHAR_EA is new ada.text_io.enumeration_io(enum => character);
05    package  NAT_EA is new ada.text_io.integer_io    ( num => natural);
06    ------------------------------------------------------------------------
07    procedure PUT    is new SCHAB_13(FODISK => character,
08                                     FOPAK1 => CHAR_EA,
09                                     FOPAK2 =>  NAT_EA);
10    ------------------------------------------------------------------------
11 begin
12    for BUCHSTABE in character range 'a'..'f' loop
13       PUT(ITEM => BUCHSTABE);
14    end loop;
15 end TESTS_13;
```

In Zeile 04 wird die Paketschablone **enumeration_io** (die sich bekanntlich im Paket **ada.text_io** befindet) mit dem diskreten Untertyp **character** als **enum**-Parameter instanziiert. Das Instanzpaket wird **CHAR_EA** genannt. In Zeile 05 wird die Paketschablone integer_io mit dem Ganzzahluntertyp natural instanziiert und das Instanzpaket wird **NAT_EA** genannt.

In den Zeilen 07 bis 09 wird eine Prozedur namens **PUT** als Instanz der Schablone **SCHAB_13** vereinbart. Als generisch aktuelle Parameter werden dabei der diskrete Untertyp **character** und die beiden Pakete **CHAR_EA** und **NAT_EA** angegeben. Diese Prozedur **PUT** sieht also, so kann man sich zumindest vorstellen, folgendermaßen aus:

```
07 procedure PUT(ITEM: character) is
08 begin
09    CHAR_EA.put    (item => ITEM);
10    ada.text_io.put(item => " Position: ");
11    NAT_EA.put     (item => character'pos(ITEM), width => 0);
12    ada.text_io.new_line;
13 end PUT;
```

Der Ausdruck **character'pos(ITEM)** in Zeile 11 liefert einen Wert des Typs **!univeral_integer**. Weil dieser Wert als **item**-Parameter der Prozedur **NAT_EA.put** verwendet wird, wird er automatisch in einen Wert des Untertyps **natural** umgewandelt (und dann ausgegeben).

Das Programm **TESTS_13** gibt folgende Zeilen zur aktuellen Ausgabe aus:

```
'a' Position: 97
'b' Position: 98
'c' Position: 99
'd' Position: 100
'e' Position: 101
'f' Position: 102
```

Man hätte die Schablone auch **ohne** formale Pakete programmieren können, etwa so:

```
01 generic
02    type FODISK is (<>);
03 procedure SCHAB_14(ITEM: FODISK);

04 with ada.text_io;
05 procedure SCHAB_14(ITEM: FODISK) is
06    package PAK1 is new ada.text_io.enumeration_io(enum => FODISK);
07    package PAK2 is new ada.text_io.integer_io    ( num => natural);
08 begin
09    PAK1.put       (item => ITEM);
10    ada.text_io.put(item => " Position: ");
11    PAK2.put       (item => FODISK'pos(ITEM), width => 0);
12    ada.text_io.new_line;
13 end SCHAB_14;
```

Bei dieser Varianten "verlangt" die Schablone keine Pakete als aktuelle Parameter, sondern "vereinbart sie selber" (in Zeile 06 und 07). In einigen Fällen würden dadurch aber unnötig viele Pakete erzeugt (nämlich dann, wenn der Benutzer der Schablone solche Pakete noch an anderen Stellen benötigt und sie noch mal vereinbart). In anderen Fällen ist es nicht möglich, die benötigten Pakete in der Schablone zu vereinbaren, weil dort die dazu nötigen Parameter nicht sichtbar sind. Auch das kann eine Motivation dafür sein, eine Schablone mit **formalen Paketen** zu versehen.

Zusammenfassung 20.5.:

- Eine Schablone kann auch **formale Pakete** als generisch formale Parameter haben.
- Ein formales Paket repräsentiert entweder **alle** Instanzen einer Paketschablone oder **all die** Instanzen, die mit bestimmten Parametern erzeugt wurden.
- Beim Instanziieren einer Schablone muß man für jedes **formale** Paket ein entsprechendes **aktuelles** Paket angeben.
- Formale Pakete sind in einer Schablone **S** dann sinnvoll, wenn man die betreffenden Instanzpakete in **S** nicht vereinbaren **kann** (weil die dazu nötigen Parameter nicht sichtbar sind) oder weil man sie nicht vereinbaren **möchte** (um die Erzeugung von "unnötig vielen Paketen" zu vermeiden).

20.6. Schablonen kombinieren

Mithilfe der Schablone **SCHAB_08** (siehe oben Beispiel 20.3.2.) kann man (fast) beliebige Reihungen **sortieren**. Mithilfe der Schablone **SCHAB_09** kann man (fast) beliebige Reihungen **einlesen** und **ausgeben**. Wenn man Reihungen eines bestimmten Untertyps sortieren **und** einlesen/ausgeben will, muß man beide Schablonen instanziieren und dabei zum Teil die **gleichen** Parameter angeben. Das ist mühsam und fehlerträchtig. Statt dessen kan man die beiden Schablonen auch zu **einer** Schablone kombinieren, so daß man nur noch diese eine Schablone instanziieren und dabei jeden Parameter nur noch einmal angeben muß. Im folgenden Beispiel werden die beiden Schablonen **SCHAB_08** und **SCHAB_09** zu einer Schablone **SCHAB_15** kombiniert.

Beispiel 20.6.1.: Schablonen kombinieren:

```
01 generic
02    type INDX is (<>);
03    type KOMP is private;
04    type REIH is array(INDX range <>) of KOMP;
05    with function SIND_OK(K1, K2: KOMP) return boolean;
06    with procedure GET(ITEM: out KOMP);
07    with procedure PUT(ITEM: in  KOMP);
```

```
...    -- Kommentare zu den generisch formalen Parametern
14 package SCHAB_15 is
15    procedure SORT(R:    in out REIH);
16    procedure GET (ITEM:    out REIH);
17    procedure PUT (ITEM: in     REIH);
...   -- Kommentare zu den Prozeduren SORT, GET und PUT.
24 end SCHAB_15;

25 with SCHAB_08, SCHAB_09, ada.text_io;
26 package body SCHAB_15 is
27    procedure XXXXX is new SCHAB_08(        INDX    => INDX,
28                                            KOMP    => KOMP,
29                                            REIH    => REIH,
30                                            SIND_OK => SIND_OK);
31    package REIH_EA is new SCHAB_09(        INDX    => INDX,
32                                            KOMP    => KOMP,
33                                            REIH    => REIH,
34                                            GET     => GET,
35                                            PUT     => PUT);
36    procedure SORT(R:    in out REIH) renames XXXXX;
37    procedure GET (ITEM:    out REIH) renames REIH_EA.GET;
38    procedure PUT (ITEM: in     REIH) renames REIH_EA.PUT;
39 end SCHAB_15;
```

In Zeile 27 bis 30 wird eine Prozedur namens **XXXXX** als Instanz der Schablone **SCHAB_08** vereinbart. Dabei werden einige der formalen Parameter der Schablone **SCHAB_15** (nämlich **INDX**, **KOMP**, **REIH** und **SIND_OK**) sozusagen "an die Schablone **SCHAB_08** weitergereicht". In Zeile 36 wird die Prozedur **XXXXX** in **SORT** umbenannt und festgelegt, daß ihr **in-out**-Parameter den Namen **R** tragen soll. Damit ist die in Zeile 15 spezifizierte Prozedur namens **SORT** vollständig vereinbart. Die **Umbenennung** in Zeile 36 dient als ihr **Rumpf**.

Entsprechend werden auch für die in Zeile 16 und 17 spezifizierten Prozeduren **GET** und **PUT** keine "von Hand programmierten Rümpfe" angegeben, sondern entsprechende **Umbenennungen** (in Zeile 37 bzw. 38).

Wenn man die Schablone **SCHAB_15** instanziiert, stehen einem drei Prozeduren namens **SORT**, **GET** und **PUT** zur Verfügung, mit denen man Reihungen eines bestimmten Untertyps sortieren, einlesen und ausgeben kann. Einen ganz ähnlichen Effekt könnte man auch dadurch erreichen, daß man die beiden Schablonen **SCHAB_08** und **SCHAB_09** instanziiert. In diesem Sinne kombiniert **SCHAB15** die Schablonen **SCHAB_08** und **SCHAB_09**.

Um **SCHAB_08** zu instanziieren, muß man **4** Parameter angeben (3 Untertypen und eine Funktion). Um **SCHAB_09** zu instanziieren, muß man **5** Parameter angeben (3 Untertypen und 2 Prozeduren). Um beiden Schablonen zu instanziieren muß man also **9** Parameter angeben (6 Untertypen und 3 Unterprogramme). Um **SCHAB_15** zu instanziieren, braucht man nur **6** Parameter angeben (3 Untertypen und 3 Unterprogramme). Noch wichtiger ist: Innerhalb von **SCHAB_15** werden die Schablonen **SCHAB_08** und **SCHAB_09** garantiert mit den gleichen Untertypen **INDX**, **KOMP** und **REIH** instanziiert.

Im folgenden Testprogramm wird die Schablone **SCHAB_15** zweimal instanziiert. Mit den beiden Instanzpaketen werden zwei Reihungen (verschiedener Typen) eingelesen, sortiert und wieder ausgegeben:

```
01 with SCHAB_15, ada.text_io;
02 procedure TESTS_15 is
03 -- Kleines Testprogramm zum Testen der Schablone SCHAB_15.
04    type FARBE is (ROT, GRUEN, BLAU);
05    type GANZ  is range 0..99;
```

```
06    type REIH1 is array(FARBE range <>) of GANZ;
07    type REIH2 is array(GANZ  range <>) of FARBE;
08    ------------------------------------------------------------------------
09    package FARBE_EA is new ada.text_io.enumeration_io(enum => FARBE);
10    package  GANZ_EA is new ada.text_io.integer_io(num => GANZ);
11    -- Die folgenden GET- und PUT-Prozeduren muessen vereinbart werden, weil
12    -- die Prozeduren FARBE_EA.get, FARBE_EA.put, GANZ_EA.get und GANZ_EA.put
13    -- mehr als einen Parameter haben, SCHAB_15 aber einstellige Prozeduren
14    -- fuer die formalen Prozeduren GET und PUT erwartet:
15    procedure GET_F(ITEM: out FARBE) is
16    begin
17       FARBE_EA.get(item => ITEM);
18    end GET_F;
19    --------------------------------
20    procedure PUT_F(ITEM: in FARBE) is
21    begin
22       FARBE_EA.put(item => ITEM, width => FARBE'width);
23    end PUT_F;
24    --------------------------------
25    procedure GET_G(ITEM: out GANZ) is
26    begin
27       GANZ_EA.get(item => ITEM);
28    end GET_G;
29    --------------------------------
30    procedure PUT_G(ITEM: GANZ) is
31    begin
32       GANZ_EA.put(item => ITEM);
33    end PUT_G;
34    ------------------------------------------------------------------------
35    package REIH1_SEA is new SCHAB_15(         INDX    => FARBE,
36                                               KOMP    => GANZ,
37                                               REIH    => REIH1,
38                                               SIND_OK => "<=",    -- aufsteigend
39                                               GET     => GET_G,
40                                               PUT     => PUT_G);
41    package REIH2_SEA is new SCHAB_15(         INDX    => GANZ,
42                                               KOMP    => FARBE,
43                                               REIH    => REIH2,
44                                               SIND_OK => ">=",    -- absteigend
45                                               GET     => GET_F, FARBE
46                                               PUT     => PUT_F);
47    ------------------------------------------------------------------------
48    R1 : REIH1(ROT..BLAU); -- Wird eingelesen, sortiert und ausgegeben
49    R2 : REIH2(17..20);    -- Wird eingelesen, sortiert und ausgegeben
50 begin
51    ada.text_io.put_line("Bitte geben Sie Ganzzahlen (0..99) ein:");
52    REIH1_SEA.GET (ITEM => R1); -- Reihung R1 einlesen
53    REIH1_SEA.SORT(R    => R1); -- Reihung R1 sortieren
54    REIH1_SEA.PUT (ITEM => R1); -- Reihung R1 ausgeben
55    ada.text_io.put_line("Bitte geben Sie ROT, GRUEN oder BLAU ein:");
56    REIH2_SEA.GET (ITEM => R2); -- Reihung R2 einlesen
57    REIH2_SEA.SORT(R    => R2); -- Reihung R2 sortieren
58    REIH2_SEA.PUT (ITEM => R2); -- Reihung R2 ausgeben
59 end TESTS_15;
```

Das Paket **REIH1_SEA** (vereinbart in Zeile 35 bis 40 als Instanz der Schablone **SCHAB_15**) enthält drei Prozeduren namens **SORT**, **GET** und **PUT**, mit denen man Reihungen des Untertyps **REIH1** (aufsteigend) sortieren, einlesen bzw. ausgeben kann.

Das Paket **REIH2_SEA** (vereinbart in Zeile 41 bis 46 als Instanz der Schablone **SCHAB_15**) enthält drei Prozeduren namens **SORT**, **GET** und **PUT**, mit denen man Reihungen des Untertyps **REIH2** (absteigend) sortieren, einlesen bzw. ausgeben kann. **o**

Wenn ein Schablonen-Instanziierer Reihungen eines bestimmten Untertyps nur **sortieren** will, kann er die Schablone **SCHAB_08** instanziieren. Wenn er seine Reihungen nur **einlesen und ausgeben** will, kann er dazu **SCHAB_09** benutzen. Wenn er beides will, sollte er die Schablone **SCHAB_15**

instanziieren. Auf diese Weise kann man ganze Hierarchien von Schablonen programmieren und dem Instanziierer für viele Zwecke eine passende Schablone als Werkzeug zur Verfügung stellen.

Zusammenfassung 20.6.:

- In einer Schablone kann man andere Schablonen **einbinden** und **instanziieren**.
- So kann eine Schablone mehrere andere Schablonen **kombinieren**.
- Durch wiederholtes Kombinieren kann man ganz **Hierarchien** von Schablonen erstellen.

Zentraldokument: Nach Filialdokument A95-20.20

21. Bruchzahltypen

In diesem Abschnitt sollen drei weitere Klassen von Typen vorgestellt werden: **Dezimale Festpunkttypen**, **gewöhnliche Festpunkttypen** und **Gleitpunkttypen**. Zusammen bilden diese Typen die Klasse der **Bruchzahltypen** und zusammen mit den **Ganzzahltypen** die Klasse der **numerischen Typen**:

Numerische Typen
- Ganzzahltypen
  - Signierte ("vorzeichenbehaftete") Ganzzahltypen
  - Modulare Ganzzahltypen
- Bruchzahltypen
  - Festpunkttypen
    - Dezimale Festpunkttypen
    - Gewöhnliche Festpunkttypen
  - Gleitpunkttypen

Anmerkung: **Bruchzahltypen** werden im Englischen offiziell als **real types** und im Deutschen entsprechend als **reelle Typen** bezeichnet. Diese Bezeichnung wird hier vermieden, da sie eine Art "mathematische Hochstapelei" darstellt und ein genaueres Verständnis dieser Typen eher erschwert. **o**

In Unterabschnitten dieses Abschnitts wird anhand von Beispielen gezeigt, wie man dezimale Festpunkttypen, gewöhnliche Festpunkttypen bzw. Gleitpunkttypen vereinbart und es wird erläutert, worauf man beim Rechnen mit dezimalen Festpunktzahlen, gewöhnlichen Festpunktzahlen bzw. Gleitpunktzahlen achten muß.

Zuvor sollen in dieser Einleitung ein paar Grundlagen behandelt werden, die für ein genaueres Verständnis von Bruchzahlen und Bruchzahltypen nützlich sind.

Bruchzahltypen wurden aus **zwei Gründen** in Programmiersprachen aufgenommen:

1. Um das **Bearbeiten von Bruchzahlen** (d.h. das Einlesen, Ausgeben und Berechnen von Bruchzahlen) zu unterstützen und zu erleichtern.
2. Um die **Kosten** bestimmter Berechnungen **zu senken**.

Von diesen beiden Gründen ist der **zweite** wesentlich wichtiger als der erste.

Berechnungen mit Bruchzahlen kann man in vielen Fällen ganz leicht durch **Berechnungen mit Ganzzahlen** ersetzen. Statt mit DM-Beträgen und 2 Stellen nach dem Komma kann man einfach mit ganzen Pfennigbeträgen rechnen. Statt mit der Einheit Kilogramm und 6 Stellen nach dem Komma kann man mit ganzen Milligramm rechnen etc.. Nur beim **Einlesen** und **Ausgaben** von Zahlen muß man dann an der richtigen Stelle ein Dezimalkomma (bzw. einen Dezimalpunkt) einfügen. Festpunkttypen realisieren diese Idee: Sie stellen Bruchzahlen (intern im Rechner) durch Ganzzahlen dar.

Anmerkung: Statt mit einem **Komma** (wie im Deutschen übliche) werden wir Bruchzahlen im folgenden stets mit einem **Punkt** notieren (wie im Englischen und in Ada üblich). **o**

Bevor man eine umfangreiche Berechnung von einem Computer ausführen läßt, muß man häufig die **Kosten** dieser Berechnung abschätzen. Diese Kosten beschreibt man meist durch zwei Größen: durch die benötigte **Rechenzeit** und die Größe des benötigten **Speichers**.

Viele auf den ersten Blick ganz harmlos erscheinende Berechnungen lassen sich nicht durchführen, weil sie zu teuer sind. Ein bekanntes Beispiel für eine sehr "teure" Funktion ist die **Fakultätsfunktion** (die Fakultät einer natürlichen Zahl **n** ist das Produkt aller natürlichen Zahlen von **1** bis **n**). Die Fakultät der Zahl **20** ist gleich **243_290_200_817_664_000** (rund **243** Billiarden). Die Fakultät von **100** ist eine Zahl, die aus mehr als **150** Ziffern besteht und die Fakultät von **1_000** besteht aus mehr als **2_500** Ziffern. Um die Fakultät von **1_000_000** zu berechnen braucht auch ein schneller PC ziemlich lange.

Aufgabe 21.1.: Besorgen Sie sich einen **Taschenrechner** und stellen Sie fest, für welche natürliche Zahlen **n** man damit den Wert der **Fakultätsfunktion** berechnen kann. **o**

Ebenfalls ziemlich teuer ist das **wiederholte Quadrieren** einer Zahl. Dabei beginnt man mit einer (ganzen oder gebrochenen) Zahl **z**. Bei jedem Schritt ersetzt man **z** durch $\mathbf{z^2}$. Wenn man **z** wie üblich als Dezimalbruch (oder als Binärbruch) darstellt, besteht $\mathbf{z^2}$ im allgemeinen aus doppelt so vielen Ziffern wie **z**. Führt man 10 Quadrierungsschritte nacheinander aus, so ist das Ergebnis etwa 1_000 mal so lang wie die Zahl, von der man ausging. Nach 20 Schritten ist das Ergebnis etwa 1_000_000 mal so lang und nach 30 Schritten etwa 1_000_000_000 mal so lang wie die Ausgangszahl. Um eine Zahl abzuspeichern, die aus 1_000_000_000 Dezimalziffern besteht, braucht man etwa 1 Gigabyte Speicher. Auch ein schneller PC braucht ziemlich lange, um eine solch lange Zahl zu quadrieren.

Aufgabe 21.2.: Besorgen Sie sich einen **Taschenrechner** und stellen Sie fest, wie oft man damit z.B. die Zahl **9.0** (oder die Zahl **5.5**) **wiederholt quadrieren** kann. **o**

Schon lange vor der Erfindung von Computern, als man Berechnungen noch mit Papier und Bleistift oder mit Hilfe von mechanischen Rechenmaschinen durchführte, haben Mathematiker entdeckt, wie man die **Kosten** vieler Berechnungen **drastisch senken** kann: Indem man auf die **mathematische Exaktheit** des Ergnisses **verzichtet** und sich mit einer **Näherungslösung** begnügt.

Z.B. kann man beim Berechnen der Fakultät von 20 auf die letzten 15 Ziffern des Ergebnisses verzichten und sich mit den ersten drei Ziffern begnügen: 243 Billiarden. Diese Näherung ist zwar um ein paar Billionen falsch, aber dieser Fehler macht weniger als 1 % der Zahl aus. Näherungslösungen mit einem so geringen relativen ("prozentualen") Fehler sind für viele praktische Anwendungen noch akzeptabel und brauchbar.

Das Beispiel der Fakultätsfunktion soll deutlich machen: **Näherungslösungen** sind nicht nur bei Zahlen "mit vielen Stellen nach dem Punkt" nützlich, sondern auch beim Rechnen mit **ganzen** Zahlen. Beim Berechnen von intergalakischen Entfernungen kommt es häufig nicht auf ein paar Milliarden Kilometer an und beim Planen des deutschen Bundeshaushaltes wäre es unsinnig, sich um Beträge unter 10_000 DM zu kümmern. Bruchzahltypen wurden in erster Linie eingeführt, um die **Kosten** von Berechnungen (an Speicherplatz und Rechenzeit) zu **senken**. Nur in zweiter Linie sollen sie die Bearbeitung von **Bruchzahlen** erleichtern.

Ganzzahltypen könnte man auch als "numerische Typen zur Durchführung **mathematisch exakter** Berechnungen" bezeichnen, und im Gegensatz dazu **Bruchzahltypen** als "numerische Typen zur

Berechnung von **Näherungslösungen**". Vermutlich werden sich diese Bezeichnungen aber nicht verbreiten.

Dem Vorteil von Bruchzahltypen, die Kosten von Berechnungen zu senken, stehen zwei **Nachteile** gegenüber:

1. Das Ergebnis einer Berechnung mit Werten eines Bruchzahltyps ist im allgemeinen nicht mathematisch exakt, sondern nur eine **Näherungslösung** (z.B. 243 Billiarden statt der exakten Lösung 243_290_200_817_664_000).
2. "**Wie gut**" diese Näherungslösung ist oder ob sie für eine bestimmte Anwendung "**noch gut genug**" ist, muß der Programmierer durch eine sorgfältige **mathematische Analyse** seines Programms feststellen.

In der Praxis werden häufig Berechnungen mit Gleitpunktzahlen programmiert, ohne daß der Programmierer eine mathematische Fehlerabschätzung durchführt. Solange auf diese Weise nur die Bilder eines Videospiels berechnet werden, mag eine solche "Einsparung" noch akzeptabel sein. In anderen Fällen ist sie gefährlich. Das Beispielprogramm **BRUCH_11** soll deutlich machen, daß eine Näherungslösung in bestimmten Fällen ohne weiteres um den Faktor **1_000** oder mehr vom mathematisch exakten Ergebnis abweichen kann.

Aufgabe 21.3.: Übergeben Sie das Programm **BRUCH_11** einem maschinellen Ada-Ausführer und lassen Sie es (eventuell mehrfach) ausführen. Während Sie sich die Ausgaben des Programms ansehen (in der am weitesten rechts stehenden Spalte müßte eigentlich immer die Zahl 1.0 ausgegeben werden, nähere Erläuterungen findet man im Programmtext), sollten Sie sich vorstellen, in einem Jumbo-Jet zu sitzen, welches gerade bei dichtem Nebel auf dem alten Flugplatz von Hongkong landet, natürlich vollautomatisch mit Hilfe komplizierter Bruchzahlrechnungen seiner Computer. **o**

Empfehlung:

1. Wenn möglich, sollte man Berechnungen so programmieren, daß sie **mathematisch exakt** durchgeführt werden.
2. Wenn das nicht möglich ist, sollte man eine genaue **Fehlerabschätzung** durchführen, ehe man das Programm einem Benutzer zur Verfügung stellt.
3. Wenn man die für eine Fehlerabschätzung nötigen Vorkenntnisse nicht besitzt, sollte man keine Bruchzahltypen verwenden und entsprechende Programmieraufgaben anderen Programmierern überlassen.
4. Ausnahmen sind nur solche Programme, bei denen falsche Ergebnisse keinen ernst zu nehmenden Schaden anrichten können.

Aufgabe 21.4.: Übergeben Sie das Beispielprogramm **BRUCH_10** einem maschinellen Ada-Ausführer und lassen Sie es mehrfach ausführen. Das Programm berechnet Werte der **Fakultätsfunktion**, und zwar mit Hilfe eines **Ganzzahltyps** und mit Hilfe von drei verschiedenen **Bruchzahltypen**. **o**

21.1. Dezimale Festpunkttypen

Was ein **dezimaler Festpunkttyp** ist, wie man ihn vereinbart und was man damit machen kann, wird anhand des folgenden Beispiels erläutert:

Beispiel 21.1.1.: Vereinbarung eines dezimalen Festpunkttyps mit Zubehör:

```
01 type    DFIX1    is delta 0.01 digtis 8;
02 package DFIX1_EA is new ada.text_io.decimal_io(num => DFIX1);
03 A1 :    DFIX1 := 123_456.78;
```

In Zeile 01 wird ein **dezimaler Festpunkttyp** !DFIX1 und sein erster **Untertyp** namens DFIX1 vereinbart. Zu diesem Typ (und zu seinem Untertyp DFIX1) gehören alle Dezimalzahlen, die sich mit einem Vorzeichen (+ oder -) und maximal 8 Dezimalziffern darstellen lassen, von denen höchstens 2 Ziffern nach und höchstens 6 Ziffern vor dem Dezimalpunkt stehen dürfen.

Zum Typ !DFIX1 gehören also unter anderem die folgenden Werte:
-999_999.99, -17.83, -17.5, -17.0, -0.1, -0.07, -0.01, 0.0,
+999_999.99, +17.83, +17.5, +17.0, +0.1, +0.07, +0.01

Wenn man Werte des Typs !DFIX1 durch **Literale** darstellt, kann man das Vorzeichen "+" natürlich auch weglassen. Den **Dezimalpunkt** und mindestens eine Stelle davor und eine danach muß man aber angeben.

Nicht zum Typ !DFIX1 gehören unter anderem die folgenden Werte:
0.123 (mehr als 2 Ziffern **nach** dem Dezimalpunkt), -1_000_000.0 (mehr als 6 Ziffern **vor** dem Dezimalpunkt), 17 (keine Bruchzahl, weil der **Punkt** fehlt).

Der Typ !DFIX1 wird als **dezimaler Festpunkttyp** bezeichnet, weil jeder Wert dieses Typs eine **feste** Anzahl von Stellen nach dem Dezimalpunkt hat, nämlich 2 Stellen.

Das in Zeile 02 vereinbarte **Paket** namens **DFIX1_EA** enthält **get**- und **put**-Prozeduren, mit denen man Werte des Untertyps DFIX1 als Texte einlesen bzw. ausgeben kann.

In Zeile 03 wird eine **Variable** namens **A1** vom Untertyp **DFIX1** mit dem Anfangswert **123_456.78** vereinbart. **o**

Allgemein gilt: Wenn man einen dezimalen Festpunkttyp vereinbart (wie oben in Zeile 01), muß man eine **delta-Zahl** und eine **digits-Zahl** angeben.

Als **digits-Zahl** muß man eine ganze Zahl größer oder gleich 1 angeben. Dieses Zahl legt fest, mit wieviel Dezimalziffern sich die Werte des Typs darstellen lassen müssen. Wie groß die **digits-Zahl** maximal sein darf, ist von Ada-Ausführer zu Ada-Ausführer verschieden. Z.B. erlaubt der Gnat-Compiler, Version 3.10 für PCs **digits-Zahlen** bis 18. Der Object-Ada-Compiler von Aonix, Version 7.0 ebenfalls für PCs erlaubt **digits-Zahlen** nur bis 8.

Als **delta-Zahl** muß man eine der folgenden Zahlen angeben:

..., **10_000.0**, **1_000.0**, **100.0**, **10.0**, **1.0**, **0.1**, **0.01**, **0.001**, **0.000_1**, **0.000_01**, ...

Das sind positive **Zehnerpotenzen** mit positivem oder negativen Exponenten (..., 10^4, 10^3, 10^2, 10^1, 10^0, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, ...). Man beachte, daß man die **delta-Zahl** immer als **Bruchzahl** angeben muß (mit einem Punkt darin), auch wenn es sich um eine ganze Zahl wie 1.0 oder 10.0 etc. handelt.

Zu dem vereinbarten dezimalen Festpunkttyp gehören nur solche Zahlen, die sich als ganzzahlig Vielfache der **delta-Zahl** darstellen lassen. Für den Typ **DFIX1** sind das die Zahlen:
0 * 0.01, ±1 * 0.01, ±2 * 0.01, ±3 * 0.01, ..., ±99_999_999 * 0.01.

Die **delta-Zahl** eines dezimalen Festpunkttyps ist häufig kleiner als **1.0**, kann aber auch größer oder gleich **1.0** sein, wie das folgende Beispiel zeigt:

Beispiel 21.1.2.: delta-Zahlen gleich oder größer als 1.0:

```
10 type DFIX2 is delta     1.0 digits 3;
11 type DFIX3 is delta 1_000.0 digits 2;
12 A2 : DFIX2 := 321.0;
13 A3 : DFIX3 := 21_000.0;
```

Zum Typ **!DFIX2** gehören nur ganze Zahlen, die sich mit 3 Dezimalziffern vor dem Punkt darstellen lassen. Das sind gerade die folgenden ganzzahlig Vielfachen der **delta-Zahl** 1.0:

0 * 1.0, ±1 * 1.0, ±2 * 1.0, ±3 * 1.0, ..., ±999 * 1.0.

Zum Typ **!DFIX3** gehören nur "glatte Tausenderzahlen", die sich mit 2 Dezimalziffern darstellen lassen. Das sind gerade die folgenden ganzzahlig Vielfachen der **delta-Zahl** 1_000.0:

0 * 1_000.0, ±1 * 1_000.0, ±2 * 1_000.0, ±3 * 1_000.0, ..., ±99 * 1_000.0.

Ein "typischer Ada-Ausführer" stellt Werte eines dezimalen Festpunkttyps intern als (dezimale) **Ganzzahlen** dar und merkt sich, mit welcher **delta-Zahl** er diese Ganzzahl z.B. vor einer Ausgabe multiplizieren muß. Z.B. stellt er die Zahl **123.45** vom Typ **!DFIX1** intern durch die Ganzzahl **12_345** dar und merkt sich, daß er diese Ganzzahl mit der **delta-Zahl** 0.01 des Typs multiplizieren muß.

Anmerkung: Die meisten heute üblichen Computer können intern nicht nur mit **Binärzahlen** rechnen, sondern auch mit **Dezimalzahlen**. Die Ziffern dieser Dezimalzahlen werden dabei einzeln durch je vier Bits als Binärzahlen kodiert. Z.B. stellt die Bitkette 0001 0010 als Dezimalzahl gelesen die Zahl 12 dar. Man nennt solche Dezimalzahlen auch **binär kodierte Dezimalzahlen** (binary coded decimals).

Wenn man will, kann man bei der Vereinbarung eines dezimalen Festpunktuntertyps auch eine **Bereichseinschränkung** angeben, etwa so:

Beispiel 21.1.3.: Untertypen mit Bereichseinschränkungen:

```
01    type DFIX4 is delta 0.1   digits 4 range -100.0  ..   +100.0;
02    type DFIX5 is delta 0.01  digits 7 range    0.0  ..+50_000.0;
03    type DFIX6 is delta 0.001 digits 6 range -333.333..   +555.555;
```

Zum Untertyp **DFIX4** gehören nur die ganzzahlig Vielfachen von **0.1**, die im Bereich **-100.0** bis **+100.0** liegen. Für **DFIX5** und **DFIX6** gilt entsprechendes. **o**

Das folgende Beispiel soll die wichtigsten Rechenregeln für dezimale Festpunktwerte verdeutlichen. Einge dieser Regeln (insbesondere die für die **Multiplikation** und **Division**) weichen deutlich von den entsprechenden Regeln für Ganzzahlen ab.

Beispiel 21.1.4.: Rechenregeln für dezimale Festpunktwerte:
"NPS" soll "Nachpunktstellen" heißen. Damit sind Dezimalziffern rechts vom Dezimalpunkt gemeint.

```
01 declare
02    A4, B4 : DFIX4 := 4.1;
03    A5, B5 : DFIX5 := 5.01;
04    A6, B6 : DFIX6 := 6.001;
05 begin
06    A6 := A5;                -- Typfehler, verboten.
07    A6 := A6 + B6;           -- Erlaubt, Ergebnis ist exakt.
08    A4 := A4 + A5;           -- Typfehler, verboten.
09    A4 := A5 - A4;           -- Typfehler, verboten.
10    A6 := A6 + DFIX6(A5);    -- Erlaubt, explizite Umwandlung.
11    A6 := A5 * A4;           -- Erlaubt, Ergebnis ist exakt.
12    A5 := A4 * B4;           -- Erlaubt, Ergebnis ist exakt.
13    A6 := A6 * B6;           -- Erlaubt, 3 NPS werden abgeschnitten.
14    A6 := A6 * 3;            -- Erlaubt, Ergebnis ist exakt.
15    A6 := 5  * B6;           -- Erlaubt, Ergebnis ist exakt.
16    A6 := A6 / B6;           -- Erlaubt, Ergebnis hat 3 NPS
17    A4 := A6 / B6;           -- Erlaubt, Ergebnis hat 1 NPS
18    A4 := A5 / A6;           -- Erlaubt, Ergebnis hat 1 NPS
19    A6 := A6 / 3;            -- Erlaubt, Ergebnis hat 3 NPS
```

Addieren und **Subtrahieren** darf man also nur solche dezimalen Festpunktwerte, die zum **selben Typ** gehören (siehe Zeile 07 und 10). Wenn eine Addition oder Subtraktion nicht die Ausnahme **constraint_error** auslöst, dann ist das Ergebnis mathematisch exakt. **Multiplizieren** und **Dividieren** darf man dagegen auch solche Werte, die zu **verschiedenen** Festpunkttypen gehören. Außerdem darf man das Ergebnis einer solchen dezimalen Festpunktmultiplikation oder -division einer Variablen eines **beliebigen Festpunkttyps** zuweisen (siehe z.B. Zeile 11). Hat diese Variable weniger Nachpunktstellen als das Ergebnis, werden vom Ergebnis entsprechend viele Stellen abgeschnitten. Dadurch verliert das Ergebnis im allgemeinen seine mathematische Exaktheit (siehe Zeile 17). Wenn die Zielvariable genügend viele Nachpunktstellen hat, wird das mathematisch exakte Ergebnis zugewiesen (siehe Zeile 16). Das Ergebnis einer Division ist im allgemeinen nicht exakt (siehe Zeile 16 bis 19). Festpunktzahlen darf man auch mit **Ganzzahlen** des vordefinierten Typs **!integer** multiplizieren oder durch Werte dieses Typs dividieren (siehe Zeile 14, 15 und 19).

Der wichtigste **Vorteil** von **dezimalen Festpunkttypen** gegenüber anderen Arten von Bruchzahltypen ist der folgende: Der Ada-Ausführer rechnet mit dezimalen Festpunktwerten weitgehend wie ein üblicher Taschenrechner oder wie die Angestellten eines Finanzamtes. Die Fehler, die beim Multiplizieren und Dividieren auftreten, kann man sich im allgmeinen sehr leicht erklären, indem man die Rechnungen "von Hand nachvollzieht". In diesem Sinne sind dezimale Festpunkttypen deutlich **menschenfreundlicher** als **gewöhnliche Festpunkttypen** oder **Gleitpunkttypen**. Kommerzielle Berechnungen (bei denen es meistens um Geldbeträge geht) sollte man grundsätzlich nur mit Ganzzahlen oder mit **dezimalen Festpunktzahlen** durchführen, aber nie mit **Gleitpunktzahlen**.

Aufgabe 21.1.1.: Übergeben Sie die Beispielprogramme **BRUCH_01**und **BRUCH_02** einem maschinellen Ada-Ausführer und lassen Sie sie mehrfach ausführen. Was passiert, wenn Sie Bruchzahlen mit zuvielen Nachpunktstellen eingeben? Was passiert, wenn Sie statt Bruchzahlen einfach Ganzzahlen eingeben? Machen Sie sich so vertraut mit diesen Programmen, daß sie ihre

Ausgaben für beliebige Eingaben exakt "voraussagen" können (insbesondere die nicht-exakten Ergebnisse der Multiplikationen und Divisionen). **o**

Das folgende Beispiel soll ein paar kleine "technische Feinheiten" verdeutlichen.

Beispiel 21.1.4.: Technische Feinheiten

```
01 declare
02    type DFIX1 is delta 0.01 digits 5;
03    A1 : DFIX1;
04 begin
05    A1 := 17;                -- Verboten, 17 ist kein Bruchzahlliteral!
06    A1 := 0.12;              -- Erlaubt.
06    A1 := 0.123;             -- Verboten, 0.123 hat zuviele Nachpunktstellen
07    A1 := 2.0 * 0.12345;     -- Erlaubt, A1 bekommt den Wert 0.24
```

Daß die Zuweisung in Zeile 07 erlaubt ist, folgt aus den allgemeinen Regeln für die Multiplikation von Festpunktzahlen und ist keine "extra Regel". **o**

Aufgabe 21.1.2.: Schreiben Sie ein Programm namens **BRUCH_12**, welches zwei Bruchzahlen einliest und ihr Produkt ausgibt. Die eingelesenen Zahlen sollen maximal zwei Dezimalstellen nach dem Punkt haben dürfen. Das Produkt soll zunächst exakt berechnet werden (mit **vier** Nachpunktstellen). Dann soll es auf **zwei** Stellen gerundet werden (indem man 0.005 dazu addiert) und schließlich mit zwei Nachpunktstellen ausgegeben werden. Beispiel: Eingaben **1.88** und **1.77**, exaktes Produkt **3.3276**, auf zwei Stellen gerundet **3.3326**, Ausgabe **3.33**. **o**

Das Beispielprogramm **BRUCH_03** zeigt, wie man beim Ausgeben von dezimalen Festpunktwerten das **Layout** beeinflußen kann (Anzahl der Stellen vor und nach dem Punkt, mit oder ohne Exponent).

Zusammenfassung 21.1.:

- Ein dezimaler **Festpunkttyp** legt genau fest, wie viele Dezimalstellen seine Werte vor und nach dem Punkt haben.
- Beim **Addieren** und **Subtrahieren** müssen beide Operanden zum selben Typ T gehören. Das Ergebnis kann man nur einer Variablen vom Typ T zuweisen.
- Beim **Multiplizieren** und **Dividieren** dürfen die Operanden zu beliebigen Festpunkttypen gehören. Das Ergebnis darf man einer Variablen eines beliebigen Festpunkttyps zuweisen.
- Festpunktwerte darf man auch mit Ganzzahlen des Typs **!integer multiplizieren** oder durch solche Werte **dividieren**.
- Das Ergebnis einer Festpunkt-**Addition** bzw. -**Subtraktion** ist mathematisch **exakt**.
- Das Ergebnis einer **Multiplikation**, so kann man sich vorstellen, wird exakt berechnet. Beim Zuweisen an eine Variable werden aber evt. Stellen abgeschnitten.
- Das Ergebnis einer **Division** ist im allgemeinen **nicht exakt**.
- **Kommerzielle Berechnungen** (bei denen es um Geldbeträge geht), sollte man immer mit dezimalen Festpunktzahlen durchführen und nie mit Gleitpunktzahlen.

21.2. Gewöhnliche Festpunkttypen

Zu einem **dezimalen** Festpunkttyp gehören bestimmte ganzzahlig Vielfache einer **Zehnerpotenz** (..., 10^2, 10^1, 10^0, 10^{-1}, 10^{-2}, ...). Zu einem **gewöhnlichen** Festpunkttyp gehören bestimmte ganzzahlig Vielfache einer **Zweierpotenz** (..., 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, ...). Ansonsten haben beide Arten von Festpunkttypen große Ähnlichkeit miteinander.

Anmerkung: Eigentlich sollte man gewöhnliche Festpunkttypen als **binäre Festpunkttypen** bezeichnen. Hauptsächlich historische Gründe, die auf Ada83 zurückgehen, haben verhindert, daß diese naheliegende Bezeichnung offiziell eingeführt wurde.

Beispiel 21.2.1.: Vereinbarung eines gewöhnlichen Festpunkttyps mit Zubehör:

```
01 type     GFIX1    is delta 0.1 range -10_000.0 .. +10_000.0;
02 package  GFIX1_EA is new ada.text_io.fixed_io(num => GFIX1);
03 A1, B1 : GFIX1 := 123.4567;
```

Wenn man einen gewöhnlichen Festpunkttyp vereinbart (wie in Zeile 01), **muß** man eine **delta-Zahl** und einen **Bereich** angeben. Die delta-Zahl darf im Prinzip eine beliebige (als Bruchzahl notierte) Zahl sein, z.B. 0.1 oder 0.0123 oder 17.56 etc. Eine digits-Zahl darf man **nicht** angeben.

Der Ada-Ausführer (nicht der Programmierer!) wählt für den gewöhnlichen Festpunkttyp eine **Zweierpotenz**, die kleiner oder gleich der vom Programmierer angegeben **delta-Zahl** ist. Diese Zweierpotenz wird als **small-Zahl** des Typs bezeichnet. Zu dem vereinbarten Typ gehören dann alle ganzzahlig Vielfachen der **small-Zahl**, die in dem vom Programmierer angegeben Bereich liegen.

Viele Ada-Ausführer wählen als **small-Zahl** immer die **größte** Zweierpotenz, die kleiner oder gleich der **delta-Zahl** ist. Hier eine **Liste von Zweierpotenzen** in verschiedenen Notationen:

...	**...**	**...**
2^4	16	16.0
2^3	8	8.0
2^2	4	4.0
2^1	2	2.0
2^0	1	1.0
2^{-1}	1/2	0.5
2^{-2}	1/4	0.25
2^{-3}	1/8	0.125
2^{-4}	1/16	0.0625
2^{-5}	1/32	0.03125
2^{-6}	1/64	0.015625
2^{-7}	1/128	0.0078125
...	**...**	**...**

Der Typ **!GFIX1** wurde mit einer **delta-Zahl** von **0.1** vereinbart. Die größte Zweierpotenz kleiner oder gleich **0.1** ist $\mathbf{2^{-4}}$, d.h. **1/16** oder **0.0625**. Ein typischer Ada-Ausführer wird diesen Wert als **small-Zahl** für !GFIX1 wählen und die Werte des Typs als ganzzahlig Vielfache von 1/16 darstellen. **o**

Wenn man einer Variablen des Untertyps **GFIX1** einen Wert zuweist, dann muß dieser Wert zu einem **ganzzahlig Vielfachen von 1/16** (der **small-Zahl** des Typs) auf- oder abgerundet werden. Z.B. hat die Variable **A1** nach der Zuweisung **A1 := 1.0 / 10.0;** nicht etwa den Wert **0.1**, sondern den Wert **2 * 0.0625** oder **0.125**. Mit solchen Fehlern muß man rechnen, wenn man eine Berechnung mit gewöhnlichen (d.h. binären) Festpunktzahlen nachvollziehen und verstehen will.

Aufgabe 21.2.1.: Übergeben Sie die Beispielprogramme **BRUCH_04** und **BRUCH_05** einem maschinellen Ada-Ausführer und lassen Sie sie mehrmals ausführen. Welche **small-Zahl** hat der Ausführer für die Typen **!GFIX1** und **!GFIX2** gewählt? Versuchen Sie, die Ausgaben der Programme exakt vorauszusagen. Dazu müssen Sie die Eingabezahlen in ganzzahlig Vielfache der small-Zahl des betreffenden Typs umwandeln. **o**

Seitdem es (in Ada95) **dezimale Festpunkttypen** gibt, haben die **gewöhnlichen Festpunkttypen** an Bedeutung verloren und werden vermutlich nur noch selten verwendet.

Das Beispielprogramm **BRUCH_06** zeigt, wie man beim Ausgeben von gewöhnlichen Festpunktwerten das **Layout** beeinflußen kann (Anzahl der Stellen vor und nach dem Punkt, mit oder ohne Exponent).

Zusammenfassung:

- Bei der Vereinbarung eines gewöhnlichen Festpunkttyps muß man eine **delta-Zahl** und einen **Bereich** angeben.
- Der Ausführer wählt dann als **small-Zahl** des Typs eine geeignete Zweierpotenz (die kleiner oder gleich der delta-Zahl ist).
- Zu dem Typ gehören dann all die **ganzzahligen Vielfachen** der **small-Zahl**, die in dem vom Programmierer angegeben **Bereich** liegen.
- Für das Rechnen mit **gewöhnlichen** Festpunktwerte gelten weitgehend die gleich Regeln wie für das Rechnen mit **dezimalen** Festpunktwerten.
- Die **Fehler**, die beim Multiplizieren und Dividieren von **gewöhnlichen** Festpunktzahlen auftreten, sind "schwieriger nachzuvollziehen", als die entsprechenden Fehler bei **dezimalen** Festpunktzahlen.

21.3. Gleitpunkttypen

Gleitpunktzahlen wurden entwickelt, um die **Kosten** von umfangreichen Berechnungen zu **senken**, insbesondere von Berechnungen mit Zahlen, die auf dem Zahlenstrahl "weit weg von der Zahl 0" oder "sehr nahe bei der Zahl 0" liegen.

Beispiel 21.3.1.: Die Zahl 1_000_000_000_000_000_000_000_000 (eine Trilliarde) liegt ziemlich **weit weg** von der Zahl 0. Ihr Kehrwert, die Zahl 1/1_000_000_000_000_000_000_000_000 (ein Trilliardstel) liegt entsprechend **nahe bei** der Zahl 0. **o**

Anmerkungen: Um die Erläuterungen in diesem Abschnitt einfach zu halten, werden nur **positive** Zahlen (und die Zahl 0) behandelt. Alle Erläuterungen gelten aber, entsprechend "übersetzt", auch für negative Zahlen. Als **kleine Zahlen** werden im folgenden Zahlen bezeichnet, die nahe bei der 0 liegen (z.B. **ein Trilliardstel**), und nicht Zahlen, die auf dem Zahlenstrahl sehr weit links liegen wie z.B. **minus eine Trilliarde**. **o**

Fast alle heute üblichen Computer enthalten spezielle Hardwareteile für die Durchführung von Gleitpunktrechnungen (sogenannte **Gleitpunktprozessoren**). Es gibt erstaunlich viele verschiedene Gleitpunktprozessoren, die sich in bisweilen wichtigen Details voneinander unterscheiden. Ein Standard (z.B. der Gleitpunktstandard **IEEE 754**) scheint sich nur sehr allmählich zu verbreiten.

Gleitpunktzahlen beruhen auf einer einfachen **Grundidee**: Man stellt eine Bruchzahl **z** im wesentlichen durch zwei Zahlen dar: durch eine Bruchzahl **m** (Mantisse) und eine Ganzzahl **e** (Exponent).

Beispiel 21.3.1.: Ein paar dezimale Gleitpunktzahlen:

Zahl z:	Mantisse m:	Exponent e:	Bedeutung von m und z:
257.3	25.73	1	$25.73 * 10^{1}$
0.000123	12.3	-5	$12.3 * 10^{-5}$
45600.0	4.56	4	$4.56 * 10^{4}$
98.76	98.76	0	$98.76 * 10^{0}$

Man kann jede Bruchzahl auf **verschiedene Weise** als Gleitpunktzahl darstellen, wie das folgende Beispiel verdeutlichen soll:

Beispiel 21.3.2.: Verschiedene Darstellungen der Zahl **12.34**:

Zahl z:	Mantisse m:	Exponent e:	Bedeutung von m und z:
...	...	...	...
12.34	1234.0000	-2	$1234.0000 * 10^{-2}$
12.34	0123.4000	-1	$0123.4000 * 10^{-1}$
12.34	0012.3400	0	$0012.3400 * 10^{0}$
12.34	0001.2340	1	$0001.2340 * 10^{1}$
12.34	0000.1234	2	$0000.1234 * 10^{2}$
...	...	...	...

Hier wird die Bezeichnung "Gleitpunktzahl" deutlich: Man kann den Dezimalpunkt der Mantisse nach links oder rechts "gleiten" lassen, indem man den Exponenten entsprechend erhöht bzw. erniedrigt. Am Wert der dargestellten Zahl ändert sich dadurch nichts.

Um unter den vielen Gleitpunktzahlen, die dieselben Zahl **z** darstellen, **eine** auszuzeichnen, hat man den Begriff der **normalisierten Darstellung** eingeführt. Eine Gleipunktzahl ist **normalisiert**, wenn ihre Mantisse **vor** dem Dezimalpunkt **genau eine** (von 0 verschiedene) Ziffer besitzt.

Beispiel 21.3.3.: Ein paar **normalisierte** Gleitpunktzahlen:

Zahl z:	Mantisse m:	Exponent e:	Bedeutung von

			m und z:
257.3	2.573	2	$2.573 * 10^{2}$
0.000123	1.23	-4	$1.23 * 10^{-4}$
45600.0	4.56	4	$4.56 * 10^{4}$
98.76	9.876	1	$9.876 * 10^{1}$

Für jede Bruchzahl z gibt es **genau eine** Darstellung als normalisierte Gleitpunktzahl. Nur die Zahl **0.0** ist eine Ausnahme von dieser Regel, für die man gesondert festlegen muß, welche Darstellung als normalisiert gelten soll (z.B. die Darstellung mit der Mantisse **0.0** und dem Exponenten **0**).

Die **Größe** einer normaliserten Gleitpunktzahl wird haupsächlich durch ihren **Exponenten** festgelegt. Die **Mantisse** bestimmt dagegen die **Genauigkeit** der Zahl und hat nur einen sehr kleinen Einfluß auf ihre Größe.

Aufgabe 21.3.1.: Angenommen, zwei (positive, dezimale) normalisierte Gleitpunktzahlen haben die **gleichen Exponenten**. Um welchen **Faktor** können die beiden Zahlen sich dann höchstens voneinander unterscheiden? Was gilt im entsprechenden Fall für **binäre** Gleitpunktzahlen? **o**

Aufgabe 21.3.2.: Wie würden Sie vorgehen, um von zwei (positiven) normalisierten Gleitpunktzahlen festzustellen, ob eine von beiden **größer** ist als die andere? Welche Teile der Zahlen würden Sie zuerst miteinander vergleichen, die Mantissen oder die Exponenten? Warum? **o**

Zwei normalisierte Gleitpunktzahlen mit gleichen Exponenten, aber **unterschiedlich langen Mantissen**, unterscheiden sich vor allem durch ihre **Genauigkeit**, weniger durch ihre **Größen**.

Beispiel 21.3.4.: Sei **z1** gleich **123_456_789.012_345** und **z2** gleich **123_456 * 10^{3}**. Die beiden Zahlen **z1** und **z2** sind **fast gleich groß** (sie unterscheiden sich nur um einen kleinen Bruchteil eines Prozentes), aber sie unterscheiden sich deutlich durch ihre **Genauigkeiten**: **z1** wurde mit einer **Genauigkeit** von **15** Dezimalziffern angegeben, **z2** dagegen nur mit einer **Genauigkeit** von **6** Ziffern. Wenn man **z1** und **z2** normalisiert darstellt, wird der **Unterschied** ihrer **Genauigkeit** und die **Ähnlichkeit** ihrer **Größe** noch deutlicher:

z1 = 1.234_567_890_123_45 * 10^{8}
z2 = 1.234_56 * 10^{8}
o

Im Kern beruhen Gleitpunktzahlen auf der Idee, die **Größe** einer Zahl getrennt von den **Ziffern** der Zahl darzustellen. Bei sehr großen Zahlen ("weit weg von der 0") und sehr kleinen Zahlen ("nahe bei der 0") stellt man nur "die wichtigsten Ziffern" der Zahl dar. Die "weniger wichtigen Ziffern" stellt man nicht dar und braucht sie deshalb auch beim Rechnen nicht zu verarbeiten. Dadurch werden die Rechenoperationen deutlich billiger und die Zahlen benötigen weniger Speicherplatz..

Aufgabe 21.3.3.: Wie muß man vorgehen, um zwei Gleitpunktzahlen zu **multiplizieren**? Welche Rechenoperationen muß man mit ihren Mantissen **m1** und **m2** und ihren Exponenten **e1** und **e2** durchführen? Wie muß man beim **dividieren** vorgehen? **o**

Heute arbeiten die meisten Gleitpunktprozessoren mir **binären** Gleitpunktzahlen. Vereinzelt gibt es aber auch Prozessoren, die mit **dezimalen** oder mit **hexadezimalen** Gleitpunktzahlen arbeiten. Bei dezimalen Gleitpunktzahlen besteht die Mantisse aus dezimalen Ziffern, die durch je 4 Bits binär kodiert sind (binary coded decimals). Zwischen **binären** und **hexadezimalen** Gleitpunktzahlen

besteht nur ein subtiler Unterschied, denn bei beiden werden die Mantissen als Binärzahlen dargestellt. Aber wenn man bei einer **hexadezimalen** Gleitpunktzahl den Exponenten um 1 verändert, entspricht das einer Verschiebung der Mantisse um **vier Binärstellen**, bei einer **binären** Gleitpunktzahl dagegen nur um **eine Stelle**.

In Ada gibt es einen vordefinierten Gleitpunktuntertyp namens **float**. Es empfiehlt sich aber in aller Regel, eigene **Gleitpunkttypen zu vereinbaren** und zu benutzen. Die Gründe dafür sind die gleichen wie die, die gegen eine Verwendung des vordefinierten Ganzzahluntertyps **intger** und für die Vereinbarung von eigenen Ganzzahltypen sprechen (siehe Abschnitt 7.). Das folgende Beispiel zeigt, wie man einen Gleitpunkttyp vereinbart:

Beispiel 21.3.4.: Vereinbarung eines Gleitpunkttyps mit Zubehör:

```
01 type     GLEIT1    is digits 7 range -10.0..+10.0;
02 package  GLEIT1_EA is new ada.text_io.float_io(num => GLEIT1);
03 A1, B1 : GLEIT1 := 1.234_567_890;
```

Bei der Vereinbarung eines Gleitpunkttyps (siehe Zeile 01) legt man eine **Mindestgenauigkeit** fest, mit der die Werte dieses Typs vom Ausführer dargestellt werden müssen. Die Angabe **digits 7** bezeichnet eine Genauigkeit von **7 Dezimalziffern**. Diese **dezimale** Genauigkeit wird in eine entsprechende **binäre** Genauigkeit umgerechnet. Dabei entsprechen einer dezimalen Ziffer ungefähr 3.322 binäre Ziffern. Einer Mantisse mit 7 dezimalen Ziffern entspricht somit eine binäre Mantisse von 7 * 3.322 gleich 24 Bits. Der Ausführer muß die Werte des Typs **!GLEIT1** also mit Mantissen darstellen, die mindestens 24 Bit lang sind. Wenn er dazu nicht in der Lage ist, muß er schon beim Übergeben des Programms ("zur Compilezeit") eine entsprechende Fehlermeldung ausgeben.

Außer der **digits**-Zahl kann man auch eine **Bereichseinschränkung** angeben (im Beispiel: **range -10.0..+10.0**). Als Grenzen muß man **Bruchzahlen** angeben (**range -10..+10** wäre falsch). Wenn man keine Bereichseinschränkung angibt, legt der Ausführer nach einer bestimmten Formel einen "zu der digits-Zahl passenden Bereich" fest. Die Formel findet man im (ARM 3.5.7).

Mit dem Paket **GLEIT1_EA** (vereinbart in Zeile 02) kann man Werte des Untertyps **GLEIT1** in Textform ausgeben und einlesen.

Wenn man eine Gleitpunktvariable mit Hilfe eines **Literals** initialisiert (siehe Zeile 03) oder ihr den Wert eines Literals zuweist, dann darf das Literal im Prinzip beliebig viele Stellen nach dem Komma haben. Der (mathematisch exakte) Wert des Literals wird dann in einen Wert des betreffenden Gleitpunkttyps umgewandelt und zugewiesen. Die Umwandlung ist im allgemeinen nicht exakt (z.B. kann man den Wert des Literals **0.1** nicht exakt in eine binäre Gleitpunktzahl umwandeln). **o**

Für Werte eines Gleitpunkttyps stehen die üblichen Vergleichsoperationen "<", "<=", "=", "/=", ">=", ">" und die Rechenoperationen "+", "-", "*", "/", "**" und **abs** zur Verfügung. Der rechte Operand der Potenzierungsfunktion "**" muß zum Typ **!integer** gehören. Die Funktion **abs** liefert den **Betrag** (absolute value) einer Zahl. Wenn man **elementare Funktionen** wie **sqrt** (Wurzel), **log** (Logarithmus), **sin**, **cos** etc. benötigt, sollte man das generische Paket **ada.numerics.-generic_elementary_functions** mit dem "selbst vereinbarten Gleitpunkttyp" instanziieren. Man erhält dann auch eine Funktion namens **"**"**, die das Potenzieren einer Gleitpunktzahl mit einer Gleitpunktzahl erlaubt. Das Paket **ada.numerics** enthält Konstanten namens **pi** und **e** mit 50

Dezimalstellen nach dem Punkt. Zufallszahlen des vordefinierten Gleitpunktuntertyps **float** kann man mit den Unterprogrammen im Paket **ada.numerics.float_random** erzeugen lassen.

Aufgabe 21.3.4.: Übergeben Sie die Beispielprogramme **BRUCH_07** und **BRUCH_08** einem maschinellen Ada-Ausführer und lassen Sie sie wiederholt ausführen. **o**

Aufgabe 21.3.5.: Mit dem Beispielprogramm **BRUCH_15** kann man sich ansehen, wie Gleitpunktzahlen "wirklich aussehen", d.h. durch welche **Bitketten** sie intern im Computer dargestellt werden. Das Programm funktioniert nur auf solchen Rechnern richtig, die das 64-Bit-Format des Gleitpunktstandards IEEE 754 unterstützen. Weitere Hinweise findet man im Text des Programms. Lassen Sie das Programm **BRUCH_15** mehrmals von einem maschinellen Ada-Ausführer ausführen und schauen Sie sich dabei die interne Darstellung von verschiedenen Gleitpunktzahlen an. **o**

Ada ist die einzige verbreitete Programmiersprache, die es dem Programmierer gestattet, **eigene Gleitpunkttypen** zu vereinbaren und dabei eine **Mindestgenauigkeit** anzugeben. Damit ist es möglich, in Ada numerische Berechnungen unabhängig von der gerade vorhandenen Hardware und **portabel** zu programmieren.

Das Beispielprogramm **BRUCH_09** zeigt, wie man beim Ausgeben von Gleitpunktwerten das **Layout** beeinflußen kann (Anzahl der Stellen vor und nach dem Punkt, mit oder ohne Exponent).

Zusammenfassung 21.3.:

- Eine **Gleitpunktzahl** besteht aus einer **Mantisse** und einem **Exponenten**.
- Die **Größe** einer normalisierten Gleitpunktzahl ist fast nur vom **Exponenten** abhängig.
- Mit der **Genauigkeit** einer Gleitpunktzahl ist die Länge ihrer **Mantisse** gemeint (z.B. 7 Dezimalziffern oder 24 Binärziffern etc.).
- Wenn man Gleitpunktzahlen **addiert**, **subtrahiert**, **multipliziert** oder **dividiert** ist das Ergebnis im allgemeinen nicht mathematisch exakt sonder nur eine **Näherung**.
- Führt man viele Gleitpunktrechnungen nacheinander durch, kann der **Gesamtfehler** erheblich werden und das Ergebnis **unbrauchbar** machen (siehe Beispielprogramm BRUCH_11).
- Für Berechnungen mit Gleitpunktzahlen sollte in aller Regel eine **Fehlerabschätzung** durchgeführt werden.

Zentraldokument: Nach Filialdokument A95-21-21

22. Zeigertypen

Ein **Zeiger** ist **Wert** eines **Zeigertyps** (so ähnlich wie eine **Ganzzahl** ein **Wert** eines **Ganzzahltyps** ist). Zwischen **Zeigern** und **Namen** gibt es wichtige **Gemeinsamkeiten**:

Namen:
Ein Name **bezeichnet** etwas, z.B. eine Variable oder eine Konstante oder ein Unterprogramm etc..
Mithilfe eines Namens kann man auf das **zugreifen**, was er bezeichnet.

Zeiger:
Ein Zeiger **zeigt auf** etwas, z.B. auf eine Variable oder auf eine Konstante oder auf ein Unterprogramm.
Mithilfe eines Zeigers kann man auf das **zugreifen**, auf das er zeigt.

Andererseits gibt es auch wichtige **Unterschiede** zwischen **Zeigern** und **Namen**:

Ein Name kann **nicht** als **Wert einer Variablen** gespeichert werden
Namen werden vom **Programmierer** festgelegt

Ein Zeiger kann als **Wert einer Variablen** gespeichert werden
Zeiger werden vom **Ausführer** festgelegt

Zeiger sind ziemlich **maschinennahe** Konstrukte. Sie abstrahieren nur wenig von den **Adressen**, mit denen heute übliche Computer die Variablen, Konstanten und Unterprogramme etc. in ihrem Hauptspeicher adressieren. Jeder Computer arbeitet mit verschiedenen Arten von Adressen (mit absoluten Adressen, relativen Adressen, virtuellen Adressen etc.) und die genaue Form dieser Adressen ist bei verschiedenen Computertypen verschieden. **Zeiger** in einer höheren Sprache wie Ada abstrahieren von den Unterschieden zwischen verschiedenen Arten und Formen von Adressen und haben nur die Eigenschaften, die allen Adressen gemeinsam sind (siehe oben).

Weil Zeiger so **maschinennah** sind, sind sie sehr **mächtig** und gleichzeitig sehr **gefährlich**. Zeigerfehler unterlaufen auch erfahrenen Programmierern sehr leicht und sind besonders schwer zu finden. Mit Zeigern zu programmieren erfordert besondere Sorgfalt und die Fähigkeit, Programme zu überprüfen, indem man sie im Kopf oder mit Papier und Bleistift ausführt.

Mit Zeigern kann man unter anderem sogenannte **dynamische Datenstrukturen** realisieren. Eine solche Datenstruktur besteht aus **Verbunden** (records), die durch **Zeiger** miteinander verbunden sind. **Dynamisch** heißen diese Datenstrukturen, weil man sie im Verlauf einer Programmausführung ("dynamisch") vergrößern oder verkleinern kann, indem man Verbunde hinzufügt bzw. entfernt. Je nachdem, wie die Verbunde einer dynamischen Datenstruktur "miteinander verzeigert" sind, unterscheidet man **einfach verkettete Listen** (jeder Verbund enthält einen Zeiger, der auf den nächsten Verbund zeigt), **doppelt verkettete Listen** (jeder Verbund enthält zwei Zeiger, einer zeigt auf den vorigen und einer auf den nächsten Verbund), **Bäume** (jeder Verbund enthält zwei oder mehr Zeiger, die auf Unterbäume zeigen) oder **allgemeine Graphen** (die Verbunde können "beliebig wild" miteinander verzeigert sein).

Zeiger spielen aber nicht nur bei dynamischen Datenstrukturen und anderen komplizierten Konstruktionen eine wichtige Rolle. Im folgenden Unterabschnitt wird gezeigt, daß Zeiger sehr viel mit dem wichtigsten Grundkonzept der meisten Programmiersprachen zu tun haben: mit dem Konzept einer **Variablen**.

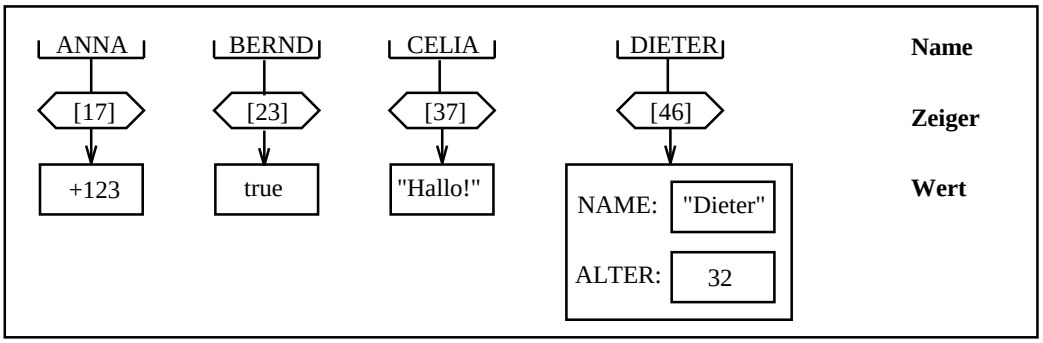
22.1. Grundlagen: Bojen, Zeiger und Zeigertypen

Bisher wurde eine **Variable** einfach als ein **Wertebehälter** beschrieben, d.h. als eine Art Kasten, in den man einen Wert hineintun kann. In diesem Abschnitt soll eine etwas genauere Vorstellung von der **Struktur einer Variablen** entwickelt werden.

Eine **Variable** besteht im Kern aus einem **Zeiger** und einem **Wert**. Außerdem kann sie noch einen **Namen** besitzen. Der Name wird vom **Programmierer** festgelegt, der Zeiger dagegen vom **Ausführer**. Der Zeiger einer Variablen zeigt auf eine bestimmte Stelle im Speicher des Ausführers. An dieser Stelle speichert der Ausführer den Wert der Variablen. Diese Speicherstelle entspricht also dem Wertebehälter, als den wir eine Variable bisher beschrieben haben.

Wenn der Ausführer ein Programm ausführt, benützt er im allgemeinen keine **Namen**, um auf verschiedene Stellen seines Speichers zuzugreifen, sondern nur noch die entsprechenden **Zeiger**. Das geschieht vor allem aus Gründen der Effizienz: Mit **Zeigern** kann der Ausführer wesentlich **schneller** umgehen als mit **Namen**. Daß eine Variable aus einem **Wert**, einem **Zeiger** und evt. einem **Namen** besteht, wird durch die sogenannte **Bojendarstellung** besonders deutlich hervorgehoben:

Beispiel 22.1.1.: Ein paar Variablen in Bojendarstellung



Die erste Variable besteht aus dem **Namen** ANNA, dem **Zeiger** [17] und dem **Wert** +123. Zwischen diesen Teilen und der Variablen bestehen folgende Relationen:

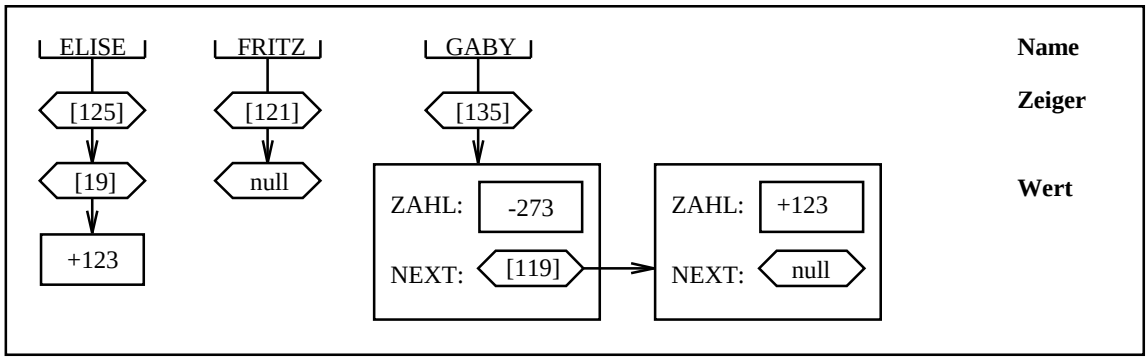
1. Der Name ANNA	**steht für**	den Zeiger [17]
2. Der Zeiger [17]	**zeigt auf**	eine Speicherstelle, die den Wert +123 enthält
3. Die Variable ANNA	**hat den Wert**	+123

o

Typen werden in der Bojendarstellung von Variablen häufig **nicht** dargestellt. Man unterscheidet aber deutlich zwischen **Zeigern** (sechseckige Kästchen) und **anderen Werten** (rechteckige Kästchen).

Eine **normale Variable** wie ANNA, BERND, CELIA, DIETER etc. hat eine Ganzzahl oder einen Aufzählungswert oder eine Reihung oder einen Verbund etc. als Wert. Im Gegensatz dazu hat eine **Zeigervariable** einen **Zeiger** als Wert, wie das folgende Beispiel verdeutlichen soll:

Beispiel 22.1.2.: Zeigervariablen und Verbundvariablen mit Zeigerkomponenten



Die **Zeigervariable** ELISE besteht aus dem **Namen** ELISE, dem **Zeiger** [125] und dem **Zeigerwert** [19]. Zwischen diesen Teilen und der Variablen bestehen folgende Relationen:

1. Der Name ELISE	**steht für**	den Zeiger [125]
2. Der Zeiger [125]	**zeigt auf**	eine Speicherstelle, die den Zeigerwert [19] enthält
3. Die Variable ELISE	**hat den Wert**	[19]
4. Der Zeiger [19]	**zeigt auf**	eine Speicherstelle, die den Wert +123 enthält
5. Die Variable ELISE	**zeigt auf**	eine Speicherstelle, die den Wert +123 enthält

Häufig identifiziert man eine **Variable** (die aus einem Namen, einem Zeiger und einem Wert besteht) mit ihrem **Wert**. Die **Variable** ELISE ist eine Zeigervariable, weil ihr **Wert** ein Zeigerwert ist. Die **Variable** ELISE zeigt auf die Speicherstelle, auf die ihr **Wert** zeigt (siehe 4. und 5.).

Der Zeigerwert [19] und die Speicherstelle, auf die er zeigt, bilden zusammen eine weitere **Variable**. Diese Variable **hat den Wert** +123, aber **keinen Namen**. Man kann auf sie nur zugreifen, weil ihr Zeiger [19] der Wert einer anderen Variablen ist.

Die Zeigervariable FRITZ hat den besonderen Wert **null**. Jeder andere, "normale" Zeigerwert zeigt auf eine Speicherstelle. Nur der Zeigerwert **null** zeigt auf keine Speicherstelle. Das ist seine definierende Eigenschaft. Würde man dem Ausführer einen Befehl geben wie z.B.: "Schreibe eine 17 an die Speicherstelle, auf die die Zeigervariable FRITZ zeigt!", dann würde der Ausführer die Ausnahme **constraint_error** auslösen, weil die Zeigervariable FRITZ auf keine Speicherstelle zeigt.

GABY ist offenbar keine **Zeigervariable** (weil ihr Wert-Kästchen nicht sechseckig ist), sondern eine **Verbundvariable**. Der Verbundwert von GABY hat zwei Komponenten namens ZAHL und NEXT. Die NEXT-Komponente enthält einen **Zeigerwert** (nämlich [119]), der auf einen weiteren Verbund zeigt. Dessen NEXT-Komponente hat den Wert **null** und zeigt somit auf keinen weiteren Verbund. **o**

Achtung: Das reservierte Wort "**null**" hat mehrere verschiedene Bedeutungen. Unter anderem bezeichnet es die **null-Anweisung** und den **null-Zeigerwert**. Was gemeint ist, muß man aus dem jeweiligen Zusammenhang schließen.

Im vorigen Beispiel wurden Zeigervariablen als Bojen dargestellt. Das folgende Beispiel soll zeigen, wie man diese Variablen in Ada vereinbaren kann:

Beispiel 22.1.3.: Vereinbarungen von **Zeigertypen** und **Zeigervariablen**

```
01 type  GANZ                  is range -500 .. +500;
02 type  ZEIGER_AUF_GANZ       is access GANZ;
03 type  PERLE;                -- Mit Perlen kann man Ketten bilden.
04 type  ZEIGER_AUF_PERLE      is access PERLE;
05 type  PERLE                 is record
06    ZAHL: GANZ;
07    NEXT: ZAP;
08 end record;
09 FRITZ: ZEIGER_AUF_GANZ;
10 ELISE: ZEIGER_AUF_GANZ   := new GANZ'(+123);
11 GABY : PERLE := (ZAHL=>-273, NEXT=>new PERLE'(ZAHL=>+123, NEXT=>null));
```

In Zeile 02 wird ein **Zeigertyp** !ZEIGER_AUF_GANZ (und sein erster Untertyp namens **ZEIGER_AUF_GANZ**) vereinbart. Fast alle Werte dieses Typs **zeigen** auf Variablen vom Untertyp **GANZ**. Die einzige Ausnahme von dieser Regel ist der Zeigerwert **null**, der auf **keine Variable** zeigt. Zu jedem Zeigertyp gehört ein solcher **null**-Wert.

Den Typ **!GANZ** (bzw. den Untertyp **GANZ**) bezeichnet man auch als den **Zieltyp** (bzw. als den **Zieluntertyp**) des Zeigertyps **!ZEIGER_AUF_GANZ**.

In Zeile 03 steht eine **unvollständige Typvereinbarung**. Sie legt nur fest, daß der Name "**PERLE**" einen Untertyp (eines neuen Typs **!PERLE**) bezeichnet. Diese unvollständige Typvereinbarung ist hier notwendig, weil die Vereinbarung des Zeigertyps **!ZEIGER_AUF_PERLE** (in Zeile 04) und die Vereinbarung des Verbundtyps **!PERLE** (in Zeile 05 bis 08) gegenseitig voneinander abhängen. Läßt man die unvollständige Typvereinbarung weg, meldet der Ausführer in Zeile 04 einen Fehler, weil dann der Name "**PERLE**" dort noch keine Bedeutung hat.

Wenn der Programmierer bei der Vereinbarung einer Zeigervariablen keinen anderen Anfangswert vorschreibt, **initialisiert** der Ausführer die Zeigervariable **automatisch** mit dem Zeigerwert **null**. Dies gilt im Beispiel für die Variable **FRITZ** (vereinbart in Zeile 09).

Der Allokator **new GANZ** (siehe Zeile 10) erzeugt eine neue Variable vom Untertyp **GANZ** und liefert als Ergebnis einen Zeigerwert, der auf die neue **GANZ**-Variable zeigt. Mit diesem Zeigerwert wird die Variable **ELISE** initialisiert. Die neue **GANZ**-Variable wird mit dem Wert **+123** initialisiert.

Ganz entsprechend erzeugt der Allokator **new PERLE** (siehe Zeile 11) eine neue Variable vom Untertyp **PERLE** und liefert als Ergebnis einen Zeigerwert, der auf die neue Variable zeigt. Die neue **PERLE**n-Variable wird mit dem Wert **(ZAHL=>+123, NEXT=>null)** initialisiert. **o**

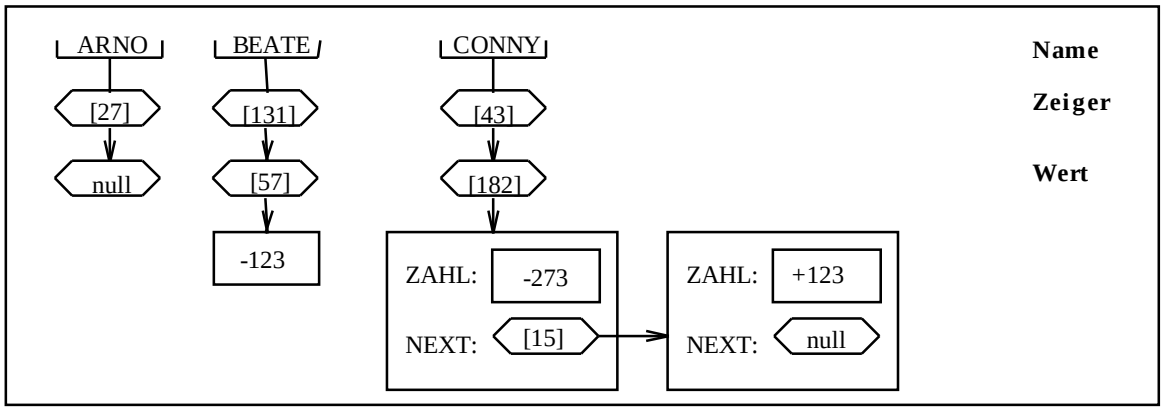
Zur Darstellung von Zeigerwerten: Wie Zeigerwerte konkret auszusehen haben, wird vom ARM nicht festgelegt. Tatsächlich benützen verschiedene maschinelle Ada-Ausführer ("Computer") sehr unterschiedliche Bitketten als Zeigerwerte. In diesem Skript werden Zeigerwerte als **ganze Zahlen in eckigen Klammern** dargestellt, z.B. **[19]**, **[125]** etc.. Die ganzen Zahlen werden möglichst "willkürlich und durcheinander" gewählt um zu verdeutlichen, daß der Ausführer diese Zeigerwerte so festlegen kann, "wie es ihm paßt" und daß er dabei an kein festes Schema gebunden ist. Der Ausführer muß nur jeder neuen Variablen einen **eigenen** Zeiger zuordnen, d.h. kein Zeiger darf gleichzeitig zwei verschiedenen Variablen zugeordnet sein ("an einer Stelle im Speicher kann nur eine Variable stehen"). Das gilt sowohl für **vereinbarte Variablen** als auch für solche Variablen, die mit dem Allokator **new** erzeugt werden (**allokierte Variablen**). **o**

Aufgabe 22.1.1.: Stellen Sie folgende Variablen als Bojen dar. Wählen Sie alle benötigten Zeigerwerte so, wie es Ihnen paßt, aber keinen Wert zweimal.

```
01 ALFRED: PERLE;
02 BERTA : PERLE(ZAHL => -17, NEXT => null);
03 CARL  : PERLE(ZAHL => +1,  NEXT => new
04           PERLE'(ZAHL => +2, NEXT => new
05             PERLE'(ZAHL => +3, NEXT => null)));
```

o

Aufgabe 22.1.2.: Betrachten Sie die folgenden Bojendarstellungen.



Vereinbaren Sie die hier als Bojen dargestellten Variablen und benützen Sie dabei die Untertypen **GANZ**, **ZEIGER_AUF_GANZ**, **PERLE** und **ZEIGER_AUF_PERLE** aus dem vorigen Beispiel 22.1.1.. **o**

Aufgabe 22.1.3.: Beantworten Sie die folgenden Fragen anhand der Bojendarstellung in der vorigen Aufgabe:
1. Wofür steht der Name **ARNO**? Ebenso für **BEATE** und **CONNY**.
2. Welchen Wert hat die Variable **ARNO**? Ebenso für **BEATE** und **CONNY**.
3. Worauf zeigt der Zeiger **[57]**? Ebenso für die Zeiger **[182]** und **[15]**.
4. Worauf zeig die Zeigervariable **ARNO**? Ebenso für **BEATE** und **CONNY**.

Die **Zeigervariable** BEATE besteht aus dem Namen **BEATE**, dem Zeiger **[131]** und dem (Zeiger-) Wert **[57]**. Sie zeigt auf eine **namenlose Variable**, die aus dem Zeiger **[57|** und dem Wert **-123** besteht. Diese beiden Variablen (die Zeigervariable und die Variable, auf die sie zeigt) "liegen nah beieinander". Gerade deshalb empfiehlt es sich, sie sorgfältig voneinander zu unterscheiden. Das folgende Beispiel soll zeigen, wie man auf diese beiden Variablen zugreifen kann.

Beispiel 22.1.4.: Auf Zeigervariablen und gezeigte Variablen zugreifen
Hier wird angenommen, daß die Variable **CONNY** zum Untertyp **ZEIGER_AUF_PERLE** und die Variablen **ARNO** und **BEATE** beide zum Untertyp **ZEIGER_AUF_GANZ** gehören.

```
01 begin
02    ARNO      := new GANZ;           -- Zuweisung an Zeigervariable
03    ARNO.all  := +321;               -- Zuweisung an gezeigte Variable
04    ARNO      := new GANZ'(-321);    -- Beides zusammen
05    BEATE.all := ARNO.all;           -- Zuweisung zwischen gezeigten Variablen
06    ARNO.all  := +777;               -- Zuweisung an gezeigte Variable
07    BEATE     := ARNO;               -- Zuweisung zwischen Zeigervariablen
08    ARNO.all  := +123;               -- Auch BEATE.all hat jetzt den Wert +123!
09    CONNY.all.ZAHL          := 333;
```

```
10    CONNY.all.NEXT.all.ZAHL := 444;
11    CONNY.all.NEXT.all.NEXT := CONNY;
```

Anfangs hat die Zeigervariable **ARNO** den Wert **null**. In Zeile 02 wird ihr ein neuer (Zeiger-) Wert zugewiesen. **ARNO.all** bezeichnet **"all das, worauf ARNO zeigt"**. ARNO zeigt auf eine GANZ-Variable. Dieser GANZ-Variablen wird in Zeile 03 der Wert **+321** zugewiesen.

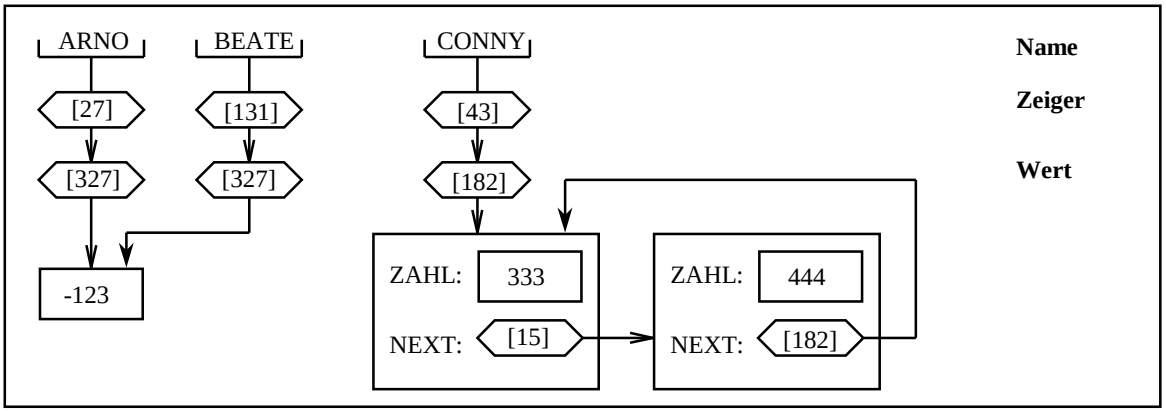
In Zeile 04 wird der Zeigervariablen **ARNO** erneut ein neuer (Zeiger-) Wert zugewiesen. Dieser Zeigerwert zeigt auf eine GANZ-Variable, die mit dem Wert **-321** initialisiert ist. Die GANZ-Variable mit dem Wert +321, auf die ARNO bisher gezeigt hat, bleibt zwar bestehen, wird aber **unerreichbar**.

Nach der Zuweisung in Zeile 05 hat die Variable, auf die BEATE zeigt, den Wert -321.
Nach der Zuweusing in Zeile 06 hat die Variable, auf die ARNO zeigt, den Wert +777.

Nach der Zuweisung in Zeile 07 haben die Zeigervariablen BEATE und ARNO beide den **gleichen** (Zeiger-) Wert (z.B. den Zeigerwert [327]). Damit zeigen sie auf **dieselbe** GANZ-Variable (nicht auf zwei GANZ-Variablen mit **gleichen** Werten!). Dieser GANZ-Variablen wird in Zeile 08 der Wert +123 zugewiesen.

CONNY.all bezeichnet "all das, worauf CONNY zeigt"., d.h. eine Verbundvariable vom Untertyp PERLE. **CONNY.all.ZAHL** (bzw. **CONNY.all.NEXT**) bezeichnet die **ZAHL**-Komponente (bzw. die **NEXT**-Komponente) dieser Verbundvariablen.

Nach Ausführung aller Zuweisungen in den Zeilen 02 bis 11 sehen die Variablen ARNO, BEATE und CONNY etwa so aus (der neue Zeigerwert [327] wurde wie immer willkürlich gewählt):



o

Aufgabe 22.1.4.: Sehen Sie sich das Beispielprogramm **ZEIGR_01** an, führen Sie es mit Papier und Bleistift aus und geben Sie dabei als Benutzer die Zahlen +17, -35, +82 und 0 ein. Stellen Sie alle Variablen, die im Programm vereinbart oder allokiert werden, als **Bojen** dar. Wie sehen diese Bojen in dem Moment aus, in dem der Ausführer die Zeile 30 erreicht? Übergeben Sie das Programm dann einem maschinellen Ada-Ausführer und lassen Sie es von ihm ausführen. Stimmen die Ausgaben des maschinellen Ausführers mit Ihren "Papier-und-Bleistift-Ausgaben" überein? o

Aufgabe 22.1.5.: Das Beispielprogramm **ZEIGER_02** ist eine Variante von **ZEIGR_01**. Lesen Sie es genau durch und führen Sie es dann mit Papier und Bleistift aus. o

Zusammenfassung 22.1.:

- Ein **Zeiger** zeigt auf ein Objekt (konkreter: auf eine Stelle im Speicher des Computers).
- **Jede** Variable besteht aus einem **Zeiger** und einem **Wert**. Der Wert steht an der Stelle im Speicher, auf die der Zeiger zeigt.
- Bei einer **Zeigervariablen** ist auch der **Wert** ein **Zeiger**.
- Ein **Zeigertyp** ist ein Typ, dessen Werte Zeiger sind.
- Man kann Variablen **vereinbaren** oder **allokieren** (mit dem Allokator **new**).
- **Vereinbarte** Variablen haben einen **Namen**, **allokierte** Variablen haben **keinen** Namen (nur einen Zeiger).
- Ist **Z** eine Zeigervariable, dann bezeichnet **Z.all** "all das, worauf Z zeigt".
- Mit Zeigern kann man **dynamische Datenstrukturen** bilden, z.B. verkettete Listen (wie in ZEIGR_01 und ZEIGR_02).
- Wenn die Werte eines Zeigertyps **ZT** auf Variablen eines Typs **T** zeigen, dann bezeichnet man **T** auch als den **Zieltyp** des Zeigertyps **ZT**.
- Die **Bojendarstellung** eignet sich besonders zur Darstellung von **Zeigervariablen** und **dynamischen Datenstrukturen**.

22.2. Zeiger auf vereinbarte Variablen

Wenn eine Variable **mehrere Namen** besitzt oder man über **mehrere Zeiger** auf sie zugreifen kann, dann bezeichnet man sie auch als **Alias-Variable** (aliased variable). Wenn z.B. die beiden Zeigervariablen **ARNO** und **BEATE** auf dieselbe GANZ-Variable zeigen, dann ändert man durch eine Zuweisung wie z.B. **ARNO.all := +123;** nicht nur die Variable **ARNO.all**, sondern auch die Variable **BEATE.all** (weil die Variable **ARNO.all** identisch ist mit der Variablen **BEATE.all**). Wenn der **Leser** eines Programms nicht mit Alias-Variablen rechnet, können sie ihn irreführen.

Eine **Alias-Variable** liegt auch dann vor, wenn eine Variable einen **Namen** hat und gleichzeitig eine **Zeigervariable** auf sie zeigt. Solche Alias-Variablen sind manchmal nützlich und auch in Ada erlaubt. Damit aber der **Leser** eines Programms durch solche Alias-Variablen nicht irregeführt wird, **muß** der Ada-Programmierer ihn an **zwei Stellen** auf ihr Vorhandensein aufmerksam machen: In der **Vereinbarung der Variablen** und in der **Vereinbarung des Zeigertyps**.

Beispiel 22.2.1.: Zeiger auf **vereinbarte** Variablen

```
01 type ZEIGER_AUF_STRING_2 is access all string;
02 T1   : aliased string := "Hallo!";
03 T2   : aliased string := "Wie geht es?";
04 T3   : aliased string := "Na, es geht so!");
05 TAB : array(1..3) of ZEIGER_AUF_STRING_2 :=
06    (1 => T1'access,
07     2 => T2'access,
08     3 => T3'access);
```

Die Werte des Zeigertyps **!ZEIGER_AUF_STRING** dürfen nicht nur auf allokierte ("mit **new** erzeugte") Variablen zeigen, sondern auch auf **vereinbarte** Variablen. Das folgt aus der Angabe des Wortes **all** in seiner Vereinbarung (Zeile 01).

In Zeile 02 wird eine Variable namens **T1** vereinbart. Das Wort **aliased** macht es möglich, daß später auch Zeigervariablen auf die Variable **T1** zeigen. Das Gleiche gilt für die Variablen **T2** und **T3**.

Die Reihung namens **TAB** (vereinbart in Zeile 05) besteht aus drei Komponenten vom **Untertyp ZEIGER_AUF_STRING_2**. Das Attribut **T1'access** bezeichnet einen Zeigerwert, der auf die (vereinbarte) Variable **T1** zeigt. Die Reihung **TAB** wird also mit drei Zeigerwerten initialisiert, die auf die vereinbarten Variablen **T1**, **T2** bzw. **T3** zeigen.

Das Attribut **T1'access** darf nur verwendet werden, weil **T1** als **aliased** vereinbart wurde. Die Komponenten der Reihung **TAB** dürfen deshalb auf vereinbarte Variablen wie **T1**, **T2** und **T3** zeigen, weil ihr Untertyp **ZEIGER_AUF_STRING** mit dem Wort **all** vereinbart wurde. **o**

Aufgabe 22.2.1.: Stellen Sie die Reihung **TAB** aus dem vorigen Beispiel ("und alles was daranhängt") als **Boje** dar. **o**

Zusammenfassung 22.2.:

- Die Werte eines "normalen" Zeigertyps können nur auf **allokierte** Variablen zeigern, aber nicht auf **vereinbarte** Variablen.
- **Keine** Zeigervariable kann auf eine "normale" vereinbarte Variable zeigen.
- Nur wenn man eine Variable als **aliased** vereinbart, können auch Zeigervariablen auf sie zeigen.
- Wenn man einen Zeigertyp mit **access all** vereinbart, können seine Wert auch auf (als **aliased**) vereinbarte Variablen zeigen.

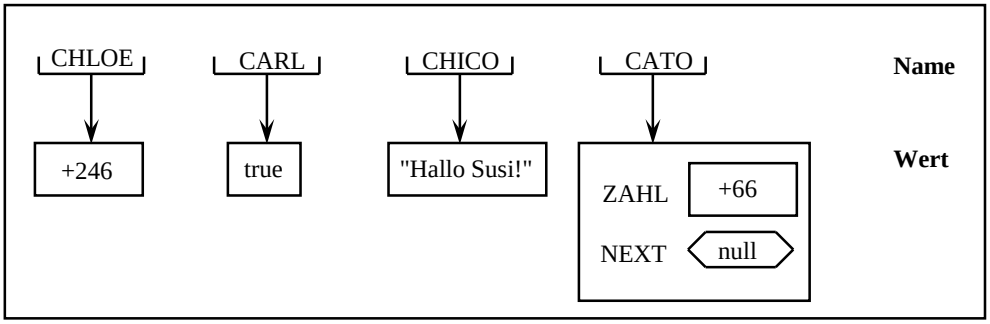
22.3. Zeiger ohne Schreibberechtigung

In diesem Abschnitt geht es um Konstanten. Insbesondere soll der subtile Unterschied zwischen einem **konstanten Zeiger** und einem **Zeiger, der auf eine Konstante zeigt**, deutlich gemacht werden.

Eine Konstante besteht nur aus einem Namen und einem Wert. In der Bojendarstellung sehen Konstanten so aus:

Beispiel 22.3.1.: Konstanten in Bojendarstellung:

```
01 CHLOE : constant GANZ    := +246;
02 CARL  : constant boolean := true;
03 CHICO : constant string  := "Hallo Susi!";
04 CATO  : constant PERLE   := (ZAHL => +66, NEXT => null);
```

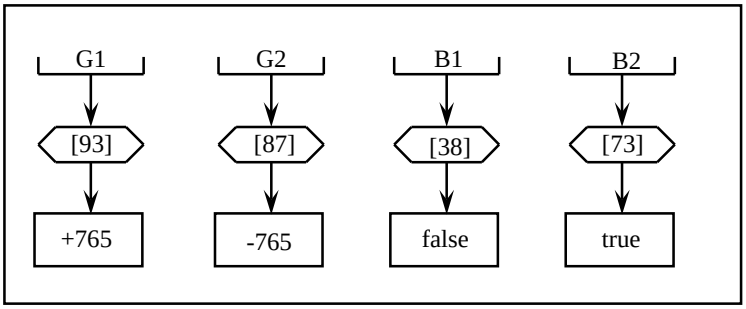


o

Das folgende Beispiel soll deutlich machen, daß eine **Konstante eines Zeigertyps** ("eine Zeigerkonstante") sehr große Ähnlichkeit mit einer "normalen Variablen" hat.

Beispiel 22.3.2.: Zeigerkonstanten und normale Variablen

```
01 type GANZ               is range -500..+500;
02 type ZEIGER_AUF_GANZ    is access GANZ;
03 type ZEIGER_AUF_BOOLEAN is access boolean;
04 G1 :          GANZ               := +765;
05 G2 : constant ZEIGER_AUF_GANZ    := new GANZ'(-765);
06 B1 :          boolean            := false;
07 B2 : constant ZEIGER_AUF_BOOLEAN := new boolean'(true);
```



Die **GANZ**-Variable **G1** hat den Wert **+765**. Die **ZEIGER_AUF_GANZ**-Konstante **G2** hat den (Zeiger-) Wert **[87]**. Sie zeigt auf eine **GANZ**-Variable mit dem Wert **-765**. Die **boolean**-Variable **B1** hat den Wert **false**. Die **ZEIGER_AUF_BOOLEAN**-Konstante **B2** hat den (Zeiger-) Wert **[73]**. Sie zeigt auf eine **boolean**-Variable mit dem Wert **true**.

Man beachte den feinen Unterschiede zwischen **GANZ-Variablen** und **ZEIGER_AUF_GANZ-Konstanen**: Die GANZ-Variable **G1** hat den GANZ-Wert **+765**. Die Zeigerkonstante **G2** hat den Zeigerwert **[87]**. Das Attribut **G1'access** bezeichnet den Zeiger **[93]** der Variablen **G1**. Der Ausdruck **G2.all** bezeichnet den Wert **-765**, auf den **G2** zeigt.

Die Werte der bisher eingeführten Zeigertypen (vereinbart mit **access** ... und **access all** ...) können nicht auf **Konstanten** zeigen. Man kann einen Zeigertyp aber so vereinbaren, daß seine Zeigerwerte ausdrücklich **nicht** mit einer **Schreibberechtigung** verbunden sind. Solche Zeigerwerte können auf Konstanten und Variablen zeigen:

Beispiel 22.3.3.: Zeiger ohne Schreibberechtigung:

```
01 type ZEIGER_AUF_STRING_3 is access constant string;
02 S1 : aliased constant string      := "Wie geht es?";
03 S2 : aliased          string      := "Na, es geht so!";
04 Z1 :          ZEIGER_AUF_STRING_3 := new string'("Hallo!");
05 Z2 :          ZEIGER_AUF_STRING_3 := S1'access;
06 Z3 :          ZEIGER_AUF_STRING_3 := S2'access;
07 Z4 : constant ZEIGER_AUF_STRING_3 := new string'("Hallo!");
08 Z5 : constant ZEIGER_AUF_STRING_3 := S1'access;
09 Z6 : constant ZEIGER_AUF_STRING_3 := S2'access;
```

Wenn man über einen Zeiger des Typs **!ZEIGER_AUF_STRING_3** auf einen String zugreift, dann darf man diesen String **nicht verändern**, unabhängig davon, ob es sich bei dem String um eine **Konstante** oder **Variable** handelt. Das folgt aus dem Wort **constant** in der Vereinbarung in Zeile 01.

Die Werte des Zeigertyps **!ZEIGER_AUF_STRING_3** dürfen auf allokierte Objekte und auf vereinbarte Objekte zeigen. Die Zeigervariable **Z1** zeigt auf ein allokiertes Stringobjekt mit dem Wert "Hallo!". Den Wert dieses Stringobjektes kann man nicht verändern, wohl aber den Wert der Zeigervariablen **Z1**. **Z2** ist eine Zeigervariable, die auf die Stringkonstante **S1** zeigt. Die Zeigervariable **Z3** zeigt auf die Stringvariable **S2**, aber "ohne Schreibberechtigung". **Z4** bis **Z6** sind Zeigerkonstanten.

Aufgabe 22.3.1.: Welche der folgenden Zuweisungen sind erlaubt und welche sind verboten?

```
01 Z1     := null;
02 Z1     := Z4;
03 Z1.all := "WIE GEHT ES?";
04 Z2     := null;
05 Z2     := Z5;
06 Z2.all := S1;
07 Z3     := null;
08 Z3     := Z6;
09 Z3.all := S2;
10 Z4     := null;
11 Z5     := new string("Mal was anderes!");
12 Z6.all := S2;
```

o

Aufgabe 22.3.2.: Im Beispielprogramm **ZEIGR_03** werden drei verschiedene Zeigertypen vereinbarten, mit **access**, mit **access all** und **access constant**. Führen Sie dieses Programm mit Papier und Bleistift aus. Überprüfen Sie Ihr Ergebnis, indem Sie das Programm auch von einem maschinellen Ada-Ausführer ausführen lassen. Das Beispielprogramm **ZEIGR_04** enthält einen weiteren access-all-Zeigertyp. **o**

Zusammenfassung 22.3.:

- Eine **Zeigerkonstante**, die auf eine Variable eines Untertyps T zeigt, hat große Ähnlichkeit mit einer **Variablen** des Untertyps T.
- Die Werte eines Zeigertyps, der mit **access** vereinbart wurde, können nur auf **allokierte Variablen** zeigen.
- Die Werte eines Zeigertyps, der mit **access all** vereinbart wurde, können auf **allokierte** und auf **vereinbarte Variablen** zeigen.
- Die Werte eines Zeigertyps, der mit **access constant** vereinbart wurde, können auf **allokierte** und **vereinbarte Variablen** und **Konstanten** zeigen.
- Mit einem **access-constant-Zeiger** kann man auf das "gezeigte Objekt" nur lesend zugreifen, man kann es aber nicht verändern (auch wenn es eine Variable ist).

22.4. Unterprogramme mit Zugriffsparametern (access parameter)

Zeiger kann man unter anderem auch als **Parameter** an Unterprogramme übergeben. Typischerweise bearbeitet das Unterprogramm dann nicht den Zeiger selbst, sondern das Objekt (die Konstante oder Variable), auf die der Zeiger zeigt. In diesem Zusammenhang kann ein kleines Problem mit dem **Typ des Zeigers** auftreten, wie das folgende Beispiel zeigen soll:

Beispiel 22.4.1.: Mehrere Zeigertypen mit demselben Zieltyp:

```
01 procedure ZEIGER_05 is
02    type      NORMALE_MELDUNG         is access string;
03    type      DRINGENDE_MELDUNG   is access string;
04    TEXT1:    NORMALE_MELDUNG     := new string'("Alles in Ordnung!");
05    TEXT2:    DRINGENDE_MELDUNG   := new string'("Abschnitt 3 brennt!");
06    procedure VERARBEITE_MELDUNG(M: in NORMALE_MELDUNG) is
07       ... -- Die Meldung M.all wird verarbeitet
08    end VERARBEITE_MELDUNG;
09 begin
10    VERARBEITE_MELDUNG(M => TEXT1);
11    ...
12    VERARBEITE_MELDUNG(M => TEXT2);  -- Typfehler!
```

Die beiden Zeigertypen **!NORMALE_MELDUNG** und **!DRINGENDE_MELDUNG** haben **denselben** Zieltyp **!string**. Die Prozedur **VERARBEITE_MELDUNG** erwartet als Parameter einen Zeiger M, der auf eine Meldung zeigt, und verarbeitet dann diese Meldung **M.all**.

Problem: Die Prozedur **VERARBEITE_MELDUNG** kann nur normale Meldungen verarbeiten, weil ihr Parameter M zum Untertyp **NORMALE_MELDUNG** gehört. Die allgemeinen Typenregeln von Ada verbieten den Prozeduraufruf in Zeile 12, obwohl er eigentlich sinnvoll wäre. **o**

In der Prozedur **VERARBEITE_MELDUNG** ist der genaue **Typ des Zeigerparameters M** eigentlich **unwesentlich**. Wichtig ist nur, daß M auf ein Objekt vom Typ **string** zeigt. Genau das kann der Programmierer ausdrücken, indem er M zu einem **Zugriffsparameter** macht, wie im folgenden Beispiel gezeigt wird:

Beispiel 22.4.2.: Ein Unterprogramm mit einem **Zugriffsparameter** (access parameter):
(Zeile 01 bis 05 genau wie im vorigen Beispiel)

```
06    procedure VERARBEITE_MELDUNG(M: access string) is
07       ... -- Die Meldung M.all wird verarbeitet
08    end VERARBEITE_MELDUNG;
09 begin
10    VERARBEITE_MELDUNG(M => TEXT1);  -- kein Typfehler!
11    ...
12    VERARBEITE_MELDUNG(M => TEXT2);  -- kein Typfehler!
```

Hier wurde **M** als **Zugriffsparameter** vereinbart (Zeile 06), indem anstelle eines konkreten Zeigertyps nur **access string** angegeben wurde. Das bedeutet, daß man (in einem Aufruf der Prozedur) als aktuellen Parameter für M **irgendeinen Zeiger** angeben darf, der auf eine **string**-Variable zeigt. Zu welchem Typ der Zeiger selbst gehört, ist egal (solange sein Typ den Untertyp **string** als **Zielunter-typ** hat).

In Zeile 10 gehört der aktuelle Parameter **TEXT1** zum Zeigertyp **!NORMALE_MELDUNG**, in Zeile 12 gehört der aktuelle Parameter **TEXT2** zum Zeigertyp **!DRINGENDE_MELDUNG** (siehe voriges Beispiel 22.4.1.). **o**

Zugriffsparameter haben automatisch den Modus **in**. Man darf diesen Modus aber **nicht** ausdrücklich angeben (bei "normalen in-Parametern" darf man den Modus **in** angeben oder weglassen).

Zusammenfassung 22.4.:

- **Verschiedene** Zeigertypen können **denselben** Zieluntertyp haben.
- Mit einem **Zugriffsparameter** (access parameter) kann man von den Unterschieden zwischen solchen Zeigertypen abstrahieren.
- Zugriffsparameter haben automatisch den Modus **in**.

22.5. Zeiger auf Unterprogramme

Zeiger können nicht nur auf **Variablen** und **Konstanten** zeigen, sondern auch auf **Unterprogramme**. Insbesodere kann man einen Zeiger, der auf ein Unterprogramm zeigt, einem anderen Unterprogramm als Parameter übergeben. Diese Möglichkeit wurde in Ada95 aufgenommen, um die Zusammenarbeit zwischen Ada-Programmen und C++-Programmen zu erleichtern. Beim Entwurf von Ada83 waren Unterprogramme als Parameter von Unterprogrammen ausdrücklich ("als schwieriges und fehlerträchtiges Konstrukt") abgelehnt worden. Den Benutzern von Ada83 wurde empfohlen, statt dessen **generische Einheiten** zu verwenden, die man unter anderem auch mit Unterprogrammen parametrisieren kann.

Auch solche Zeiger, die auf Unterprogramme zeigen, sind in Ada natürlich **streng getypt** und können nur auf Unterprogramme eines bestimmten **Profils** zeigen, wie das folgende Beispiel deutlich machen soll:

Beispiel 22.5.1.: Zeiger auf Unterprogramme und Unterprogramm-Profile

```
01 declare
02    type ZEIGER_AUF_PROZ is
03       access procedure (A: in integer; B: out character);
04    type ZEIGER_AUF_FUNK is
05       access function  (X: natural) return natural;
```

Die Werte des Zeigertyps **!ZEIGER_AUF_PROZ** können nur auf **Prozeduren** zeigen, die zwei Parameter besitzen. Der erste Parameter muß den Modus **in** haben und zum Untertyp **integer** gehören. Der zweite Parameter muß den Modus **out** haben und zum Untertyp **character** gehören. Wie die Parameter der "gezeigten Prozedur" heißen, ist egal.

Ganz entsprechend können die Zeiger des Typs **!ZEIGER_AUF_FUNK** nur auf **Funktionen** mit einem **natural**-Parameter und einem **natural**-Ergebnis zeigen. **o**

Zum **Profil** eines Unterprogramms gehören folgende Informationen:

1. Die Angabe, ob es sich um eine **Prozedur** oder um eine **Funktion** handelt
2. Die **Anzahl** der Parameter
3. Für jeden Parameter: Sein **Modus** und sein **Untertyp**. Dabei ist die **Reihenfolge** der Parameter signifikant (siehe dazu die Prozedur **EMIL** im folgenden Beispiel).
4. Bei Funktionen: Der Untertyp des Funktionsergebnisses.

Die **Namen** der Parameter gehören **nicht** zum Profil eines Unterprogramms.

Ein **Zeigertyp**, dessen Werte auf Unterprogramme zeigen, legt das **Profil** dieser Unterprogramme genau fest. Man bezeichnet dieses Profil auch als das **Zielprofil** (designated profile) des Zeigertyps.

Beispiel 22.5.2.: Die Typen **!ZEIGER_AUF_PROZ** und **!ZEIGER_AUF_FUNK** wurden im vorigen Beispiel vereinbart:

```
04    procedure OTTO(SUMME: in integer; GRUPPE: out character) is ... end OTTO;
05    procedure EMIL(GRUPPE: out character; SUMME: in integer) is ... end EMIL;
06    function  ANNA(PARAM: natural) return natural is ... end ANNA;;
07    function  BERTA(X:    natural) return natural is ... end BERTA;
08    function  CELIA(X:   positive) return natural is ... end CELIA;
09    PROZ1: ZEIGER_AUF_PROZ := OTTO'access;
10    PROZ2: ZEIGER_AUF_PROZ := EMIL'access;         -- Falsches Profil, verboten.
11    FUNK1: ZEIGER_AUF_PROZ := ANNA'access;
12    FUNK2: ZEIGER_AUF_FUNK := CELIA'access;        -- Falsches Profil, verboten.
13    CHAR1: character;
14    NAT1 : natural;
15 begin
16    PROZ1.all(A => 123, B => CHAR1);               -- aufgerufen wird OTTO.
17    NAT1  := FUNK1.all(X => +123);                 -- aufgerufen wird ANNA.
18    FUNK1 := BERTA'access;
19    NAT1  := FUNK1.all(X => -123);                 -- aufgerufen wird BERTA.
20    ...
```

Das **Profil** der Prozedur **OTTO** (siehe Zeile 04) stimmt mit dem Zielprofil des Zeigertyps **ZEIGER_AUF_PROZ** überein. Deshalb ist die Initialisierung in Zeile 09 erlaubt. Das Attribut **OTTO'access** bezeichnet einen Zeigerwert, der auf die Prozedur **OTTO** zeigt. Mit diesem Zeigerwert wird die Variable namens **PROZ1** initialisiert. Anschließend zeigt die Variable **PROZ1** auf die Prozedur **OTTO**.

Das Profil der Prozedur **EMIL** (siehe Zeile 05) ist nicht gleich dem Zielprofil des Zeigertyps **ZEIGER_AUF_PROZ**, denn die Reihenfolge der Parameter ist signifikant.

PROZ1.all bezeichnet "all das, worauf der Zeiger **PROZ1** zeigt", in Zeile 16 also die Prozedur **OTTO**. Als Abkürzung für **PROZ1.all** dürfte man hier auch einfach **PROZ1** schreiben. Die Parameternamen **A** und **B** wurden in der Vereinbarung des Zeigertyps **!ZEIGER_AUF_PROZ** festgelegt. Obwohl hier (in Zeile 15) die Prozedur **OTTO** aufgerufen wird, dürfen die Namen ihrer Parameter (**SUMME** und **GRUPPE**) nicht verwendet werden.

Aufgabe 22.5.1.: Führen Sie das Beispielprogramm **ZEIGR_05** mit Papier und Bleistift aus. Welche Texte gibt das Programm zur aktuellen Ausgabe aus? Überprüfen Sie Ihr Ergebnis, indem Sie das Programm auch von einem maschinellen Ada-Ausführer ausführen lassen. **o**

Zusammenfassung 22.5.:

- Zeiger können nicht nur auf Objekte, sondern auch auf **Unterprogramme** zeigen.
- Ist **U** ein Unterprogramm, dann bezeichnet **U'access** einen Zeigerwert, der auf **U** zeigt.
- Ein **Zeigertyp**, dessen Werte auf Unterprogramme zeigen, legt das **Profil** dieser Unterprogramme genau fest.
- Zum **Profil** eines Unterprogramms gehört 1. die Angabe, ob es sich um eine **Funktion** oder **Prozedur** handelt, 2. die **Anzahl** der Parameter, 3. für jeden Parameter der **Modus** und der **Untertyp** dieses Parameters und 4. bei Funktionen der **Untertyp** des Funktionsergebnisses.

22.6. Lebensdauer von allokierten Variablen

Der Ausführer **erzeugt** eine Variable, indem er einen freien Platz in seinem Speicher reserviert (und der Variablen die Adresse dieses Speicherplatzes als **Zeiger** zuordnet). Er **zerstört** die Variable, indem er ihren Speicherplatz (und ihren Zeiger) wieder freigibt. Je früher eine Variable wieder zerstört wird, desto früher kann ihr Speicherplatz wieder für andere Zwecke verwendet werden (z.B. zum Erzeugen neuer Variablen).

Wenn man eine Variable V am Anfang eines Unterprogramms U **vereinbart**, dann **erzeugt** der Ausführer die Variable jedesmal, wenn er mit einer Ausführung von U beginnt und **zerstört** sie wieder, wenn er diese Ausführung von U beendet. Das Erzeugen und Zerstören von **vereinbarten** Variablen kostet den Ausführer relativ **wenig** Arbeit.

Statt eine Variable zu **vereinbaren** kann man sie auch mit dem Allokatorbefehl **new** erzeugen lassen. Formal ist ein solcher Allokatorbefehl ein **Funktionsaufruf**, der als Ergebnis einen Zeiger liefert (der auf ein neues Objekt zeigt). Allokatorbefehle dürfen nicht nur im **Vereinbarungsteil** einer Programmeinheit verwendet werden (z.B. zum Initialisieren von Zeigervariablen), sondern auch im **Anweisungsteil** (z.B. in Zuweisungen an Zeigervariablen). Das Erzeugen und Zerstören von **allokierten** Variablen kann den Ausführer ziemlich **viel** Arbeit kosten.

Eine **allokierte Variable** wird in dem Moment **erzeugt**, in dem der Ausführer den Allokatorbefehl **new** ausführt. Wann eine allokierte Variable wieder **zerstört** wird, soll in diesem Abschnitt näher behandelt werden.

Grundsätzlich gilt: Der Ausführer **darf** allokierte Variablen beliebig lange leben lassen (maximal bis zum Ende der betreffenden Programmausführung). Er **darf** eine allokierte Variable aber auch jederzeit und **sofort** zerstören, wenn die Variable **unerreichbar** ist, d.h. wenn **keine Zeigervariable mehr darauf zeigt**.

Beispiel 22.6.1.: Eine allokierten Variablen wird **unerreichbar**:

```
01 declare
02    type ZAS is access string;
03    Z1 : ZAS;
04 begin
05    ...
06    Z1 := new string'("Hallo!");
07    ada.text_io.put(item  => Z1.all);
08    Z1 := null;
09    ...
```

In Zeile 06 wird mit **new** eine neue **string**-Variable erzeugt (allokiert) und ihr Zeigerwert wird der Variablen **Z1** zugewiesen. Nach Ausführung der Zuweisung in Zeile 08 ist diese **string**-Variable aber **unerreichbar** und der Ausführer darf sie zerstören. **o**

Im allgemeinen ist es für den Ausführer ziemlich aufwendig, die **Unerreichbarkeit** einer allokierten Variablen "so früh wie möglich" zu erkennen. Eine viel "billigere" Strategie besteht darin, allokierte Variablen zusammen mit dem **zuständigen Zeigertyp** zu zerstören. Diese Strategie beruht auf folgenden Überlegungen:

1. Alle **Zeigervariablen**, die auf ein allokiertes Objekt **a** zeigen, müssen zum **selben** Zeigertyp **ZT** gehören (weil nur dann "kopierende Zuweisungen" zwischen diesen Zeigervariablen erlaubt sind). Diesen Zeigertyp **ZT** bezeichnen wir hier als den **für a zuständigen Zeigertyp**.
2. Keine Variable lebt länger, als ihr Typ. Insbesondere lebt keine **Zeigervariable** länger als ihr Zeigertyp **ZT**.
3. Wenn der Ausführer einen Zeigertyp **ZT** zerstört, gibt es also keine **ZT**-Variablen mehr und er kann somit auch alle allokierten Objekte **a** zerstören, für die **ZT zuständig** ist.

Das folgende Beispiel soll diese etwas abstrakten Überlegungen konkret illustrieren:

Beispiel 22.6.2.: Der zugehörige Zeigertyp eines allokierten Objekts:

```
01 BLOCK1: declare
02    type ZAS1 is access string;         -- ein relativ langlebiger Typ
03    Z11: ZAS1;
04    ...
05 begin
06    ...
07    BLOCK2: declare
08       Z12: ZAS1 := new string'("Hallo!");
09       type ZAS2 is access string;      -- ein relativ kurzlebiger Typ
10       Z21: ZAS2 := new string'("Wie geht es?");
11    begin
12       Z11 := Z12;
13       ...
14    end BLOCK2;
15    ...
16 end BLOCK1;
```

Der Zeigertyp **ZAS1** wird im äußeren **BLOCK1** vereinbart und lebt solange, wie dieser äußere Block ausgeführt wird. Der Zeigertyp **ZAS2** wird im inneren **BLOCK2** vereinbart und lebt nur solange, wie dieser innere Block ausgeführt wird. **ZAS2**-Variablen können also nur im **BLOCK2** vereinbart werden und leben nur relativ kurz. **ZAS1**-Variablen können dagegen im **BLOCK1** und im **BLOCK2** vereinbart werden und leben entsprechend lange bzw. kurz.

Für die in Zeile 10 allokierte **string**-Variable ist der Zeigertyp **!ZAS2** zuständig. Wenn der **BLOCK2** fertig ausgeführt ist, kann diese **string**-Variable zusammen mit dem für sie zuständigen Zeigertyp **!ZAS2** zerstört werden. Für die in Zeile 08 allokierte **string**-Variable ist dagegen der Zeigertyp **!ZAS1** zuständig. Diese **string**-Variable ist auch nach Beendigung von **BLOCK2** noch über die Zeigervariable **Z11** erreichbar und darf erst bei Beendigung von **BLOCK1** (zusammen mit dem zuständigen Zeigertyp **!ZAS1**) zerstört werden. **o**

Das Zerstören von allokierten Variablen wird auch als **Speicherbereinigung** (garbage collection) bezeichnet. Die Sprache Ada schreibt **keine** bestimmte Vorgehensweise zur Speicherbereinigung vor. Ein Ada-Ausführer kann wählen, ob er allokierte Variablen automatisch "so früh wie möglich",

zusammen mit dem zuständigen Zeigertyp, überhaupt nicht oder nach einer anderen Strategie zerstört. Einfache Algorithmen zur automatischen Speicherbereinigung haben den Nachteil, daß sie in schwer vorhersagbaren Momenten einer Programmausführung relativ viel Prozessorzeit kosten können. Programme, die "harte Realzeitanforderungen" erfüllen müssen (z.B. die Programme zum Landen eines Flugzeuges) werden deshalb traditionsgemäß für Ada-Ausführer **ohne** automatische Speicherbereinigung geschrieben. Fortschritte bei der Entwicklung von modernen Speicherbereinigungsalgorithmen könnten das aber in Zukunft ändern.

Statt sich auf die Speicherbereinigung durch den Ausführer zu verlassen, kann man allokierte Variablen auch durch entsprechende Anweisungen zerstören lassen, wie das folgende Beispiel zeigen soll:

Beispiel 22.6.3.: Mit einer Instanz der Prozedurschablone **unchecked_deallocation** allokierte Variablen zerstören:

```
01 with ada.unchecked_deallocation, ada.text_io;
02 procedure P7 is
03    type ZAS1 is access string;
04    type ZAS2 is access string;
05    procedure GIB_FREI is new ada.unchecked_deallocation
06       (object => string, name => ZAS1);
07    Z1: ZAS1 := new string'("Hallo!");
08    Z2: ZAS2 := new string'("Wie geht es?");
09 begin
10    ada.text_io.put_line(item => Z1.all);
11    GIB_FREI(X => Z1);
12    Z1 := new string'("Na es geht so!");
13    ...
```

Die Prozedur **GIB_FREI** wird in Zeile 05 bis 06 als Instanz der Schablone **ada.unchecked_deallocation** vereinbart. Dabei muß man zwei **Untertypen** (als **object**- und als **name**-Parameter) angeben. Der **object**-Untertyp darf ein beliebiger Untertyp sein. Allokierte Objekte dieses Untertyps kann man mit der Prozedur **GIB_FREI** zerstören. Als **name**-Untertyp muß man einen **Zeigeruntertyp** angeben, dessen Werte auf Objekte des **object**-Untertyps zeigen. Im Beispiel zeigen die Werte des Zeigeruntertyps **ZAS1** auf Objekte des Untertyps **string**.

In Zeile 07 wird eine **string**-Variable mit dem Wert "Hallo!" allokiert. In Zeile 11 wird diese **string**-Variable zerstört, d.h. ihr Speicherplatz wird freigegeben. Die Zeigervariable **Z1** hat nach dem Aufruf der Prozedur **GIB_FREI** den Wert **null**.

Ohne den **GIB_FREI**-Befehl in Zeile 11 würde die **string**-Variable mit dem Wert "Hallo!" durch die Zuweisung in Zeile 12 **unerreichbar** und könnte **nicht** mehr vom Programmierer per Anweisung zerstört werden.

Man beachte, daß man mit der Prozedur **GIB_FREI** in diesem Beispiel nur solche **string**-Variablen zerstören kann, auf die ein Zeiger des Untertyps **ZAS1** zeigt, d.h. für die **ZAS1** zuständig ist. Um z.B. die **string**-Variable **Z2.all** zu zerstören, für die **ZAS2** zuständig ist, müßte man eine weitere Instanz der Prozedurschablone **ada.unchecked_deallocation** vereinbaren (mit object => string und name => **ZAS2**).

Daß der zweite Parameter der Prozedurschablone **ada.unchecked_deallocation** von den Ada-Entwicklern **name** genannt wurde, soll wohl an die Gemeinsamkeiten von **Zeigern** und **Namen** erinnern. **o**

Das Zerstören von allokierten Variablen mit einer Instanz der Prozedurschablone ada.unchecked_deallocation ist **gefährlich**, denn mit dem Objekt wird nur **eine** Zeigervariable gelöscht, die darauf zeigt. Ob der Programmierer auch alle anderen Zeigervariablen löscht, die auf dasselbe Objekt zeigen, wird nicht geprüft. Das folgende Beispiel zeigt, wie man **fehlerhaft** auf den Speicherplatz einer zerstörten Variablen zugreifen kann:

Beispiel 22.6.4.: Fehlerhafter Zugriff auf den Speicherplatz einer zerstörten Variablen:

```
01 with ada.unchecked_deallocation, ada.text_io;
02 procedure ZEIGR_07 is
...
07    type ZAS is access string;
08    procedure GIB_FREI is new ada.unchecked_deallocation
09       (object => string, name => ZAS);
10    Z1 : ZAS1 := new string'("Hallo!");
11    Z2 : ZAS1 := Z1;
12 begin
13    ada.text_io.put_line(item => Z1.all);
14    GIB_FREI(X => Z1);
15    Z1 := new string'("hello!");
16    ada.text_io.put_line(item => Z2.all);
17 end ZEIGR_07;
```

Anfänglich zeigen die beiden Zeigervariablen **Z1** und **Z2** auf **diesselbe** allokierte **string**-Variable mit dem Wert "Hallo!". Der Wert dieser **string**-Variablen wird in Zeile 13 ausgegeben. In Zeile 14 wird die **string**-Variable **Z1.all** zerstört und der **ZAS**-Variablen **Z1** wird der Wert **null** zugewiesen. Trozdem zeigt die Zeigervariable **Z2** weiterhin auf den Speicherplatz der zerstörten Variablen. Der **put_line**-Befehl in Zeile 16 ist deshalb **fehlerhaft**. Ob dieser **put_line**-Befehl eine Ausnahme auslöst oder eine andere Wirkung hat (z.B. die Ausgabe einer Zeichenkette) wird im ARM ausdrücklich **nicht** festgelegt. Der Programmierer ist für alle Folgen solcher fehlerhafter Befehle selbst verantwortlich. **o**

Aufgabe 22.6.1.: Übergeben Sie das Beispielprogramm **ZEIGR_07** einem maschinellen Ada-Ausführer und lassen Sie es ausführen. Können Sie sich die Ausgaben des Programms erklären? **o**

22.7. Binäre Bäume

Zu den interessantesten und nützlichsten Anwendungen von **Zeigern** gehören sogenannte **Bäume**, insbesondere **binäre Bäume**.

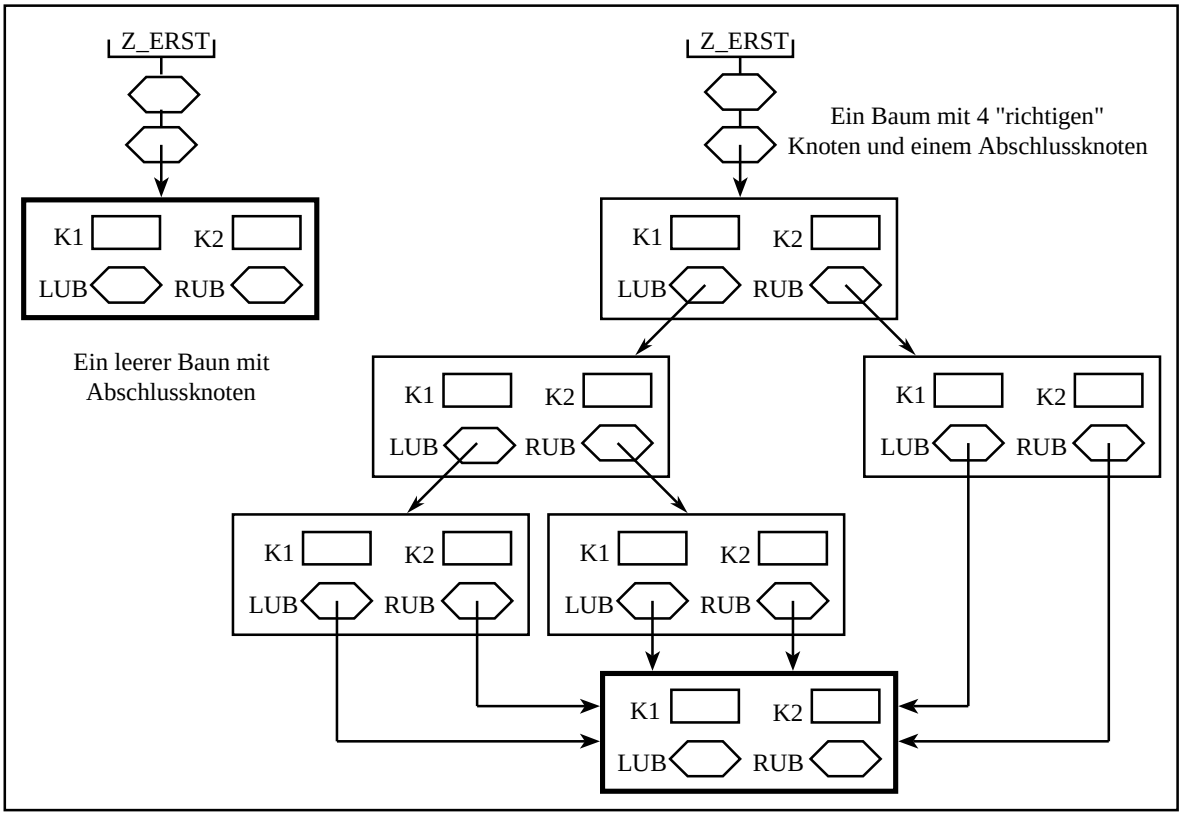
Ein binärer Baum ist entweder **leer** oder er besteht aus einem **Knoten**, an dem zwei binäre Bäume hängen. Jeder dieser beiden Bäume kann seinerseits wieder leer sein oder aus einem Knoten bestehen, an dem zwei binäre Bäume hängen u.s.w..

Die Knoten eines binären Baumes sind **Verbunde**. Das folgende Beispiel zeigt, wie die Vereinbarung eines entsprechenden Verbundtyps aussehen kann:

Beispiel 22.7.1.: Ein Verbundtyp für die **Knoten eines binären Baumes**:

```
01 type KNOTEN;                          -- Unvollstaendige Typvereinbarung
02 type ZAK    is access KNOTEN;         -- Zeiger auf Knoten
03 type KNOTEN is record                 -- Vollstaendige Typvereinbarung
04    K1:  DATENTYP;                     --   Datenkomponente 1
05    K2:  DATENTYP;                     --   Datenkomponente 2
06    LUB: ZAK;                          --   Zeiger auf linken  Unterbaum
07    RUB: ZAK;                          --   Zeiger auf rechten Unterbaum
08 end record;
```

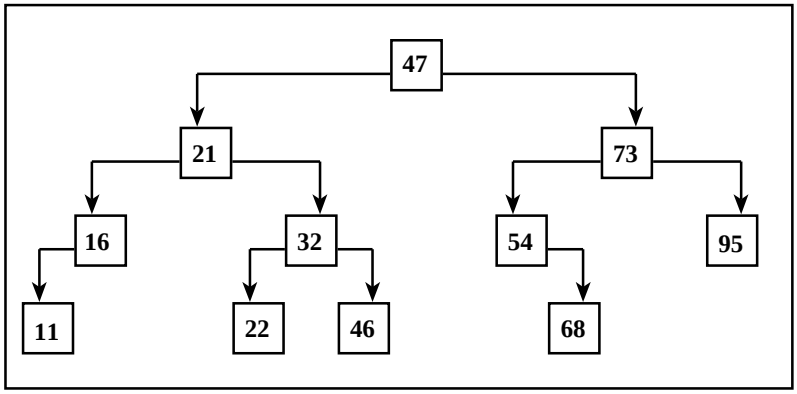
Die **unvollständige Typvereinbarung** in Zeile 01 ist notwendig, damit der Ausführer in Zeile 02 "**KNOTEN**" als Namen eines Untertyps erkennt. Die Komponenten **K1** und **K2** sind in diesem Beispiel die eigentlichen **Nutzdaten**, die gespeichert werden sollen. Sie können zu **irgendwelchen** Typen gehören. Die Zeiger **LUB** und **RUB** sind dagegen **Verwaltungsdaten**. Sie dienen dazu, den binären Baum aufzubauen und müssen zum Zeigertyp **!ZAK** gehören. Binäre Bäume, die aus solchen **KNOTEN** bestehe, können in Bojendarstellung etwa so aussehen:



Der **leere Baum** (links) besteht aus einer Zeigervariablen namesn **Z_ERST** und einem **Abschluss-knoten**. Dieser Abschlussknoten kann das **Suchen** im Baum erleichtern (siehe unten). Auf ihn zeigen alle **LUB**- und **RUB**-Zeiger im Baum, die nicht auf einen "richtigen Unterbaum" zeigen.

Ein binärer Baum ist nach einer Datenkomponente K1 **sortiert**, wenn für jeden Knoten **K** des Baumes gilt: Die K1-Komponente des Knotens **K** ist **größer** als alle K1-Komponenten in seinem **linken** Unterbaum und **kleiner** als alle K1-Komponenten in seinem **rechten** Unterbaum.

Beispiel 22.7.2.: Ein **sortierter** binärer Baum:
(Dargestellt ist für jeden Knoten nur **die** Komponente, nach der der Baum **sortiert** ist. Auf die Darstellung des abschliessenden Knotens wurde hier verzichtet).



Der oberste Knoten hat den Wert 47. Alle Werte in seinem **linken** Unterbaum sind **kleiner** als 47, alle Werte in seinem **rechten** Unterbaum sind **größer** als 47. Entsprechendes gilt für jeden Knoten des Baumes. **o**

Aufgabe 22.7.1.: Wo könnte man im sortierten binären Baum des vorigen Beispiels zusätzliche Knoten mit den Werten 20, 80, 50 und 99 einfügen, ohne die schon vorhandenen Knoten "zu verändern oder zu verschieben"? Natürlich soll der Baum auch **nach** dem Einfügen des neuen Knotens noch **sortiert** sein. Wie findet man allgemein die Stelle, an der man einen neuen Knoten mit einem bestimmten Wert einfügen kann? **o**

Aufgabe 22.7.2.: Fügen Sie (mit Papier und Bleistift) in einen anfangs leeren binären Baum Knoten mit den folgenden Werten so ein, daß der Baum nach jeder Einfügung **sortiert** ist: 46, 16, 54, 73, 95, 47, 68, 22, 32, 11, 21. **o**

Das folgende Beispiel zeigt eine **Prozedur**, mit der man Knoten in einen sortierten Baum einfügen kann. Dabei wird angenommen, daß die Knoten zu dem oben im Beispiel 22.7.1. vereinbarten Untertyp **KNOTEN** gehören, daß der Baum nach der **K1**-Komponente sortiert ist, daß der Zeiger **Z_ERST** auf den ersten Knoten des Baums zeigt (bzw. auf den **Abschlussknoten**, falls der Baum leer ist) und daß der Zeiger **Z_LETZT** immer auf den **Abschlussknoten** zeigt.

Beispiel 22.7.3.: Eine Prozedur zum Einfügen von Knoten in einen sortierten binären Baum:

```
01    procedure FUEGE_EIN(HIER: in out ZAK; N1, N2: DATENTYP) is
02    -- Fuegt einen neuen Knoten mit K1 gleich N1 und K2 gleich N2 in den
03    -- binaeren Baum ein, auf dessen Wurzelknoten der Zeiger HIER zeigt.
04    begin
05       if HIER = Z_LETZT then              -- Am "Ende des Baums" angekommen!
06          HIER := new KNOTEN'(K1 => N1, K2 => N2, LUB | RUB => Z_LETZT);
07       elsif N1 < HIER.all.K1 then         -- Weiter nach links
08          FUEGE_EIN(HIER => HIER.LUB, N1 => N1, N2 => N2);
09       else                                -- Weiter nach rechts
10          FUEGE_EIN(HIER => HIER.RUB, N1 => N1, N2 => N2);
11       end if;
12    end FUEGE_EIN;
```

Diese Prozedur ist **rekursiv**, d.h. sie ruft sich (in Zeile 08 und in Zeile 09) unter bestimmten Bedingungen und mit etwas veränderten Parametern selbst auf. **o**

Angenommen, wir haben einen bestimmten **K1**-Wert und wollen in einem binären Baum nach einem Knoten mit diesem **K1**-Wert suchen. Dann empfiehlt es sich, den gesuchten **K1**-Wert zuerst in den **Abschlussknoten** zu schreiben und dann erst mit der eigentlichen Suche zu beginnen. Dadurch ist man sicher, daß man den gesuchten **K1**-Wert **finden** wird (entweder in einem "richtigen

Knoten" oder spätenstens beim Erreichen des Abschlussknotens) und braucht beim Suchen nicht dauernd abzufragen, ob man schon "am Ende des Baumes" angekommen ist.

Wenn der Baum nach der **K1**-Komponenten **sortiert** ist, kann eine Prozedur zum Suchen nach einem Knoten mit einer bestimmten K1-Komponenten etwa so aussehen:

Beispiel 22.7.4.: Eine **SUCHE**-Prozedur:

```
01    procedure SUCHE(HIER: ZAK; N1: DATENTYP) is
02    begin
03       if HIER.all.K1 = N1 then          -- N1 als K1-Komponente gefunden!
04          GEFUNDEN(HIER => HIER);
05       elsif N1 < HIER.all.K1 then       -- Weiter nach links
06          SUCHE(HIER => HIER.LUB, N1 => N1);
07       else                              -- Weiter nach rechts
08          SUCHE(HIER => HIER.RUB, N1 => N1);
09       end if;
10    end SUCHE;
```

Ein Aufruf dieser Prozedur kann etwa so aussehen:

```
31    Z_LETZT.all.K1 := 37;            -- Den gesuchten Wert in den Abschluss-
32    SUCHE(HIER => Z_ERST, N1 => 37); -- knoten bringen und dann suchen
```

Zuerst wird der gesuchte **K1**-Wert in den Abschlussknoten geschrieben (siehe Zeile 31), dann wird die Prozedur **SUCHE** aufgerufen. Die Prozedur **SUCHE** findet den gesuchten **K1**-Wert garantiert (spätestens im Abschlussknoten) und ruft dann die Prozedur **GEFUNDEN** auf (in Zeile 04), die etwa so aussehen kann:

```
41    procedure GEFUNDEN(HIER: ZAK) is
42    -- Wird von der Prozedur SUCHE aufgerufen, wenn sie einen K1-Wert
43    -- im Baum gefunden hat:
44    begin
45       if HIER = Z_LETZT then
46          -- Der K1-Wert wurde erst im Abschlussknoten gefunden,
47          -- kommt im Baum also nicht wirklich vor.
...         ...
50       else
            -- Der K1-Wert wurde in einem "richtigen Knoten" gefunden,
            -- auf den der Zeiger HIER zeigt.
...         ...
54       end if;
55    end GEFUNDEN;
```

o

Aufgabe 22.7.3.: Führen Sie einen Aufruf der Prozedur **SUCHE** mit Papier und Bleistift aus. Zuvor sollten Sie folgende Vorbereitungen durchführen:
1. Zeichnen Sie einen **sortierten, binären Baum mit Abschlussknoten** (in Bojendarstellung wie im Beispiel 22.7.1. oder etwas vereinfacht wie im Beispiel 22.7.2., aber den Abschlussknoten sollten sie unbedingt einzeichnen).
2. Zeichnen Sie zwei Zeigervariablen namens **Z_ERST** und **Z_LETZT** so, daß **Z_ERST** auf den **ersten** Knoten des Baums und **Z_LETZT** auf den **Abschlussknoten** zeigt.
3. Schreiben Sie den **K1**-Wert, nach dem Sie suchen wollen, in die **K1**-Komponente des Abschlussknotens. **o**

Die Knoten eines binären Baumes kann man sich in **Ebenen** eingeteilt denken. Zur **Ebene 1** gehört nur der erste Knoten ("der Wurzelknoten") des Baums, zur **Ebene 2** gehören die beiden Nachfolgerknoten des Wurzelknotens, zur **Ebene 3** deren Nachfolger etc..

Aufgabe 22.7.4.: Sei **n** ein ganze Zahl, die größer oder gleich 1 ist. Maximal wieviele Knoten können zur Ebene **n** eines binären Baums gehören? Maximal wieviele Knoten kann ein binärer Baum enthalten, der aus **n** Ebenen besteht? **o**

In einem sortierten, binären Baum sucht man immer "von oben nach unten". Dabei führt man auf jeder Ebene, durch die man kommt, genau einen Vergleich aus. Spätestens wenn man auf der untersten Ebene angekommen ist, hat man den gesuchten Wert gefunden oder ist sicher, daß dieser Wert nirgends im Baum vorkommt. Für eine Suche muß man also maximal so viele "Vergleichsschritte" durchführen, wie der Baum Ebenen hat. Die folgenden Beispielzahlen sollen die erstaunliche Effizienz von binären Bäumen illustrieren:

Anzahl der Ebenen eines binären Baumes	**Anzahl der Knoten des Baumes (maximal)**
10	ca. 1_000
20	ca. 1_000_000
30	ca. 1_000_000_000

Angenommen, man hat eine Million Knoten als lineare **Liste** organisiert und sucht in dieser Liste einen Knoten mit einem bestimmen Wert. Dann muß man durchschnittlich **500_000** Abfrageschritte durchführen. Wenn man die Million Knoten dagegen als sortierten, binären **Baum** organisiert (in dem alle Ebenen voll mit Knoten besetzt sind), braucht man maximal nur **20** Abfrageschritte.

Aufgabe 2.7.5.: Wieviele Abfrageschritte muß man durchschnittlich ausführen, um in einer **Liste** von **einer Milliarde** Knoten einen Knoten mit einem bestimmten Wert zu finden? Wieviele Abfrageschritte braucht man maximal, wenn die Knoten als sortierter, binärer **Baum** organisiert sind (bei dem alle Ebenen voll mit Knoten besetzt sind)? **o**

Zentraldokument: Nach Filialdokument A95-22-22

23. Klassenweite Typen und etikettierte Verbunde

Ada ist eine **objektorientierte** Programmiersprache. Mit der Methode der objektorientierten Programmierung hofft man, die **Wiederverwendung** von Programmteilen zu erleichtern und die **Wartung** komplexer Programmsysteme zu vereinfachen. **Klasse** und **Objekt** sind die wichtigsten Grundbegriffe der objektorientierten Programmierung.

Der Begriff einer **Klasse** ist eine Weiterentwicklung des **Typbegriffs**. Hier die wichtigsten "Parallelen" zwischen Typen und Klassen:

Typen und Variablen	**Klassen und Objekte**
Ein Typ ist ein Bauplan für Variablen.	Eine Klasse ist ein Bauplan für Objekte
Der Typ einer Variablen legt fest, welche Operationen man auf die Variable anwenden darf.	Die Klasse eines Objekts legt fest, welche Operationen man auf das Objekt anwenden darf.

Bei **Typen** und Variablen stellt man sich vor, daß die Variablen und die Unterprogramme zu ihrer Bearbeitung im Prinzip "irgendwo" vereinbart werden und "herumliegen" können. Der **Ausführer** achtet aber darauf, daß man auf eine Variable nur "die richtigen" Unterprogramme anwendet. Wenn man z.B. versucht zwei **boolean**-Variablen zu addieren, meldet der Ausführer einen Typenfehler.

Bei **Klassen** und Objekten stellt man sich vor, daß die Unterprogramme zur Bearbeitung eines Objekts in dem Objekt selbst enthalten sind. Ein Objekt hat also große Ähnlichkeit mit einem **Modul** (d.h. mit einem Ada-Paket). Es enthält **Datenkomponenten** und **Unterprogramme** zu ihrer Bearbeitung, von denen einige nur **innerhalb** des Objekts sichtbar und "geschützt" sind und die anderen auch **außerhalb** des Objekts gesehen und benutzt werden können. Welche **Merkmale** (Datenkomponenten und Unterprogramme) ein Objekt enthält und welche davon wo sichtbar sind, wird natürlich vom Bauplan des Objekts (d.h. von seiner **Klasse**) festgelegt.

Eine **Klasse** ist also so etwas wie ein **Typ**, dessen Werte (oder Variablen) **Module** sind.

Ob Datenkomponenten und Unterprogramme zu einem **Objekt** zusammengefaßt werden, oder ob Variablen und Unterprogramme "getrennt voneinander herumliegen", macht **keinen** wesentlichen Unterschied: In beiden Fällen garantiert der Ausführer, daß man nur "die richtigen Operationen" auf ein Objekt bzw. eine Variable anwenden kann.

Unterschiede bestehen vor allem in der Art und Weise, wie man **Typen** bzw. **Klassen vereinbart**. Wenn man in Ada einen neuen Typ **T** vereinbart, legt man vor allem fest, ob T ein Ganzzahltyp, ein Aufzählungstyp, ein Verbundtyp etc. sein soll, d.h. man legt die **Typklasse** von **T** fest. Je nach Typklasse bekommt man dann zum neuen Typ **T** automatisch bestimmte Operationen dazu: falls **T** ein Ganzzahltyp ist, die Operationen **+**, **-**, *****, /, **rem** etc., falls **T** ein Aufzählungstyp ist, die Attribute wie **T'succ** und **T'pred** etc., falls **T** ein Verbundtyp ist, die **Punktnotation**, mit der man auf die Komponenten eines T-Verbundes zugreifen kann, etc..

Wenn man eine neue Klasse **K** vereinbart, läßt man den Ausführer häufig zuerst eine "**Kopie**" einer schon vorhandenen Klasse **S** anfertigen. Dann verändert man in dieser Kopie einige Merkmale (Datenkomponenten und/oder Unterprogramme) und fügt ein paar neue Merkmale hinzu. Man sagt in diesem Falle, daß die neue Klasse K die Merkmale der alten Klasse S **geerbt** hat und bezeichnet S als die **Superklasse** von K und K als eine **Unterklasse** von S.

Analogie: Wenn man einen Bauplan für eine neue Art von Einfamilienhäuser erstellen will, ist es häufig günstig, mit einer Kopie eines bewährten Bauplans anzufangen und dann ein paar Details zu ändern und ein paar hinzuzufügen.

Wenn man einen "normalen" **Typ** T vereinbart, bekommt man automatisch bestimmte **Unterprogramme** (Operationen) dazu, die der **Typklasse** von T entsprechen. Wenn man eine **Klasse** K vereinbart, kann man dabei die **Unterprogramme** und **Datenkomponenten** einer anderen (Super–) **Klasse** S erben.

Das Typensystem der Sprache Ada ist so aufgebaut, daß man Ada-**Typen** ganz ähnlich vereinbaren und benutzen kann, wie **Klassen** in anderen objektorientierten Sprachen. Insbesondere kann man einen Ada-Typ **T** als **Kopie** eines schon vorhandenen Typs **S** vereinbaren, so daß **T** alle Operationen von **S** (seinem Supertyp) **erbt**. Außerdem kann man zum neuen Typ **T** weitere **Operationen** und (falls **S** ein sogenannter etikettierter Verbundtyp ist) **Datenkomponenten** hinzufügen. Unter einem **Objekt** versteht man in Ada einfach eine **Konstante** oder **Variable** irgend eines Typs und statt von **Klassen** spricht man in Ada einfach von **Typen**.

23.1. Einen Typ von einem Typ ableiten und seine Operationen erben

Mithilfe der Operation **new** kann man einen **neuen Typ** als **Kopie** eines schon **vorhandenen Typs** vereinbaren. Diese Operation **new** zum "**Kopieren von Typen**" hat nichts mit dem Allokator **new** zu tun, mit dem man neue **Variable** erzeugen lassen kann.

Beispiel 23.1.1.: Einen Typ mit **new** kopieren

```
01 declare
02    type    AEPFEL   is range 0..500;
03    type    BIRNEN   is new AEPFEL;
04    type    MELONEN  is new BIRNEN range 0..200;
05    subtype ZITRONEN is     BIRNEN range 0..200;
06    A1, A2 : AEPFEL   := 350;
07    B1, B2 : BIRNEN   := 150;
08    M1, M2 : MELONEN  := 130;
09    Z1, Z2 : ZITRONRN := 120;
```

Typvereinbarungen wie in Zeile 02 kamen in diesem Skript schon häufig vor. Der Untertyp **AEPFEL** besteht aus einem neuen Typ **!AEPFEL** und der Einschränkung **range 0..500**. Der neue Typ **!AEPFEL** ist in aller Regel eine Kopie eines "Hardwaretyps", mit dessen Werten der Ausführer "besonders gut umgehen" kann (siehe dazu Abschnitt 7.).

Der Untertyp **BIRNEN** besteht aus einem neuen Typ **!BIRNEN** und der Einschränkung **range 0..500**. Der Typ **!BIRNEN** ist eine **Kopie** des Typs **!AEPFEL**. Das bedeutet: Für jede Operation, die man auf **!AEPFEL**-Werte anwenden kann (z.B. **+**, **-**, ***, /, **rem** etc.), gibt es eine entsprechende Operationen für **!BIRNEN**-Werte. Trotzdem darf man "Äpfel und Birnen nicht addieren" oder sonstwie miteinander verrechnen, denn **!BIRNEN** ist ein anderer Typ als **!AEPFEL**

Analogie: Die Typen **!AEPFEL** und **!BIRNEN** sind wie Zwillinge, die "genau gleich aussehen", aber doch zwei verschiedene Personen sind. **o**

Anmerkung: Man kann in Ada jeden Wert eines numerischen Typs **T1** ausdrücklich in einen entsprechenden Wert eines anderen numerischen Typs **T2** umwandeln lassen. Z.B. bezeichnet der Ausdruck **AEPFEL(M1)** einen Wert vom Typ **!AEPFEL**. Der Ausdruck **MELONEN(A1)** bezeichnet eine Wert vom Typ **!MELONEN**, löst aber evtl. die Ausnahme **constraint_error** aus, weil der Wert von **A1** zu groß ist. **o**

Der Untertyp **MELONEN** besteht aus einem neuen Typ **!MELONEN** und der Einschränkung **range 0..200**. Der Typ **!MELONEN** ist eine Kopie des Typs **!BIRNEN** und sieht somit "genauso aus" wie der Typ **!AEPFEL**. **!MELONEN**-Werte und **!BIRNEN**-Werte kann man nicht miteinander verrechnen.

Der Untertyp **ZITRONEN** besteht aus dem alten Typ **!MELONEN** und der Einschränkung **range 0..200**. **MELONEN**- und **ZITRONEN**-Werte **darf man** miteinander verrechnen, da sie zum selben Typ **!MELONEN** gehören. **o**

In diesem ersten Beispiel hatte es keinen großen Vorteil, den Typ **!BIRNEN** mit **new** als Kopie des Typs **!AEPFEL** zu vereinbaren. Der Typ **!BIRNEN** erbt zwar alle Operationen (**+**, **-**, *****, /, **rem** etc.) vom Typ **!AEPFEL**. Aber dieselben Operationen hätte er auch bekommen, wenn man ihn "ganz normal ohne **new**" vereinbart hätte. Im nächstens Beispiel ist es **lohnender**, den Typ **!AEPFEL** mit **new** zu kopieren.

Beispiel 23.1.2.: Einen Typ in einer **Paketspezifikation** vereinbaren und dann kopieren:

```
01 package OTTO is
02    type AEPFEL is range 0..500;
03    function  MAL2(A: AEPFEL) return AEPFEL;
04    procedure LOESCHE(A: in out AEPFEL);
05    ...
06 end OTTO;
```

Der Typ **!AEPFEL** wird hier (zusammen mit seinem ersten Untertyp AEPFEL) in einer **Paketspezifikation** vereinbart. In derselben Paketspezifikation werden die beiden Unterprogramme **MAL2** und **LOESCHE** vereinbart, die einen **AEPFEL**-Parameter bzw. ein **AEPFEL**-Ergebnis haben. Damit gehören diese Unterprogramme zu den **primitiven Operationen** des Typs **!AEPFEL**. Wenn man den Typ **!AEPFEL** mit **new** kopiert, **erbt** die Kopie all seine primitiven Operationen, z.B. so:

```
10 with OTTO;
11 procedure EMIL is
12    type BIRNEN is new OTTO.AEPFEL;
13    B1, B2 : BIRNEN := 15;
14    ...
15 begin
16    B1 := MAL2(A => B2);
17    LOESCHE(A => B2);
18    ...
```

Der Typ **!BIRNEN** ist eine Kopie des Typs **!AEPFEL** und erbt damit unter anderem Kopien der Unterprogramme **MAL2** und **LOESCHE**. Die geerbten Unterprogramme haben folgende Spezifikationen:

```
function  MAL2(A: BIRNEN) return BIRNEN;
procedure LOESCHE(A: in out BIRNEN);
```

Überall da, wo in den Spezifikationen der Originalunterprogramme der Untertyp **AEPFEL** steht, steht in den Spezifikationen der Kopien der Untertyp **BIRNEN**. o

Allgemein gilt: Wenn man Unterprogramme zur Bearbeitung von Werten eines Typs **T** zusammen mit dem Typ **T** in einer **Paketspezifikation** vereinbart, dann gehören diese Unterprogramme zu den **primitiven Operationen** von **T**. Wenn man **T** mit **new** kopiert, erbt die Kopie alle primitiven Operationen von **T**.

Im vorigen Beispiel wurde der Typ **!AEPFEL** in einer Paketspezifikation vereinbart und wurde dann einmal in einer Prozedur mit **new** kopiert. Das Kopieren kann auch schon in der Paketspezifikation stattfinden und "über mehrere Stufen gehen", wie das folgende Beispiel zeigt:

Beispiel 23.1.3.: Kopien kopieren:

```
01 package BERTA is
02    type AEPFEL is range 0..500;
03    function  MAL2(A: AEPFEL) return AEPFEL;
04    procedure LOESCHE(A: in out AEPFEL);
05    ...
06    type BIRNEN is new AEPFEL;
07    function  MAL5(B: BIRNEN; I: integer) return BIRNEN;
08    procedure LOESCHE(B: in out BIRNEN);
09    ...
10    type KIWIS is new BIRNEN;
11    function  MAL5(K: KIWIS; B: boolean) return KIWIS;
12    procedure LOESCHE(K: in out KIWIS; I: in integer);
```

Der Typ **!BIRNEN** erbt Kopien der Unterprogramme **MAL2** und **LOESCHE** und hat selbst zwei weitere primitive Operationen namens **MAL5** und **LOESCHE**. Weil diese neue **LOESCHE**-Prozedur für Birnen die gleiche Spezifikation wie die geerbete **LOESCHE**-Prozedur hat, wird die geerbte Prozedur **verdeckt**. Zum Typ **!BIRNEN** gehören also (außer den üblichen Ganzzahloperationen **+**, **-**, *****, /, **rem** etc.) **drei** primitive Operationen namens **MAL2**, **MAL5** und **LOESCHE**.

Der Typ **!KIWIS** erbt alle primitiven Operationen vom Typ **!BIRNEN** und hat selbst **zwei** weitere namens **MAL5** und **LOESCHE**. Keines der Unterprogramme wird von einem anderen verdeckt. Also gehören zum Typ **!KIWI** (außer den üblichen Ganzzahloperationen **+**, **-**, *****, /, **rem** etc.) insgesamt **fünf** primitive Operationen. Davon heißt eine **MAL2**, zwei heißen **MAL5** und die letzten beiden heißen **LOESCHE**. o

Aufgabe 23.1.1.: Geben Sie die **Spezifikationen** der drei primitiven Operationen an, die zum Typ **!BIRNEN** gehören. Ebenso für die fünf primitiven Operationen des Typs **!KIWIS**. o

Ein Unterprogramm U1 **verdeckt** ein Unterprogramm U2, wenn die Unterprogramme den **gleichen Namen** haben und ihre Spezifikationen **typkonform** (type conformant) sind. Das ist der Fall wenn gilt:
1. Beide Unterprogramme haben die **gleiche Anzahl von Parametern**
2. Wenn eines der Unterprogramme eine **Funktion** ist, dann ist auch das andere Unterprogramm eine Funktion und beide Funktionen haben **denselben Ergebnistyp**.
3. Entsprechende **Parameter** der beiden Unterprogramme (in der angegebenen Reihenfolge) haben **denselben Typ**.

Die **Namen** der Parameter und ihre **Modi** (**in**, **out** oder **in out**) haben keinen Einfluß auf die Typkonformität der Unterprogramme. Siehe dazu auch (ARM 6.3.1(15)).

Zusammenfassung 23.1.:

- Mit **new** kann man einen neuen Typ als Kopie eines alten Typs vereinbaren (**type T is new S**).
- Der alte und der neue Typ sind zwei **verschiedene** Typen.
- Der neue Typ **erbt** alle **primitiven Operationen** des alten Typs.
- Ein Unterprogramm U gehört zu den **primitiven Operationen** eines Typs T, wenn U einen T-Parameter oder ein T-Ergebnis hat und U und T in derselben Paketspezifikation vereinbart wurden.

23.2. Etikettierte Verbundtypen vereinbaren, kopieren und erweitern

Im vorigen Abschnitt wurde unter anderem gezeigt, wie man die Menge der **primitiven Operationen** eines Typs erweitern kann. In diesem Abschnitt wird gezeigt, wie man die **Datenkomponenten** eines Typs erweitern kann. In diesem Sinne **erweiterbar** sind in Ada die sogenannten **etikettierten Verbundtypen** (tagged record types). Warum diese Typen **etikettiert** heißen, wird erst später erläutert. Hier ein Beispiel dafür, wie man solche Typen vereinbart und erweitert:

Beispiel 23.2.1.: Etikettierte Verbundtypen vereinbaren und erweitern:

```
01 package TAG_A_01 is
02    type GLEIT is digits 6 range -10_000.0..+10_000.0;
03    ------------------------------------------------------------------------
04    type PUNKT is tagged record
05       X, Y: GLEIT;
06    end record;
07    function ABSTAND_U(P: PUNKT) return GLEIT;
08    -- Liefert den Abstand zwischen P und dem Ursprung (0.0, 0.0)
09    procedure SPIEGEL_U(P: in out PUNKT);
10    -- Spiegelt P am Ursprung
11    procedure PUT(ITEM: PUNKT);
12    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
13    ------------------------------------------------------------------------
14    type QUADRAT is new PUNKT with record   -- QUADRAT ist Sohn von PUNKT
15       DELTA_X: GLEIT;
16    end record;
17    function FLAECHE(Q:QUADRAT) return GLEIT;
18    -- Liefert die Flaeche von Q
19    function UMFANG(Q: QUADRAT) return GLEIT;
20    -- Liefert den Umfang von Q
21    procedure PUT(ITEM: QUADRAT);
22    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
23    ...
```

Der Typ **!GLEIT** (siehe Zeile 02) ist ein ganz normaler Gleitpunkttyp.

Der Typ **!PUNKT** ist ein **etikettierter Verbundtyp**. Jeder Wert dieses Typs besteht aus zwei Komponenten namens **X** und **Y**. Zu den primitiven Operationen dieses Typs gehören die Unterprogramme **ABSTAND_U**, **SPIEGEL_U** und **PUT**.

Der Typ **!QUADRAT** ist ebenfalls ein **etikettierter Verbundtyp** und wird hier (in Zeile 14 bis 16) als eine **Erweiterung** des Typs **!PUNKT** vereinbart. Jeder Wert des Typs **!QUADRAT** besteht aus **drei** Komponenten namens **X**, **Y** (geerbt) und **DELTA_X** (neu). Wir bezeichnen den Typ **!PUNKT** auch als den **Vatertyp** des Typs **!QUADRAT** und **!QUADRAT** als einen **Sohn** von **!PUNKT**.

Der Typ **!QUADRAT** erbt von seinem Vatertyp **!QUADRAT** die beiden Datenkomponenten **X** und **Y** und die primitiven Operationen **ABSTAND_U**, **SPIEGEL_U** und **PUT**. Die geerbte **PUT**-Prozedur wird aber durch eine "eigene PUT-Prozedur für QUADRAT-Objekte" (spezifiziert in Zeile 21) **verdeckt**. Außerdem gehören zum Typ **!QUADRAT** zwei neue primitive Operationen namens **FLAECHE** und **UMFANG**. **o**

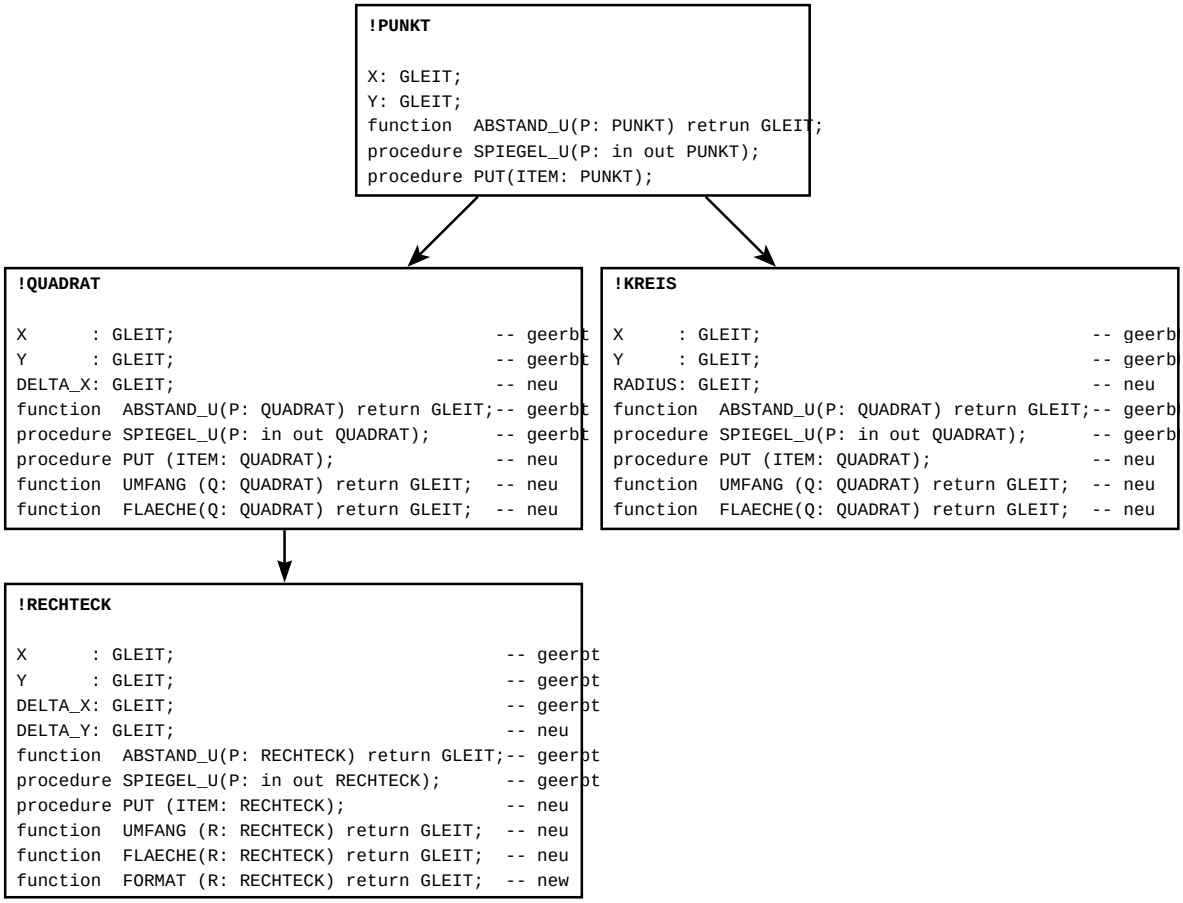
Aufgabe 23.2.1.: Ergänzen Sie die Spezifikation des Paketes **TAG_A_01**, indem Sie dort noch folgende Größen vereinbaren:
1. Einen Typ **!RECHTECK** (als Erweiterung von **!QUADRAT**) mit einer zusätzlichen Komponente **DELTA_Y** und neuen primitiven Operationen **UMFANG**, **FLAECHE**, **PUT** und **FORMAT** (das Format eines Rechtecks ist der Quotient seiner Seitenlängen).
2. Einen Typ **!KREIS** (als Erweiterung von **!PUNKT**) mit einer zusätzlichen Komponenten **RADIUS** und neuen primitiven Operationen **UMFANG**, **FLAECHE** und **PUT**. **o**

Aufgabe 23.2.2.: Geben Sie für jeden der Typen **!PUNKT**, **!QUADRAT**, **!RECHTECK** und **!KREIS** (vereinbart in der Spezifikation des Paketes **TAG_A_01**) eine Liste seiner **primitiven Operationen** an. Geben Sie in dieser Liste nicht nur die Namen der Unterprogramme an, sondern ihre vollständigen Spezifikationen. Welches "neue" Unterprogramm verdeckt welches geerbte Unterprogramm? **o**

Aufgabe 23.2.3.: Schreiben Sie einen geeigneten Rumpf für das Paket **TAG_A_01**. Um den Abstand eines Punktes vom Ursprung zu berechnen, empfiehlt es sich, die Paketschablone **ada.numerics.generic_elemenatary_functions** (siehe (ARM A.5.1)) mit dem Typ **GLEIT** zu instanziieren. In dem Instanzpaket findet man unter anderem eine Quadratwurzelfunktion namens **sqrt**. Die Komponente **DELTA_X** eines Quadrats oder Rechtecks soll die **Hälfte** der Länge einer Seite in x-Richtung enthalten (den Abstand des Mittelpunkts von den Seiten). Ensprechend soll **DELTA_Y** die **Hälfte** der Länge einer Seite in y-Richtung enthalten. Im Paket **ada.numerics** ist eine Konstante namens **pi** vereinbart, die bei der Berechnung eines Kreisumfangs oder einer Kreisfläche nützlich ist.

Aufgabe 23.2.4.: Schreiben Sie eine Prozedur namens **TAG_A_02**, die das Paket **TAG_A_01** einbindet und die darin enthaltenen Typen und Unterprogramme benützt. Vereinbaren Sie ein paar **Punkte**, **Quadrate**, **Rechtecke** und **Kreise**, Spiegeln Sie sie am Ursprung, berechnen Sie ihre Umfänge und Flächen etc. und geben Sie alle Ergebnisse zur aktuellen Ausgabe aus. **o**

Die im Paket **TAG_A_01** vereinbarten Typen **!PUNKT**, **!QUADRAT**, **!RECHTECK** und **!KREIS** (siehe oben Beispiel 23.2.1. und die Musterlösung zu Aufgabe 23.2.1.) bilden eine **Ist-Erweiterung-von-Hierarchie** oder Baumstruktur:



Jedes **QUADRAT**-Objekt hat alle **Merkmale** (Datenkomponenten und Unterprogramme) eines **PUNKT**-Objekts, aber ein **PUNKT**-Objekt hat nicht alle Merkmale eines **QUADRAT**-Objekts. Deshalb darf man ein **QUADRAT**-Objekt in ein **PUNKT**-Objekt umwandeln, aber eine umgekehrte Umwandlung ist verboten. Sei **P1** ein **PUNKT**-Objekt und **Q1** ein **QUADRAT**-Objekt. Dann bezeichnet **PUNKT(Q1)** das in **Q1** enthaltene **PUNKT**-Objekt, aber eine Umwandlung wie **QUADRAT(P1)** ist nicht erlaubt. Für die anderen Typen gilt entsprechendes.

Entsprechend dieser Typhierarchie ist jedes **QUADRAT**-Objekt und jedes **KREIS**-Objekt auch ein **PUNKT**-Objekt und jedes **RECHTECK**-Objekt ist auch ein **QUADRAT**-Objekt. Diese Typhierarchie hat also nichts mit der üblichen mathematischen Klassifizierung zu tun, die besagt, daß jedes **Quadrat** auch ein **Rechteck** ist und ein **Quadrat** oder **Kreis** nur in einem extremen Sonderfall ein **Punkt** ist. **Typhierarchien** haben häufig eine ganz andere Struktur, als Klassifizierungen und Hierarchien, die aus anderen Zusammenhängen und Wissensgebieten stammen.

Zusammenfassung 23.2.:

- Einen **etikettierten Verbundtyp** EV2 kann man als **Erweiterung** eines schon vorhandenen etikettierten Verbundtyps EV1 vereinbaren.
- Der neue Typ EV2 **erbt** alle **Datenkomponenten** des alten Typs EV1 und kann weitere Datenkomponenten hinzufügen.

- Außerdem **erbt** der der neue Typ alle **primitiven Operationen** des alten Typs und kann diese durch neue Operationen verdecken oder ergänzen.

23.3. Klassenweite Typen und dynamische Wahlaufrufe (dispatching calls)

Das Beispiel in diesem Abschnitt stützt sich auf das Paket **TAG_A_01**, welches im vorigen Abschnitt vorgestellt wurde.

Angenommen, wir wollen Objekte der Typen **!PUNKT**, **!QUADRAT**, **!RECHTECK** und **!KREIS** verschieben (um **DX** in x-Richtung und um **DY** in y-Richtung) und anschließend mit der **PUT**-Prozedur des jeweiligen Typs ausgeben. Dann könnten wir für jeden der vier Typen ein eigenes Unterprogramm schreiben, welches diese kleine Aufgabe löst. Eine elegantere und wartungsfreundlichere Lösung besteht aus **einem** einzigen Unterprogramm, mit dem man Objekte aller vier Typen verschieben und ausgeben kann.

Beispiel 23.3.1.: Die Prozedur SCHIEBE: Eine für alle:

Die folgende Prozedur kann man z.B. im Paket **TAG_A_01** vereinbaren:

```
01 procedure SCHIEBE(CP: in out PUNKT'class; DX, DY: GLEIT) is
02 begin
03    CP.X := CP.X + DX;
04    CP.Y := CP.Y + DY;
05    PUT(ITEM => CP);    -- Welche PUT-Prozedur wird hier aufgerufen?
06 end SCHIEBE;
```

Der formale Parameter **CP** der Prozedur **SCHIEBE** gehört nicht zu einem spezifischen Untertyp wie **PUNKT**, **QUADRAT** etc., sondern zu dem **klassenweiten Untertyp PUNKT'class**. Zu diesem Untertyp gehören alle Objekte, die zum Untertyp **PUNKT** oder zu einem von **PUNKT** mit **new** abgeleiteten Untertyp gehören. Konkret bedeutet das hier: Wenn man die Prozedur **SCHIEBE** aufruft, darf man als aktuellen Parameter für **CP** wahlweise ein **PUNKT**-Objekt, ein **QUADRAT**-Objekt, ein **RECHTECK**-Objekt oder ein **KREIS**-Objekt angeben. Der aktuellen Parameter muß nur mindestens alle Merkmale eines **PUNKT**-Objektes haben.

In Zeile 05 wird eine Prozedur namens **PUT** aufgerufen. Nun gibt es zahlreiche Prozeduren, die alle **PUT** heißen, unter anderem eine für **PUNKT**-Objekte, eine andere für **QUADRAT**-Objekte, eine weitere für **RECHTECK**-Objekte und schließlich eine für **KREIS**-Objekte. Welche von diesen vier **PUT**-Prozeduren ausgeführt wird, legt der Ausführer nicht wie üblich **statisch** fest (wenn man ihm das Programm übergibt, zur "Compilezeit"), sondern **dynamisch** (wenn er das Programm ausführt, "zur Laufzeit"). Jedesmal, wenn der Ausführer die Prozedur **SCHIEBE** ausführt, wählt er (in Zeile 05) die "zu CP passende" **PUT**-Prozedur aus: Ist **CP** ein **PUNKT**-Objekt, wird die **PUT**-Prozedur für **PUNKT**-Objekte ausgeführt, ist **CP** ein **QUADRAT**-Objekt, wird die **PUT**-Prozedur für **QUADRAT**-Objekte ausgeführt etc.. In Zeile 05 steht also ein Aufruf, bei dem die aufgerufene Prozedur **dynamisch bestimmt** wird, und zwar anhand des aktuellen Wertes eines Parameters (im Beispiel: **CP**). **o**

Im Englischen bezeichnet man solche Unterprogrammaufrufe als **dispatching calls**. Hier werden sie als **dynamische Wahlaufrufe** oder kurz als **Wahlaufrufe** bezeichnet, weil der Ausführer unter mehreren Unterprogrammen "das richtige" wählen muß.

Damit der Ausführer diese Wahl korrekt durchführen kann, ist jedes Objekt eines etikettierten Verbundtyps mit einer **zusätzlichen Komponenten** versehen, in der ihr Typ steht (in irgendwie kodierter Form). Diese Komponente wird im Englischen als **tag** und hier als **Etikett** bezeichnet. Wegen dieser Komponenten werden die entsprechenden Typen als **etikettierte Verbundtypen** (tagged record types) bezeichnet.

Wenn der Ausführer die Prozedur **SCHIEBE** ausführt und zur Zeile 05 kommt, prüft er das Etikett von **CP** und wählt anhand seines Wertes die richtige **PUT**-Prozedur aus.

Wahlaufrufe (dispatching calls) sind in Adaprogrammen die einzigen Stellen, an denen während der **Ausführung** eines Programms ("zur Laufzeit") geprüft werden muß, zu welchem **Typ** ein Objekt gehört. Alle anderen Typprüfungen und damit zusammenhängenden Maßnahmen können schon bei der **Übergabe** eines Programms ("zur Compilezeit") durchgeführt werden. Deshalb brauchen die Objekte "normaler Typen" auch kein Typetikett und dieses Etikett unterscheidet normale Typen von etikettierten Verbundtypen.

23.4. Abstrakte Typen erweitern

Der Typ **!PUNKT** wurde im Paket **TAG_A_01** als **konkreter** Typ vereinbart. Deshalb kann ein Paketbenutzer, der in seinem Programm eine Variable **P1** vom Untertyp **TAG_A_01.PUNKT** vereinbart hat, auf die Komponenten der Variablen (**P1.X** und **P1.Y**) zugreifen. Insbesondere darf er seiner Variablen **P1** ein Aggregat wie z.B. **(X => 3.0, Y => 2.0)** zuweisen.

Damit sind bestimmte nachträgliche **Verbesserungen** des Paketes praktisch ausgeschlossen. Z.B. kann man die kartesischen Koordinaten **X** und **Y** der **PUNKT**-Objekte nicht durch Polarkoordinaten **RO** und **DELTA** ersetzen (wobi **RO** ein Winkel zur x-Achse und **DELTA** ein Abstand vom Ursprung ist).

Häufig ist es günstiger, einen Typ wie **!PUNKT** als **abstrakten** Typ zu vereinbaren, so daß Benutzer des Typs nicht auf die Komponenten von **PUNKT**-Variablen zugreifen können. In Ada muß man den Typ dazu als **privaten** oder **limitiert privaten** Typ vereinbaren. Das folgende Beispiel zeigt, wie man einen abstrakten (d.h. privaten) Typ **erweitern** kann.

Beispiel 23.4.: Einen privaten Typ erweitern

```
01 package TAG_B_01 is
02    type GLEIT is digits 6 range -10_000.0..+10_000.0;
03    ------------------------------------------------------------------------
04    type PUNKT is tagged private;
05
06    function  MAKE       (MX, MY: GLEIT) return PUNKT;
07    -- Macht aus MX und MY einen Punkt und liefert ihn als Ergebnis
08    procedure MACHE_PUNKT(MX, MY: GLEIT; P: in out PUNKT);
09    -- Bringt die Werte MX und MY in den Punkt P hinein.
10    function ABSTAND_U(P: PUNKT) return GLEIT;
11    -- Liefert den Abstand zwischen P und dem Ursprung (0.0, 0.0)
12    procedure PUT(ITEM: PUNKT);
13    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
14    ------------------------------------------------------------------------
15    type QUADRAT is new PUNKT with private;
16
17    function MAKE(MX, MY: GLEIT)        return QUADRAT;
18    -- Diese Funktion muss sein, weil der Typ QUADRAT vom Typ PUNKT eine
```

```
19    -- abstrakte Funktion MAKE (mit 2 GLEIT-Parametern) erbt!
20    function MAKE(MX, MY, SEITE: GLEIT) return QUADRAT;
21    -- Eigentliche MAKE-Funktion fuer QUADRAT (mit 3 GLEIT-Paramtern).
22    -- Macht aus MX, MY und SEITE ein Quadrat und liefert es als Ergebnis.
23    procedure MACHE_QUADRAT(MX, MY, SEITE: GLEIT; Q: in out QUADRAT);
24    -- Bringt die Werte MX, MY und SEITE in das Quadrat Q hinein.
25
26    function FLAECHE(Q:QUADRAT) return GLEIT;
27    -- Liefert die Flaeche von Q
28    function UMFANG(Q: QUADRAT) return GLEIT;
29    -- Liefert den Umfang von Q
30    procedure PUT(ITEM: QUADRAT);
31    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
32    ------------------------------------------------------------------------
33 private
34    type PUNKT is tagged record
35       X, Y: GLEIT;
36    end record;
37
38    type QUADRAT is new PUNKT with record
39       DELTA_X: GLEIT;
40    end record;
41 end TAG_B_01;
```

Der Typ **!PUNKT** wird hier als **privater** Typ vereinbart (in Zeile 02 steht die unvollständige und im privaten Teil in Zeile 34 bis 36 die vollständige Vereinbarung). Damit ist der Typ **abstrakt** und ein Paketbenutzer kann einer **PUNKT**-Variablen **P1** nicht einfach ein Aggregat wie z.B. **(X => 3.0, Y => 2.0)** zuweisen. Als Ersatz muß das Paket **TAG_B_01** seinen Benutzern sogenannte **Konstruktoren** zur Verfügung stellen. Das sind Funktionen oder Prozeduren, mit denen man den Wert eines Objekts eines abstrakten Typs festlegen oder verändern kann.

Anmerkung: In anderen objektorientierten Sprachen sind **Konstruktoren** ganz besondere Befehle, für die **spezielle Regeln** gelten (z.B. werden sie **nicht** vererbt). In Ada sind Konstruktoren "ganz normale Unterprogramme", für die es keine besonderen Regeln gibt. Man bezeichnet ein Unterprogramm als **Konstruktor**, wenn es dazu dient, ein Objekt aus seinen Datenkomponenten zusammenzusetzen. **o**

In Zeile 06 wird eine **Konstruktor-Funktion** namens **MAKE** für **PUNKT**-Objekte und in Zeile 08 eine **Konstruktor-Prozedur** namens **MACHE_PUNKT** spezifiziert.

Konstruktor-**Funktionen** sind ein bißchen eleganter als **Prozeduren**, weil man sie schon bei der Vereinbarung einer Variablen zum Initialisieren verwenden kann. Im Zusammenhang mit Typerweiterung verursachen sie leider ein **Problem**, welches unten beschrieben wird. Konstruktor-**Prozeduren** sind weniger elegant als Funktionen, aber auch bei Typerweiterungen **unproblematisch**.

Der Typ **!QUADRAT** wird im öffentlichen Teil der Paketspezifikation als Erweiterung des Typs **!PUNKT** vereinbart (Zeile 15). Deshalb erbt er die primitiven Operationen des Typs **!PUNKT**.

Unter anderem erbt **!QUADRAT** eine Kopie der Konstruktor-Funktion **MAKE**. Die geerbte Kopie hat folgende Spezifikation:

```
function MAKE(MX, MY: GLEIT) return QUADRAT;
```

Hier tritt ein Problem auf: Die Original-Funktion **MAKE** liefert ein **PUNKT**-Objekt. Die Kopie der Funktion soll ein **QUADRAT**-Objekt liefern. Aber welchen Wert soll die kopierte Funktion der Komponenten **DELTA_X** ihres Ergebnisses geben?

Dieses Problem tritt immer dann auf, wenn das Ergebnis einer Funktion zu einem **privaten Typ** gehört und die Funktion **vererbt** wird. In solchen Fällen wird die Funktion nur als **abstrakte** Funktion vererbt. Praktisch bedeutet das: Der Programmierer **muß** die geerbte Funktion mit einer "selbst programmierten konkreten" Funktion **verdecken**. Das kann ziemlich lästig sein. In Zeile 17 wird eine entsprechende **MAKE**-Funktion spezifiziert. Im Rumpf dieser Funktion kann man der **DELTA_X**-Komponente (durch die sich ein Quadrat von einem Punkt unterscheidet) einen Standardwert zuweisen, z.B. 0.0. Als Alternative kann man seinen Kollegen den Aufruf dieser **MAKE**-Funktion verbieten und nur eine Ausnahme auslösen, wenn sie doch aufgerufen wird. Beide Lösungen sind nicht besonders schön.

Bei Konstruktor-**Prozeduren** tritt dieses Problem nicht auf. In Zeile 08 wird eine Konstruktor-Prozedur für Punkte und in Zeile 23 eine für Quadrate spezifiziert. Der Typ **!QUADRAT** erbt zwar eine Kopie von **MACHE_PUNKT** mit der Spezifikation

```
procedure MACHE_PUNKT(MX, MY: GLEIT; P: out QUADRAT);
```

Aber diese Prozedur **muß man nicht verdecken**. Man kann hoffen, daß niemand eine Prozedur namens **MACHE_PUNKT** aufruft, um einem **QUADRAT**-Objekt einen Wert zuzuweisen. Wenn es doch passieren sollte, bleibt die **DELTA_X**-Komponente von **P** einfach unverändert. **o**

In Zeile 15 könnte man statt **is new PUNKT with private** auch einfach **is private** schreiben. Dann wäre "in der Öffentlichkeit" aber nicht bekannt, daß **!QUADRAT** eine Erweiterung von **!PUNKT** ist, und damit würde **!QUADRAT** in der Öffentlichkeit auch nichts von **!PUNKT** erben. Nur im privaten Teile der Paketspezifikation und im Rumpf des Paketes **TAG_B_01** wäre sichtbar, daß **!QUADRAT** eine Erweiterung von **!PUNKT** ist und nur dort würde **!QUADRAT** von **!PUNKT** erben.

Zusammenfassung 23.4.:

- Auch etikettierte Verbundtypen kann man als **private** Typen vereinbaren.
- Auch **private** etikettierte Typen kann man **erweitern**.
- Für private Typen muß man den Benutzern **Konstruktoren** zur Verfügung stellen.
- Konstruktor-**Funktionen** sind elegant, werden aber als **abstrakte** Funktionen vererbt und müssen dann **verdeckt** werden.
- Konstruktor-**Prozeduren** sind weniger elegant, werden aber als **konkrete** Prozeduren vererbt und müssen somit **nicht** verdeckt werden.

23.5. Abstrakte Typen in verschiedenen Paketen erweitern

Im vorigen Abschnitt wurden zwei abstrakte Typen vereinbart, von denen der eine (**!QUADRAT**) eine **Erweiterung** des anderen (**!PUNKT**) ist. Beide Typen wurden im selben Paket (TAG_B_01) vereinbart.

Häufig hat man einen abstrakten Typ **T1** in einem Paket **P1** vereinbart, und möchte dann einen Typ **T2** als Erweiterung von **T1** vereinbaren. Das Paket **P1** mit dem Typ **T1** ist vielleicht schon seit

einiger Zeit in Gebrauch und wird von vielen Programmen **B1**, **B2** etc. benützt. Es wäre ungünstig, den neuen erweiterten Typ **T2** ebenfalls im Paket **P1** zu vereinbaren, denn dazu müßte man "das bewährte Paket **P1**" mit einem Editor verändern. Dabei könnten sich Fehler einschleichen und bewirken, daß die Programme **B1**, **B2** etc. mit der neuen Version des Paketes **P1** nicht mehr laufen. Aus solchen Gründen möchte man das Paket P1 lieber unangetastet lassen und die Erweiterung **T2** in einem neuen Paket **P2** vereinbaren.

Dabei ist es wichtig, **zwei Fälle** zu unterscheiden, je nachdem, "welche Eigenschaften" des Typs **T1** (der erweitert werden soll) man sehen muß:

Fall 1: Man kommt mit den **öffentlichen** Eigenschaften des Typs **T1** aus.
Fall 2: Man muß auch auf die **privaten** Eigenschaften des Typs **T1** zugreifen.

Im **Fall 1** kann man die Erweiterung **T2** in einem "normalen" Paket **P2** vereinbaren. Im **Fall 2** muß man die Erweiterung **T2** in einem sogenannten **Kindpaket** (child package) von **P1** vereinbaren. Die folgenden beiden Beispiele sollen deutlich machen, was das konkret bedeutet. In diesen Beispielen passiert folgendes:

1. In einem Paket namens **TAG_C_01** wird ein privater Typ **!PUNKT** vereinbart.
2. In einem Paket namens **TAG_C_02** wird ein privater Typ **!QUADRAT** als Erweiterung von **!PUNKT** vereinbart. Dabei wird nur auf die **öffentlichen** Eigenschaften von **!PUNKT** zugegriffen.
3. In einem Paket namens **TAG_C_02.KIND1** wird ein privater Typ **!RECHTECK** als Erweiterung von **!QUADRAT** vereinbart. Dabei wird auch auf **private** Eigenschaften von **!QUADRAT** zugegriffen.

Beispiel 23.5.1.: Einen privaten Typ öffentlich erweitern:

Im Paket **TAG_C_01** wird ein privater Typ **!PUNKT** vereinbart:

```
01 with ada.text_io;
02 package TAG_C_01 is
03    type GLEIT is digits 6 range -10_000.0..+10_000.0;
04    type PUNKT is tagged private;
05    procedure MACHE_PUNKT(MX, MY: GLEIT; P: out PUNKT);
06    -- Macht aus P einen Punkt mit Koordinaten MX und MY
07    function  ABSTAND_U(P: PUNKT) return GLEIT;
08    -- Liefert den Abstand zwischen P und dem Ursprung (0.0, 0.0)
09    procedure PUT(ITEM: PUNKT);
10    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
11    procedure SCHIEBE(CP: in out PUNKT'class; DX, DY: GLEIT);
12    -- Verschiebt CP um (DX, DY) und gibt CP aus (mit
13    -- der zu CP passenden PUT-Prozedur)
14    package GLEIT_EA is new ada.text_io.float_io(num => GLEIT);
15 private
16    type PUNKT is tagged record
17       X, Y: GLEIT;
18    end record;
19 end TAG_C_01;
```

Die Unterprogramme **MACHE_PUNKT**, **ABSTAND_U** und **PUT** gehören zu den primitiven Operationen des privaten Typs **!PUNKT**.

Mit der Prozedur **SCHIEBE** kann man nicht nur Objekte des hier und jetzt vereinbarten Typs **!PUNKT** verschieben und ausgeben, sondern auch Objekte, deren Typen erst später in anderen

Paketen vom Typ **!PUNKT** abgeleitet werden. Darin liegt die besondere Stärke von klassenweiten Typen wie **!PUNKT'class** und dynamischen Wahlaufrufen.

Der Rumpf des Paketes **TAG_C_01** wird hier nicht wiedergegeben.

Im Paket **TAG_C_02** wird ein privater Typ **!QUADRAT** als Erweiterung des Typs **!PUNKT** vereinbart. Die "privaten Eigenschaften" von **!PUNKT** sind im Paket **TAG_C_02 nicht** sichtbar.

```
01 with TAG_C_01;
02 use type TAG_C_01.GLEIT;
03 package TAG_C_02 is
04    subtype GLEIT is TAG_C_01.GLEIT;
05    type QUADRAT is new TAG_C_01.PUNKT with private;
06    procedure MACHE_QUADRAT(MX, MY, SEITE: GLEIT; Q: out QUADRAT);
07    -- Macht aus Q ein Quadrat mit Mittelpunkt bei (MX, MY) und
08    -- Seiten der Laenge SEITE
09    function  FLAECHE(Q: QUADRAT) return GLEIT;
10    -- Liefert die Flaeche von Q
11    procedure PUT(ITEM: QUADRAT);
12    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
13 private
14    type QUADRAT is new TAG_C_01.PUNKT with record
15       DELTA_X: GLEIT;
16    end record;
17 end TAG_C_02;
```

Der private Typ **!QUADRAT** wird als Erweiterung des privaten Typs **!TAG_C_01.PUNKT** vereinbart (siehe Zeile 05 und Zeile 14 bis 16).

Die **use**-Klausel in Zeile 02 dient der Bequemlichkeit. Sie macht die Operationen **+**, **-**, *****, **/** etc. des Typs **GLEIT** direkt sichtbar. Statt **TAG_C_01."-"(3.0)** kann man jetzt einfach **-3.0** schreiben.

Die Untertypvereinbarung in Zeile 04 dient auch nur der Bequemlichkeit. Ohne sie müßte man immer **TAG_C_01.GLEIT** schreiben statt einfach **GLEIT**.

Jedes **QUADRAT**-Objekt besteht aus einem **PUNKT**-Objekt und aus einer zusätzlichen Komponente **DELTA_X**. Ein **PUNKT**-Objekt besteht aus zwei Komponenten namens **X** und **Y** (siehe oben, Paket **TAG_C_01**). Auf diese "privaten Eigenschaften" des Typs **!PUNKT** wird im Paket **TAG_C_02** aber **nicht** direkt zugegriffen, auch nicht in seinem Rumpf:

```
01 with ada.text_io;
02 package body TAG_C_02 is
03    subtype PUNKT is TAG_C_01.PUNKT;
04    package GLEIT_EA renames TAG_C_01.GLEIT_EA;
05
06    procedure MACHE_QUADRAT(MX, MY, SEITE: GLEIT; Q: out QUADRAT) is
07    begin
08       TAG_C_01.MACHE_PUNKT(MX => MX, MY => MY, P => PUNKT(Q));
09       Q.DELTA_X := SEITE / 2.0;
10    end MACHE_QUADRAT;
11
12    function FLAECHE(Q: QUADRAT) return GLEIT is
13    begin
14       return 4.0 * Q.DELTA_X**2;
15    end FLAECHE;
16
17    procedure PUT(ITEM: QUADRAT) is
18    begin
19       ada.text_io.put(item => "Quadrat mit Mittelpunkt bei ");
20       TAG_C_01   .PUT(ITEM =>  PUNKT(ITEM));
21       ada.text_io.put(item => "Seite:    ");
22       GLEIT_EA   .put(item =>  2.0 * ITEM.DELTA_X);
```

```
23       ada.text_io.new_line;
24       ada.text_io.put(item => "Flaeche:  ");
25       GLEIT_EA   .put(item =>  FLAECHE(Q => ITEM));
26       ada.text_io.new_line(spacing => 2);
27    end PUT;
28 end TAG_C_02;
```

Die Zeilen 03 und 04 dienen der Bequemlichkeit. Statt **TAG_C_01.PUNKT** kann man danach einfach **PUNKT** und statt **TAG_C_01.GLEIT_EA** einfach **GLEIT_EA** schreiben.

In der Konstruktor-Prozedur **MACHE_QUADRAT** wird die Konstruktor-Prozedur **MACHE_-PUNKT** aufgerufen (Zeile 08). Dieser Aufruf einer **öffentlichen** Prozedur erspart einen direkten Zugriff auf die **privaten** Komponenten **Q.X** und **Q.Y**.

Die gleiche Technik wird in der **PUT**-Prozedur (Zeile 17 bis 27) angewendet. Der **PUNKT**-Anteil des **QUADRAT**-Objekts **ITEM** wird mit der öffentlichen Prozedur **TAG_C_01.PUT** ausgegeben (Zeile 20).

Wichtig an diesem Beispiel ist: Der private Typ **!PUNKT** aus dem Paket **TAG_C_01** wurde in dem "ganz normalen" Paket **TAG_C_02** zu einem (ebenfalls privaten) Typ **!QUADRAT** erweitert. Auf **private Eigenschaften** von **!PUNKT** konnte dabei **nicht** zugegriffen werden.

Beispiel 23.5.2.: Einen privaten Typ privat erweitern

In der folgenden Paketspezifikation wird ein Typ **!RECHTECK** als Erweiterung von **!QUADRAT** vereinbart:

```
01 package TAG_C_02.KIND1 is
02    type RECHTECK is new QUADRAT with private;
03    procedure MACHE_RECHTECK(MX, MY, SEITE_X, SEITE_Y: GLEIT;
04                             R: out RECHTECK);
05    -- Macht aus R ein Rechteck mit MX, MY, SEITE_X und SEITE_Y.
06    function  FLAECHE(R: RECHTECK) return GLEIT;
07    -- Liefert die Flaeche von R.
08    procedure PUT(ITEM: RECHTECK);
09    -- Gibt ITEM in Textform zur aktuellen Ausgabe aus
10 private
11    type RECHTECK is new QUADRAT with record
12       DELTA_Y: GLEIT;
13    end record;
14 end TAG_C_02.KIND1;
```

Aus dem Namen **TAG_C_02.KIND1** folgt, daß es sich um eine **Kindeinheit** (child unit) des Paketes **TAG_C_02** handelt. **TAG_C_02** wird auch als das **Elternpaket** der Kindeinheit **TAG_C_02.KIND1** bezeichnet. Kindeinheiten genießen bei ihrem Elternpaket gewisse Privilegien: Im privaten Teil und im Rumpf von **TAG_C_01.KIND1** ist der private Teil des Paketes **TAG_C_01** sichtbar. Im Kindpaket **TAG_C_01.KIND1** ist somit die **innere Struktur** eines **QUADRAT**-Objektes sichtbar.

Im Rumpf des Kindpaketes ist vor allem die Funktion **FLAECHE** interessant:

```
01 with ada.text_io;
02 package body TAG_C_02.KIND1 is
...
06    procedure MACHE_RECHTECK ... end MACHE_RECHTECK;
...    ...
12    function FLAECHE(R: RECHTECK) return GLEIT is
13    begin
```

```
14       return 4.0 * R.DELTA_X * R.DELTA_Y;
15    end FLAECHE;
16
17    procedure PUT ... end PUT;
...   ...
30 end TAG_C_02.KIND1;
```

In Zeile 14 wird auf die Komponenten **R.DELTA_X** und **R.DELTA_Y** zugegriffen. Daß jedes **RECHTECK**-Objekt eine Komponente namens **DELTA_Y** hat, wurde hier im Paket **TAG_C_-01.KIND1** festgelegt (siehe oben in der Spezifikation Zeile 12). Dagegen gehört die **DELTA_X**-Komponente zu den "privaten Eigenschaften" des Typs **!QUADRAT** und wurde schon im Paket **TAG_C_02** vereinbart. Hier darf auf diese Komponente nur deshalb zugegriffen werden, weil **TAG_C_01.KIND1** eine Kindeinheit von **TAG_C_01** ist.

Wichtig an diesem Beispiel ist: Der private Typ **!QUADRAT** aus dem Paket **TAG_C_02** wurde in dem **Kindpaket TAG_C_02.KIND1** zu einem (ebenfalls privaten) Typ **!RECHTECK** erweitert. Dabei konnte auf private Eigenschaften von **!QUADRAT** zugegriffen werden (in der Funktion **FLAECHE**). **o**

Aufgabe 23.5.1.: Schreiben Sie die Prozeduren **MACHE_RECHTECK** und **PUT**, die im Rumpf von **TAG_C_02.KIND1** nur angedeutet sind. Müssen Sie dabei auf private Eigenschaften des Typs **!QUADRAT** zugreifen, oder kommen Sie mit Aufrufen der öffentlichen Unterprogramme aus? **o**

Aufgabe 23.5.2.: Übergeben Sie die drei Pakete **TAG_C_01**, **TAG_C_02** und **TAG_C_01.KIND1** und die Prozedur **TAG_C_03** einem maschinellen Ada-Ausführer und lassen Sie das Programm **TAG_C_03** ausführen. Verändern Sie die Prozedur **TAG_C_03** ein bißchen, indem Sie dort weitere **PUNKT**-Objekte und **QUADRAT**-Objekte **RECHTECK-Objekte** vereinbaren und bearbeiten. **o**

Zusammenfassung 23.5.:

- Wenn ein abstrakter Typ **T1** in einem Paket **P1** vereinbart wurde, kann man eine Erweiterung **T2** von **T1** auch ("später") in einem anderen Paket **P2** vereinbaren.
- Ist **P2** ein "normales Paket", dann sind die "privaten Eigenschaften" von **T1** dort **nicht** sichtbar.
- Ist **P2** dagegen ein **Kindpaket** von **P1**, dann sind dort die "privaten Eigenschaften" von **T1** sichtbar.
- Wenn möglich (aber nur dann!) sollte man die Erweiterung **T2 ohne** Zugriff auf die privaten Eigenschaften von **T1** vereinbaren.

23.6. Nur-Vater-Typen und ein polymorpher Speicher

Normalerweise kann der Programmierer einen **Typ** T auf **zwei** verschiedene Weisen benützen:

1. Er kann den Ausführer **Objekte** dieses Typs erzeugen lassen.
2. Er kann T als **Vatertyp** benützen, d.h. mit **new** weitere Typen von T ableiten.

Einen **nur-Vater-Typ** kann man nur als Vatertyp, aber nicht zum Erzeugen von Objekten verwenden.

Im folgenden Beispiel wird mit Hilfe eines nur-Vater-Typs ein **Speicher** realisiert, in dem man Werte ganz verschiedener Typen ablegen kann. Dabei soll der Benutzer des Speichers bestimmen können, wieviele Werte von welchem Typ im Speicher ablegt.

Zum Vergleich: In einer **Reihung** kann man nur Werte **eines** bestimmten (Komponenten-) Typs speichern. In einem Verbund kann man zwar Werte **verschiedener** Typen ablegen, aber wieviele Werte zu welchem Typ gehören wird durch den Bauplan des Verbundes festgelegt, nicht vom Benutzer des Verbundes.

Beispiel 23.6.1.: Mit einem **nur-Vater-Typ** fängt es an:

```
01 package TAG_D_01 is
02    type SPEICHERBAR is abstract tagged null record;
03    procedure PUT   (ITEM: in SPEICHERBAR) is abstract;
04    procedure PUT_CW(ITEM: in SPEICHERBAR'class);
05    type Z_SPEICHERBAR is access all SPEICHERBAR'class;
06    type INDEX         is range 1..5; -- 5 Speicherplaetze
07    procedure TUE_REIN(WOHIN: in INDEX; WAS: in  Z_SPEICHERBAR);
08    procedure HOL_RAUS(WOHER: in INDEX; WAS: out Z_SPEICHERBAR);
09 end TAG_D_01;
```

Der Typ **!SPEICHERBAR** ist ein etikettierter Typ ohne Komponenten (**null record**). Daß er ein **nur-Vater-Typ** ist, folgt aus der Angabe **is abstract**.

Anmerkung zur Terminologie: Es liegt nahe, Typen wie !SPEICHERBAR als **abstrakte** Typen zu bezeichnen, weil sie mit der Angabe **is abstract** vereinbart werden. Das widerspricht aber dem verbreiteten Sprachgebrauch, demzufolge ein **abstrakter Typ** so etwas wie ein **privater** oder **limitiert privater** Typ in Ada ist. Deshalb werden Typen wie !SPEICHERBAR hier **nicht** als abstrakte Typen sondern als **nur-Vater-Typen** bezeichnet. **o**

In Zeile 03 wird **PUT** als abstraktes Unterprogramm vereinbart. Das bedeutet zweierlei:
1. Im Rumpf des Paketes **TAG_D_01** muß (und darf) kein Rumpf für die Prozedur **PUT** angegeben werden.
2. Wenn man vom nur-Vater-Typ **!SPEICHERBAR** einen konkreten Typ T ableitet, muß man für T eine entsprechende konkrete Prozedur **PUT** angeben.

Zur Typklasse **SPEICHERBAR'class** gehören der Typ **!SPEICHERBAR** und alle Typen, die direkt oder indirekt von diesem Typ abgeleitet wurden.

Aus obigen Vereinbarungen folgt: Zu jedem Typ der Typklasse **SPEICHERBAR'class** gibt es eine passende **PUT**-Prozedur. Die Prozedur **PUT_CW** mit dem **klassenweiten** Parameter ITEM soll mit einem dynamischen Wahlaufruf jeweils die zu ITEM passende **PUT**-Prozedur aufrufen (ähnlich wie die Prozedur **SCHIEBE** im vorigen Abschitt).

Ein Zeiger des Typs **!Z_SPEICHERBAR** kann auf irgendein Objekt zeigen, dessen Typ zur Typklasse **SPEICHERBAR'class** gehört. Der Speicher, um den es hier geht, wird im **Rumpf** des Paketes als **Reihung** solcher **Zeiger** realisiert:

```
01 package body TAG_D_01 is
02    ------------------------------------------------------------------------
03    procedure PUT_CW(ITEM: in SPEICHERBAR'class) is
04    begin
05       -- Die jeweils "passende Prozedur PUT" wird aufgerufen:
```

```
06       PUT(ITEM => ITEM);
07    end PUT_CW;
08    ------------------------------------------------------------------------
09    -- In diesem SPEICHER werden die speicherbaren Objekte abgelegt
10    -- (genauer: abgelegt werden Zeiger, die auf die Objekte zeigen)
11    SPEICHER: array(INDEX) of Z_SPEICHERBAR;
12    ------------------------------------------------------------------------
13    procedure TUE_REIN(WOHIN: in INDEX; WAS: in  Z_SPEICHERBAR) is
14    begin
15       SPEICHER(WOHIN) := WAS;
16    end TUE_REIN;
17    ------------------------------------------------------------------------
18    procedure HOL_RAUS(WOHER: in INDEX; WAS: out Z_SPEICHERBAR) is
19    begin
20       WAS := SPEICHER(WOHER);
21    end HOL_RAUS;
22    ------------------------------------------------------------------------
23 end TAG_D_01;
```

Damit man Werte eines bestimmten Typs **T** in diesem Speicher ablegen kann, muß man **T** vom Typ **!SPEICHERBAR** ableiten. Falls das nicht geht (weil **T** schon auf andere Weise vereinbart wurde), muß man einen **Verbundtyp** mit einer **T-Komponenten** von **!SPEICHERBAR** ableiten. Um z.B. Werte des vordefinierten Untertyps **integer** "speicherbar" zu machen, kann man das folgende Paket vereinbaren:

Beispiel 23.6.2.: integer-Objekte speicherbar machen:

```
01 with TAG_D_01;
02 package TAG_D_02 is
03 ------------------------------------------------------------------------
04 -- Jedes Objekt vom Typ SPEICHERBAR1 enthaelt eine Ganzzahl.
05 -- Die Prozedur TUE_REIN1 tut ein SPEICHERBAR1-Objekt in den Speicher, mit
06 -- PUT kann man einen SPEICHERBAR1-Wert zur aktuellen Ausgabe ausgeben.
07 ------------------------------------------------------------------------
08    type SPEICHERBAR1 is new TAG_D_01.SPEICHERBAR with record
09       INT: integer;
10    end record;
11    procedure PUT(ITEM: SPEICHERBAR1);
12    procedure TUE_REIN1(WOHIN: TAG_D_01.INDEX; WAS: integer);
13 end TAG_D_02;
```

Der Typ **!SPEICHERBAR1** wird hier als Erweiterung des Typs **!SPEICHERBAR** vereinbart. Deshalb muß auch eine PUT-Prozedur für SPEICHERBAR1-Objekte spezifiziert werden.

Ein **SPEICHERBAR1**-Objekt ist im Grunde nur ein "verpacktes" **integer**-Objekt. Die Prozedur **TUE_REIN1** nimmt dem Benutzer das "Verpacken" von **integer**-Objekten und den Umgang mit Zeigern ab. Hier der Rumpf des Paketes TAG_D_02:

```
01 with ada.text_io;
02 package body TAG_D_02 is
03    ------------------------------------------------------------------------
04    package INTEGER_EA is new ada.text_io.integer_io(num => integer);
05    ------------------------------------------------------------------------
06    procedure PUT(ITEM: SPEICHERBAR1) is
07    begin
08       ada.text_io.put(item => "Ein Wert vom Typ SPEICHERBAR1: ");
09       INTEGER_EA .put(item => ITEM.INT, width => 0);
10       ada.text_io.new_line;
11    end PUT;
12    ------------------------------------------------------------------------
13    procedure TUE_REIN1(WOHIN: TAG_D_01.INDEX; WAS: integer) is
14       Z: TAG_D_01.Z_SPEICHERBAR := new SPEICHERBAR1'((INT => WAS));
15    begin
16       TAG_D_01.TUE_REIN(WOHIN => WOHIN, WAS => Z);
17    end TUE_REIN1;
```

```
18    ------------------------------------------------------------------
19 end TAG_D_02;
```

Die Prozedur **TUE_REIN1** nimmt das **integer**-Objekt **WAS**, verpackt es in einem **SPEICHERBAR1**-Verbund und läßt den Zeiger **Z** auf diesen Verbund zeigen (siehe Zeile 14). Der Zeiger Z wird dann mit der Prozedur **TUE_REIN** in den Speicher getan. **o**

Aufgabe 23.6.1.: Der Untertyp **STRING6** sei folgendermaßen vereinbart:

```
   subtype STRING6 is string(1..6);
```

Schreiben Sie ganz analog zum Paket **TAG_D_02** ein Paket **TAG_D_03**, mit dessen Hilfe man Werte des Untertyps **STRING6** im Speicher des Pakets **TAG_D_01** ablegen kann. **o**

Werte verschiedener Typen in einen Speicher **hineinzutun** ist einfacher, als sie wieder herauszuholen. Das liegt daran, daß man beim Hineintuen eines Wertes seinen **Typ** kennt. Beim Herausholen eines Wertes weiß man dagegen im allgemeinen nicht, "was für einen Wert man erwischt". Technisch bekommt man ja nur einen **Zeiger** Z, der auf irgendein Objekt zeigt, dessen Typ zur Typklasse **SPEICHERBAR'class** gehört. Was kann man mit dem Objekt **Z.all** machen, dessen genauen Typ man nicht kennt?

Eine Möglichkeit besteht darin, dieses Objekt formal in ein Objekt eines spezifischen Typs wie **SPEICHERBAR1** bzw. **SPEICHERBAR2** umzuwandeln. Das ist erlaubt, aber wenn man dabei einen Fehler macht, wird eine Ausnahme ausgelöst.

Sicherer und eleganter ist es, das Objekt **Z.all** mit **dynamischen Wahlaufrufen** zu bearbeiten, wie es in dem folgenden kleinen Testprogramm **TAG_D_04** geschieht:

Beispiel 23.6.3.: Werte in den Speicher tun, wieder herausholen und bearbeiten:

```
01 with TAG_D_01, TAG_D_02, TAG_D_03, ada.text_io;
02 procedure TAG_D_04 is
03 ------------------------------------------------------------------------
04 -- Dient zum testen der Pakete TAG_D_01, TAG_D_02 und TAG_D_03, die einen
05 -- "polymorphen Speicher" realisieren. Drei Ganzzahlen und zwei Zeichen-
06 -- ketten werden 1. in diesen Speicher hineingetan, 2. wieder herausgeholt
07 -- und schliesslich 3. zur aktuellen Ausgabe ausgegeben.
08 ------------------------------------------------------------------------
09    -- Der ZEIGER Z kann auf beliebige "speicherbare Objekte" zeigen:
10    Z: TAG_D_01.Z_SPEICHERBAR;
11 begin
12    -- 5 Objekte in den Speicher hineintun:
13    TAG_D_02.TUE_REIN1(WOHIN => 1, WAS =>  123);
14    TAG_D_03.TUE_REIN2(WOHIN => 2, WAS => "Hallo!");
15    TAG_D_02.TUE_REIN1(WOHIN => 3, WAS =>  456);
16    TAG_D_02.TUE_REIN1(WOHIN => 4, WAS =>  789);
17    TAG_D_03.TUE_REIN2(WOHIN => 5, WAS => "Ende?!");
18
19    -- Die 5 Objekt wieder aus dem Speicher holen und zur aktuellen Ausgabe
```

```
20    -- ausgeben:
21    for I in TAG_D_01.INDEX loop
22       TAG_D_01.HOL_RAUS(WOHER => I, WAS => Z);
23       TAG_D_01.PUT(ITEM => Z.all);
24    end loop;
25    ada.text_io.put_line("TAG_D_04: Das war's erstmal!");
26 end TAG_D_04;
```

In Zeile 23 wird die Prozedur **PUT** mit dem klassenweiten **ITEM**-Parameter aufgerufen. Im Rumpf dieser **PUT**-Prozedur wird die zum jeweiligen Objekt **Z.all** passende **PUT**-Prozedur dynamisch ausgewählt und aufgerufen. **o**

Das Wort "polymorph" bedeutet im allgemeinen so etwas wie "vielförmig" oder "vielgestaltig". Der in diesem Abschnitt behandelte Speicher wird **polymorph** genannt, weil man darin "verschiedenförmige" Werte (d.h. Werte verschiedener Typen) ablegen kann.

Ein **nur-Vater-Typ** muß nicht unbedingt als **leerer** Typ (mit den Worten **null record**) vereinbart werden, sondern kann auch **Komponenten** haben. Außerdem können zu einem nur-Vater-Typ nicht nur abstrakte, sondern auch normale ("konkrete") Unterprogramme gehören, etwa so:

Beispiel 23.6.4.: Ein nur-Vater-Typ mit Komponenten und konkreten primitiven Operationen:

```
package TAG_D_05 is
   type FARBIGE_GANZZAHL is abstract tagged record
      X: integer;
      Y: FARBE;
   end record;
   procedure MACHE_FG(G: integer; F: FARBE; FG: out FARBIGE_GANZZAHL);
   function  WERT(FG: FARBIGE_GANZZAHL) return integer;
   procedure DREHE_RUM(FG: FARBIGE_GANZZAHL) is abstract;
   ...
```

Obwohl man keine Objekte vom Typ **!FARBIGE_GANZZAHL** erzeugen lassen kann, legt dieser Typ doch fest, daß seine Objekte zwei Komponenten namens X und Y haben sollen. Diese Festlegung wird wirksam, wenn man von diesem nur-Vater-Typ weitere Typen ableitet. Die **konkreten** Unterprogramme **MACHE_FG** und **WERT** und das **abstrakte** Unterprogramm **DREHE_RUM** gehören zu den primitiven Operationen des Typs **!FARBIGE_GANZZAHL** und werden von ihm vererbt. Im **Rumpf** des Paketes **TAG_D_05** muß man nur für die konkreten Unterprogramme Rümpfe angeben. **o**

Zusammenfassung 23.6.:

- Von einem **nur-Vater-Typ** kann man weitere Typen **ableiten**, aber keine Objekte vereinbaren.
- Zu den **primitiven Operationen** eines nur-Vater-Typs können **abstrakte** und normale ("konkrete") Unterprogramme gehören.
- Wenn man einen konkreten Typ von einem **nur-Vater-Typ** ableitet, muß man für jedes **abstrakte** Unterprogramm des Vatertyps ein entsprechendes konkretes Unterprogramm angeben.

23.7. Wurzeltypen, universelle Typen und andere klassenweite Typen

Die folgenden Begriffe können nützlich sein, wenn man Ada mit anderen objektorientierten Programmiersprachen vergleichen will oder eine Antworten auf folgende Frage sucht: Zu welchem Typ gehören Ganzzahlliterale und Bruchzahlliterale eigentlich?

Eine **Typklasse** besteht aus einem Typ **!T** und allen von **!T** direkt oder indirekt (mit **new**) **abgeleiteten** Typen **!A1**, **!A2** etc.. Den Typ **!T** bezeichnet man auch als den **Wurzeltyp** (root type) seiner Typklasse.

Zu jedem **etikettierten** Typ **!T**, den man vereinbart hat, gibt es automatisch auch einen **klassenweiten** Typ **!T'class**. Die Wertemenge dieses klassenweiten Typs ist die **Vereinigung** der Wertemengen aller Typen !T, !A1, !A2, ..., die zur Typklasse von !T gehören. Siehe dazu auch (ARM 3.4) und insbesondere (ARM 3.4.1).

Einen klassenweiten Typ **!T'class** kann man nicht direkt vereinbaren, sondern nur indirekt, indem man den Typ **!T** vereinbart. Im Gegensatz zu seinem **klassenweiten** Typ **!T'class** bezeichnet man den vereinbarten Typ **!T** auch als einen **spezifischen** Typ.

Es gibt in Ada einen spezifischen Typ **!root_integer**. Von diesem Typ, so kann man sich vorstellen, leitet der Ausführer **alle Ganzzahltypen** ab (den vordefinierten Ganzzahltyp **!integer** und alle vom Programmierer vereinbarten signierten und modularen Ganzzahltypen wie !GANZ, !AEPFEL, !MODU etc.). Zum spezifischen Typ **!root_integer** gibt es eine Art klassenweiten Typ **!universal_integer**. Für diesen universellen Typ gilt:

1. Alle **Ganzzahlliterale** wie 0, 17, 123, 2#101# etc. gehören zum Typ !universal_integer.
2. Man kann **Konstanten** dieses Typs vereinbaren (sogenannte **benannte Zahlen**, named numbers, siehe das folgende Beispiel).
3. Man kann **keine Variablen** des Typs !universal_integer vereinbaren.
4. Es gibt **Attribute**, die einen Wert dieses Typs bezeichnen bzw. liefern, z.B. die Attribute 'size, 'length und 'pos (siehe ARM K).
5. Einen Ausdruck des Typs !universal_integer darf man in einem Ada-Programm überall da verwenden, wo ein Wert irgendeines anderen Ganzzahltyps !GT erwartet wird. Der Wert des !universal_integer-Ausdrucks wird dann automatisch in einen Wert des Typs !GT umgewandelt.

Beispiel 23.7.1.: Automatische Umwandlungen von Werten des Typs **!universal_integer**:

```
01 declare
02    type GANZ is range -500..+1000;
03    GUTE_DINGE : constant := 100_000_000_000_000_000_000_000_000_003 -
04                             100_000_000_000_000_000_000_000_000_000;
05    I1         : integer  := 123;
06    G1         : GANZ     := 123;
07    I2         : integer  := GUTE_DINGE + 1;
08    G2         : GANZ     := GUTE_DINGE + 1;
```

Die Konstante **GUTE_DINGE** wurde ohne Angabe eines Typs vereinbart. Sie wird als eine **benannte Zahl** (named number) bezeichnet und gehört zum Typ **!universal_integer**. Der Ausführer muß ihren Wert (3) exakt berechnen, auch wenn der Ausdruck rechts vom Zuweisungszeichen ":=" sehr große Zahlen enthält. Der Wert des Literals **123** gehört ebenfalls zum Typ **!univesal_integer**. In Zeile 05 (bzw. in Zeile 06) wird dieser Wert automatisch in einen Wert des Typs **!integer** (bzw. des Typs **!GANZ**) umgewandelt und zugewiesen. Ganz entsprechend wird der

Wert des **!universal_integer**-Ausdrucks **GUTE_DINGE + 1** in Zeile 07 bzw. 08 umgewandelt und zugewiesen. **o**

Es gibt in Ada einen spezifischen Typ **!root_real**. Von diesem Typ, so kann man sich vorstellen, leitet der Ausführer **alle Bruchzahltypen** ab (den vordefinierten Gleitpunkttyp **!float** und alle vom Programmierer vereinbarten Gleitpunkt-, gewöhnlichen Festpunkt- und dezimalen Festpunkttypen wie !GLEIT, !GFIX1, !DFIX1 etc.). Zum spezifischen Typ **!root_real** gibt es eine Art klassenweiten Typ **!universal_real**. Für diesen universellen Typ gilt:

1. Alle **Bruchzahlliterale** wie 0, 17, 123, 2#101# etc. gehören zum Typ !universal_real.
2. Man kann **Konstanten** dieses Typs vereinbaren (sogenannte **benannte Zahlen**, named numbers, siehe das folgende Beispiel).
3. Man kann **keine Variablen** des Typs !universal_real vereinbaren.
4. Es gibt **Attribute**, die einen Wert dieses Typs bezeichnen, z.B. die Attribute 'delta und 'small (siehe ARM K).
5. Einen Ausdruck des Typs !universal_real darf man in einem Ada-Programm überall da verwenden, wo ein Wert irgendeines anderen Bruchzahltyps !BT erwartet wird. Der Wert des !universal_real-Ausdrucks wird dann automatisch in einen Wert des Typs !BT umgewandelt.

Beispiel 23.7.2.: Automatische Umwandlungen von Werten des Typs **!universal_real**:

```
01 declare
02    type GLEIT is digits 6;
03    type GFIX1 is delta 0.25 range -100..+100;
04    type DFIX1 is delta 0.01 digits 7;
05    PI  : constant := 3.141_592_653_589_793_238_462_643_383_279_502_884_197;
06    GL1 : GLEIT    := PI ** 2 / 2;
07    GF1 : GFIX1    := 17.5;
08    DF1 : DFIX1    := 2 * PI
```

Die Konstante **PI** wurde (in Zeile 05) ohne Angabe eines Typs vereinbart. Sie wird als **benannte Zahl** bezeichnet und gehört zum Typ **!universal_real**. Der Ausdruck **PI ** 2 / 2** in Zeile 06 gehört ebenfalls zum Typ **!universal_real**. Sein Wert muß vom Ausführer exakt berechnet und dann automatisch in einen Wert des Typs **!GLEIT** umgewandelt werden. In Zeile 07 bzw. 08 wird entsprechend ein **!universal_real**-Werte automatisch in einen Wert des Typs **!GFIX1** (bzw. **!DFIX1**) umgewandelt und dann zugewiesen. **o**

Schließlich gibt es in Ada noch einen spezifischen Typ !root_fixed. Von diesem Typ, so kann man sich vorstellen, leitet der Ausführer alle Festpunkttypen ab (den vordefinierten gewöhnlichen Festpunkttyp **!duration** und alle vom Programmierer vereinbarten gewöhnlichen und dezimalen Festpunkttypen wie !GFIX1, !DFIX1 etc.). Zum spezifischen Typ **!root_fixed** gibt es eine Art klassenweiten Typ **!universal_fixed**. Für diesen universellen Typ gilt:

Die Multiplikationsoperation "*****" und die Divisionsoperation "/" für Festpunktwerte haben folgende Spezifikationen:

```
function "*"(left, right: !universal_fixed) return !universal_fixed;
function "/"(left, right: !universal_fixed) return !universal_fixed;
```

Das bedeutet praktisch: Man kann diese Operationen auf Werte **beliebiger** Festpunkttypen anwenden (auch auf zwei Werte von zwei **verschiedenen** Festpunkttypen). Das Ergebnis einer Festpunktmultiplikation bzw. -division gehört erstmal zum Typ **!universal_fixed** und wird dann

automatisch entsprechend dem Zusammenhang in einen anderen Festpunkttyp umgewandelt, etwa so:

Beispiel 23.7.3.: Mit verschiedenen Festpunkttypen rechnen:

```
01 declare
02    type GFIX1 is delta 0.25   range -100..+100;
03    type DFIX1 is delta 0.01   digits 7;
04    type DFIX2 is delta 0.0001 digits 9;
05    GF1  : GFIX1 := 1.234;
06    DF11 : DFIX1 := 37.39;
07    DF12 : DFIX1 := 51.77;
08    DF21 : DFIX2 := DF11 * DF12; -- Ohne Genauigkeitsverlust!
09    DF22 : DFIX2 := GF1  / DF12; -- Mit  Genauigkeitsverlust!
```

In Zeile 08 ist liefert die Multiplikationsoperation "*" ein mathematisch exaktes Ergebnis des Typs **!universal_fixed** mit vier Dezimalstellen nach dem Punkt. Dieses Ergebnis wird (ohne Verlust der Exaktheit) in einen Wert des Typs **!DFIX2** umgewandelt und der Variablen **DF21** zugewiesen.

In Zeile 09, so kann man sich vorstellen, liefert die Divisionsoperation "/" ebenfalls ein mathematisch exaktes Ergebnis des Typs **!universal_fixed**. Dieses Ergebnis hat sehr viele Stellen hinter dem Punkt, wird aber in einen Wert des Typs **!DFIX2** umgewandelt und der Variablen **DF22** zugewiesen. Bei der Umwandlung geht die Exaktheit des Divisionsergebnisses verloren, weil der Typ **!DFIX2** nur vier Dezimalstellen nach dem Punkt vorsieht. Siehe dazu auch (ARM 4.5.5(18)).
o

Zusammenfassung 23.7.:

- Eine **Typklasse** besteht aus einem **Wurzeltyp** und allen davon **abgeleiteten Typen**.
- **!root_integer** ist der Wurzeltyp der Klasse aller Ganzzahltypen.
- **!root_real** ist der Wurzeltyp der Klasse aller Bruchzahltypen.
- **!root_fixed** ist der Wurzeltyp der Klasse aller Festpunkttypen.
- Zu den Typen **!root_integer**, **!root_real** bzw. **!root_fixed** gehören die **universellen** Typen **!universal_integer**, **!universal_real** bzw. **!universal_fixed**.
- Zum Typ **!universal_integer** gehören u.a. alle **Ganzzahlliterale**.
- Zum Typ **!universal_real** gehören u.a. alle **Bruchzahlliterale**.
- Zum Typ **!universal_fixed** gehören die Parameter und das Ergebnis der Operationen "*" und "/" für Festpunktwerte.
- Werte der universellen Typen werden "**automatisch** ihrer Umgebung angepaßt".
- Zum jedem **spezifischen etikettierten** Typ **!T** gibt es einen **klassenweiten** Typ **!T'class**.
- Die **universellen** Typen sind so etwas Ähnliches wie **klassenweite** Typen für die spezifischen Typen **!root_integer**, **!root_real** und **!root_fixed**.

24. Kontrollierte Typen

Auf **Objekte** (Konstanten und Variablen) wendet der Ausführer im Prinzip **drei** fundamentale Operationen an: **Initialisierung**, **Finalisierung** und **Zuweisung**. **Initialisiert** wird ein Objekt, nachdem es erzeugt wurde. Erzeugt wird ein Objekt aufgrund einer Vereinbarung oder mit Hilfe eines Allokators **new**. **Finalisiert** wird ein Objekt, bevor es zerstört wird. Zerstört wird ein Objekt z.B., weil des Unterprogramm verlassen wird, in dem es vereinbart wurde, oder aufgrund eines Aufrufs einer Instanz der Prozedurschablone **unchecked_deallocation**. Eine **Zuweisung** erfolgt aufgrund einer Zuweisungsanweisung, als Teil einer ausdrücklichen Initialisierung, bei der Übergabe von Parametern an Unterprogramme und in ähnlichen Situationen.

Manchmal genügt dem Programmierer nicht, was der Ausführer beim Initialisieren, Finalisieren bzw. Zuweisen "von sich aus" macht. Mithilfe von **kontrollierten Typen** (controlled types) kann er die Operationen des Ausführers ergänzen. Für einen kontrollierten Typ kann er **drei** Prozeduren namens **INITIALIZE**, **FINALIZE** und **ADJUST** vereinbaren, die je einen **in-out**-Parameter des betreffenden kontrollierten Typs haben und in folgenden Momenten aufgerufen werden:

- **INITIALIZE** unmittelbar nachdem ein kontrolliertes Objekt erzeugt und "wie üblich" initialisiert wurde
- **FINALIZE** unmittelbar bevor ein kontrolliertes Objekt "wie üblich" finalisiert und zerstört wird
- **ADJUST** als letzter Schritt einer Zuweisung an ein kontrolliertes Objekt

(Das Original dieser Einleitung findet man im (ARM 7.6)).

Technisch gesehen macht man einen Typ **KT** zu einem **kontrollierten Typ**, indem man ihn mit **new** von dem etikettierten Typ **controlled** im Paket **ada.finalization** ableitet, z.B. so:

```
type KT is new ada.finalization.controlled with record
   ...
end record;
```

Einen **limitiert kontrollierten** Typ **LKT** erhält man entsprechend als Nachfolger des etikettierten Typs **limited_controlled**, etwa so:

```
type LKT is new ada.finalization.limited_controlled with record
   ...
end record;
```

Für einen limitierten Typ wie **LKT** steht dem Programmierer keine Zuweisungsanweisung zur Verfügung. Entsprechend kann er für so einen Typ auch keine **ADJUST**-Prozedur angeben.

Der Typ **!controlled** im Paket **ada.finalization** ist ein **nur-Vater-Typ** (abstract type), aber die Prozeduren **INITIALIZE**, **FINALIZE** und **ADJUST** sind nicht etwa **abstrakte** Unterprogramme, die man mit eigenen konkreten Unterprogrammen **verdecken muß**, sondern **konkrete** Unterprogramme, die man **verdecken kann**. Wenn man eines dieser Unterprogramme nicht verdeckt, wird das entsprechende "Originalunterprogramm" aufgerufen, welches einfach **nichts** macht. Für den Typ **!limited_controlled** und seine beiden Prozeduren **INITIALIZE** und **FINALIZE** gilt entsprechendes.

Kontrollierte Typen eignen sich besonders dazu, das **Allokieren** und **Deallokieren** von "gezeigten" Variablen in einem Paket zusammenzufassen und vor den Benutzern des Typs zu verbergen. Die folgenden Abschnitte enthalten Beispiele dafür, wie das konkret aussehen kann.

24.1. Zeichenketten als kontrollierte Objekte

Häufig ist es effizient, "große Objekte" indirekt über **Zeiger** zu manipulieren. Statt z.B. das ganze Objekt an eine andere Stelle zu kopieren, genügt es dann häufig, eine Kopie des betreffenden Zeigers zu verschieben.

Ein besonders einfaches Beispiel für große Objekte sind **Zeichenketten**, die aus Tausenden oder sogar Millionen von Zeichen bestehen können. Im folgenden Beispiel wird ein Typ **!ZEICHEN_KETTE** vereinbart, dessen Werte im wesentlichen aus einem **Zeiger auf einen String** bestehen. Damit die Benutzer des Typs die Strings nicht selbst allokieren und deallokieren müssen, wird der Typ als **kontrollierter Typ** vereinbart.

Beispiel 24.1.1.: Ein kontrollierter Typ **!ZEICHEN_KETTE**:

```
01 with ada.finalization;
02 package KON_A_01 is
03 ------------------------------------------------------------------------
04 -- Stellt einen privaten kontrollierten Typ !ZEICHEN_KETTE und eine Kon-
05 -- struktor-Prozedur MACHE_ZEICHEN_KETTE zur Verfuegung. Die privaten
06 -- Prozeduren INITIALIZE, FINALIZE und ADJUST geben u.a. kurze Meldungen zur
07 -- aktuellen Ausgabe aus, wenn sie aufgerufen werden.
08 ------------------------------------------------------------------------
09    type ZEICHEN_KETTE is private;
10    procedure MACHE_ZEICHEN_KETTE(S: string; ZK: in out ZEICHEN_KETTE);
11 private
12    type Z_STRING      is access string;
13    type ZEICHEN_KETTE is new ada.finalization.controlled with record
14       Z: Z_STRING;
15    end record;
16    procedure INITIALIZE(ZK: in out ZEICHEN_KETTE);
17    procedure FINALIZE  (ZK: in out ZEICHEN_KETTE);
18    procedure ADJUST    (ZK: in out ZEICHEN_KETTE);
19 end KON_A_01;
```

Jeder Wert des Typs **!ZEICHEN_KETTE** (siehe Zeile 13 bis 15) ist ein **Verbund** mit nur einer Komponente namens **Z**. Diese Komponente ist ein Zeiger, der auf einen String zeigt (oder den Wert **null** hat).

Der Rumpf des Paketes **KON_A_01** wird hier nicht wiedergegeben, aber die darin enthaltenen Unterprogramme sollen kurz beschrieben werden:

Die Konstruktor-Prozedur **MACHE_ZEICHEN_KETTE** macht aus dem String **S** eine ZEICHEN_KETTE **ZK**. Zuerst wird der String **ZK.Z.all** an die Speicherverwaltung zurückgegeben (falls der Zeiger **ZK.Z** den Wert **null** hat, passiert bei diesem ersten Schritt nichts). Dann wird mit **new** eine neue Stringvariable allokiert, die eine Kopie von **S** enthält. Der Zeiger auf diese Kopie wird der Komponenten **ZK.Z** zugewiesen.

Die Prozedur **INITIALIZE** macht nichts wichtiges und man hätte sie in diesem Beispiel auch weglassen können.

Die Prozedur **FINALIZE** gibt den String **ZK.Z.all** an die Speicherverwaltung zurück, bevor das Objekt **ZK** zerstört wird. Dadurch wird verhindert, daß der String **ZK.Z.all** "unlöschbar im Speicher liegen bleibt".

Die Prozedur **ADJUST** erzeugt mit **new** eine Kopie des Strings **ZK.Z.all** und läßt den Zeiger **ZK.Z** auf diese Kopie zeigen. Dadurch wird erreicht, daß die **Z**-Komponenten von zwei **ZEICHEN_KETTE**-Objekten **ZK1** und **ZK2** auch nach einer Zuweisung wie **ZK2 := ZK1;** nicht auf **denselben** String zeigen, sondern auf **zwei gleiche** Strings.

Alle vier Prozeduren (**MACHE_ZEICHEN_KETTE**, **INITIALIZE**, **FINALIZE** und **ADJUST**) geben kurze Meldungen zur aktuellen Ausgabe aus, so daß man "draußen vor dem Bildschirm" erkennen kann, in welcher Reihenfolge und mit welchem **ZK**-Parametern sie aufgerufen wurden.

In der folgenden Prozedur **KON_A_02** werden einige Variable des Untertyps **ZEICHEN_KETTE** vereinbart und bearbeitet:

```
01 with KON_A_01, ada.text_io;
02 procedure KON_A_02 is
03 ------------------------------------------------------------------------
04 -- Zum Testen des Paketes KON_A_01, in dem ein privater kontrollierter Typ
05 -- !ZEICHEN_KETTE vereinbart wird.
06 ------------------------------------------------------------------------
07    ZK1 : KON_A_01.ZEICHEN_KETTE;
08    ZK2 : KON_A_01.ZEICHEN_KETTE;
09    STR : string(1..9);
10 begin
11    ada.text_io.put_line(item => "Hallo1 ---------------------");
12    for I in natural range 1..3 loop
13       STR := "---" & natural'image(I) & " ---";
14       KON_A_01.MACHE_ZEICHEN_KETTE(S  => STR, ZK => ZK1);
15    end loop;
16    ada.text_io.put_line(item => "Hallo2 ---------------------");
17    ZK2 := ZK1;
18    ada.text_io.put_line(item => "Hallo3 ---------------------");
19 end KON_A_02;
```

In Zeile 13 wird der Stringvariablen **STR** ein String der Form **"--- X ---"** zugewiesen, wobei anstelle von **X** eine der Zahlen 1, 2 bzw. 3 steht. Hier die Ausgabe des Programms **KON_A_02**. Die meisten Ausgaben stammen aus dem Rumpf des Paketes **KON_A_01**, nur die Hallo-Zeilen werden direkt von der Prozedur **KON_A_02** ausgegeben:

```
01  INI  Null-Zeiger!
02  INI  Null-Zeiger!
03 Hallo1 ---------------------
04  MZK1 Null-Zeiger!
05  MZK2 --- 1 ---
06  MZK1 --- 1 ---
07  MZK2 --- 2 ---
08  MZK1 --- 2 ---
09  MZK2 --- 3 ---
10 Hallo2 ---------------------
11  FIN1 Null-Zeiger!
12  FIN2 Null-Zeiger!
13  ADJ1 --- 3 ---
14  ADJ2 --- 3 ---
15 Hallo3 ---------------------
16  FIN1 --- 3 ---
17  FIN2 Null-Zeiger!
```

```
18  FIN1 --- 3 ---
19  FIN2 Null-Zeiger!
```

In den Prozeduren **MACHE_ZEICHEN_KETTE**, **INITIALIZE**, **FINALIZE** und **ADJUST** ist der jeweils **erste** und **letzte** Befehl ein Ausgabebefehl. Ausgegeben wird jeweils ein Kennzeichen für die Prozedur (**INI** für INITIALIZE, **MZK** für **MACHE_ZEICHEN_KETTE** etc.). **MZK1** wird am **Anfang** und **MZK2** am **Ende** der Prozedur **MACHE_ZEICHEN_KETTE** ausgegeben. Jede der Prozeduren hat einen **ZEICHEN_KETTE**-Parameter, der im wesentlichen aus einem Zeiger **Z** besteht. **"Null-Zeiger!"** bedeutet, daß dieser Zeiger **Z** den Wert **null** hat. Sonst wird der String **Z.all** ausgegeben. Z.B. sieht man in Zeile 06 und 07, daß die Prozedur **MACHE_ZEICHEN_KETTE** aufgerufen wurde, daß der Zeiger **Z** am Anfang auf einen String **"--- 1 ---"** und am Ende auf einen String **"--- 2 ---"** zeigte.

Aufgabe 24.1.1.: Erklären Sie die Ausgabe des Programms **KON_A_02** Zeile für Zeile anhand des Programmtextes. Welcher Befehl in **KON_A_02** hat den Aufruf welcher Prozedur(en) verursacht? Möglicherweise hilft es, sich auch den Rumpf des Paketes **KON_A_01** anzusehen. **o**

Das Paket **KON_B_01** und die Prozedur **KON_B_02** leisten fast das gleiche, wie ihre Namensvettern mit dem 'A' im Namen, nur die Ausgaben sind ein bißchen "maschinennäher". Hier die Ausgabe des Programms **KON_B_02** (welches praktisch genauso aussieht wie **KON_A_02**):

```
01  INI  16#254FDDC#      16#0#
02  INI  16#254FDC8#      16#0#
03 Hallo1 --------------------
04  MZK1 16#254FDDC#      16#0#
05  MZK2 16#254FDDC# 16#520620# --- 1 ---
06  MZK1 16#254FDDC# 16#520620# --- 1 ---
07  MZK2 16#254FDDC# 16#520620# --- 2 ---
08  MZK1 16#254FDDC# 16#520620# --- 2 ---
09  MZK2 16#254FDDC# 16#520620# --- 3 ---
10 Hallo2 --------------------
11  FIN1 16#254FDC8#      16#0#
12  FIN2 16#254FDC8#      16#0#
13  ADJ1 16#254FDC8# 16#520620# --- 3 ---
14  ADJ2 16#254FDC8# 16#520638# --- 3 ---
15 Hallo3 --------------------
16  FIN1 16#254FDC8# 16#520638# --- 3 ---
17  FIN2 16#254FDC8#      16#0#
18  FIN1 16#254FDDC# 16#520620# --- 3 ---
19  FIN2 16#254FDDC#      16#0#
```

Die Zahlen im **16-er-System** stellen **Zeiger** ("Adressen") dar. Der erste Zeiger zeigt auf den **ZEICHEN_KETTE**-Parameter namens **ZK** des jeweiligen Unterprogramms und der zweite Zeiger ist der Wert der **Z**-Komponente dieses Parameters. Ganz rechts findet man wieder die Zeichenkette, auf die der Zeiger **Z** zeigt. Die konkreten Adressen können bei verschiedenen Ausführungen desselben Programms verschieden sein (erst recht, wenn verschiedene Ada-Ausführer benutzt werden). **o**

Aufgabe 24.1.2.: Bei welchen Adressen standen (laut obiger Ausgabe des Programms **KON_B_02**) die Variable **ZK1** und **ZK2**? Bei welchen Adressen wurden die diversen Strings allokiert? Zeichnen Sie die Variablen **ZK1** und **ZK2** als Bojen. Wie veränderten sich diese Bojen im Laufe der Programmausführung? **o**

Zusammenfassung 24.1.:

- Ein **kontrollierter Typ** KT ist ein Nachfolger des Typs **controlled** im Paket **ada.finalization**.

- **KT** erbt von seinem Vatertyp drei konkrete Prozeduren namens **INITIALIZE**, **FINALIZE** und **ADJUST.**
- Diese Prozeduren "machen nichts", können aber durch "wirkungsvollere" Prozeduren **verdeckt** werden.
- Die Prozeduren **INITIALIZE**, **FINALIZE** und **ADJUST** werden vom Ausführer in bestimmten Situationen "automatisch" ausgeführt (ohne daß man ihm das durch entsprechende Prozeduraufrufe befehlen müßte).
- **INITIALIZE** wird immer dann ausgeführt, nachdem ein **KT**-Objekt **erzeugt** wurde (aufgrund einer **Vereinbarung** oder aufgrund einer **Allokation** mit **new**).
- **FINALIZE** wird immer ausgeführt, bevor ein **KT**-Objekt zerstört wird.
- **ADJUST** wird immer ausgeführt, nachdem einem **KT**-Objekt ein neuer Wert zugewiesen wurde.

24.2. Ein limitiert kontrollierter Typ

Im folgenden Beispiel wird (im Paket **KON_C_01**) ein **limitiert kontrollierter** Typ **!ZEICHEN_-KETTE** vereinbart. Für diesen Typ gibt es **keine** Zuweisungsanweisung ":=". Dafür stellt das Paket seinen Benutzern eine Prozedur **WEISE_ZU** zur Verfügung. Alles weitere ist ziemlich ähnlich wie im vorigen Abschnitt.

Beispiel 24.2.1.: Ein limitiert kontrollierter Typ !ZEICHEN_KETTE:

```
01 with ada.finalization;
02 package KON_C_01 is
03 ------------------------------------------------------------------------
04 -- Stellt einen limitiert privaten kontrollierten Typ !ZEICHEN_KETTE, eine
05 -- Konstruktor-Prozedur MACHE_ZEICHEN_KETTE und eine Zuweisungs-Prozedur
06 -- WEISE_ZU zur Verfuegung. Wenn eine dieser oeffentlichen Prozeduren
07 -- oder die private Prozedur FINALIZE aufgerufen wird, wird unter anderem
08 -- eine Meldung zur aktuellen Ausgabe ausgegeben.
09 ------------------------------------------------------------------------
10    type ZEICHEN_KETTE is limited private;
11    procedure MACHE_ZEICHEN_KETTE(S: in string; ZK: out ZEICHEN_KETTE);
12    procedure WEISE_ZU(ZIEL: out ZEICHEN_KETTE; QUELLE: in EICHEN_KETTE);
13 private
14    type Z_STRING      is access string;
15 -- for  Z_STRING'storage_size use 200_000;
16    type ZEICHEN_KETTE is new
17         ada.finalization.limited_controlled with record
18       Z: Z_STRING;
19    end record;
20    procedure FINALIZE(ZK: in out ZEICHEN_KETTE);
21 end KON_C_01;
```

Für den Typen **limited_controlled** gibt es keine **ADJUST**-Funktion und die **INITIALIZE**-Prozedur wird auch in diesem Beispiel nicht benötigt. Deshalb wird für den Typ !ZEICHEN_KETTE nur die Prozedur **FINALIZE** mit einer eigenen Prozedur überdeckt (in Zeile 20).

Der Rumpf des Paketes **KON_C_01** wird hier nicht wiedergegeben, aber die darin enthaltenen Unterprogramme sollen kurz beschrieben werden:

Die Konstruktor-Prozedur **MACHE_ZEICHEN_KETTE** funktioniert ganz genauso wie die gleichnamige Prozedur im vorigen Abschnitt. Sie macht aus dem String **S** eine ZEICHEN_KETTE und bringt diese nach **ZK**.

Die Prozedur **WEISE_ZU** gibt zuerst den String **ZIEL.Z.all** an die Speicherverwaltung zurück, erzeugt dann mit **new** eine Kopie des Strings **QUELLE.Z.all** und läßt schließlich den Zeiger **ZIEL.Z** auf diese Kopie zeigen.

Die Prozedur **FINALIZE** funktioniert ganz genauso wie die gleichnamige Prozedur im vorigen Abschnitt. Sie gibt den String **ZK.Z.all** an die Speicherverwaltung zurück, bevor das Objekt **ZK** zerstört wird. Dadurch wird verhindert, daß der String **ZK.Z.all** "unlöschbar im Speicher liegen bleibt".

Alle drei Prozeduren (**MACHE_ZEICHEN_KETTE**, **WEISE_ZU** und **FINALIZE**) geben kurze Meldungen zur aktuellen Ausgabe aus, so daß man "draußen vor dem Bildschirm" erkennen kann, in welcher Reihenfolge sie aufgerufen wurden und wieviele Bytes momentan durch allokierte Strings belegt sind.

In der folgenden Prozedur **KON_C_02** werden einige Variable des limitiert kontrollierten Untertyps **ZEICHEN_KETTE** vereinbart und bearbeitet:

```
01 with KON_C_01, ada.text_io;
02 procedure KON_C_02 is
03 ------------------------------------------------------------------------
04 -- Dient zum Testen des Paketes KON_C_01, in dem ein limitiert kontrol-
05 -- lierter Typ !ZEICHEN_KETTE vereinbart wird.
06 ------------------------------------------------------------------------
07    S1 : string(1..100_000) := (others => '?'); -- Ein langer String
08    S2 : string(1.. 54_321) := (others => '!'); -- Auch nicht sehr kurz
09    ZK0: KON_C_01.ZEICHEN_KETTE;
10 begin
11    for I in 1..3 loop
12       ada.text_io.put_line(item => "Hallo1 ----------------------");
13       BLOCK1: declare
14          ZK1: KON_C_01.ZEICHEN_KETTE;
15          ZK2: KON_C_01.ZEICHEN_KETTE;
16       begin
17          KON_C_01.MACHE_ZEICHEN_KETTE(S => S1, ZK => ZK1);
18          KON_C_01.MACHE_ZEICHEN_KETTE(S => S2, ZK => ZK2);
19          KON_C_01.WEISE_ZU(ZIEL => ZK1, QUELLE => ZK2);
20          ada.text_io.put_line(item => "Hallo2 ----------------------");
21       end BLOCK1;
22    end loop;
23    ada.text_io.put_line(item => "Hallo3 ----------------------");
24    KON_C_01.MACHE_ZEICHEN_KETTE(S => "1234567", ZK => ZK0);
25 end KON_C_02;
```

Die Variable **ZK0** wird im Vereinbarungsteil der **Prozedur** KON_C_02 vereinbart (Zeile 09) und somit erst finalisiert und zerstört, wenn die Prozedur fertig ausgeführt ist. Dagegen werden die Variablen **ZK1** und **ZK2** im **BLOCK1** vereinbart und jedesmal finalisiert und zerstört, wenn der **BLOCK1** fertig ausgeführt ist.

Hier die Ausgabe des Programms KON_C_02. Die meisten Ausgaben stammen aus dem Rumpf des Paketes **KON_C_01**, nur die Hallo-Zeilen werden direkt von der Prozedur **KON_C_02** ausgegeben:

```
01 Hallo1 ----------------------
02 MZK:           0     100000
03 MZK:      100000     154321
04 WZU:      154321     108642
```

```
05 Hallo2 ----------------------
06 FIN:      108642      54321
07 FIN:       54321          0
08 Hallo1 ----------------------
09 MZK:           0     100000
10 MZK:      100000     154321
11 WZU:      154321     108642
12 Hallo2 ----------------------
13 FIN:      108642      54321
14 FIN:       54321          0
15 Hallo1 ----------------------
16 MZK:           0     100000
17 MZK:      100000     154321
18 WZU:      154321     108642
19 Hallo2 ----------------------
20 FIN:      108642      54321
21 FIN:       54321          0
22 Hallo3 ----------------------
23 MZK:           0          7
24 FIN:           7          0
```

Die Prozeduren **MACHE_ZEICHEN_KETTE**, **WEISE_ZU** und **FINALIZE** geben jeweils ihr Kennzeichen **MZK**, **WZU** bzw. **FIN** aus, gefolgt von zwei Ganzzahlen. Die Zahlen geben an, wieviele Bytes mit allokierten Strings belegt sind, und zwar bei Beginn der betreffenden Prozedur und unmittelbar vor ihrem Ende. Z.B. meldet in **Zeile 17** die Prozedur **MACHE_ZEICHEN_KETTE**, daß sie aufgerufen wurde, daß zu Beginn ihrer Ausführung 100_000 Bytes und nach ihrer Ausführung 154_321 Bytes mit allokierten Strings belegt waren. Offenbar hat die Prozedur einen String der Länge 54_321 Bytes allokiert. In **Zeile 21** werden von der Prozedur **FINALIZE** offenbar 54_321 Bytes deallokiert ("an die Speicherverwaltung zurückgegeben"). **o**

Aufgabe 24.2.1.: Erklären Sie die Ausgabe des Programms **KON_C_02** Zeile für Zeile. Überprüfen Sie insbesondere, ob die Anzahl der (mit allokierten Strings) belegten Bytes "an den richtigen Stellen" gleich 0 wird. Wieviele Bytes werden bei einer Programmausführung **insgesamt** allokiert (und wieder deallokiert)? **o**

Aufgabe 24.2.2.: Ändern Sie in der Prozedur **KON_C_02** die Zeile 11 (**for I in 1..3 loop**) so, daß der Rumpf der Schleife nicht nur **3** Mal, sondern **1_000** Mal oder sogar **100_000** Mal durchlaufen wird. Wieviel Speicher wird dann insgesamt für Strings allokiert (und wieder deallokiert)?. Solche "Massentests" sind notwendig um zu zeigen, daß wirklich alle Speicherzellen, die allokiert werden, auch ordnungsgemäß wieder deallokiert werden. **o**

Zusammenfassung 24.2.:

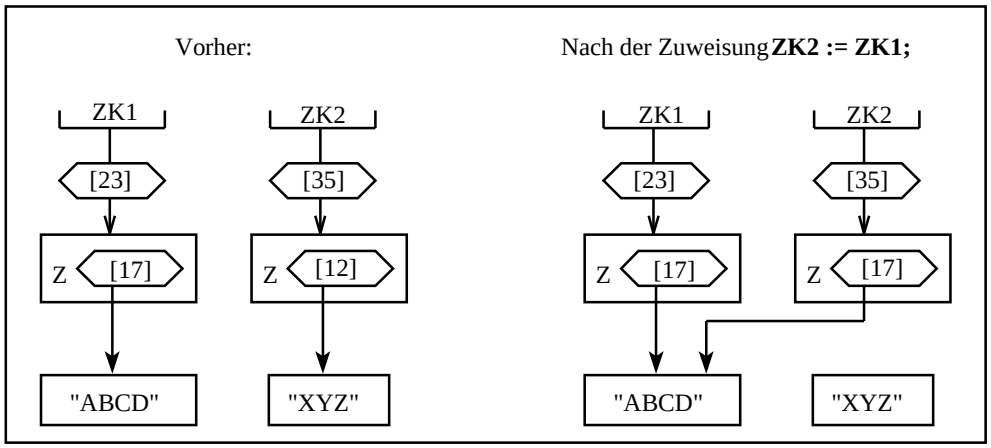
- Ein limitiert kontrollierter Typ **LKT** ist ein Nachfolger des Typs **limited_controlled** im Paket **ada.finalization**.
- **LKT** erbt von seinem Vater zwei konkrete aber "wirkungslose" Prozeduren namens **INITIALIZE** und **FINALIZE**, die man durch "wirkungsvollere" Prozeduren **verdecken** kann.
- **INITIALIZE** wird automatisch aufgerufen, nachdem ein LKT-Objekt **erzeugt** wurde.
- **FINALIZE** wird automatisch aufgerufen, bevor ein LKT-Objekt **zerstört** wird.

24.3. Zeiger mit Referenzsemantik

In den Beispielen der vorigen beiden Abschnitte waren **ZEICHEN_KETTEN**-Objekte mit einer **Kopiersemantik** versehen worden. Damit ist gemeint: Nach einer Zuweisung **ZK2 := ZK1;**

zwischen zwei **ZEICHEN_KETTE**-Objekten **ZK1** und **ZK2** zeigt der Zeiger **ZK2.Z** auf eine **Kopie** des Strings **ZK1.Z.all**, und nicht auf denselben String wie der Zeiger **ZK.Z**. Die Benutzer des Typs **ZEICHEN_KETTE** merken also gar nicht, daß ein **ZEICHEN_KETTE**-Objekt im wesentlichen ein **Zeiger** ist, sondern können sich unter einem ZEICHEN_KETTE-Objekt so etwas wie einen String vorstellen.

Von einer **Referenzsemantik** spricht man, wenn nach einer Zuweisung **ZK2 := ZK1;** die Zeiger **ZK1.Z** und **ZK2.Z** auf ein und **denselben** String zeigen (oder: auf denselben String referieren), wie das folgende Diagramm verdeutlichen soll:

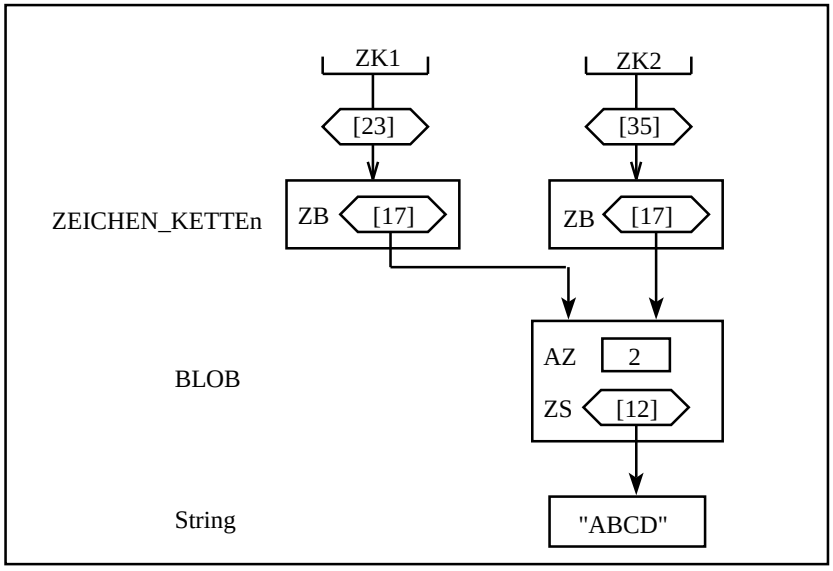


Vor der Zuweisung hat die Zeigervariable **ZK2.Z** den Wert **[12]** und zeigt auf den String **"XYZ"**. Nach der Zuweisung **ZK2 := ZK1;** hat die Variable **ZK2.Z** den Wert **[17]** und zeigt damit auf den String **"ABCD"** (auf den auch die Variable **ZK1.Z** zeigt). Bei dieser Vorgehensweise können die Benutzer des Typs **ZEICHEN_KETTE** merken, daß **ZEICHEN_KETTE**-Objekte "nur" Zeiger ("Referenzen") sind und keine Strings.

Frage: Darf man den String **"XYZ"** im Verlauf der Zuweisung **ZK2 := ZK1;** deallokieren oder nicht?
Antwort: Das kommt darauf an.

Wenn **ZK2.Z** vor der Zuweisung der **einzige** Zeiger mit dem Wert **[12]** war, dann darf und sollte man **"XYZ"** im Verlauf der Zuweisung deallokieren. Möglicherweise existieren aber noch weitere Zeiger **ZK3.Z**, **ZK4.Z** etc., die auch den Wert **[12]** haben und damit auf den String **"XYZ"** zeigen. In diesem Fall darf man den String natürlich **nicht** deallokieren.

Um diese beiden Fälle unterscheiden zu können, geht man häufig folgendermaßen vor: Man definiert Verbunde, die hier als **BLOBs** bezeichnet werden. Jeder **BLOB** besteht aus einem Zeiger **ZS**, der auf einen String zeigt, und einer Ganzzahlvariablen **AZ**, in der die Anzahl der Zeiger steht, die auf diesen **BLOB** zeigen. Ein **ZEICHEN_KETTE**-Objekt besteht dann im wesentlichen aus einem Zeiger **ZB**, der auf ein **BLOB**-Objekt zeigt. Zwei **ZEICHEN_KETTE**-Objekte **ZK2** und **ZK3**, die auf dasselbe **BLOB**-Objekt (und damit indirekt auf denselben String) zeigen, sehen dann als Bojen dargestellt etwa so aus:



Vor einer Zuweisung zwischen zwei **ZEICHEN_KETTE**-Objekten **ZK2 := ZK1;** wird **ZK2** finalisiert, indem der Zähler **ZK2.ZB.all.AZ** um 1 vermindert wird. Falls dadurch **AZ** den Wert 0 erreicht, wird zuerst der String **ZK2.ZB.all.ZS.all** und dann der BLOB **ZK2.ZB.all** deallokiert. **Nach** der Zuweisung wird der Zähler **ZK2.ZB.all.AZ** (der **identisch** ist mit dem Zähler **ZK1.-ZB.all.AZ**) um 1 erhöht.

Das folgende Beispiel zeigt, wie man **ZEICHEN_KETTE**-Objekte mit **Referenzsemantik** in Ada realisieren kann.

Beispiel 24.3.1.: Ein kontrollierter Typ **!ZEICHEN_KETTE** mit Referenzsemantik:

```
01 with ada.finalization;
02 package KON_D_01 is
03 ------------------------------------------------------------------------
04 -- Stellt einen privaten kontrollierten Typ !ZEICHEN_KETTE und eine
05 -- Konstruktor-Prozedur MACHE_ZEICHEN_KETTE zur Verfuegung. Objekte
06 -- ZK1, ZK2, etc. des Typs ZEICHEN_KETTE haben "Referenzsemantik", d.h.
07 -- nach einer Zuweisung ZK1 := ZK2 zeigen ZK1 und ZK2 auf denselben String,
08 -- nicht auf zwei gleich Strings.
09 ------------------------------------------------------------------------
10    type ZEICHEN_KETTE is private;
11    procedure MACHE_ZEICHEN_KETTE(S: string; ZK: in out ZEICHEN_KETTE);
12 private
13    type ANZAHL        is range 0..2e9; -- Zum Zaehlen der Zeiger
14    type Z_STRING      is access string;
15    type BLOB          is record
16       AZ : ANZAHL := 0; -- Anzahl Zeiger, die auf diesen BLOB zeigen
17       ZS : Z_STRING;
18    end record;
19    type Z_BLOB        is access BLOB;
20    type ZEICHEN_KETTE is new ada.finalization.controlled with record
21       ZB: Z_BLOB;
22    end record;
23    procedure ADJUST  (ZK: in out ZEICHEN_KETTE);
24    procedure FINALIZE(ZK: in out ZEICHEN_KETTE);
25 end KON_D_01;
```

Die Vereinbarungen in dieser Paketspezifikation, insbesondere die Typen **!Z_STRING**, **!BLOB**, **!Z_BLOB** und **!ZEICHEN_KETTE**, werden durch die obigen Diagramme illustriert.

Hier eine kleine Testprozedur **KON_D_02**, die das Paket **KON_D_01** einbindet und benützt:

```
01 with KON_D_01, ada.text_io;
02 procedure KON_D_02 is
03    ZK1 : KON_D_01.ZEICHEN_KETTE;
04    ZK2 : KON_D_01.ZEICHEN_KETTE;
05    ZK3 : KON_D_01.ZEICHEN_KETTE;
06 begin
07    ada.text_io.put_line(item => "Hallo1 --------------------");
08    KON_D_01.MACHE_ZEICHEN_KETTE(S => "ABC",    ZK => ZK1);
09    KON_D_01.MACHE_ZEICHEN_KETTE(S => "UVWXYZ", ZK => ZK1);
10    ada.text_io.put_line(item => "Hallo2 --------------------");
11    ZK2 := ZK1;
12    ZK3 := ZK2;
13    ada.text_io.put_line(item => "Hallo3 --------------------");
14 end KON_D_02;
```

Hier die Ausgabe des Programms **KOND_D_02**. Die meisten Zeilen stammen auch hier von den Prozeduren im Paket **KON_D_01**, und nur die Hallo-Zeilen werden direkt von der Prozedur KON_D_02 ausgegeben:

```
01 Hallo1 --------------------
02  MZK1 ZK.ZB ist gleich null!
03  MZK2 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: ABC
04  MZK1 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: ABC
05  FIN1 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: ABC
06  FIN2 ZK.ZB ist gleich null!
07  MZK2 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: UVWXYZ
08 Hallo2 --------------------
09  FIN1 ZK.ZB ist gleich null!
10  FIN2 ZK.ZB ist gleich null!
11  ADJ1 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: UVWXYZ
12  ADJ2 ZK.ZB.all.AZ:  2  ZK.ZB.all.ZS.all: UVWXYZ
13  FIN1 ZK.ZB ist gleich null!
14  FIN2 ZK.ZB ist gleich null!
15  ADJ1 ZK.ZB.all.AZ:  2  ZK.ZB.all.ZS.all: UVWXYZ
16  ADJ2 ZK.ZB.all.AZ:  3  ZK.ZB.all.ZS.all: UVWXYZ
17 Hallo3 --------------------
18  FIN1 ZK.ZB.all.AZ:  3  ZK.ZB.all.ZS.all: UVWXYZ
19  FIN2 ZK.ZB.all.AZ:  2  ZK.ZB.all.ZS.all: UVWXYZ
20  FIN1 ZK.ZB.all.AZ:  2  ZK.ZB.all.ZS.all: UVWXYZ
21  FIN2 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: UVWXYZ
22  FIN1 ZK.ZB.all.AZ:  1  ZK.ZB.all.ZS.all: UVWXYZ
23  FIN2 ZK.ZB ist gleich null!
```

Die Zeile 02 beginnt mit **MZK1** zum Zeichen dafür, daß sie zu **Beginn** einer Ausführung der Prozedur **MACHE_ZEICHEN_KETTE** ausgegeben wurde. Entsprechend beginnt die Zeile 02 mit **MZK2**, weil sie am **Ende** einer Ausführung dieser Prozedur ausgegeben wurde. Entsprechendes gilt auch für die anderen Zeilen, die mit **FIN1**, **FIN2**, **ADJ1** bzw. **ADJ2** beginnen. Jede der drei Prozeduren (MACHE_ZEICHEN_KETTE, ADJUST und FINALIZE) hat einen Parameter namens **ZK** vom Untertyp **ZEICHEN_KETTE** und gibt gewisse Informationen über diesen Parameter aus. Z.B. wird in allen **FIN2**-Zeilen mitgeteilt, daß der Zeiger **ZK.ZB** den Wert **null** hat.

In Zeile 04 bis 07 erkennt man an der Folge der Kennungen **MZK1**, **FIN1**, **FIN2**, **MZK2**, daß **innerhalb** der Prozedur **MACHE_ZEICHEN_KETTE** die Prozedur **FINALIZE** aufgerufen wird.

Aufgabe 24.3.1.: Erklären Sie die Ausgaben des Programms **KON_D_02** Zeile für Zeile. Stellen Sie die Variablen **ZK1**, **ZK2** und **ZK3** als Bojen dar. Führen Sie das Programm **KON_D_02** mit Papier und Bleistift aus und verändern Sie die Bojen entsprechend. **o**

Zusammenfassung 24.3.:

- Bei abstrakten Typen wie !ZEICHEN_KETTE unterscheidet man zwei Vorgehensweisen: Die **Kopiersemantik** und die **Referenzsemantik**.
- Bei der **Kopiersemantik** merken die Benutzer des Typs **nicht**, ob oder daß ZEICHEN_KETTE-Objekte **Zeiger** sind und können sich unter einem solchen Objekt eine **Zeichenkette** vorstellen.
- Bei der **Referenzsemantik** merken die Benutzer des Typs, daß ZEICHEN_KETTE-Objekte nur **Zeiger** ("Referenzen") sind, und keine Zeichenketten.
- In objektorientierten Sprachen wie Eiffel, C++, Java und Ada spielen **beide Arten von Semantik** eine wichtige Rolle.
- In Ada kann man **abstrakte Typen** mit Hilfe von **kontrollierten Typen** wahlweise mit einer **Kopiersemantik** oder einer **Referenzsemantik** versehen.

Zentraldokument: Nach Filialdokument A95-24-24

25. Ein-/Ausgabebefehle und Dateien

Ein-/Ausgabebefehle wie **get** und **put** sind im Vergleich zu **internen Befehlen** wie Zuweisungen, Additionen, if-Anweisungen, Schleifen etc. etwas ganz **Besonderes**. Zum einen dauert die Ausführung eines typischen Ein-Ausgabebefehls relativ **lange** (z.B. 100_000 mal so lang wie die Ausführung einer internen Addition). Zum anderen werden Ein-/Ausgabebefehle zu einem wesentlichen Teil vom verwendeten **Betriebssystem** ausgeführt, internen Befehle dagegen direkt vom **Prozessor** des Computers.

Beispiel 25.1.: Ein **get**- oder **put**-Befehl in einem Ada-Programm bewirkt typischerweise, daß eine entsprechende Routine im **Betriebssystem** aufgerufen wird. Diese Routine führt dann die nötige Übertragung von Daten zwischen dem Hauptspeicher des Computers und einem Ein-/Ausgabegerät durch. Dagegen werden **interne Befehle** wie Zuweisungen. Additionen, if-Anweisungen, Schleifen etc. (bzw. die Maschinenbefehle, in die diese Ada-Befehle von einem Compiler übersetzt wurden) direkt vom **Prozessor** des Computers und **ohne** Beteiligung des Betriebssystems ausgeführt. **o**

Die Tatsache, daß Ein-/Ausgabebefehle vom Betriebssystem ausgeführt werden, stellt die Entwickler einer neuen Programmiersprache NPS vor das folgende **Dilemma**:

Wenn sie in die Sprache NPS nur **einfache und schwache** Ein-/Ausgabebefehle aufnehmen, dann kann man relativ leicht erreichen, daß diese Befehle von allen Betriebssystemen "mit genau der gleichen Wirkung" ausgeführt werden. Allerdings werden die Benutzer der Sprache mit solchen einfachen Ein-/Ausgabebefehlen möglicherweise nicht zufrieden sein.

Wenn die Entwickler für die Sprache NPS dagegen **aufwendige und komfortable** Ein-/Ausgabebefehle vorsehen, dann können sie nur schwer oder gar nicht mehr erreichen, daß diese Befehle von allen Betriebssystemen "mit genau der gleichen Wirkung" ausgeführt werden. "Alle Betriebssysteme zu ändern und anzupassen" ist sehr schwer bis unmöglich. "Schöne" Ein-/Ausgabebefehle in einer Sprache NPS haben die Tendenz, die **Portabilität** von NPS-Programmen (ihre "Übertragbarkeit auf viele Computersysteme") zu erschweren oder einzuschränken.

Allgemein haben viele Sprachentwickler die Erfahrung gemacht, daß sich **Ein-/Ausgabebefehle** sehr viel **schwerer** normieren und standardisieren lassen als **interne Befehle**. Die Entwickler von Ada haben daraus folgende Konsequenzen gezogen:

1. Zum **Kern** der Sprache Ada gehören **keine** Ein-/Ausgabebefehle. Damit wird eine wirkungsvolle Standardisierung des Sprachkerns wesentlich erleichtert bzw. überhaupt ermöglicht.
2. Zu Ada gehören eine Reihe von **Programmeinheiten** (Pakete und Schablonen) **mit Ein-/Ausgabebefehlen** darin. Diese Einheiten sind auch **standardisiert**, aber nicht ganz so "hart" wie der Sprachkern.
3. Die **Standardeinheiten** enthalten nur relativ **einfache und schwache** Ein-/Ausgabebefehle, die von allen Betriebssystem mit gleicher Wirkung ausgeführt werden können.

Wenn die Standardeinheiten nicht ausreichen, kann man sie relativ leicht durch selbst entwickelte oder dazugekaufte Programmeinheiten ergänzen oder ersetzen. Allerdings schränkt man dadurch die **Portierbarkeit** des betreffenden Programms auf solche Computersysteme ein, auf denen auch die zusätzlichen Programmeinheiten zur Verfügung stehen.

Anmerkung: Die Entwickler der Sprache **Java** sind radikal anders vorgegangen und haben von Anfang an sehr aufwendige und komfortable Ein-/Ausgabebefehle in ihre Sprache aufgenommen. Dieser Schritt ist sehr zu begrüßen. Allerdings muß sich noch zeigen, ob eine für die Praxis ausreichende **Standardisierung** dieser mächtigen Ein-/Ausgabebefehle gelingt oder nicht. Wenn sie gelingt, werden die Entwickler von Ada sicherlich versuchen, diese Befehle auch in Ada-Programmen zugänglich zu machen, z.B. in Form von zusätzlichen (Standard-) Paketen. **o**

25.1. Überblick über Pakete und Schablonen für die Ein-/Ausgabe

Die folgenden Programmeinheiten enthalten Ein-/Ausgabebefehle und müssen **in jeder Ada-Implementierung** vorhanden sein:

```
ada.text_io                  -- Paket
ada.text_io.integer_io       -- Paketschablone
ada.text_io.modular_io       -- Paketschablone
ada.text_io.enumeration_io   -- Paketschablone
ada.text_io.decimal_io       -- Paketschablone
ada.text_io.fixed_io         -- Paketschablone
ada.text_io.float_io         -- Paketschablone
ada.sequential_io            -- Paketschablone
ada.direct_io                -- Paketschablone
ada.streams                  -- Paket
ada.streams.stream_io        -- Paket
```

ada.text_io, **ada.streams** und **ada.streams.stream_io** sind **Pakete**. Die übrigen Einheiten sind **Paketschablonen**. Wenn man eine der Paketschablonen instanziieren will, muß man einen **Untertyp U** angeben und kann dann mit dem Instanzpaket Werte des Untertyps **U** ein- und ausgeben.

Das Paket **ada.text_io** und die sechs Schablonen **integer_io** bis **float_io** dienen dazu, Daten **in einer von Menschen lesbaren Form** (kurz: **in Textform**) ein- und auszugeben. Die übrigen Schablonen und Pakete dienen dazu, Daten **in Binärform** ein- und auszugeben. Mithilfe der Schablonen **sequential_io** und **direct_io** kann man **typhomogene** Dateien schreiben und lesen. In einer solchen Datei müssen alle Datensätze zum **gleichen Untertyp** gehören. Ein **Strom** (stream) ist etwas ähnliches wie eine Datei, braucht aber **nicht** typhomogen zu sein. D.h. man darf Werte **unterschiedlicher Typen** in einen Strom schreiben. Mit dem Paket **ada.streams** kann man **allgemeine Ströme** programmieren (auch solche, die Daten im Hauptspeicher verwalten oder solche, die als Schnittstelle zu einer Datenleitung dienen). Das Paket **ada.streams.stream_io** dient etwas spezieller zur Verwaltung von sogenannten **Stromdateien** (stream files). Ströme und Stromdateien werden im Unterabschnitt 25.4. behandelt.

Das Paket **ada.wide_text_io** ist sozusagen eine Kopie von **ada.text_io**, in der überall der Untertyp **character** durch den Untertyp **wide_character** ersetzt wurde. Somit enthält das Paket **ada.wide_text_io** auch 100 Größen, darunter die sechs Paketschablonen **integer_io** bis **float_io**.

Zum Unterschied zwischen Ein-/Ausgabe in **Textform** und in **Binärform**: Heute übliche Computer stellen Daten (Zeichen, Zeichenketten, Ganzzahlen, Bruchzahlen, Aufzählungswerte, Verbunde,

Reihungen etc.) intern in ihren Speichern durch bestimmte **Bitketten** dar. Z.B. wird der Buchstabe **'a'** durch eine bestimmte Bitkette dargestellt. Ein Bildschirm kann diese Bitkette in einen **a-förmigen Lichtfleck** umwandeln und wenn man auf die **a-Taste** einer Tastatur drückt, wird diese Bitkette zum Rechner geschickt.

Eine Ganzzahl wie z.B. **-123** wird intern im Rechner ebenfalls durch eine bestimmte Bitkette dargestellt. Aber ein Bildschirm ist **nicht** in der Lage, diese Bitkette in einen Lichtfleck umzuwandeln, der der Zahl erkennbar ähnlich sieht. Um die Zahl auszugeben, muß man sie zuerst in eine entsprechende Zeichenkette umwandeln (genauer: man muß die Bitkette, die die Zahl **-123** darstellt, in die vier Bitketten für die Zeichen **'-'**, **'1'**, **'2'** und **'3'** umwandeln) und diese Zeichenkette dann zum Bildschirm ausgeben. Beim Einlesen einer Zahl von einer Tastatur geht man umgekehrt vor: Man liest zuerst eine Zeichenkette ein und wandelt diese dann in eine interne Ganzzahl um.

Einzelne **Zeichen** (**character**-Werte) und **Zeichenketten** (**string**-Werte) sind die einzigen Daten, die **nicht** umgewandelt zu werden brauchen, wenn man sie von einer Tastatur einliest oder zu einem Bildschirm ausgibt. Bei allen anderen Daten (z.B. bei Ganzzahlen, Bruchzahlen, Aufzählungswerten, Verbunden, Reihungen etc.) muß jeweils eine Umwandlung zwischen der internen Darstellung als Bitkette und einer externen Darstellung als Zeichenkette stattfinden. Wenn diese Umwandlungen stattfinden, spricht man von Ein-/Ausgabe **in einer von Menschen lesbaren Form** (kurz: in **Textform**).

Wenn man Daten nicht zu einem Bildschirm ausgibt, sondern z.B. zu einer **Magnetplatte**, dann braucht man die internen Bitketten des Rechners nicht unbedingt in Zeichenketten umzuwandeln, sondern kann sie unverändert auf die Platte schreiben. Das geht schneller und hat den zusätzlichen Vorteil, daß man auch beim Einlesen solcher Daten gleich die interne Bitkette hat und keine Umwandlung durchführen muß. Wenn man die internen Bitketten des Rechners unverändert ein- und ausgibt, spricht man von Ein-/Ausgabe **in Binärform**.

Wie die Bitketten aussehen, mit denen bestimmte Werte dargestellt werden, wird durch die Sprache Ada **nicht** festgelegt. Jeder Ada-Ausführer kann diese Bitketten im Prinzip so wählen, wie es ihm am besten paßt. Wenn man Daten mit **einem** Ada-Ausführer **in Binärform** in eine Datei schreibt und versucht, sie mit einem **anderen** Ada-Ausführer wieder zu lesen, kann es deshalb zu "Mißverständnissen" kommen. Das ist selbst dann nicht ausgeschlossen, wenn die beiden Ada-Ausführer (Compiler, Binder, Laufzeitsystem etc.) auf **demselben Rechner** laufen. Bei der Ein-/Ausgabe von Daten **in Textform** ist die Situation deutlich **besser**, aber auch da gibt es keinen perfekten Standard (z.B. wird das **Ende einer Textzeile** je nach Betriebssystem verschieden dargestellt).

Beim Ausgeben eines Wertes gehen im allgemeinen Informationen über den **Typ** des Wertes **verloren**. Man kann ganz leicht eine **AEPFEL**-Zahl ausgeben und als **BIRNEN**-Zahl wieder einlesen. Bei der Ein-/Ausgabe in Binärform sind besonders radikale Typverwechslungen möglich. Z.B. kann man den Wert einer **Ganzzahlvariablen G** als Bitkette ausgeben und die Bitkette später dann als Wert einer **Bruchzahlvariablen B** wieder einlesen. Die Werte der Variablen **B** und **G** stehen dabei in aller Regel "in keiner sinnvollen Beziehung zueinander". Dieses Problem löst man bei der **Strom-Ein-/Ausgabe** (stream input output), indem man nicht nur "die nackten Daten" ausgibt, sondern zusätzlich auch Informationen über ihren **Typ** (z.B. in Form der **Etiketten** eines etikettierten Verbundes).

Eine **Datei** ist "irgendein Ding", von dem man Daten **einlesen** oder zu dem man Daten **ausgeben** kann. Beispiele für Dateien: Eine **Tastatur** (nur einlesen möglich), ein **Bildschirm** (nur ausgeben möglich), ein **Drucker** (nur ausgeben möglich), eine **Verbindung zum Internet**, eine **Datei** auf einer Magnetplatte, eine **Datei** auf einer CD-ROM (nur einlesen möglich) etc.. Auf welche "Dinge" man als Dateien zugreifen kann, wird durch das jeweilige Betriebssystem bestimmt.

Um von einem Ada-Programm aus auf eine Datei zugreifen zu können, braucht man eine Art "**Berechtigungsausweis**". Das ist eine Variable, welche zu einem Untertyp namens **file_type** gehört. Bevor man auf eine Datei zugreift, muß man sie durch einen entsprechenden Prozeduraufruf **öffnen**. Dabei muß man einen ("leeren") **Ausweis** als Parameter vorlegen und angeben, wie man auf die Datei zugreifen möchte (nur lesend, nur schreibend, etc.). Wenn das Öffnen der Datei gelingt, trägt die Öffnungsprozedur in den Ausweis ein, daß man ab jetzt zu bestimmten Zugriffen auf die Datei berechtigt ist (nur lesend, nur schreibend, etc.). Bei jedem Zugriff auf die Datei muß man diesen Ausweis vorlegen (d.h. als Parameter des betreffenden Zugriffsunterprogramms angeben).

Mit den Prozeduren namens **create** kann man eine neue Datei erzeugen und öffnen. Mit **open** kann man eine schon existierende Datei öffnen (falls die Datei noch nicht existiert, wird die Ausnahme **name_error** ausgelöst). Mit **close** kann man eine Datei wieder schließen. Dabei wird der angegebene Ausweis "ungültig gemacht", damit man danach auch nicht versehentlich auf die Datei zugreifen kann. Mit Prozeduren namens **get** bzw **put** kann man Daten **in Textform** aus einer Datei **lesen** bzw. in die Datei schreiben. Mit Prozeduren namens **read** bzw. **write** kann man Daten **in Binärform** lesen bzw. schreiben.

Als **Textdatei** wird eine Datei bezeichnet, die mit Befehlen im Paket **ada.text_io** (bzw. **ada.wide_text_io**) bearbeitet wird. Entsprechend wird eine Datei als **sequentielle Datei**, **direkte Datei** bzw. **Stromdatei** bezeichnet, wenn sie mit Hilfe der Paketschablone **ada.sequential_io** bzw. **ada.direct_io** bzw. mit Befehlen aus dem Paket **ada.streams.stream_io** bearbeitet wird.

25.2. Textdateien (ada.text_io)

Das Paket **ada.text_io** stellt seinen Benutzern insgesamt genau **100 Größen** zur Verfügung, nämlich 5 **Typen**, 3 **Untertypen**, 1 **Konstante**, 8 Umbenennungen von **Ausnahmen**, 32 **Funktionen**, 45 **Prozeduren** und 6 **Paketschablonen** (siehe (ARM A.10.1)). Einige dieser Größen sollen im folgenden erläutert werden.

Um auf eine Textdatei zugreifen zu können, braucht man einen "Ausweis", d.h. eine Variable des Typs **!file_type**, der am Anfang des Paketes **ada.text_io** vereinbart wird. Dieser Typ ist **limitiert privat**. Damit ist es nicht möglich, Ausweise mit Zuweisungsanweisungen zu **verändern** oder sie zu **kopieren**. Nur mit Hilfe der Prozeduren **create** bzw. **open** kann man sich in einen **file_type**-Ausweis eine Zugriffsberechtigung auf eine bestimmte Datei "eintragen lassen".

Zum Aufzählungstyp **!file_mode** gehören die drei Werte **in_file**, **out_file** und **append_file**. Beim Öffnen einer Datei muß man einen dieser Werte angeben und ist dann berechtigt, nur lesend bzw. nur schreibend bzw. "anhängend" auf die Datei zuzugreifen. Den Dateimodus **append_file** benützt man, wenn man hinter die Datensätze einer schon existierenden Datei weitere Datensätze "anhängen" will.

Mit den Befehlen im Paket **ada.text_io** (und in Instanzen der Paketschablone **ada.sequential_io**) kann man Dateien grundsätzlich nur **sequentiell** bearbeiten, d.h. man muß Daten **nacheinander** in die Datei schreiben und kann sie später nur in der Reihenfolge lesen, in der man sie geschrieben hat. Mit einer Instanz der Schablone **direct_io** kann man statt sequentiell auch **direkt** auf die Datensätze einer Datei zugreifen (siehe unten Abschnitt 25.3.).

Mit der Prozedur **reset** kann man jederzeit an den Anfang einer Datei zurückkehren und den Dateimodus verändern, z.B. von **out_file** oder **append_file** zu **in_file** oder von **in_file** zu **append_file**. Die Prozedur **reset** erledigt die nötigen Veränderungen des Berechtigungsausweises.

Zu jedem Ada-Programm, welches gerade ausgeführt wird, gehört eine Tabelle mit sechs "schon ausgefüllten" **Berechtigungsausweisen** für Zugriffe auf bestimmte Dateien:

	Eingabe	**Ausgabe**	**Fehlerausgabe**
Standard	Tastatur	Bildschirm	Bildschirm
Aktuelle	Tastatur	Bildschirm	Bildschirm

Welche Dateien in der Zeile **Standard** eingetragen sind, wird vom **Betriebssystem** bestimmt. Die Eintragungen in der Standard-Zeile können im Verlauf der Programmausführung **nicht** geändert werden. Allerdings erlauben einige Betriebssysteme (z.B. UNIX und DOS), beim **Starten des Programms** die Einträge in dieser Zeile durch eine sogenannte **Dateiumlenkung** zu verändern, etwa so:

```
C:\USER> OTTO >c:\vera\temp.tmp  <c:\vera\daten.txt
```

Das Programm namens **OTTO** wird aufgerufen mit der Datei **c:\vera\temp.tmp** als **Standard-ausgabedatei** und mit **c:\vera\daten.txt** als **Standardeingabedatei**.

Die **Aktuelle**-Zeile in obiger Tabelle ist anfänglich eine **Kopie** der **Standard**-Zeile, kann aber im Verlaufe der Programmausführung **verändert** werden. Die Dateien in der Spalte **Eingabe** sind schon im Modus **in_file** ("zum Lesen"), die Dateien in den Spalten **Ausgabe** und **Fehlerausgabe** im Modus **out_file** ("zum Schreiben") geöffnet.

Wennn man in einem **get**-Befehl keinen **Ausweis** (d.h. keine **file_type**-Variable) angibt, wird der Ausweis aus der Zeile **Aktuelle** und Spalte **Eingabe** genommen, d.h. es wird von der **aktuellen Eingabe** gelesen. Gibt man in einem **put**-Befehl keinen Ausweis an, wird entsprechend zur **aktuellen Ausgabe** ausgegeben.

Mit den Prozeduren **set_input**, **set_output** bzw. **set_error** kann man andere Ausweise in die Spalte **Aktuelle** der obigen Tabelle eintragen lassen. Vorher muß man sich einen geeigneten Ausweis besorgen, indem man eine **file_type**-Variable vereinbart und sie von der Prozedur **create** bzw. **open** mit einer Zugriffsberechtigung ausfüllen läßt, etwa so:

Beispiel 25.2.1.: Eigene Dateien als **aktuelle Dateien** festlegen:

```
01 declare
02    AUSWEIS1: ada.text_io.file_type;
03    AUSWEIS2: ada.text_io.file_type;
04    CHAR: character;
04 begin
```

```
05    ada.text_io.open       (file => AUSWEIS1,
06                            mode => ada.text_io.in_file,
07                            name => "C:\VERA\DATEN.TXT");
08    ada.text_io.create     (file => AUSWEIS2,
09                            mode => ada.text_io.out_file,
10                            name => "C:\VERA\TEMP.TMP");
11    ada.text_io.put(item => "Hallo!", file => AUSWEIS2);
12    ada.text_io.get(item =>  CHAR,    file => AUSWEIS1);
13    ...
14    ada.text_io.set_input  (file => AUSWEIS1);
15    ada.text_io.set_output (file => AUSWEIS2);
16    ...
17    ada.text_io.put(item => "Hallo!");
18    ada.text_io.get(item =>  CHAR);
19    ...
```

Mit der Prozedur **open** (siehe Zeile 05 bis 07) wird eine (hoffentlich schon existierende) Datei namens **"C:\VERA\DATEN.TXT"** als Eingabedatei geöffnet. Die Berechtigung zum Lesen aus dieser Datei wird in den **AUSWEIS1** eingetragen. Mit **create** wird (in Zeile 08 bis 10) eine neue, noch leere Datei namens **"C:\VERA\TEMP.TMP"** als Ausgabedatei erzeugt. Die Berechtigung zum Schreiben in diese Datei wird in den **AUSWEIS2** eingetragen.

In Zeile 11 gibt der **put**-Befehl zur Datei **TEMP.TMP** aus.
In Zeile 12 liest der **get**-Befehl von der Datei **DATEN.TXT** ein.

In Zeile 14 wird die Datei **DATEN.TXT** zur **aktuellen Eingabe** gemacht.
In Zeile 15 wird die Datei **TEMP.TMP** zur **aktuellen Ausgabe** gemacht.

In Zeile 17 gibt der **put**-Befehl zur aktuellen Ausgabe (d.h. zur Datei **TEMP.TMP**) aus.
In Zeile 18 liest der **get**-Befehl von der aktuellen Eingabe (d.h. von der Datei **DATEN.TXT**) ein.

Mit den Prozeduren **set_input**, **set_output** und **set_error** darf man die aktuellen Dateien (die Ausweise in der Spalte **Aktuelle** der obigen Tabelle) **beliebig oft** wechseln.
o

Sprachregelung: "Die Datei, auf die der Ausweis AUSWEIS1 Zugriffe erlaubt" bezeichnt man kurz und einfach auch als "die Datei AUSWEIS1". Trotzdem ist ein **Ausweis** natürlich **keine Datei**!

Die Funktionen **standard_input**, **standard_output** und **standard_error** liefern die Ausweise (Werte vom Typ **!file_type**), die in der obigen Tabelle in der Zeile **Standard** stehen und vom Betriebssystem vorgegeben werden. Mit einem Befehl wie z.B. **ada.text_io.put(item => "Hallo!", file => standard_output);** kann man eine Meldung zur Standardausgabe ausgeben.

Die Funktionen **current_input**, **current_output** und **current_error** liefern die Ausweise (Werte vom Typ **!file_type**), die in der obigen Tabelle in der Zeile **Aktuelle** stehen. Mit einem Befehl wie z.B. **ada.text_io.put(item => "Ventil verklemmt!", file => current_error);** kann man eine Meldung zur aktuellen Fehlerausgabe ausgeben.

Mit der Funktion **is_open** kann man prüfen, ob ein Ausweis "gültig ausgefüllt" ist. Der Aufruf **is_open(file => AUSWEIS1)** liefert den Wert **true**, wenn der **AUSWEIS1** zum Zugriff auf eine Datei berechtigt ("wenn die Datei AUSWEIS1 geöffnet ist"). Wenn der **AUSWEIS1** noch oder wieder ungültig ist, wird **false** geliefert.

Wenn man mit einer Prozedur namens **get** ein Zeichen von der Tastatur einliest, muß der Benutzer nicht nur das Zeichen eingeben, sondern auch noch auf die **Return-Taste** drücken. Benutzt man eine der Prozeduren namens **get_immediate**, braucht der Benutzer **nicht** auf die **Return-Taste** zu drücken. Nähere Einzelheiten dazu findet man im (ARM A.10.7).

Mit der Funktion **end_of_file** kann man prüfen, ob man das **Ende einer Datei** erreicht hat. Der Aufruf **end_of_file(file => AUSWEIS1)** liefert den Wert **true**, wenn das Ende der Eingabedatei erreicht ist, auf die der **AUSWEIS1** Lesezugriffe erlaubt. Ist die Datei noch nicht zu Ende, wird **false** geliefert. Wenn man versucht, aus einer Datei zu lesen, bei der das Ende der Datei schon erreicht ist, wird die Ausnahme **end_error** ausgelöst. Von einer Tastatur aus kann man "das Ende der Datei" signalisieren, indem man eine bestimmte Tastenkombination eingibt. Unter UNIX und DOS etc. ist das die Kombination **Strg-z**.

Mit den **get**- und **put**-Prozeduren, die sich direkt im Paket **ada.text_io** befinden, kann man nur Werte der Typen **!character** und **!string** einlesen und ausgeben. Will man Werte eines Ganzzahltyps, eines modularen Typs, eines Aufzählungstyps, eines dezimalen Festpunkttyps, eines gewöhnlichen Festpunkttyps oder eines Gleitpunkttyps einlesen und ausgeben, muß man die entsprechende Paketschablone **integer_io**, **modular_io**, **enumeration_io**, **decimal_io**, **fixed_io** bzw. **float_io** mit einem geeigneten Untertyp **U** instanziieren. In dem Instanzpaket findet man dann **get**- und **put**-Prozeduren zum einlesen und ausgeben von Werten des Untertyps **U**.

Mit der Prozedur **delete** kann man eine Datei **löschen**. Der dabei vorgelegte Ausweis wird ungültig gemacht. Gibt man beim Erzeugen einer neuen Datei mit **create** als Dateinamen eine **leere Zeichenkette** an, wird die Datei als **temporäre Datei** behandelt und automatisch gelöscht, wenn man sie mit **close** schließt oder wenn das erzeugende Programm fertig ausgeführt ist.

Aufgabe 25.1.1.: Machen Sie sich anhand des Beispielprogramms **EATXT_01** mit den grundlegenden Befehlen für die **Ein-/Ausgabe von Daten in Textform** vertraut (Wie vereinbart man einen **Ausweis**? Wie benützt man ihn beim Zugreifen auf eine Datei? Die Unterprogramme **create**, **put**, **put_line**, **get_line**, **end_of_file**, **close** und **is_open**). o

Aufgabe 25.1.2.: Mit den Unterprogrammen **set_output**, **standard_output**, **current_output** etc. kann man die Standarddateien und die aktuellen Dateien verwalten. Schauen Sie sich als Beispiel dafür auch das Programm **EATXT_02** an. o

Aufgabe 25.1.3.: Das Beispielprogramm **EATXT_03** verlängert eine Textdatei namens **EATXT_03.AUS** im Dateimodus **append_file** um ein paar Zeilen (wenn die Datei schon existiert) bzw. erzeugt eine neue Datei (wenn die Datei noch nicht existiert). Lassen Sie das Programm mehrmals ausführen und sehen Sie sich nach jeder Ausführung die Ausgabedatei **EATXT_03.AUS** an. o

Aufgabe 25.1.4.: Schreiben Sie ein Programm namens **EATXT_04**, welches von der aktuellen Eingabe den **Namen einer Textdatei** (genauer: ihren Pfadnamen) einliest, die entsprechende Datei öffnet, sie mit **get_line** "durchliest" und zählt, aus wieviel Zeilen und Zeichen die Datei besteht. Die Ergebnisse der Zählung sollen "in lesbarer Form" zur aktuellen Ausgabe ausgegeben werden. o

Zusammenfassung 25.1.:

- Das Paket **ada.text_io** dient dazu, Daten **in Textform** einzulesen und auszugeben.

- Dabei werden beim Einlesen **Zeichenketten** in **interne Werte** umgewandelt.
- Beim Ausgeben werden **interne Werte** in **Zeichenketten** umgewandelt.
- Diese Umwandlungen werden von den Prozeduren namens **get** und **put** durchgeführt.
- Nur **character**- und **string**-Werte müssen bei der Ein-/Ausgabe in Textform **nicht** umgewandelt werden.
- Die **Standarddateien** (Standard-Eingabe, -Ausgabe und Fehlerausgabe) werden vom Betriebssystem festgelegt und können von einem Programm aus **nicht** verändert werden.
- Die **aktuellen Dateien** (die akutelle Eingabe, Ausgabe und Fehlerausgabe) werden ebenfalls vorgegeben, können aber durch **andere Dateien** ersetzt werden.
- Das Paket **ada.wide_text_io** ist eine Kopie von **ada.text_io**, in der überall **character** durch **wide_character** ersetzt wurde.
- Eine genaue und verbindliche Beschreibung des Paketes **ada.text_io** findet man in (ARM A.10).

25.2. Sequentielle Dateien (ada.sequential_io)

Eine **sequentielle Datei** besteht aus Datensätzen, die man nur in der Reihenfolge **lesen** kann, in der sie in die Datei **geschrieben** wurden. Die Datensätze einer sequentiellen Datei müssen alle zum **gleichen Untertyp** gehören. Dabei sind aber nicht nur **definite** ("als Bauplan vollständige") Untertypen erlaubt, sondern auch **indefinite** Typen, z.B. Verbunduntertypen mit Diskriminanten.

Textdateien und **sequentielle Dateien** kann man beide nur **sequentiell** bearbeiten. Der Unterschied zwischen diesen Dateiarten besteht darin, daß bei einer **sequentiellen** Datei die Ein-/Ausgabe **in Binärform** erfolgt, bei einer **Textdatei** dagegen **in Textform**.

Das folgende Beispiel zeigt, wie man die Paketschablone **ada.sequential_io** instanziieren und mit dem Instanzpaket sequentielle Dateien bearbeiten kann:

Beispiel 25.2.1.: Die Paketschablone **ada.sequential_io** benützen:

```
01 with ada.sequential_io;
02 procedure EASEQ_00 is
03    type      OTTO is ... -- irgend ein (Unter-) Typ
04    package   OTTO_SEA is new ada.sequential_io(element_type => OTTO);
05    AUSWEIS1: OTTO_SEA.file_type;
06    AUSWEIS2: OTTO_SEA.file_type;
07    OTTO1   : OTTO := ... -- ein passender OTTO-Wert
08    ...
09 begin
10    ...
11    OTTO_SEA.create(file => AUSWEIS1, name => "C:\DATEI1.TMP");
12    OTTO_SEA.open  (file => AUSWEIS2, name => "C:\DATEI2.DAT",
13                    mode => OTTO_SEA.in_file);
14    ...
15    OTTO_SEA.read  (file => AUSWEIS2, item => OTTO1);
16    ...
17    OTTO_SEA.write (file => AUSWEIS1, item => OTTO1);
18    ...
19    OTTO_SEA.reset (file => AUSWEIS1, mode => OTTO_SEA.in_file);
20    ...
21    if OTTO_SEA.end_of_file(file => AUSWEIS1) then
22       OTTO_SEA.close(file => AUSWEIS1);
23    ...
```

Das Paket **OTTO_SEA** ist eine Instanz der Paketschablone **ada.sequential_io** (Zeile 04). Mit dem Instanzpaket **OTTO_SEA** kann man beliebig viele **OTTO-Dateien** sequentiell bearbeiten. Als **OTTO-Datei** wird hier eine Datei bezeichnet, deren Datensätze alle zum Untertyp **OTTO** gehören.

Mit den beiden **file_type**-Variablen **AUSWEIS1** und **AUSWEIS2** (Zeile 05 und 06) kann man "gleichzeitig" auf zwei **OTTO-Dateien** zugreifen. Man beachte: Die beiden Untertypen namens **file_type**, die im Paket **OTTO_SEA** bzw. im Paket **ada.text_io** vereinbart sind, haben formal **nichts** miteinander zu tun und sind Untertypen **verschiedener Typen**. Die Unterprogramme im Paket **OTTO_SEA** akzeptieren nur Ausweise der Untertyps **OTTO_SEA.file_type** und die Unterprogramme im Paket **ada.text_io** bestehen genauso hartnäckig auf Ausweisen des Untertyps **ada.text_io.file_type**.

In Zeile 11 wird eine neue Datei namens **"C:\DATEI.TMP"** erzeugt und im Dateimodus **out_file** geöffnet (in eine neue sequentielle Datei kann man nur schreiben). Die Berechtigung zum **Schreiben** in die neue Datei wird von der **create**-Prozedur in den **AUSWEIS1** eingetragen.

In Zeile 12 wird eine (hoffentlich) schon existierende Datei namens **"C:\DATEI2.DAT"** im Dateimodus **in_file** geöffnet. Die Berechtigung zum Lesen aus dieser Datei wird von der **open**-Prozedur in den **AUSWEIS2** eingetragen. Wenn der Ausführer keine Datei mit dem angegebenen Namen finden kann (oder das Betriebssystem ihm keinen Zugriff darauf erlaubt) wird die Ausnahme **name_error** ausgelöst.

In Zeile 15 wird ein Datensatz aus der Datei **AUSWEIS2** in die Variable **OTTO1** gelesen. Das Lesen erfolgt **in Binärform**, d.h. **ohne** irgendwelche Umwandlungen.

In Zeile 17 wird der Wert der Variablen **OTTO1** als nächster Datensatz in die Datei **AUSWEIS2** geschrieben. Das Schreiben erfolgt **in Binärform**, d.h. **ohne** irgendwelche Umwandlungen.

In Zeile 19 wird zum Anfang der Datei AUSWEIS1 zurückgekehrt und der Dateimodus auf **in_file** gesetzt. Der **AUSWEIS1** berechtigt ab jetzt **nicht** mehr zum **Schreiben** in diese Datei, dafür aber zum **Lesen** aus dieser Datei.

In Zeile 21 wird geprüft, ob das Ende der (Eingabe-) Datei **AUSWEIS1** erreicht ist. Wenn ja, wird die Datei **AUSWEIS1** in Zeile 22 **geschlossen**. Dadurch verliert der **AUSWEIS1** seine Gültigkeit. Man könnte ihn aber durch einen erneuten Aufruf der Prozedur **open** oder **create** wieder gültig machen lassen (d.h. Ausweise sind "beliebig oft wiederverwendbar").

Aufgabe 25.2.1.: Das Beispielprogramm **EASEQ_01** erzeugt eine sequentielle Datei namens **EASEQ_01.AUS** und schreibt ein paar Verbunde eines Verbunduntertyps **PERSON** hinein. Außerdem werden diese Verbunde zur aktuellen Ausgabe ausgegeben ("damit der Benutzer sieht, was passiert"). Mit dem Beispielprogramm **EASEQ_02** kann man die Verbunde in der Datei **EASEQ_01.AUS** wieder einlesen und ebenfalls zur aktuellen Ausgabe ausgeben lassen. Finden Sie heraus: Warum stimmen die Bildschirmausgaben von **EASEQ_01** und **EASEQ_02** nicht überein? Warum ist **KARL** plötzlich **weiblich** und nicht mehr **männlich**? Es handelt sich dabei um eine sehr allgemeine und wichtige Gefahr bei der Aus- und Eingabe getypter Daten! **o**

Aufgabe 25.2.2.: Das Beispielprogramm **EASEQ_03** schreibt ein paar Verbunde eines **varianten Verbundtyps** (eines Verbundtyps mit einer **Diskriminanten**) in eine sequentielle Datei, liest sie dann von dort wieder ein und gibt sie zur aktuellen Ausgabe aus. Das Besondere an diesem Beispiel

ist, daß die einzelnen Verbunde **unterschiedlich viele Komponenten** haben. Überzeugen Sie sich davon, daß trotzdem alles richtig funktioniert. **o**

Aufgabe 25.2.3.: Schreiben Sie ein Programm namens **EASEQ_05**, welches von der aktuellen Eingabe den Namen einer zu bearbeitenden Textdatei (genauer: ihren Pfadnamen) einliest. Die entsprechende Datei soll aber nicht als **Textdatei** (mit **ada.text_io**) bearbeitet werden, sondern als **sequentielle Datei**, und zwar mit einem Paket namens **SEA** welches wie folgt vereinbart wird:

```
10 package SEA is new ada.sequential_io(element_type => character);
```

Die zu bearbeitende Datei soll "Zeichen für Zeichen durchgelesen" werden. Dabei sollen die **Anzahl der Zeichen** und die **Anzahl der Zeilen** gezählt und am Schluß "in lesbarer Form" zur aktuellen Ausgabe ausgegeben werden. Das **Ende einer Zeile** wird je nach Betriebssystem durch ein oder zwei Zeichen gekennzeichnet: Unter **DOS** durch ein **CR**- und ein **LF**-Zeichen, unter **Unix** nur durch ein **LF**-Zeichen und bei Macintosh-Dateien nur durch ein **CR**-Zeichen. Geeignete **character**-Konstanten namens **CR** und **LF** findet man im Paket **ada.characters.latin_1** (siehe (ARM A.3.3)). **o**

Zusammenfassung 25.2.:

- Eine **sequentielle Datei** besteht aus **Datensätzen**.
- Diese Datensätze brauchen **nicht** gleich lang zu sein, müssen aber zum **gleichen** (definiten oder indefiniten) **Untertyp** gehören.
- Dieser Untertyp kann z.B. ein **Verbunduntertyp** sein und darf sogar **Diskriminanten** besitzen.
- Für **sequentielle Dateien** heißen die Prozeduren zum Lesen und Schreiben **read** und **write**.
- Die Prozeduren **read** und **write** lesen bzw. schreiben **in Binärform**, d.h. sie führen **keine** Umwandlungen der Daten durch.
- Zum Vergleich: Für **Textdateien** heißen die Prozeduren zum Lesen und Schreiben **get** und **put**.
- Die Prozeduren **get** und **put** lesen bzw. schreiben **in Textform**, d.h. sie führen **Umwandlungen** durch zwischen **externen Zeichenketten** und **internen Werten**.
- Die Unterprogramme **create**, **open**, **close**, **delete** und **end_of_file** für **sequentielle Dateien** funktionieren ganz ähnlich wie die gleichnamigen Unterprogramme für **Textdateien**.
- Eine genaue und verbindliche Beschreibung der Paketschablone **ada.sequential_io** findet man in (ARM A.8).

25.3. Direkte Dateien (ada.direct_io)

Eine **direkte Datei** besteht aus Datensätzen, die alle zum **gleichen Untertyp** gehören müssen. Dieser Untertyp muß **definit** ("als Bauplan vollständig") sein. Praktisch bedeutet das: Die Datensätze einer direkten Datei müssen alle **gleich lang** sein. Uneingeschränkte Reihungsuntertypen wie z.B. **string** oder Verbunduntertypen mit Diskriminanten sind **indefinit** und somit **nicht** erlaubt.

Eine **direkte Datei** hat Ähnlichkeit mit einer **Reihung**. Eine Reihung besteht aus **Komponenten**, auf die man mit Hilfe von **Indizes** zugreifen kann. Eine direkte Datei besteht aus **Datensätzen**, auf die man mit Hilfe von **Satznummern** zugreifen kann. Auf einen Datensatz **direkt** zugreifen soll

heißen, daß man den Datensatz lesen bzw. schreiben kann, ohne vorher die davorstehenden Datensätze lesen bzw. schreiben zu müssen.

Für direkte Dateien gibt es Befehle wie etwa "Lies den Satz mit der Nummer 123" oder "Schreibe diese Daten als Satz mit der Nummer 231" etc.. Außerdem kann man die Sätze einer direkten Datei auch **sequentiell** schreiben bzw. lesen.

Der wichtigste **Unterschied** zwischen **direkten Dateien** und **Reihungen** hat mit ihrer **Größe** zu tun. Beim Erzeugen einer **Reihung** wird die Anzahl ihrer Komponenten **festgelegt** und kann später **nicht** mehr verändert werden. Die Größe einer **direkten Datei** ist dagegen **flexibel** und wird **automatisch angepaßt**. Jedesmal, wenn man einen Datensatz mit einer Nummer **n** in eine direkte Datei schreibt, wird die Datei (falls nötig) so "verlängert", daß sie aus mindestens **n** Datensätzen besteht. Wenn man z.B. in eine neu erzeugt direkte Datei einen Datensatz mit der Nummer 100 schreibt, hat die Datei anschließend Platz für 100 Datensätzen (mit den Nummern 1 bis 100). Schreibt man danach einen Datensatz mit der Nummer 87 in die Datei, wird ihre Größe nicht verändert. Aber wenn man später einen Datensatz mit der Nummer 500 schreibt, wird die Datei entsprechend verlängert etc.. Wenn der Ausführer (bzw. das verwendete Betriebssystem) eine bestimmte Dateiverlängerung nicht durchführen kann, wird eine Ausnahme ausgelöst (Welche? device_error?). Merkwürdigerweise gibt es in Ada **keinen** Befehl, mit dem man eine direkte Datei **verkleinern** kann (etwa: "reduziere auf 100 Datensätze" oder so ähnlich).

Wenn man beim **Lesen** aus einer direkten Datei eine zu große Satznummer angibt, wird die Datei nicht etwa entsprechend verlängert, sondern die Ausnahme **end_error** wird ausgelöst.

Es gibt noch zwei kleinere Unterschiede zwischen **direkten Dateien** und **Reihungen**. Die Indizes einer Reihung können zu einem **beliebigen** (vom Programmierer festgelegten) diskreten **Untertyp** gehören. Die Satznummern einer direkten Datei müssen zu einem bestimmten Ganzzahluntertyp namens **positive_count** gehören und beginnen immer bei 1. Die Obergrenze **positive_count'last** wird nicht durch die Sprache Ada festgelegt und kann von Ausführer zu Ausführer verschieden sein. Schließlich lebt eine Reihung höchstens bis zum **Ende der Programmausführung**, in deren Verlauf sie erzeugt wurde. Dagegen ist eine Datei im allgemeinen **persistent**, d.h. sie exisitiert unabhängig von irgendwelchen Programmausführungen und verschwindet nur, wenn man sie ausdrücklich löscht.

Das folgende Beispiel zeigt, wie man die Paketschablone **ada.direct_io** instanziieren und mit dem Instanzpaket **direkte Dateien** bearbeiten kann:

Beispiel 25.3.1.: Die Paketschablone ada.direct_io benützen:

```
01 with ada.direct_io;
02 procedure EADIR_00 is
03    type      EMIL is ... -- irgend ein (Unter-) Typ
04    package   EMIL_DEA is new ada.direct_io(element_type => EMIL);
05    AUSWEIS1: EMIL_DEA.file_type;
06    AUSWEIS2: EMIL_DEA.file_type;
07    EMIL1   : EMIL := ... -- ein passender EMIL-Wert
08    SNR     : EMIL_DEA.positive_count := 123;
09    ...
10 begin
11    ...
12    EMIL_DEA.create        (file => AUSWEIS1, name => "C:\DATEI1.TMP"
13                            mode => EMIL_DEA.inout_file);
14    EMIL_DEA.open          (file => AUSWEIS2, name => "C:\DATEI2.DAT",
15                            mode => EMIL_DEA.in_file);
16    ...
```

```
17    EMIL_DEA.read          (file => AUSWEIS2, item => EMIL1, from => 87);
18    ...
19    EMIL_DEA.write         (file => AUSWEIS1, item => EMIL1, to => SNR+2);
20    ...
21    EMIL_DEA.reset         (file => AUSWEIS1, mode => EMIL_DEA.in_file);
22    EMIL_DEA.set_index     (file => AUSWEIS1, to => 25);
23    while not EMIL_DEA.end_of_file(file => AUSWEIS1) loop
24       EMIL_DEA.read(file => AUSWEIS1, item => EMIL1);
25       ...
26    end loop;
27    ...
```

Das Paket **EMIL_DEA** ist eine Instanz der Schablone **ada.direct_io** (Zeile 04). Mit dem Instanzpaket **EMIL_DEA** kann man beliebig viele **EMIL**-Dateien direkt bearbeiten. Als **EMIL-Datei** wird hier eine Datei bezeichnet, deren Datensätze alle zum Untertyp **EMIL** gehören.

Mit den beiden **file_type**-Variablen **AUSWEIS1** und **AUSWEIS2** (Zeile 05 und 06) kann man "gleichzeitig" auf zwei **EMIL**-Dateien zugreifen. Man beachte, daß der Typ **!file_type** im Paket **EMIL_DEA** verschieden ist von dem Typ **!file_type** im Paket **ada.text_io**.

In Zeile 12 bis 13 wird eine neue Datei erzeugt und im Dateimodus **inout_file** geöffnet. Diesen Modus gibt es **nur für direkte Dateien** und nicht für Textdateien oder sequentielle Dateien. Er berechtigt gleichzeitig zum **Lesen** und zum **Schreiben**. Diese Berechtigung wird in den **AUSWEIS1** eingetragen.

In Zeile 14 bis 15 wird eine (hoffentlich) schon existierende Datei im Modus **in_file** geöffnet. Die Berechtigung zum Lesen aus dieser Datei wird in den **AUSWEIS2** eingetragen.

In Zeile 17 wird der Datensatz mit der Nummer **87** aus der Datei **AUSWEIS2** in die Variable **EMIL1** gelesen. Als **from**-Parameter muß man irgendeinen Ausdruck des Untertyps **EMIL_DEA.-positive_count** angeben.

In Zeile 19 wird der Wert der Variablen **EMIL1** als Datensatz mit der Nummer **SNR+2** in die Datei **AUSWEIS1** geschrieben. Falls nötig wird die Datei vorher entsprechend verlängert.

In Zeile 21 wird zum Anfang der Datei **AUSWEIS1** zurückgekehrt und der Dateimodus von **inout_file** zu **in_file** geändert. Ab jetzt berechtigt der **AUSWEIS1** nur noch zum Lesen aus der Datei.

In Zeile 22 wird die **aktuelle Satznummer** der Datei **AUSWEIS1** auf 25 gesetzt. Da im nächsten **read**-Befehl (Zeile 24) kein **from**-Parameter angegeben wurde, liest dieser Befehl den Satz mit der **aktuellen Satznummer** und erhöht diese anschließend um eins. Konkret bedeutet das: Mit der **while**-Schleife (Zeile 23 bis 26) werden ab dem Satz mit der Nummer 25 alle Sätze der Datei AUSWEIS1 **sequentiell gelesen**. Ganz entsprechend kann man Sätze auch **sequentiell** in eine direkte Datei **schreiben**, indem man im **write**-Befehl keinen **to**-Parameter angibt.

Die Prozeduren **EMIL_DEA.close** und **EMIL_DEA.delete** funktionieren bei direkten Dateien genauso, wie die gleichnamigen Prozeduren für andere Dateien. **o**

Aufgabe 25.3.1.: Das Beispielprogramm **EADIR_01** erzeugt eine direkte Datei, in der Strings als Datensätze stehen. Der Benutzer kann eine Satznummer eingeben und bekommt dann den entsprechenden String angezeigt. Schauen Sie sich das Programm an und probieren Sie es aus. **o**

Zusammenfassung 25.3.:

- Eine **direkte Datei** besteht aus **Datensätzen**, auf die man mit Hilfe von **Satznummern** zugreifen kann.
- Die Datensätze müssen alle zum gleichen **definiten Untertyp** gehören, d.h. sie müssen alle **gleich lang** sein.
- Wenn man einen Datensatz mit der Nummer **n** in eine direkte Datei **schreibt**, wird die Datei (falls nötig) automatisch so **verlängert**, daß sie Platz für **n** Datensätze hat.
- Die Satznummern beginnen immer bei **1**.
- Es gibt **keinen** Ada-Befehl zum **Verkleinern** einer direkten Datei.
- Man kann Sätze auch **sequentiell** in eine **direkte Datei** schreiben bzw. aus ihr lesen.
- Für direkte Dateien gibt es (außer den üblichen Modi **in_file**, **out_file** und **append_file**) auch den Modus **inout_file**, der für Textdateien und sequentielle Dateien **nicht** zur Verfügung steht.
- Eine genaue und verbindliche Beschreibung der Paketschablone **ada.direct_io** findet man in (ARM A.8.4 und A.8.5).

25.4. Ströme und Stromdateien (ada.streams und stream_io)

Ein **Strom** ist (ganz ähnlich wie eine **Datei**) ein "Ding", in das man Daten **hineinschreiben** und aus dem man Daten **herauslesen** kann. Ein Strom ist aber (anders als eine Datei) **nicht typhomogen**, sondern kann Werte verschiedener Typen aufnehmen und wieder abgeben, d.h. ein Strom ist im allgemeinen **typheterogen**.

In Ada kann der Programmierer selbst Ströme mit neuen, "maßgeschneiderten" Eigenschaften programmieren. Dabei kann er einen Strom z.B. durch einen **Bereich im Hauptspeicher** realisieren (als sogenannte "RAM-Disk") oder als Schnittstelle zu einer **Datenleitung** oder mit Hilfe einer **Datei**. Außerdem kann er das **Protokoll** festlegen, nach dem Daten von einem Strom aufgenommen und wieder abgegeben werden, z.B. wie bei einer **Schlange** in FIFO-Folge (first in, first out), wie bei einem **Stapel** in LIFO-Folge (last in, first out), wie bei einer (sequentiellen oder direkten) **Datei** oder nach einem anderen Protokoll.

Das Paket **ada.streams** enthält Typen und Unterprogramme, mit deren Hilfe man selbst Ströme mit "maßgeschneiderten" Eigenschaften programmieren kann. Zur Lösung vieler Stromprobleme genügen aber die Ströme "von der Stange", die das Paket **ada.streams.stream_io** einem fertig zur Verfügung stellt. Dabei handelt es sich um sogenannte **Stromdateien** (stream files).

Eine **Stromdatei** kombiniert die Eigenschaften eines **Stromes** mit den Eigenschaften einer **Datei**. Weil eine Stromdatei ein **Strom** ist, kann man Werte verschiedener Typen hineinschreiben. Weil eine Stromdatei eine **Datei** ist, sind ihre Werte persistent und man kann sie beliebig oft lesen etc..

Technisch und etwas genauer gilt in Ada folgendes:

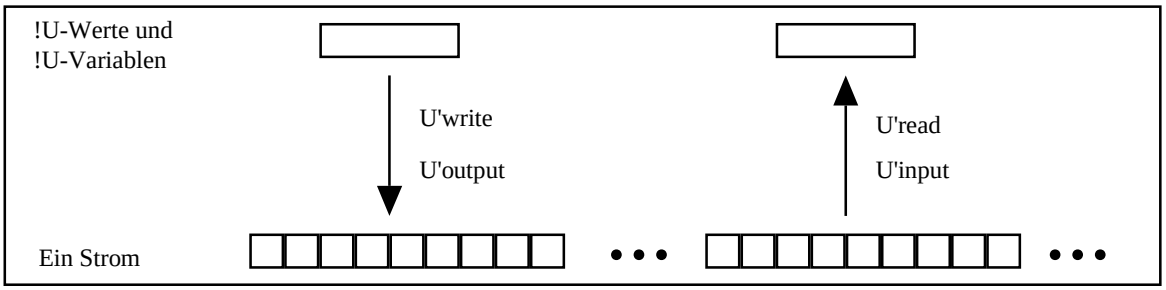
Ein **Stromtyp** ist ein Typ, der direkt oder indirekt von dem Typ **root_stream_type** im Paket **ada.streams** abgeleitet wurde. Ein **Strom** ist eine Variable eines Stromtyps.

Der Typ **root_stream_type** ist ein **nur-Vater-Typ** (abstract type) mit zwei abstrakten Prozeduren namens **read** und **write**. Wenn man einen eigenen "maßgeschneiderten" Stromtyp vom Typ **root_stream_type** ableitet, muß man ("wie immer") diese abstrakten Prozeduren durch selbst programmierte, konkrete Prozeduren verdecken. Dadurch kann man praktisch alle Eigenschaften der neuen Ströme selbst bestimmen.

Konkret besteht ein **Strom** aus sogenannten **Stromelementen**, die zum Typ **!stream_element** gehören. Dieser Typ ist im Paket **ada.streams** als modularer Ganzzahltyp vereinbart. "Wie groß" ein Stromelement ist, wird nicht durch die Sprache Ada festgelegt, sondern vom jeweiligen Ausführer bestimmt. Typischerweise entspricht ein Stromelement einem **Byte** oder einem **Speicherwort** des verwendeten Prozessors. Die konkrete Prozedur **write**, die zu jedem Stromtyp gehört, schreibt eine **Reihung von Stromelementen** in einem Strom und entsprechend liest die Prozedur **read** eine **Reihung von Stromelementen** aus einem Strom. Allerdings ruft der Programmierer diese Prozeduren **write** und **read** fast nie "direkt" auf. Meistens ruft er andere Unterprogramme auf, die ihrerseits **write** bzw. **read** aufrufen.

Zu jedem Untertyp **U** (zu jedem!) gibt es vier Attributunterprogramme namens **U'write**, **U'read**, **U'output** und **U'input**. Mit den Prozeduren **U'write** bzw. **U'output** kann man einen Wert des Typs **!U** in einen Strom **hineinschreiben**. Mit der Prozedur **U'read** bzw. der Funktion **U'input** kann man einen Wert des Typs **!U** aus einem Strom **herauslesen**.

Man kann die Unterprogramme **U'write**, **U'output**, **U'read** und **U'input** als "Vermittler" zwischen zwei Abstraktionsebenen verstehen. Die **Werte des Typs !U** gehören zur höheren ("abstrakteren") Ebene. Zur unteren ("konkreteren") Ebene gehören **Stromelemente**, **Reihungen von Stromelementen** und **Ströme**. Die Prozeduren **U'write** und **U'output** übertragen Daten von der höheren zur tieferen Ebene. Die Unterprogramme **U'read** und **U'input** übertragen Daten in der umgekehrten Richtung. Beim Übertragen finden Umwandlungen zwischen **!U-Werten** (auf der höheren Ebene) und **Reihungen von Stromelementen** (auf der unteren Ebene) statt, etwa so:



Wenn der Programmierer die Prozedur **U'write** aufruft, muß er zwei aktuelle Parameter angeben: Einen **!U-Wert** und einen **Strom**. Die Prozedur **U'write** wandelt den **!U-Wert** in eine **Reihung von Stromelementen** um und schreibt diese Reihung mit der Prozedur **write** des entsprechenden Stromtyps in den Strom.

Wenn der Programmierer die Prozedur **U'read** aufruft, muß er eine **!U-Variable item** und einen **Strom** als aktuelle Parameter angeben. Die Prozedur **U'read** liest mit der Prozedur **read** des entsprechenden Stromtyps eine **Reihung von Stromelementen** aus dem Strom, macht daraus einen **!U-Wert** und weist ihn der Variablen **item** zu. Dabei "weiß" die Prozedur **U'read**, wieviele Stromelemente sie lesen muß, um daraus einen **!U-Wert** zu rekonstruieren.

Die Prozeduren **U'write** und **U'read** schreiben bzw. lesen nur einen "nackten !U-Wert". Die Unterprogramme **U'output** und **U'input** leisten ganz ähnliches, schreiben bzw. lesen aber in Abhängigkeit vom Untertyp U außer dem U-Wert noch die folgenden "Zusatzdaten":

wenn U ein **Reihungsuntertyp** ist:	Die **Indexgrenzen** des U-Wertes
wenn U ein **varianter Verbundtyp** ist:	Die **Diskrimanten** des U-Wertes

Wenn man z.B. Strings mit **string'write** in einen Strom schreibt, geht die **Länge** der einzelnen Strings verloren. Beim Lesen mit **string'read** muß man dann jeweils angeben, "wieviel String" man haben will. Wenn man die Strings dagegen mit **string'output** in den Strom schreibt, werden die Indexgrenzen mit abgespeichert und beim Lesen mit **string'input** bekommt man genau den String, den man geschrieben hat. Siehe dazu die Beispielprogramme **EAS_A_02** und **EAS_B_02**.

Wenn U ein **elementarer Untertyp** ist (z.B. ein Ganzzahluntertyp oder ein Aufzählungsuntertyp etc.) haben die Prozeduren **U'write** und **U'output** die gleiche Wirkung und **U'read** und **U'input** unterscheiden sich nur dadurch, daß **U'read** eine **Prozedur** und **U'input** eine **Funktion** ist.

Mit den Unterprogrammen **U'class'output** und **U'class'input** kann man einen etikettierten Verbund des klassenweiten Untertyps **U'class** zusammen mit seinem **Etikett** (d.h. **mit voller Typinformation**) in einen Strom hineinschreiben bzw. aus einem Strom herauslesen. Die Prozedur **U'class'output** schreibt zuerst das **Etikett** des Verbundes in den Strom und ruft dann per Wahlaufruf (dispatching call) die zum Etikett passende Prozedur **'output** auf. Umgekehrt liest die Funktion **U'class'input** zuerst ein Etikett aus dem Strom und ruft dann die dazu passende Funktion **'input** auf.

Ströme sind von ihrer Grundidee her **nicht typsicher** und damit gefährlich. Wenn man z.B. einen **Ganzzahlwert** G in einen Strom S schreibt und später (wenn man beim Lesen an der entsprechenden Stelle von S angekommen ist) aus dem Strom einen **Bruchzahlwert** einliest, stellt der Ausführer **keinen Typfehler** fest, sondern interpretiert die Stromelemente der **Ganzzahl** G einfach als Stromelemente einer **Bruchzahl**. Die Beispielprogramme **EAS_A_06** und **EAS_B_06** zeigen, welche "unsinnigen Verwandlungen" dadurch zustande kommen können. Mithilfe der Unterprogramme **U'class'output** und **U'class'input** kann der Programmierer bestimmte Typenfehler vermeiden bzw. automatisch erkennbar machen. Grundsätzlich sollte er sich für eine von zwei Alternativen entscheiden:

1. Er schreibt und liest mit den Prozeduren **'write** und **'read** nur "nackte Werte" ohne Typinformationen. Diese Alternative ist sehr **effizient** aber auch gefährlich, weil "Typverwechslungen" möglich sind und nicht automatisch entdeckt werden.
2. Er schreibt und liest mit den Unterprogrammen **U'class'output** und **U'class'input** nur **etikettierte Verbunde** eines klassenweiten Untertyps **U'class**. Diese Alternative ist weniger effizient (weil auch Etiketten geschrieben und gelesen werden müssen). Dafür sind bestimmte "Typverwechslungen" nicht möglich.

"Ganz typensicher" ist ein Strom aber **nie**. Auch wenn der Programmierer sich vorgenommen hat, zur Bearbeitung eines Stroms nur die Unterprogramme **U'class'output** und **U'class'input** zu benützen, kann er doch jederzeit aus Versehen z.B. mit einer der Prozeduren **integer'write** oder **boolean'write** etc. einen "typenmäßig falschen Wert" in den Strom schreiben. Beim Zusammentreffen unglücklicher Umstände entdeckt der Ausführer trotzdem keinen Fehler (weder beim Übergeben des Programms noch während seiner Ausführung).

Zum Abschluß noch ein kleines technisches Detail: Die Unterprogramme **U'read**, **U'write**, **U'input**, **U'output**, **U'class'input** und **U'class'output** erwarten als ersten Parameter nicht einen **Strom**, sondern "nur" einen **Zeiger**, der auf einen Strom zeigt. Der Zeiger darf zu einem **beliebigen** Zeigertyp gehören, dessen Zieltyp ein Stromtyp ist. Durch diese Festlegung wird der Umgang mit Stömen an einigen Stellen etwas einfacher und flexibler.

Daß man selbst einen neuen Stromtyp (mit "maßgeschneiderten Eigenschaften") vereinbaren muß, kommt relativ selten vor. Trotzdem wird hier damit begonnen, damit der Leser sich ganz allgemein mit **Strömen** vertraut machen kann, bevor er sich mit den vorgefertigten, speziellen **Stromdateien** befaßt.

Analogie: Es schadet nichts, während der ersten Fahrstunde einen Blick unter die Motorhaube eines Autos zu werfen. Auch wenn man nicht Automechaniker werden will, ist es gut zu wissen, was ein Keilriemen ist, wo sich die Lichtmaschine befindet und was die beiden miteinander zu tun haben.

Beispiel 25.4.1.: Selbst einen neuen **Stromtyp** vereinbaren:

```
01 with ada.streams;
02 package EAS_A_01 is
03 ------------------------------------------------------------------------
04 -- Stellt seinen Benutzern einen Stromtyp (stream type) namens SCHLANGE und
05 -- die beiden dazugehoehrenden Prozeduren WRITE und READ zur Verfuegung.
06 -- Ein SCHLANGE-Objekt ist ein FIFO-Speicher (first in, first out).
07 ------------------------------------------------------------------------
08    package AST renames ada.streams; -- Eine Abkuerzung
09    type SCHLANGE is new AST.root_stream_type with private;
10    procedure WRITE(STROM:    in out SCHLANGE;
11                    ELEMENTE: in     AST.stream_element_array);
12    -- Schreibt die ELEMENTE in den STROM.
13    procedure READ (STROM:    in out SCHLANGE;
14                    ELEMENTE:    out AST.stream_element_array;
15                    LETZTER_I:   out AST.stream_element_offset);
16    -- Holt soviele Stromelemente aus dem STROM, wie in die Reihung ELEMENTE
17    -- passen.
18 private
19    type SCHLANGE is new AST.root_stream_type with record
20       SPEICHER: AST.stream_element_array(1..1_000);
21       LW      : AST.stream_element_offset := 0; -- last written
22       LR      : AST.stream_element_offset := 0; -- last read
23    end record;
24    -- SPEICHER(LW) ist die letzte durch WRITE "gefuellte" Komponente
25    -- SPEICHER(LR) ist die letzte durch READ  "geleerte"  Komponente
26 end EAS_A_01;
```

Weil in diesem Paket **EAS_A_01** an vielen Stellen "Dinge aus dem Paket **ada.streams**" benützt werden, wird in Zeile 08 der Name **AST** als Abkürzung für **ada.streams** vereinbart.

Der Typ **!SCHLANGE** wird (in Zeile 09) vom Typ **!root_stream_type** (im Paket **AST**) abgeleitet. Damit ist **!SCHLANGE** per Definition ein **Stromtyp** und jede Variable dieses Typs ist ein **Strom**.

Der Typ **!root_stream_type** hat zwei **abstrakte** Prozeduren namens **WRITE** und **READ**. Diese müssen durch entsprechende **konkrete** Prozeduren für den Typ **!SCHLANGE** verdeckt werden. Entsprechende konkrete Prozeduren werden in Zeile 10 bis 12 bzw. 13 bis 17 spezifiziert. Die **Rümpfe** dieser Prozeduren stehen natürlich im Rumpf des Paketes **EAS_A_01**.

Möglicherweise hat der Typ **!SCHLANGE** von seinem Vatertyp **!root_stream_type** ein paar Verbundkomponenten geerbt. Aber weil **!root_stream_type** ein limitiert privater Typ ist (siehe dazu die Spezifikation des Paketes **ada.streams** in (ARM 13.13.1)), sind diese Komponenten außerhalb des Paketes **AST** nicht sichtbar.

In Zeile 19 bis 23 wird festgelegt, daß ein Strom des Typs **SCHLANGE** (zusätzlich zu den nicht sichtbaren Komponenten seines Vatertyps **!root_stream_type**) 3 Komponenten namens **SPEICHER**, **LW** und **LR** besitzt. Die **SPEICHER**-Komponente ist eine Reihung, die aus 1_000 Stromelementen (Komponenten vom Untertyp **AST.stream_element**) besteht und Indizes vom Ganzzahluntertyp **AST.stream_element_offset** hat. **LW** und **LR** sind Indexvariablen für die **SPEICHER**-Komponente. **LW** enthält stets den Index, bis zu dem schon Stromelemente in die Reihung **SPEICHER** geschrieben wurden. **LR** enthält stets den Index, bis zu dem schon Stromelemente aus der Reihung **SPEICHER** gelesen wurden. Der Wert 0 in diesen Variablen bedeutet, daß noch **keine** Stromelemente in den **SPEICHER** geschrieben bzw. aus dem **SPEICHER** gelesen wurden.

Der Rumpf des Paketes **EAS_A_01** wird hier nicht wiedergegeben. Es soll aber ausdrücklich vor ihm **gewarnt** werden. Um die Rümpfe der Prozeduren **WRITE** und **READ** möglichst leicht verständlich zu machen, wurden sie sehr einfach und **unsicher** programmiert. Z.B. prüft **WRITE** nicht, ob im **SPEICHER** noch genügend Platz ist (wenn nicht, wird einfach ein **constraint_error** ausgelöst) und **READ** stellt nicht sicher, daß der **LR**-Index immer unterhalb des **LW**-Index bleibt. Nur wenn man einen Strom des Typs **!SCHLANGE** "vorsichtig" benutzt, funktioniert er wie eine richtige Schlange. Wenn der Leser sich mit den einfachen Prozeduren **WRITE** und **READ** vertraut gemacht hat, sollte er sie verbessern und "sicher machen".
o

Beispiel 25.4.2.: Einen Strom des Untertyps **SCHLANGE** benützen:

```
01 with ada.text_io, EAS_A_01;
02 procedure EAS_A_02 is
03 ------------------------------------------------------------------------
04 -- Schreibt Strings unterschiedlicher Laenge in einen Strom (stream) namens
05 -- STROM1, liest sie dann wieder aus dem STROM1 und gibt sie zur aktuellen
06 -- Ausgabe aus. Dies geschieht zweimal: einmal mit string'write/string'read
07 -- und dann noch mal mit string'output/string'input.
08 ------------------------------------------------------------------------
09    STROM1: aliased EAS_A_01.SCHLANGE;
10    T1    : string := "Vom Eise befreit sind Strom und Baeche";
11    T2    : string := "Durch des Fruehlings holden, belebenden Blick;";
12    T3    : string := "Im Tale gruenet Hoffnungsglueck.";
13    S29   : string(1..29); -- Zum Einlesen mit der Prozedur string'read
14 begin
15    ---------------------------------------------------------------
16    -- Mit string'write schreiben und mit string'read lesen:
17    string'write (STROM1'access, T1);
18    string'write (STROM1'access, T2);
19    string'write (STROM1'access, T3);
20
21    for I in 1..4 loop
22       string'read  (STROM1'access, S29);
23       ada.text_io.put_line(item => S29);
24    end loop;
25
26    ada.text_io.new_line;
27    ---------------------------------------------------------------
28    -- Mit string'output schreiben und mit string'input lesen:
29    string'output(STROM1'access, T1);
30    string'output(STROM1'access, T2);
```

```
31    string'output(STROM1'access, T3);
32
33    for I in 1..3 loop
34       ada.text_io.put_line(item => string'input(STROM1'access));
35    end loop;
36
37 end EAS_A_02;
```

Der Typ **!EAS_A_01.SCHLANGE** ist ein **Stromtyp** und somit ist die Variablen **STROM1** (vereinbart in Zeile 09) ein **Strom**. Auf eine vereinbarte Variable wie **STROM1** darf nur dann ein Zeiger zeigen, wenn die Variable als **aliased** verinbart wurde. Das geschieht in Zeile 09, weil der Zeiger **STROM1'acces** später gebraucht wird.

Die Strings **T1**, **T2** und **T3** (vereinbart in Zeile 10 bis 12) sollen in den **STROM1** geschrieben und dann von dort wieder gelesen werden. Wie man leicht nachzählen kann, bestehen die drei Strings zusammen aus 116 Zeichen und 116 ist gleich 4 mal **29**. Die String-Variablen **S29** (vereinbart in Zeile 13) besteht aus 29 Komponenten.

Mit den drei Aufrufen der Prozedur **string'write** in Zeile 17 bis 19 werden die Strings **T1** bis **T3** in den **STROM1** geschrieben, aber **ohne** ihre Indexgrenzen (d.h. ohne ihre Längen). Anschließend stehen im **STROM1** einfach 116 Zeichen hintereinander.

Die Schleife in Zeile 21 bis 24 liest **viermal** mit der Prozedur **string'read** je 29 Zeichen aus dem **STROM1** in die Variable **S29** und gibt sie zur aktuellen Ausgabe aus.

Mit den drei Aufrufen der Prozedur **string'output** in Zeile 29 bis 31 werden die Strings **T1** bis **T3** noch einmal in den **STROM1** geschrieben, diesmal aber **mit** ihren Indexgrenzen. Anschließend stehen im **STROM1** 3 Strings bekannter Länge.

Die Schleife in Zeile 33 bis 35 liest **dreimal** mit der Funktion **string'input** einen String aus dem **STROM1** und gibt ihn zur aktuellen Ausgabe aus.

Die Ausgabe des Programms **EAS_A_02** sieht wie folgt aus:

```
01 Vom Eise befreit sind Strom u
02 nd BaecheDurch des Fruehlings
03  holden, belebenden Blick;Im
04 Tale gruenet Hoffnungsglueck.
05
06 Vom Eise befreit sind Strom und Baeche
07 Durch des Fruehlings holden, belebenden Blick;
08 Im Tale gruenet Hoffnungsglueck.
```

Mit diesem Beispiel sollte die Verwendung eines **Stroms** und insbesondere der Unterschied zwischen den Prozeduren **string'write/string'read** einerseits und den Unterprogrammen **string'output/string'input** andererseit verdeutlicht werden. **o**

Im vorigen Beispiel wurden in den **STROM1** nur Werte **eines** Typs geschrieben (nämlich **!string**-Werte), d.h. der **STROM1** wurde **typhomogen** benutzt. Im Beispielprogramm **EAS_A_03** wird ein Strom **typheterogen** benutzt. Dazu wird ein dezimaler Festpunkttyp namens **BETRAG** vereinbart und es werden **!string**-Werte und **!BETRAG**-Werte "in wilder Reihenfolge" in den Strom geschrieben.

Das Paket **EAS_A_04** stellt seinen Benutzern einen etikettierten nur-Vater-Typ (abstract type) **!WAGEN**, zwei davon abgeleitete Typen **!PERSONEN_WAGEN** und **!LAST_WAGEN** sowie drei Ausgabeprozeduren zur Verfügung. Zwei der Prozeduren heißen PUT und geben **PERSONEN_WAGEN**-Werte bzw. **LAST_WAGEN**-Werte in Textform zur aktuellen Ausgabe aus. Die dritte Prozedur heißt **PUT_CW** ("CW" wie "class wide") und hat einen Parameter namens **ITEM** vom klassenweiten Untertyp **WAGEN'class**. Diese Prozedur ruft per Wahlaufruf (dispatching call) die zu ihrem **ITEM**-Parameter passende **PUT**-Prozedur auf.

Das Beispielprogramm **EAS_A_05** demonstriert die "typgesicherte" Benützung eines Stroms mit Hilfe der Unterprogramme **'class'output** und **'class'input**. Das Paket **EAS_A_04** wird von der Prozedur **EAS_A_05** mit **with** eingebunden. In **EAS_A_05** werden mit der Prozedur **WAGEN'class'output** ein paar **PERSONEN_WAGEN**- und ein paar **LAST_WAGEN**-Werte "in wilder Reihenfolge" in einen Strom geschrieben, mit der Funktion **WAGEN'class'input** von dort wieder gelesen und dann mit der Prozedur **PUT_CW** zur aktuellen Ausgabe ausgegeben.

Das Beispielprogramm **EAS_A_06** soll die grundsätzlich mit Strömen verbundenen **Gefahr** von Typfehlern deutlich machen. In einen Strom werden vier Werte **geschrieben**. Anschließend werden vier Werte aus dem Strom **gelesen** und in Textform zur aktuellen Ausgabe ausgegeben. Geschrieben und gelesen werden folgende Werte:

In den Strom geschrieben wird:	**Aus dem Strom gelesen wird**:
die Ganzzahl 1	eine Ganzzahl
die Gleitpunktzahl 1.0	eine Gleitpunktzahl
die Gleitpunktzahl 1.0	eine Ganzzahl
die Ganzzahl 1	eine Gleitpunktzahl

Auf dem Bildschirm erscheinen daraufhin die folgenden vier Zeilen:

```
01 1
02  1.00000
03 1065353216
04  0.00000
```

Die ersten beiden Zeilen sehen "wie erwartet" aus (Ganzzahl 1, Gleitpunktzahl 1.0). Die dritte Zeile zeigt: Die Bitkette der **Gleitpunktzahl** 1.0 ist gleich der Bitkette der **Ganzzahl** 1065353216. Die vierte Zeile zeigt umgekehrt, daß die Bitkette der **Ganzzahl** 1 als **Gleitpunktzahl** gelesen den Gleitpunktwert **0.0** darstellt. **o**

Aufgabe 25.4.1.: Machen Sie sich mit den Beispielprogrammen **EAS_A_02**, **EAS_A_03**, **EAS_A_05** und **EAS_A_06** vertraut, lassen Sie sie von einem maschinellen Ada-Ausführer ausführen und schreiben Sie selbst ein paar ähnliche Programme, in denen Ströme des Typs **!SCHLANGE** aus dem Paket **EAS_A_01** benützt werden. **o**

Aufgabe 25.4.2.: Verbessern Sie die Rümpfe der Prozeduren **WRITE** und **READ** im Rumpf des Paketes **EAS_A_01**, so daß die Prozeduren "sicher" funktionieren und im Falle einer falschen Benützung verständliche Fehlermeldungen ausgeben (nicht nur "Irgendwo ist ein **constraint_error** aufgetreten!"). **o**

Die Beispielprogramme der **B**-Serie (**EAS_B_02**, **EAS_B_03**, **EAS_B_05** und **EAS_B_06**) leisten im wesentlichen das Gleiche wie die entsprechenden Programme der **A**-Serie, benützen aber

anstelle eines Stroms vom Typ **!SCHLANGE** eine **Stromdatei** (und anstelle des "selbstgestrickten" Paketes **EAS_A_01** das Standardpaket **ada.streams.stream_io**). Hier das Programm **EAS_B_02** als Beispiel:

Beispiel 25.4.3: Eine **Stromdatei** mit Hilfe des Paketes **ada.streams.stream_io** benützen:

```
01 with ada.text_io, ada.streams.stream_io;
02 procedure EAS_B_02 is
03 ------------------------------------------------------------------------
04 -- Schreibt Strings unterschiedlicher Laenge in eine Stromdatei, liest sie
05 -- von dort wieder ein und gibt sie zur aktuellen Ausgabe aus. Dies
06 -- geschieht zweimal: einmal mit string'write/string'read und dann nochmal
07 -- mit string'output/string'input.
08 ------------------------------------------------------------------------
09    package    TIO renames ada.text_io;
10    package    SIO renames ada.streams.stream_io;
11    AUSWEIS1: SIO.file_type;
12    STROM1_Z: SIO.stream_access;
13    T1      : string := "Vom Eise befreit sind Strom und Baeche";
14    T2      : string := "Durch des Fruehlings holden, belebenden Blick;";
15    T3      : string := "Im Tale gruenet Hoffnungsglueck.";
16    S29     : string(1..29); -- Zum Einlesen mit der Prozedur string'read
17 begin
18    ------------------------------------------------------------------------
19    -- Eine neue, leere Stromdatei wird erzeugt. Dem Zeiger STROM1_Z
20    -- wird ein Wert zugewiesen, so dass er auf diese Datei zeigt.
21    SIO.create(file => AUSWEIS1, name => "EAS_B_02.AUS");
22    STROM1_Z := SIO.stream(file => AUSWEIS1);
23    ------------------------------------------------------------------
24    -- Zwei mal drei Strings werden in die Stromdatei geschrieben:
25    string'write    (STROM1_Z, T1); -- Einmal      mit string'write
26    string'write    (STROM1_Z, T2);
27    string'write    (STROM1_Z, T3);
28    string'output   (STROM1_Z, T1); -- und nochmal mit string'output
29    string'output   (STROM1_Z, T2);
30    string'output   (STROM1_Z, T3);
31    ------------------------------------------------------------------
32    -- Aus der Stromdatei werden mit der Prozedur string'read vier
33    -- Strings der Laenge 29 gelesen und zur akt. Ausgabe ausgegeben:
34    SIO.reset(file => AUSWEIS1, mode => SIO.in_file);
35    for I in 1..4 loop
36       string'read(STROM1_Z, S29);
37       TIO.put_line(item =>  S29);
38    end loop;
39    TIO.new_line;
40    ------------------------------------------------------------------
41    -- Aus der Stromdatei werden mit der Funktion string'input die drei
42    -- uebrigen Strings gelesen und zur aktuellen Ausgabe ausgegeben:
43    while not SIO.end_of_file(file => AUSWEIS1) loop
44       TIO.put_line(item => string'input(STROM1_Z));
45    end loop;
46    ------------------------------------------------------------------
47    SIO.close(file => AUSWEIS1);
48 end EAS_B_02;
```

Die Namen **TIO** und **SIO** werden (in Zeile 10 und 11) als Abkürzungen für die Namen **ada.text_io** und **ada.streams.stream_io** vereinbart.

In den **AUSWEIS1** (vereinbart in Zeile 11) kann man sich die Berechtigung zum Zugreifen auf eine **Stromdatei** eintragen lassen. Der Zeiger **STROM1_Z** (vereinbart in Zeile 12) wird zwar mit dem Wert **null** initialisiert, kann aber (und wird später) auf eine **Stromdatei** zeigen.

Mit dem Befehl **SIO.create** wird (in Zeile 21) eine neue, leere **Stromdatei** namens **"EAS_B_02.-AUS"** erzeugt und im Dateimodus **out_file** geöffnet (weil kein anderer Modus angegeben wurde).

Die Funktion **SIO.stream** macht aus einem **Ausweis**, der zum Zugriff auf eine Datei berechtigt, einen **Zeiger**, der auf eine **Stromdatei** zeigt. Nach der Zuweisung in Zeile 22 zeigt der Zeiger **STROM1_Z** somit auf die **Stromdatei** namens **"EAS_B_02.AUS"**. Zur Zeit darf man in diese Stromdatei nur **schreiben**, weil die darunterliegende Datei **AUSWEIS1** den Dateimodus **out_file** hat.

In Zeile 25 bis 30 werden **sechs** Strings in die Stromdatei **STROM1_Z.all** geschrieben, drei mit **string'write** (**ohne** Indexgrenzen) und drei mit **string'output** (**mit** Indexgrenzen).

Mit dem Befehl **SIO.reset** wird (in Zeile 34) zum Anfang der Datei **AUSWEIS1** zurückgekehrt und der Modus der Datei von **out_file** zu **in_file** verändert. Ab jetzt darf man in die Stromdatei **STROM1_Z.all** nicht mehr **schreiben**, dafür aber daraus **lesen**.

Ansonsten funktioniert dieses Programm **EAS_B_02** ganz ähnlich wie das Programm **EAS_A_02** und produziert die gleichen Bildschirmausgaben (siehe oben Beispiel 25.4.2.). Zusätzlich existiert nach einer Ausführung von **EAS_B_02** eine Datei namens **EAS_B_02.AUS**, in der zweimal die Zeichenketten **T1**, **T2** und **T3** (vereinbart in Zeile 13 bis 15) stehen, einmal **ohne** Indexgrenzen und einmal **mit** Indexgrenzen. **o**

Aufgabe 25.4.3.: Übergeben Sie das Beispielprogramm **EAS_B_02** einem maschinellen Ada-Ausführer ausführen, lassen Sie es ausführen und schauen Sie sich anschließend mit einem Editor die Datei namens **EAS_B_02.AUS** an. Besonders günstig ist ein Editor, der die Datei auch **binär** (eigentlich: hexadezimal) anzeigen kann, z.B. der Editor **TextPad** der Firma Helios Software Solutions. Können Sie die Indexgrenzen der Strings erkennen? **o**

Zentraldokument: Nach Filialdokument A95-25-25

26. Darstellungsbefehle

Normalerweise braucht der Ada-Programmierer sich **nicht** darum zu kümmern, durch welche konkreten Bitketten der Ausführer bestimmte Werte darstellt. Von dieser Regel gibt es zwei Arten von Ausnahmen. Manchmal will der Programmierer sich darüber **informieren**, wie bestimmte Werte dargestellt werden. Z.B. möchte er wissen, wieviele Bits eine bestimmte Variable belegt. In anderen Fällen muß der Programmierer die Darstellung von Werten **beeinflußen** und **festlegen**. Z.B. muß er sicherstellen, daß die Komponenten einer großen Reihung möglichst platzsparend dargestellt und "eng gepackt" werden oder er muß die Lage von Komponenten innerhalb eines Verbundes "bitgenau" festlegen.

Die Darstellungsbefehle, mit denen man sich über die konkrete Darstellung von Größen informieren bzw. diese Darstellung beeinflußen kann, sind im allgemeinen stark **implementierungsabhängig**, d.h. ihre Wirkung kann von Ausführer zu Ausführer verschieden sein. Im folgenden werden zwei konkrete Ada-Ausführer erwähnt:

1. Der **Gnat-Ausführer**. Damit ist der **Gnat-Compiler** Version 3.10 unter Windows 95/NT ("und alles was dazugehört") gemeint. "Gnat" steht für "Gnu Ada Translator" und "Gnu" steht (rekursiv) für "Gnu is not Unix". Der Gnat-Compiler wurde und wird von der **Free Software Foundation** entwickelt.

2. Der **Aonix-Ausführer**. Damit ist der **ObjectAda-Compiler** ("und alles was dazugehört") der Firma Aonix gemeint, und zwar die Version 7.1, die ebenfalls unter Windows 95/NT läuft.

In den folgenden Unterabschnitten werden einige **Darstellungsbefehle** anhand von typischen Anwendungsbeispielen erläutert. Eine **vollständige Beschreibung** aller Darstellungsbefehle findet man im (ARM 13).

26.1. Das 'size-Attribut, das Pragma pack und Größenklauseln

Das '**size**-Attribut eines Ganzzahluntertyps gibt an, wieviele Bits der Ausführer **mindestens** benötigt, um die Werte dieses Untertyps darzustellen. Das '**size**-Attribut einer Variablen gibt an, wieviele Bits diese Variable tatsächlich belegt. Mit dem Pragma **pack** kann der Programmierer den Ausführer auffordern, die Werte eines Reihungsuntertyps oder eines Verbunduntertyps so darzustellen, daß sie **möglichst wenig Speicherplatz** belegen. Eine **Größenklausel** (size clause) ist eine **Darstellungsklausel** der Form **for U'size use G;**. Dabei muß **U** ein **Untertyp** und **G** ein **statischer Ganzzahlausdruck** sein. Mit einer Größenklausel weist man den Ausführer an, für die Darstellung eines U-Wertes möglichst genau G Bits zu verwenden. Wenn der Ausführer dazu nicht in der Lage ist, muß er eine entsprechende Fehlermeldung ausgeben.

Beispiel 26.1.1.: Das '**size**-Attribut, das Pragma **pack** und **Größenklauseln**:

```
01 with ada.text_io;
02 procedure DARST_01 is
...   ...
10    type    GANZ0 is       range -2**31..+2**31-1; -- 32 Bit
11    subtype GANZ1 is GANZ0 range -2**15..+2**15-1; -- 16 Bit
```

```
12    subtype GANZ2 is GANZ1 range -2** 7..+2** 7-1; --  8 Bit
13    type    GANZ3 is       range    300..307;      -- 16, 9 oder 3 Bit?
14    type    GANZ4 is       range    300..307;      -- 16, 9 oder 3 Bit?
15    for     GANZ4'size  use 3; -- 3 Bits genuegen!
16    GVAR0: GANZ0 :=   0;
17    GVAR1: GANZ1 :=   1;
18    GVAR2: GANZ2 :=   2;
19    GVAR3: GANZ3 := 303;
20    GVAR4: GANZ4 := 304;
21    type    RT_2U is array(1..10) of GANZ2; -- ungepackt
22    type    RT_2P is array(1..10) of GANZ2; -- gepackt
23    pragma pack(RT_2P);
24    type    RT_3U is array(1..10) of GANZ3; -- ungepackt
25    type    RT_3P is array(1..10) of GANZ3; -- gepackt
26    pragma pack(RT_3P);
27    type    RT_4U is array(1..10) of GANZ4; -- ungepackt
28    type    RT_4P is array(1..10) of GANZ4; -- gepackt
29    pragma pack(RT_4P);
...     ...
```

Werte des Ganzzahluntertyps **GANZ4** belegen auf einem PC normalerweise **zwei Byte** (gleich **16 Bit**). Es ist aber relativ leicht möglich, diese Werte mit nur **9 Bit** darzustellen. Noch weniger Platz braucht man, wenn man die Werte des Untertyps um **GANZ4'first** vermindert darstellt und die Zahl **GANZ4'first** nur einmal abspeichert. Bei dieser **verschobenen Darstellung** (biased representation) braucht man pro **GANZ4**-Wert nur noch **3 Bit**. Oder andersherum: Da zum Untertyp **GANZ4** nur **acht** verschiedene Werte gehören, kann man jeden dieser Werte durch eine Bitkette der Länge **drei** kodieren.

Die **Größenklausel** in Zeile 15 weist den Ausführer an, die Werte des Untertyps **GANZ4** mit nur je **3 Bits** darzustellen. Wenn der Ausführer **verschobene Darstellungen** nicht realisieren kann, muß er (bei der Übergabe des Programms, "zur Compilezeit") eine entsprechende Fehlermeldung ausgeben.

Das Beispielprogramm **DARST_01** dient dazu, die Werte verschiedener **'size**-Attribute zur aktuellen Ausgabe auszugeben. Wenn man das Programm vom **Gnat-Ausführer** ausführen läßt, sieht die Ausgabe so aus:

```
01 GANZ0'size:   32
02 GANZ1'size:   16
03 GANZ2'size:    8
04 GANZ3'size:    9
05 GANZ4'size:    3
06 GVAR0'size:   32
07 GVAR1'size:   32
08 GVAR2'size:   32
09 GVAR3'size:   16
10 GVAR4'size:    8
11 RT_2U'size:  320
12 RT_2P'size:   80
13 RT_3U'size:  160
14 RT_3P'size:   96
15 RT_4U'size:   80
16 RT_4P'size:   30
17 DARST_01: Das war's erstmal!
```

Das **'size**-Attribut einer **Variablen** muß nicht mit dem **'size**-Attribut ihres **Untertyps** übereinstimmen. Z.B. hat der Untertyp **GANZ2** ein **'size**-Attribut von 8 (siehe Zeile 03). Trotzdem belegt die **GANZ2**-Variable **GVAR2** volle **32 Bit**, und nicht nur **8 Bits** (siehe Zeile 08).

Das Attribut **GANZ2'size** ist aber nicht bedeutungslos, wie die Zeilen 11 und 12 zeigen sollen. Eine Reihung des Typs **!RT_2U** besteht aus 10 Komponenten des Untertyps **GANZ2**, von denen jede 32

Bit belegt. Eine Reihung des Typs **!RT_2P** besteht ebenfalls aus 10 Komponenten des Untertyps **GANZ2**. Diese Komponenten belegen aber nur je 8 Bits (d.h. je **GANZ2'size** viele Bits). Somit belegt eine **RT_2U**-Reihung **320** Bit (Zeile 11), eine **RT_2P**-Reihung aber nur **80** Bit. Zu der kompakteren 80-Bit-Darstellung wird der Ausführer durch das Pragma **pack** veranlaßt (in Zeile 23 der Prozedur **DARST_01**).

Die Ganzzahluntertypen **GANZ3** und **GANZ4** haben den gleichen Wertebereich (300..307) und unterscheiden sich nur durch die **Darstellung** ihrer Werte. Entsprechend sind auch die Reihungstypen **!RT_3U**, **!RT_3P**, **!RT_4U** und **!RT_4P** "im wesentlichen gleich" und unterscheiden sich nur durch die **Darstellung** ihrer Werte. Eine **RT_3U**-Reihung besteht aus 10 Komponenten á 16 Bit (siehe Zeile 13). Eine **RT_3P**-Reihung belegt (siehe Zeile 14) **12 Byte** (gleich **96 Bit**, 90 Bit für 10 Komponenten á 9 Bit und 6 freie Bits, damit insgesamt eine ganze Zahl von Bytes belegt wird). Eine **RT_4U**-Reihung belegt **80 Bit** (10 Komponenten á 3 Bit, jede Komponente auf 8 Bit verlängert) und eine **RT_4P**-Reihung belegt nur **30 Bit** (siehe Zeile 15 und 16).

Ein Zugriff auf eine **3-Bit**-Komponente einer **RT_4P**-Reihung erfordert mehrere Maschinenbefehle. Ein Zugriff auf eine **8-Bit**-Komponente einer **RT_4U**-Reihung erfordert nur einen Maschinenbefehl. Wenn man Reihungen mit Hilfe von Darstellungsbefehlen kompakt darstellen läßt, **spart** man im allgemeinen **Speicherplatz**, **erhöht** dafür aber die **Zugriffszeiten**.

Der **Aonix-Ausführer** kann keine **verschobenen Darstellungen** (biased representations) realisieren und akzeptiert das Beispielprogramm **DARST_01** nur, wenn man die Größenklausel in Zeile 15 (**for GANZ4'size use 3;**) entfernt. Die Ausgabe des Programms sieht dann so aus:

```
01 GANZ0'size:   32
02 GANZ1'size:   16
03 GANZ2'size:    8
04 GANZ3'size:    9
05 GANZ4'size:    9
06 GVAR0'size:   32
07 GVAR1'size:   16
08 GVAR2'size:    8
09 GVAR3'size:   16
10 GVAR4'size:   16
11 RT_2U'size:   80
12 RT_2P'size:   80
13 RT_3U'size:  160
14 RT_3P'size:  160
15 RT_4U'size:  160
16 RT_4P'size:  160
17 DARST_01: Das war's erstmal!
```

An den Zeilen 12 und 14 dieser Ausgabe kann man auch erkennen, daß der Aonix-Ausführer die Komponenten von Reihungen nicht so "eng packt" wie der Gnat-Compiler. **o**

Aufgabe 26.1.1.: Übergeben Sie das Beispielprogramm **DARST_01** einem maschinellen Ada-Ausführer und lassen sie es ausführen. Akzeptiert der Ausführer das Programm, "so wie es ist", oder erst nach gewissen Anpassungen? Vergleichen Sie die Ausgaben Ihres Ausführers mit den oben wiedergegebenen Ausgaben des **Gnat-Ausführers** und des **Aonix-Ausführers**. **o**

Aufgabe 26.1.2.: Schreiben Sie ein Programm namens **DARST_02**, in dem verschiedene **Gleitpunkttypen**, **dezimale Festpunkttypen** und **dezimale Festpunktvariablen** vereinbart werden. Die Werte der **'size**-Attribute dieser Typen und Variablen sollen zur aktuellen Ausgabe ausgegeben werden (ähnlich wie im Beispielprogramm **DARST_01**). **o**

Aufgabe 26.1.3.: Das Beispielprogramm **DARST_03** realisiert eine sogenannten **Bittabelle**, in der man ca. **20 Millionen Zahlen** sehr kompakt speichern kann. Es handelt sich dabei um eine besonders wirkungsvolle Anwendung des Pragmas **pack**. Schauen Sie sich das Programm an und lassen Sie es von einem maschinellen Ada-Ausführer ausführern. Beim Starten des Programms müssen Sie eventuell ein paar Sekunden warten, bis der Ausführer 20 Millionen **boolean**-Variablen mit dem Wert **false** initialisiert hat. **o**

Viele heute übliche Computer arbeiten intern mit sogenannten **Worten**, die aus zwei (oder vier oder acht etc.) Byte bestehen. Der Computer kann in seinem Speicher besonders schnell auf solche Worte zugreifen, wenn sie an einer durch zwei (bzw. vier oder acht) **teilbaren** Adresse beginnen. Man spricht in solchen Fällen auch von einer **Ausrichtung** (alignment) der Worte **auf zwei Byte** (bzw. auf vier Byte oder acht Byte).

Innerhalb eines **Verbundes** kann es durch die Ausrichtung einzelner Komponenten zu **Lücken** (gaps) kommen. Mit dem Pragma **pack** kann man den Ausführer auffordern, solche Lücken möglichst zu **vermeiden**. Dadurch **spart** man **Speicherplatz**, **verlangsamt** aber den Zugriff auf die nicht mehr ausgerichteten Komponenten.

Beispiel 26.1.2.: Das Pragma **pack** für Verbunduntertypen:

```
01 with ada.text_io;
02 procedure DARST_04 is
...   ...
07    type V_TYP1 is record
08       K1: character;
09       K2: integer;
10       K3: string(1..3);
11       K4: float;
12    end record;
13    type V_TYP2 is record
14       K1: character;
15       K2: integer;
16       K3: string(1..3);
17       K4: float;
18    end record;
19    pragma pack(V_TYP2); -- Auf dieses Pragma kommt es an!
20    V_VAR1: V_TYP1 := ('X', -123, "ABC", 654.321);
21    V_VAR2: V_TYP2 := ('Y', +123, "DEF", 321.654);
```

Der Verbundtyp **!V_TYP2** unterscheidet sich von **!V_TYP1** nur durch das Pragma **pack** in Zeile 19. Für den **Gnat-Ausführer** und den **Aonix-Ausführer** gilt:

Eine **Verbundvariable** wird normalerweise auf **vier** Byte **ausgerichtet** (d.h. der Ausführer läßt sie an einer durch vier teilbaren Adresse beginnen). Die Komponente **K1** vom Untertyp **character** belegt **ein** Byte und wird **nicht** ausgerichtet. Die Komponente **K2** vom Untertyp **integer** belegt **vier** Byte und wird normalerweise auf **vier** Byte **ausgerichtet**. Die Komponente **K3** vom Untertyp **string(1..3)** belegt **drei** Byte und wird **nicht** ausgerichtet. Die Komponente **K4** vom Untertyp **float** belegt **vier** Byte und wird normalerweise auf **vier** Byte **ausgerichtet**.

Eine **V_TYP1**-Variable enthält somit **zwei** Lücken: Eine 3-Byte-Lücke zwischen **K1** und **K2** und eine 1-Byte Lücke zwischen **K3** und **K4**. Die ganze Verbundvariable belegt somit **16 Byte** (gleich 128 Bit). Aufgrund des Pragmas **pack** in Zeile 19 werden **V_TYP2**-Variablen **gepackt** dargestellt. Sie enthalten **keine Lücken** und belegen nur **12 Byte** (gleich 96 Bit). **o**

Aufgabe 26.1.4.: Übergeben Sie das Beispielprogramm **DARST_04** einem maschinellen Ada-Ausführer und lassen sie es ausführen. Akzeptiert der Ausführer das Programm, "so wie es ist", oder erst nach gewissen Anpassungen? Erläutern Sie die Ausgaben des Programms einem Kollegen. **o**

Zusammenfassung 26.1.:

- Das '**size**-Attribut eines **Untertyps** gibt an, wieviele Bits der Ausführer mindestens braucht, um einen Wert dieses Untertyps darzustellen.
- Das '**size**-Attribut einer Variablen gibt an, wieviele Bits die Variable belegt.
- Das '**size**-Attribut einer Variablen muß nicht mit dem '**size**-Attribut ihres Untertyps übereinstimmen.
- Mit einer Darstellungsklausel der Form **for X use Y;** kann man den Ausführer dazu auffordern, Werte des Untertyps **X** mit Y vielen Bits darzustellen. Wenn der Ausführer einer solchen Aufforderung nicht nachkommen kann, lehnt er sie mit einer Fehlermeldung ab.
- Mit dem Pragma **pack** kann man den Ausführer dazu auffordern, die Werte eines Reihungstyps oder einer Verbundtyps möglichst platzsparend darzustellen. Wenn der Ausführer einem Pragma nicht entsprechen kann, braucht er keine Fehlermeldung auszugeben.

26.2. Das Layout von Verbunden bitgenau festlegen

Angenommen, ein altes Magnetband wurde vor vielen Jahren von einem exotischen Rechner mit Datensätzen der folgenden Form beschrieben (innerhalb eines Datensatzes werden die einzelnen Bits von links nach rechts mit 0 beginnend durchnummeriert):

Bits	Inhalt
0 bis 2	Irrelevant
3 bis 7	Eine **Ganzzahl** im Bereich 0..31
8 bis 31	Ein **String**, bestehend aus drei Zeichen zu je acht Bit
32 bis 36	Irrelevant
37 bis 42	Eine **Ganzzahl** im Bereich -30..+30
43 bis 47	Irrelevant

Diese Datensätze sollen jetzt mit einem Ada-Programm und dem Gnat-Ausführer bearbeitet werden. Das folgende Beispiel zeigt, wie man einen entsprechenden Verbundtyp vereinbaren und sein Layout bitgenau festlegen kann.

Beispiel 26.2.1.: Das **Layout** eines Verbundttyps **bitgenau** festlegen:

```
01 with ada.text_io, ada.sequential_io;
02 procedure DARST_05 is
...   ...
09    type V_TYP is record              -- Ein Verbundtyp
10       K1 : natural range 0..31;      --  5 Bit
11       K2 : string(1..3);             -- 24 Bit
12       K3 : integer range -30..+30;   --  6 Bit
13    end record;
14    -- Verbund-Darstellungsklausel (record representation clause),
15    -- die Bits 0 bis 2 und die Bits 32 bis 36 bleiben unbenutzt:
16    for V_TYP use record
17       K1 at 0 range  3.. 7;          --  5 Bit
18       K2 at 0 range  8..31;          -- 24 Bit
19       K3 at 0 range 37..42;          --  6 Bit
```

```
20    end record;
21    -- Groessenattribut festlegen (attribut definition clause)
22    -- for V_TYP'size use 64;
```

In Zeile 09 bis 13 wird ganz normal ein Verbundtyp mit drei Komponenten vereinbart. Die **irrelevanten Abschnitte** der Datensätze werden hier **nicht** erwähnt.

In Zeile 16 bis 20 wird das Layout eines **V_TYP**-Verbundes festgelegt. Beschrieben werden insgesamt **43 Bits** (von Bit 0 bis Bit 42). Der Gnat-Compiler rundet diese Länge auf **48 Bit** auf, damit ein Verbund eine **ganze** Anzahl von Bytes (zu je 8 Bit) belegt. Daraus ergibt sich, daß ein **V_TYP**-Verbund **drei** Lücken enthält: Bit 0 bis 2, Bit 32 bis 36 und Bit 43 bis 47.

In den Zeilen 17 bis 19 kann man jeweils hinter **at** die Nummer einer **Speichereinheit** angeben. Beim Gnat-Ausführer ist eine Speichereinheit ein **8-Bit-Byte**. Das kann man daran erkennen, daß die Konstante **storage_unit** im Paket **system** den Wert **8** hat. In Zeile 18 und 19 könnte man somit auch folgendes schreiben:

```
18a      K2 at 1 range 0..23;
19a      K3 at 4 range 6..11;
```

Falls ein **V_TYP**-Verbund nicht **6 Byte** (48 Bit) lang sein soll, sondern z.B. **8 Byte** (64 Bit), kann man das durch eine **Größenklausel** festlegen, wie sie in Zeile 22 als Kommentar angegeben ist. **o**

Aufgabe 26.2.1.: Schauen Sie sich das Beispielprogramm **DARST_05** an und lassen Sie es von einem maschinellen Ada-Ausführer ausführen. Schauen Sie sich danach die Datei **DARST_05.AUS** an, möglichst mit einem Editor, der Daten auch **binär** (bzw. **hexadezimal**) anzeigen kann. Können Sie das Layout der Datensätze erkennen? **o**

Zusammenfassung 26.2.:

- Das **Layout** eines Verbundtyps kann man **bitgenau** festlegen (siehe obiges Beispiel).
- Mit einer Größenklausel der Form **for U'size use G;** kann man festlegen, daß ein Wert des Untertyps **U** genau **G** viele Bits belegen soll.

26.3. Die Adresse einer Konstanten oder Variablen festlegen

Zur Lösung bestimmter Probleme muß man auf ganz bestimmte Adressen im Hauptspeicher eines Computers zugreifen. In Ada kann man die Adresse einer Konstanten oder Variablen mit Hilfe einer sogenannten **Adreßklausel** festlegen, etwa so:

Beispiel 26.3.1.: Die **Adresse** einer Variablen festlegen:

```
01 with ada.text_io, system.storage_elements;
02 procedure DARST_06 is
...   ...
16    type MODU is mod 2**32;
17    M1:  MODU;
...   ...
22    ADR1: system.storage_elements.integer_address := 16#1_FFF0#;
```

```
...   ...
27    for M1'address use system.storage_elements.to_address(ADR1);
...   ...
```

Die Werte des Typs **!MODU** (vereinbart in Zeile 16) belegen bei vielen Ada-Ausführern genau vier Byte (32 Bit) und MODU-Variablen werden normalerweise auf vier Byte ausgerichtet. Die Variable **M1** ist eine solche 32-Bit-Variable. Normalerweise würde der **Ausführer** die Adresse dieser Variablen festlegen. In diesem Beispiel schreibt der **Programmierer** eine bestimmte Adresse für **M1** vor.

Die Variable **ADR1** (vereinbart in Zeile 22) gehört zum Untertyp **integer_address**, der im Paket **system.storage_elements** vereinbart ist. Die Variable **ADR1** hat den Wert **16#1_FFF0#**. Auf einem PC mit einem 486- oder 586-Prozessor entspreicht dieser Wert einer Adresse, die aus dem **Selektor 1** und dem **Offset FFF0** besteht. Was ein **Selektor** und ein **Offset** ist, wird in der Beschreibung der entsprechenden Prozessoren genau erklärt (fragen Sie Ihren Intel-Vertreter oder lesen Sie ein entsprechendes Buch über X86-Prozessoren). Der Typ **!integer_address** ist ein **Ganzzahltyp** (bei manchen Ausführern ein **signierter** Ganzzahltyp, bei anderen ein **modularer** Ganzzahltyp). Werte dieses Typs kann man also z.B. einlesen und ausgeben (mit einer Instanz der Paketschablon **integer_io** bzw. **modular_io**).

Die Funktion **to_address** (aufgerufen in Zeile 27) wandelt einen Wert des Ganzzahltyps **!integer_address** in einen entsprechenden Wert des Typs **!system.address** um. Einen solchen **address**-Wert muß man in der Darstellungsklausel in Zeile 27 angeben. Die Darstellungsklausel legt fest, daß die Variable **M1** im Speicher bei der Adresse **16#1_FFF0#** anfangen soll.

Wie Adressen konkret aussehen und auf welche Adressen ein Programm zugreifen darf, wird nicht durch die Sprache Ada, sondern vor allem durch die verwendete **Hardware** und das **Betriebssystem** festgelegt.

Aufgabe 26.3.1.: Schauen Sie sich das Beispielprogramm **DARST_06** an und versuchen Sie, es von einem maschinellen Ada-Ausführer ausführen zu lassen (möglichst von einem solchen Ausführer, der einen 486- oder 586-Prozessor verwendet und unter Windows 95/NT läuft). Seien Sie nicht enttäuscht, wenn die Ausführung mit einer Ausnahme **program_error** abgebrochen wird. Versuchen Sie in diesem Fall, die Werte der Variablen **ADR1** bis **ADR4** so zu verändern, daß die Ausnahme nicht mehr auftritt. Verbringen Sie aber nicht zu viel Zeit mit diesen Versuchen. **o**

Zusammenfassung 26.3.:

- Mit einer Adreßklausel der Form **for V'address use A;** kann man den Wert von **A** als Anfangsadresse der Variablen (oder Konstanten) **V** festlegen.
- Dabei muß **A** ein statischer Ausdruck sein und zum Typ **!address** (vereinbart im Paket **system**) gehören.
- Die Funktion **to_address** (vereinbart im Paket **system.storage_elements**) wandelt Werte des Ganzzaltyps **!integer_address** (vereinbart im Paket **system.storage_elements**) in Werte des Typs **!address** (vereinbart im Paket **system**) um.
- Eine nützliche und erfolgreiche Anwendung von Adreßklauseln setzt Vertrautheit mit der verwendeten **Hardware** und dem **Betriebssystem** voraus.

26.4. Die Funktionsschablone unchecked_conversion

Normalerweise schützt das **Typensystem** von Ada den Programmierer davor, z.B. zwei **boolesche Werte** zu addieren oder zwei **Bruchzahlen** mit der Operation **and** zu verknüpfen etc.. Es gibt aber Situationen, in denen der Programmierer diesen Schutz an bestimmten Stellen seines Programm ausschalten oder umgehen möchte. Die Funktionsschablone **ada.unchecked_conversion** ist ein sehr allgemeines und mächtiges Mittel dazu.

Beim Instanziieren der Schablone **ada.unchecked_conversion** muß man **zwei Unterypen** angeben: Einen **Quelluntertyp** (source subtype) und einen **Zieluntertyp** (target subtype). Mit der Instanzfunktion kann man dann Werte des Quelluntertyps in Werte des Zieluntertyps "umwandeln", etwa so:

Beispiel 26.4.1.: Die Funktionsschablone **ada.unchecked_conversion** instanziieren:

```
01 declare
02    subtype STRING4 is string(1..4);
03    function NATURAL_TO_STRING4 is new ada.unchecked_conversion
04       (source => natural, target => STRING4);
05    function STRING4_TO_FLOAT is new ada.unchecked_conversion
06       (source => STRING4, target => float);
07    N1 : natural := 17;
08    F1 : float   := float(N1);                      -- inhaltliche Umwandlung
09    N2 : natural := natural(F1);                    -- inhaltliche Umwandlung
10    S1 : STRING4 := NATURAL_TO_STRING4(N1);         -- formale Umwandlung
11    F2 : float   := STRING4_TO_FLOAT(S1);           -- formale Umwandlung
```

Die Funktion **NATURAL_TO_STRING4** wird (in Zeile 03 bis 04) als Instanz der Schablone **ada.unchecked_conversion** vereinbart. Diese Instanziierung der Schablone funktioniert nur dann fehlerfrei, wenn **natural**-Werte und **STRING4**-Werte intern durch **gleichlange** Bitketten dargestellt werden, d.h. wenn **natural'size** gleich **STRING4'size** ist. Dies ist z.B. beim **Gnat-Ausführer** und beim **Aonix-Ausführer** der Fall.

Die Funktion **STRING4_TO_FLOAT** wird (in Zeile 05 bis 06) ebenfalls als Instanz der Schablone **ada.unchecked_conversion** vereinbart. Die Instanzfunktionen **NATURAL_TO_STRING4** und STRING4_TO_FLOAT haben folgende Spezifikationen:

```
function NATURAL_TO_STRING4(S: natural) return STRING4.
function STRING4_TO_FLOAT  (S: STRING4) return FLOAT;
```

In Zeile 08 findet eine "ganz normale inhaltliche Umwandlung" statt. Der **natural**-Wert **17** wird in den entsprechenden **float**-Wert **17.0** umgewandelt und der Variablen **F1** zugewiesen. Ganz ähnlich wird in Zeile 09 der **float**-Wert 17.0 in den **natural**-Wert 17 umgewandelt und der Variablen **N2** zugewiesen.

In Zeile 10 findet eine ganz andere, "rein formale" Umwandlung statt: Die Bitkette, die in der Variablen **N1** steht, wird **unverändert** der Variablen **S1** zugewiesen. Entsprechend wird in Zeile 11 die gleiche Bitkette der Variablen **F2** zugewiesen. Danach stehen in den Variablen **N1**, **S1** und **F2** zwar gleiche Bitketten, aber völlig verschiedene Werte.

Aufgabe 26.4.1.: Im Beispielprogramm **DARST_07** werden Werte eines Ganzzahltyps **!GANZ** formal in "Bittabellen" eines Reihungstyps **!BIT_TAB** umgewandelt und dann "bitweise" zur aktu-

ellen Ausgabe ausgegeben. Schauen Sie sich das Programm an und lassen Sie es von einem maschinellen Ada-Ausführer ausführen. **o**

Aufgabe 26.4.2.: Schreiben Sie ein Programm namens **BRUCH_15**, welches eine **Gleitpunktzahl** einliest und bitweise wieder ausgibt (ähnlich wie das im Programm **DARST_07** mit **Ganzzahlen** geschieht). **o**

Zentraldokument: Nach Filialdokument A95-26-26

27. Tasks und geschützte Objekte

Alle bisher behandelten Ada-Befehle müssen vom Ausführer **sequentiell** ("einer nach dem anderen") ausgeführt werden. In einigen Fällen erlaubt das ARM dem Ausführer, eine von mehreren **verschiedenen** Reihenfolgen auszuwählen, aber es verlangt (für die bisher behandelten Befehle) **immer**, daß der Ausführer die Befehle in einer **Reihenfolge** ("nacheinander") ausführt.

Beispiel 27.1.: Festgelegte und nicht festgelegte Reihenfolge bei der Ausführung von Befehlen:

```
01 declare
02    function FANNY(N: natural) return natural is
03    ...
04    end FANNY;
05    NAT1: natural := FANNY(2) + FANNY(6);
06    ...
```

Das ARM schreibt vor, daß der Ausführer **zuerst** die Funktionsvereinbarung in Zeile 02 bis 04 und **danach** die Variablenvereinbarung in Zeile 05 abarbeiten **muß**. Dagegen darf der Ausführer die Werte der (Teil-) Ausdrücke **FANNY(2)** und **FANNY(6)** in **beliebiger Reihenfolge** berechnen. Der Ausführer darf die Werte dieser Teilausdrücke aber nicht **gleichzeitig** oder **zeitlich überlappt** berechnen, d.h. er muß die Teilausdrücke nacheinander ("in einer **Reihenfolge**") auswerten. **o**

Wenn alle Befehle eines Programms **sequentiell** ausgeführt werden müssen (nach den Regeln der verwendeten Sprache und Umgebung), bezeichnet man das Programm als ein **sequentielles Programm**. Alle bisher vorgestellten Beispielprogramme sind sequentielle Programme.

Es gibt Probleme, die man mit einem sequentiellen Programm **nicht** lösen kann, z.B. das folgende:

Problem 1: Es sollen Zahlen eingelesen und abgespeichert werden. Die Zahlen sollen von **zwei** (oder mehr) **Benutzern** über **zwei** (oder mehr) **Tastaturen** eingegeben werden. Dabei soll es natürlich möglich sein, daß die Benutzer **unterschiedlich schnell** und mit **wechselnden Geschwindigkeiten** Zahlen eingeben.

Zwei Befehle (oder Befehlsfolgen oder ganze Programme), die nicht **nacheinander** ausgeführt werden müssen, sondern **zeitlich unabhängig voneinander** ausgeführt werden dürfen, bezeichnet man auch als **nebenläufige** Befehle (bzw. Befehlsfolgen bzw. Programme).

Weitere Probleme aus der Praxis, die üblicherweise mit **nebenläufigen Befehlsfolgen** gelöst werden:

Problem 2: Ein Computer soll "im Hintergrund" eine Datei ausdrucken, während man "im Vordergrund" eine andere Datei mit einem Editor bearbeitet.

Problem 3: Man öffnet mit einem Textverarbeitungsprogramm (z.B. mit Word 6.0 von Microsoft) eine Datei, die **viele Seiten** enthält. Die Seitennummern stehen nicht fest im Text, sondern werden (beim Öffnen der Datei und nach bestimmten Änderungen) jeweils **neu berechnet**. Das Berechnen der Seitennummern kann **viele Sekunden** oder sogar Minuten dauern und wird deshalb meist von einer **nebenläufigen Befehlsfolge** durchgeführt. Während diese Befehlsfolge läuft, kann der Benutzer die Datei schon mit allen Befehlen bearbeiten, die **keine** Seitennummern voraussetzen.

Viele Betriebssysteme sind in der Lage, sogenannte **Prozesse** zu verwalten. Ein Prozeß ist eine **Befehlsfolge**, die **nebenläufig** zu anderen Prozessen ausgeführt werden darf. Gleichzeitig ist ein Prozeß häufig eine "Verwaltungseinheit des Betriebssystems", für die bestimmte **Betriebsmittel** reserviert werden können, z.B. Hauptspeicherbereiche, einzelne Dateien und ganze Ein-/Ausgabegeräte. Diese Reservierung von Betriebsmitteln macht die Verwaltung von Prozessen in heute (1998) üblichen Betriebssystemen im allgemeinen relativ **aufwendig** und schwerfällig. Man hat deshalb noch eine zweite Art von nebenläufigen Befehlsfolgen eingeführt, die sogenannten **Kontrollfäden** (threads). Ein Kontrollfaden ist eine Befehlsfolge, die nebenläufig zu anderen Kontrollfäden ausgeführt werden darf. Die Befehle **innerhalb** eines Kontrollfadens müssen in aller Regel **sequentiell** ausgeführt werden. Mehrere **Kontrollfäden** können zum selben **Prozeß** gehören. Betriebsmittel reservieren kann man aber nur für den **Prozeß**, nicht für die einzelnen Kontrollfäden. Dadurch belastet die Verwaltung eines Kontrollfadens einen Rechner im allgemeinen **wesentlich weniger**, als die Verwaltung eines Prozesses. Um ihren Charakter zu verdeutlichen bezeichnet man Kontrollfäden bisweilen auch als "leichte Prozesse".

Wie man **Prozesse** strukturieren und organisieren muß, wird in aller Regel von dem verwendeten **Betriebssystem** festgelegt. Z.B. funktionieren Prozesse unter UNIX ein bißchen anders als Prozesse unter Windows 95, und unter Windows NT gibt es wiederum etwas andere Regeln für Prozesse. Ob und wie man **Kontrollfäden** programmieren kann, wird in aller Regel von der verwendeten **Programmiersprache** festgelegt.

Vor 1980 konnte man nebenläufige Befehlsfolgen praktisch nur als **Prozesse** nach den Regeln des jeweils verwendeten Betriebssystems programmieren. Nach wesentlichen **Änderungen** am Betriebssystem oder bei einem **Wechsel** des Betriebssystems mußte man die Prozesse anpassen bzw. neu programmieren. **Ada83** war die erste verbreitete Programmiersprache, in der man nebenläufige Befehlsfolgen (sogenannte **Tasks**) unabhängig vom Betriebssystem nach den Regeln einer Programmiersprache formulieren konnte. Das **Task-Konzept von Ada83** war ein "mutiger Schritt in die richtige Richtung", es erwies sich in der Praxis aber als "zu schwach" und "lückenhaft". Es wurde deshalb (unter Mitwirkung zahlreicher Praktiker und Theoretiker) überarbeitet und ergänzt, so daß **Ada95** ein deutlich besseres und ausgereifteres **Task-Konzept** enhält.

Ob man die **Tasks** eines Ada-Programms als **Kontrollfäden** implementieren sollte, oder ob (bzw. unter welchen Umständen) es günstiger ist, sie auf die **Prozesse** des verwendeten Betriebssystems abzubilden, ist unter Fachleuten ein interessantes Diskussionsthema. Der Gnat-Compiler Version 3.10 und der Aonix-Compiler Version 7.0 (beide unter Windows 95/NT) realisieren Ada-Tasks als **Kontrollfäden**.

In den folgenden Unterabschnitten soll anhand von Beispielen gezeigt werden, wie man in Ada nebenläufige Befehlsfolgen (**Tasks**) programmieren kann. Es ist nicht ganz einfach, die Nebenläufigkeit von Tasks **sichtbar** und anschaulich zu machen. Eine naheliegende Methode besteht darin, die Tasks ("unabhängig voneinander") Daten zu **zwei** verschiedenen Bildschirmen oder zu zwei verschiedenen Stellen eines Bildschirms ausgeben zu lassen. Leider haben viele PCs nur **einen** Bildschirm und zum Ada-Standard gehört **kein** Befehl, mit dem man den Cursor eines Bildschirms zu einer bestimmten Stelle des Bildschirms bewegen kann. Somit ist es nicht ganz einfach, die Nebenläufigkeit zweier Tasks anschaulich zu machen. Als Notlösung wird in einigen Beispielprogrammen folgender "Trick" angewendet: Mehrere Tasks geben ihren Namen zur aktuellen Ausgabe aus (normalerweise also zum einzigen Bildschirm des Rechners). Die Nebenläufigkeit

("zeitliche Unabhängigkeit") der Task kann der Benutzer dann anhand von **zwei Phänomenen** erkennen:

1. Bei **einer** Ausführung des Beispielprogramms erscheinen die Tasknamen "in wirrer Reihenfolge" auf dem Bildschim.
2. Läßt man das Beispielprogramm **mehrmals** ausfühen, erscheinen die Tasknamen **nicht** immer in der gleichen Reihenfolge.

Diese Phänomene sind allerdings nicht besonders "robust" und hängen von schwer zu kontrollierenden Größen ab (vom verwendeten Ada-Ausführer, insbesondere vom Betriebssystem, der Geschwindigkeit des Prozessors etc.). Sollte die Ausgabe der Beispielprogramme zu **regelmäßig** und **leicht vorhersagbar** sein, kann man die Tasks in den Beispielprogrammen unterschiedlich mit "Rechenarbeit" belasten und versuchen, damit ihre Nebenläufig deutlicher sichtbar zu machen.

27.1. Ganz einfache Tasks

Jedes Ada-Programm wird als eine Task ("**Haupttask**") ausgeführt. Läßt der Programmierer innerhalb des Programms **weitere Tasks** erzeugen, so werden diese **nebenläufig zueinander** und **nebenläufig zur Haupttask** ausgeführt. Das folgende Beispiel soll diese grundlegenden Tatsachen deutlich machen.

Beispiel 27.1.1.: Eine **Haupttask** und zwei weitere **Tasks**:

```
01 with ada.text_io;
02 procedure TASKS_01 is
03 ------------------------------------------------------------------------
04 -- Drei Tasks (ALICE, BERTA und die Haupttask) geben "um die Wette"
05 -- Zeichenketten zur aktuellen Ausgabe aus.
06 ------------------------------------------------------------------------
07    procedure BELASTUNG(WIE_STARK: natural) is
08    -- Dient zum "Belasten" von Tasks mit "Rechenarbeit".
09       INT1: integer := -1;
10    begin
11       for I in natural range 0..WIE_STARK loop
12          INT1 := INT1 * INT1; -- Rechenarbeit
13       end loop;
14    end BELASTUNG;
15    ------------------------------------------------------------------------
16    procedure PUT(ITEM: string) is
17    -- Gibt ITEM zehnmal zur aktuellen Ausgabe aus, jeweils mit einer
18    -- Ziffer zwischen '0' und '9' und einem Blank dahinter:
19       BLANK: constant character := ' ';
20    begin
21       for C2 in character range '0'..'9' loop
22          ada.text_io.put(item => ITEM & C2 & BLANK);
23          BELASTUNG(WIE_STARK => 50_000);
24       end loop;
25    end PUT;
26    ------------------------------------------------------
27    task ALICE;               -- Spezifikation von ALICE
28    task BERTA;               -- Spezifikation von BERTA
29    ------------------------------------------------------
30    task body ALICE is        -- Rumpf von ALICE
31    begin
32       for C1 in character range '0'..'5' loop
33          PUT(ITEM => "Alice" & C1);
```

```
34          BELASTUNG(WIE_STARK => 1_000);
35       end loop;
36    end ALICE;
37    -------------------------------------------------------
38    task body BERTA is        -- Rumpf von Berta
39    begin
40       for C1 in character range '0'..'5' loop
41          PUT(ITEM => "Berta" & C1);
42          BELASTUNG(WIE_STARK => 100);
43       end loop;
44    end BERTA;
45    -------------------------------------------------------
46 begin
47    for C1 in character range '0'..'5' loop         -- Anweisungen
48       PUT(ITEM => "Haupt" & C1);                   -- des Haupt-
49       BELASTUNG(WIE_STARK => 100);                 -- programms.
50    end loop;
51 end TASKS_01;
```

Die Hilfsprozedur **BELASTUNG** (vereinbart in Zeile 07 bis 14) dient dazu, im Rahmen der einzelnen Tasks "belastende Berechnungen" durchzuführen. Man kann eine Task **verlangsamen**, indem man sie stärker mit Berechnungen belastet.

Mit der Prozedur **PUT** (vereinbart in Zeile 16 bis 25) geben die einzelnen Tasks wiederholt ihren Namen zur aktuellen Ausgabe (d.h. zum Bildschirm) aus. Jede Task hängt an ihren **Namen** noch eine Ziffer **C1** (aus dem Bereich '0' bis '5') an. Dahinter hängt die Prozedur **PUT** noch eine weitere Ziffer **C2** (aus dem Bereich '0' bis '9') an. Die beiden Ziffern sollen dem Benutzer am Bildschirm deutlich machen, wie oft der betreffende Taskname schon ausgegeben wurde.

Jede Task besteht aus einer **Spezifikation** und einem **Rumpf**. In der Spezifikation wird der **Name** der Task festgelegt. Außerdem werden dort die **Eingänge** (entries) der Task beschrieben.

Die **Spezifikation** einer Task namens **ALICE** steht in Zeile 27. Diese Task hat **keine** Eingänge. Ihr **Rumpf** wird in Zeile 30 bis 36 vereinbart.

Ganz entsprechend steht in Zeile 28 die **Spezifikation** einer Task namens **BERTA** (ebenfalls **ohne** Eingänge) und in Zeile 38 bis 44 wird ihr **Rumpf** vereinbart.

Nachdem der Ausführer alle im Vereinbarungsteil der Prozedur **TASKS_01** vereinbarten Größen (die beiden Prozeduren **BELASTUNG** und **PUT** und die beiden Tasks **ALICE** und **BERTA**) fertig **erzeugt** hat, beginnt er damit, die folgenden **drei** Befehlsfolgen **nebenläufig zueinander** ("zeitlich unabhängig voneinander") auszuführen:

- die **Haupttask**, d.h. die Anweisungen der Prozedur **TASKS_01** in den Zeile 47 bis 50
- die Task **ALICE** (siehe Zeile 30 bis 36)
- die Task **BERTA** (siehe Zeile 38 bis 44)

Jede dieser drei Tasks gibt (mit Hilfe der Prozedur **PUT**) mehrfach ihren Namen "verziert" mit zwei Ziffern zur aktuellen Ausgabe aus und führt (in der Prozedur **BELASTUNG**) ein paar Berechnungen durch. Die **Ausgabe** des Programms **TASKS_01** kann z.B. so aussehen:

```
Alice00 Berta00 Haupt00 Alice01 Berta01 Alice02 Berta02 Berta03 Alice03 Alice04
Haupt01 Berta04 Haupt02 Berta05 Haupt03 Berta06 Haupt04 Berta07 Haupt05 Berta08
Haupt06 Berta09 Alice05 Haupt07 Haupt08 Alice06 Alice07 Haupt09 Haupt10 Alice08
Haupt11 Alice09 Berta10 Haupt12 Berta11 Alice10 Berta12 Berta13 Alice11 Haupt13
Haupt14 Alice12 Alice13 Haupt15 Berta14 Alice14 Berta15 Alice15 Haupt16 Berta16
Alice16 Berta17 Haupt17 Alice17 Alice18 Berta18 Haupt18 Alice19 Haupt19 Berta19
Haupt20 Alice20 Haupt21 Alice21 Berta20 Haupt22 Berta21 Alice22 Haupt23 Berta22
```

```
Alice23 Haupt24 Berta23 Haupt25 Berta24 Alice24 Haupt26 Berta25 Haupt27 Berta26
Alice25 Haupt28 Alice26 Haupt29 Berta27 Alice27 Berta28 Haupt30 Berta29 Alice28
Haupt31 Berta30 Haupt32 Alice29 Haupt33 Berta31 Berta32 Alice30 Haupt34 Alice31
Haupt35 Berta33 Alice32 Berta34 Alice33 Haupt36 Haupt37 Alice34 Berta35 Haupt38
Alice35 Haupt39 Berta36 Berta37 Alice36 Haupt40 Berta38 Alice37 Haupt41 Haupt42
Berta39 Alice38 Haupt43 Berta40 Alice39 Haupt44 Haupt45 Berta41 Alice40 Alice41
Haupt46 Berta42 Haupt47 Alice42 Berta43 Haupt48 Berta44 Alice43 Haupt49 Alice44
Berta45 Haupt50 Alice45 Alice46 Berta46 Haupt51 Alice47 Haupt52 Berta47 Berta48
Alice48 Haupt53 Berta49 Alice49 Haupt54 Berta50 Alice50 Haupt55 Haupt56 Berta51
Alice51 Haupt57 Berta52 Haupt58 Alice52 Haupt59 Alice53 Berta53 Alice54 Berta54
Alice55 Berta55 Alice56 Berta56 Alice57 Berta57 Berta58 Alice58 Berta59 Alice59
```

Man beachte, daß die Prozeduren **BELASTUNG** und **PUT** nicht zu einer bestimmten Task gehören, sondern von allen drei Tasks aufgerufen werden. Dabei können gleichzeitig mehrere Ausführungen derselben Prozedur aktiv sein. Während die Task **ALICE** mit Hilfe der Prozedur **PUT** die Zeichenketten "ALICE00", "ALICE01", "ALICE02" etc. ausgibt, wird im Rahmen der Task **BERTA** eine **zweite Ausführung** derselben Prozedur gestartet und diese zweite **PUT**-Ausführung gibt die Zeichenketten "BERTA00", "BERTA01", "BERTA02" etc. aus. Gleichzeitig gibt eine **dritte Ausführung** der Prozedur **PUT** im Rahmen der **Haupttask** die Zeichenketten "Haupt00", "Haupt01", "Haupt02" etc. aus.

Ganz ähnlich sind zu bestimmten Zeiten vermutlich auch bis zu **drei Ausführungen** der Prozedur **BELASTUNG** gleichzeitig aktiv, je eine im Rahmen der Task **ALICE**, der Task **BERTA** und der **Haupttask**. Während dieser Zeiten existieren **drei** Exemplare der **integer**-Variablen namens **INT1**, deren Vereinbarung in Zeile 09 steht. **o**

Aufgabe 27.1.1.: Übergeben Sie das Beispielprogramm **TASKS_01** einem maschinellen Ada-Ausführer und lassen sie es **mehrmals** ausführen. Vergleichen Sie die Ausgaben des Programms miteinander und mit der oben wiedergegebenen Ausgabe. Verändern Sie die "Belastung der einzelnen Tasks mit Rechenarbeit", indem Sie in den Aufrufen der Prozedur **BELASTUNG** (in Zeile 34, 42 bzw. 49) den **WIE_STARK**-Parameter verändern. Können Sie dadurch die Ausgabe des Programms gezielt beeinflussen? **o**

Aufgabe 27.1.2.: Führen **Sie** zusammen mit **zwei weiteren Personen** das Beispielprogramm **TASKS_01** "von Hand" aus. Nachdem Sie die Vereinbarungen des Programms gemeinsam abgearbeitet haben, soll jede Person eine der drei Tasks (**ALICE**, **BERTA** und die **Haupttask**) übernehmen und unabhängig von den anderen Personen ausführen. Als **Bildschirm** kann ein Blatt Papier dienen, auf das alle drei Personen Zugriff haben. Zu diesem gemeinsamen Bildschirm geben alle drei Tasks ihre "verzierten Namen" aus. **o**

Aufgabe 27.1.3.: Erweitern Sie das Beispielprogramm **TASKS_01** um eine weitere Task namens **CARLO**, die sich "ganz ähnlich verhalten soll", wie die schon vorhandenen Tasks. **o**

Zusammenfassung 27.1.:

- Jedes Ada-Programm wird im Rahmen einer **Haupttask** ausgeführt.
- Der Programmierer kann **weitere Tasks** vereinbaren, die dann **nebenläufig** zueinander und zur Haupttask ausgeführt werden.
- Ein Unterprogramm kann gleichzeitig **im Rahmen verschiedener Tasks** ausgeführt werden.
- Wenn man eine **Task** vereinbart, muß man eine **Spezifikation** und einen **Rumpf** angeben.

- Eine **Task**, die im Vereinbarungsteil **VT** einer Programmeinheit **PE** vereinbart wurde, wird vom Ausführer **gestartet**, nachdem alle Vereinbarungen in **VT** abgearbeitet wurden und unmittelbar bevor die **Anweisungen** der Einheit **PE** ausgeführt werden.

27.2. Ein Rendezvous an einem Taskeingang

Im vorigen Abschnitt wurden Tasks vorgestellt, die **völlig unabhängig voneinander** ausgeführt werden können. Im Prinzip durfte der Ada-Ausführer die einzelnen Tasks so schnell oder so langsam ausführen, wie es ihm beliebte.

In diesem Abschnitt soll gezeigt werden, wie man die Tasks **ALICE** und **BERTA** an bestimmten Stellen ihrer Ausführungen **synchronisieren** kann: Nach jeweils 10 Ausgaben soll die bis dahin schnellere Task auf die langsamere Task warten. Die **Haupttask** soll weiterhin unabhängig von **ALICE** und **BERTA** (d.h. unsynchronisiert) ausgeführt werden.

Eine Synchronisation zwischen zwei Tasks kann man in Ada mit einem sogenannten **Rendezvous** erreichen. Dazu muß eine der beiden Tasks einen sogenannten **Eingang** (entry) besitzen und **für diesen Eingang** eine **accept**-Anweisung ausführen. Damit erklärt sie sich zu einem Rendezvous an diesem Eingang bereit. Die andere Task muß **den Eingang aufrufen** (ähnlich wie man eine Prozedur aufruft) und drückt damit ihre Bereitschaft zu einem Rendezvous an diesem Eingang aus. Entscheidend wichtig ist: Wenn eine der beiden Tasks ihre Bereitschaft zu einem Rendezvous **früher** bekundet als die andere, muß sie auf die andere Task **warten**. Wenn beide Tasks ihre Bereitschaft bekundet haben, findet das Rendezvous statt und in dem Moment sind die beiden Tasks **synchronisiert**. Nach dem Rendezvous laufen die beiden Tasks wieder nebenläufig zueinander ("zeitlich unabhängig voneinander") und können dabei beliebig stark "unsynchronisiert" werden.

Beispiel 27.2.1.: Zwei Tasks **synchronisieren** sich mehrmals durch ein **Rendezvous**:

```
01 with ada.text_io;
02 procedure TASKS_02 is
03 ------------------------------------------------------------------------
...   ...
...   -- Die Prozdeduren BELASTUNG und PUT wie in TASKS_01
...   ...
27    ------------------------------------------------------
28    task ALICE is               -- Spezifikation von ALICE
29       entry EINGANG1;          -- ALICE hat einen Eingang!
30    end  ALICE;
31    task BERTA;                 -- Spezifikation von BERTA (ohne Eingaenge)
32    ------------------------------------------------------
33    task body ALICE is          -- Rumpf von ALICE
34    begin
35       for C1 in character range '0'..'5' loop
36          PUT(ITEM => "Alice" & C1);
37          accept EINGANG1 do
38             ada.text_io.new_line;
39          end EINGANG1;
40          BELASTUNG(WIE_STARK => 1_000);
41       end loop;
42    end ALICE;
43    ------------------------------------------------------
44    task body BERTA is          -- Rumpf von Berta
45    begin
46       for C1 in character range '0'..'5' loop
47          PUT(ITEM => "Berta" & C1);
```

```
48          ALICE.EINGANG1;
49       end loop;
50       BELASTUNG(WIE_STARK => 100);
51    end BERTA;
52    -------------------------------------------------------
53 begin
54    for C1 in character range '0'..'5' loop          -- Anweisungen
55       PUT(ITEM => "Haupt" & C1);                    -- des Haupt-
56       BELASTUNG(WIE_STARK => 100);                  -- programms
57    end loop;
58 end TASKS_02;
```

Die Prozeduren **BELASTUNG** und **PUT** erfüllen hier die gleiche Aufgabe wie im vorigen Abschnitt (im Beispiel 27.1.1.).

Die Task **ALICE** (spezifiziert in Zeile 28 bis 30) hat jetzt einen **Eingang** (entry) namens **EINGANG1**. Die Task **BERTA** (spezifiziert in Zeile 31) hat **keinen** Eingang (wie im vorigen Beispiel).

Die Task **ALICE** ruft die Prozedur **PUT** auf (in Zeile 36) und gibt damit ihren **Namen** (mit "Verzierungen") zehn Mal zur aktuellen Ausgabe aus. Danach (in Zeile 37 bis 39) führt **ALICE** eine **accept**-Anweisung für ihren **EINGANG1** aus und erklärt sich damit zu einem Rendezvous an diesem Eingang bereit. Alle Anweisungen zwischen **do** und **end EINGANG1** werden **während** des Rendezvous ausgeführt. In diesem einfachen Beispiel steht dort nur ein **new_line**-Befehl (siehe Zeile 38).

Die Task **BERTA** ruft die Prozedur **PUT** auf (in Zeile 47) und gibt damit ihren Namen (mit "Verzierungen") zehnmal zur akutellen Ausgabe aus. Danach ruft sie den **EINGANG1** der Task **ALICE** auf (in Zeile 48). Dieser **Eingangsaufruf** sieht genauso aus wie der **Aufruf** einer (parameterlosen) **Prozedur**.

Falls **ALICE** die **accept**-Anweisung (in Zeile 37) erreicht, bevor **BERTA** den **Eingangsaufruf** (in Zeile 48) ausführt, läßt der Ausführer **ALICE** auf **BERTA** warten. Falls umgekehrt **BERTA** den Eingangsaufruf erreicht, bevor **ALICE** die **accept**-Anweisung ausführt, läßt der Ausführer **BERTA** auf **ALICE** warten. Die Rendezvous-Befehle (d.h. der **new_line**-Befehl in Zeile 38) werden also ausgeführt, nachdem die beiden Tasks **ALICE** und **BERTA** ihren Namen mit der Prozedur **PUT** jeweils zehnmal ausgegeben haben.

Da die **accept**-Anweisung von **ALICE** und der dazu passende **Eingangsaufruf** von **BERTA** sich jeweils im Rumpf einer **Schleife** befinden, findet das Rendezvous zwischen **ALICE** und **BERTA** nicht nur einmal statt, sondern **mehrmals** (insgesamt genau sechsmal). Die **Haupttask**, in deren Rahmen die Anweisungen der Prozedur **TASKS_02** (in Zeile 54 bis 57) ausgeführt werden, läuft unabhängig von **ALICE** und **BERTA** und wird nicht mit ihnen synchronisiert.

Die **Ausgabe** des Programms **TASKS_02** kann z.B. so aussehen:

```
Alice00 Berta00 Haupt00 Berta01 Alice01 Alice02 Alice03 Haupt01 Berta02 Alice04
Haupt02 Berta03 Berta04 Alice05 Haupt03 Alice06 Alice07 Haupt04 Alice08 Haupt05
Berta05 Haupt06 Haupt07 Alice09 Berta06 Haupt08 Haupt09 Berta07 Berta08 Haupt10
Berta09 Haupt11 Haupt12
Alice10 Haupt13 Haupt14 Berta10 Alice11 Haupt15 Berta11 Alice12 Haupt16 Berta12
Berta13 Haupt17 Berta14 Alice13 Haupt18 Haupt19 Berta15 Alice14 Haupt20 Berta16
Alice15 Alice16 Alice17 Berta17 Berta18 Alice18 Berta19 Alice19 Haupt21
Alice20 Haupt22 Berta20 Alice21 Haupt23 Alice22 Haupt24 Berta21 Berta22 Alice23
Alice24 Berta23 Haupt25 Haupt26 Berta24 Alice25 Haupt27 Berta25 Haupt28 Haupt29
Berta26 Berta27 Haupt30 Haupt31 Haupt32 Berta28 Alice26 Haupt33 Berta29 Alice27
Haupt34 Alice28 Haupt35 Alice29
Alice30 Haupt36 Haupt37 Berta30 Alice31 Berta31 Alice32 Berta32 Berta33 Alice33
```

```
Haupt38 Berta34 Alice34 Berta35 Alice35 Haupt39 Berta36 Alice36 Berta37 Berta38
Alice37 Haupt40 Berta39 Haupt41 Haupt42 Alice38 Haupt43 Alice39
Alice40 Haupt44 Berta40 Alice41 Haupt45 Alice42 Berta41 Alice43 Haupt46 Haupt47
Alice44 Haupt48 Alice45 Alice46 Haupt49 Berta42 Berta43 Alice47 Berta44 Alice48
Haupt50 Berta45 Haupt51 Haupt52 Berta46 Haupt53 Alice49 Berta47 Berta48 Haupt54
Berta49 Haupt55 Haupt56 Haupt57 Haupt58
Alice50 Alice51 Haupt59 Berta50 Berta51 Alice52 Alice53 Berta52 Berta53 Alice54
Berta54 Alice55 Berta55 Alice56 Berta56 Berta57 Alice57 Alice58 Alice59 Berta58
Berta59
```

Es ist günstig, die Ausgaben der **Haupttask** zunächst zu "übersehen" und sich auf die Ausgaben von **ALICE** und **BERTA** zu konzentrieren. Dann erkennt man, daß der erste **new_line**-Befehl ausgeführt wurde, nachdem **ALICE** den Text **"Alice09"** und **BERTA** den Text **"Berta09"** ausgegeben hat. Der zweite **new_line**-Befehl wurde nach der Ausgabe von **"Alice19"** und **"Berta19"** ausgeführt und der dritte **new_line**-Befehl nach der Ausgabe von **"Alice29"** und **"Berta29"** etc.. Die **new_line**-Befehle werden jeweils **während** eines Rendezvous ausgeführt. **Zwischen** zwei solchen Synchronisationspunkten laufen **ALICE** und **BERTA** unabhäng voneinander und können sich gegenseitig überholen.

Daß nach jedem Rendezvous zunächst die Task **ALICE** ihren Namen ausgibt ("Alice10", "Alice20", "Alice30" etc.) ist eine "zufällige Angewohnheit des verwendeten Ada-Ausführers" und beruht **nicht** auf einer Vorschrift des ARM.

Allgemein gilt: Vor jedem Eingang **E** einer Task **T0** gibt es eine **Warteschlange**. Wenn mehrere Tasks **T1**, **T2**, ... den Eingang **E** aufrufen, werden sie normalerweise zuerst in diese Warteschlange eingereiht und müssen dort warten, bis die Task **T0** ihnen ein Rendezvous gewährt. Im Unterabschnitt 27.6. wird gezeigt, wie eine rufende Task **vermeiden** kann, in diese Warteschlange eingereiht zu werden.

Aufgabe 27.2.1.: Verändern Sie in der Prozedur **TASKS_02** die **Rendezvous-Befehle** (zwischen **do** und **end EINGANG1** in Zeile 39). Lassen Sie während eines Rendezvous z.B. den Text **"ALICE und BERTA treffen sich gerade!"** ausgeben. **o**

Aufgabe 27.2.2.: Beschreiben Sie möglichst genau, **wann** der Ausführer (bei der Ausführung des Programms **TASKS_02**) mit der Ausführung der Tasks **ALICE** und **BERTA** beginnt. Welche Arbeiten erledigt der Ausführer garantiert vorher? **o**

Zusammenfassung 27.2.:

- Tasks kann man mit Hilfe des **Rendezvous**-Mechanismus **synchronisieren**.
- Damit zwei Tasks **T1** und **T2** ein **Rendezvous** durchführen können, muß eine der Tasks (z.B. **T1**) einen **Eingang** (entry) besitzen.
- Die Task **T1** muß für ihren Eingang eine **accept**-Anweisung ausführen.
- Die Task **T2** muß den Eingang von T1 **aufrufen**.
- Ein **Rendezvous** findet statt, nachdem **T1** die **accept**-Anweisung und **T2** einen entsprechenden **Eingangsaufruf** ausgeführt hat.
- Die Task, die "**früher** zum Rendezvous kommt" als die andere, muß auf die **später** kommende Task **warten**. Darin besteht die Synchronisation.
- Vor jedem **Eingang** einer Task **T1** befindet sich eine **Warteschlange**. Wenn mehrere Tasks denselben Eingang aufrufen, werden sie in die Warteschlange des Eingangs eingereiht und müssen dort warten, bis sie an der Reihe sind und ihr Rendezvous mit **T1** stattfindet.

27.3. Eingänge mit Parametern

Ein **Eingang** einer Task hat Ähnlichkeit mit einer **Prozedur** und kann (genau wie eine Prozedur) **formale Parameter** haben. Jeder formale Parameter muß zu einem bestimmten **Untertyp** gehören und einen **Modus** (**in**, **out** oder **in out**) haben. Wenn man den Eingang (von einer anderen Task aus) **aufruft**, muß man entsprechende **aktuelle Parameter** angeben.

Das Beispielprogramm **TASKS_03** ist eine Variante des Programms **TASKS_02** (siehe oben Beispiel 27.2.1.). Im Programm **TASKS_03** synchronisiert sich die Task **ALICE** nicht nur mit **BERTA**, sondern auch mit der **Haupttask**. Bei jedem Rendezvous zwischen **ALICE** und **BERTA** (bzw. zwischen **ALICE** und der **Haupttask**) wird der Text **"Alice trifft Berta"** (bzw. **"Alice trifft Haupt"**) ausgegeben.

Damit **ALICE** "merken kann", mit welcher Task sie gerade ein Rendezvous hat, wurde ihr **EINGANG1** mit einem Parameter namens **WER_DA** versehen. Über diesen Parameter teilt die den Eingang aufrufende Task (**BERTA** bzw. die **Haupttask**) der Task **ALICE** jeweils ihren **Namen** mit.

Beispiel 27.3.1.: Ein **Eingang** mit einem **Parameter**:

```
01 with ada.text_io;
02 procedure TASKS_03 is
03 ------------------------------------------------------------------------
...   ...
...   -- Die Prozeduren BELASTUNG und PUT wie in TASKS_01
...   ...
29    ---------------------------------------------------------
30    task ALICE is             -- Spezifikation von ALICE
31       entry EINGANG1(WER_DA: string); -- Eingang mit Parameter
32    end  ALICE;
33    task BERTA;               -- Spezifikation von BERTA (ohne Eingaenge)
34    ---------------------------------------------------------
35    task body ALICE is        -- Rumpf von ALICE
36    begin
37       for C1 in character range '0'..'5' loop
38          PUT(ITEM => "Alice" & C1);
39          accept EINGANG1(WER_DA: string) do
40             ada.text_io.new_line;
41             ada.text_io.put_line("Alice trifft " & WER_DA & " ----------");
42          end EINGANG1;
43          BELASTUNG(WIE_STARK => 1_000);
44       end loop;
45    end ALICE;
46    ---------------------------------------------------------
47    task body BERTA is        -- Rumpf von Berta
48    begin
49       for C1 in character range '0'..'2' loop
50          PUT(ITEM => "Berta" & C1);
51          ALICE.EINGANG1(WER_DA => "Berta");
52       end loop;
53       BELASTUNG(WIE_STARK => 100);
54    end BERTA;
55    ---------------------------------------------------------
56 begin
57    for C1 in character range '0'..'2' loop          -- Anweisungen
58       PUT(ITEM => "Haupt" & C1);                    -- des Haupt-
59       ALICE.EINGANG1(WER_DA => "Haupt");            -- programms
60       BELASTUNG(WIE_STARK => 100);
61    end loop;
62 end TASKS_03;
```

ALICE führt **sechsmal** eine **accept**-Anweisung aus (Zeile 39 bis 42). **BERTA** ruft **dreimal** den **EINGANG1** auf (in Zeile 51). Die **Haupttask** ruft ebenfalls **dreimal** den **EINGANG1** auf (in Zeile 59). Somit finden insgesamt **sechs** Rendezvous statt, **drei** zwischen **ALICE** und **BERTA** und **drei** zwischen **ALICE** und der **Haupttask**. Die Ausgabe des Programms **TASKS_03** kann z.B. so aussehen:

```
Alice00 Alice01 Berta00 Alice02 Haupt00 Alice03 Haupt01 Berta01 Berta02 Alice04
Haupt02 Haupt03 Berta03 Alice05 Haupt04 Alice06 Haupt05 Berta04 Alice07 Alice08
Berta05 Alice09 Berta06 Berta07 Haupt06 Haupt07 Haupt08 Berta08 Haupt09 Berta09
Alice trifft Berta ----------
Alice10 Berta10 Berta11 Alice11 Berta12 Alice12 Berta13 Alice13 Alice14 Berta14
Berta15 Alice15 Berta16 Alice16 Berta17 Alice17 Berta18 Alice18 Alice19
Alice trifft Haupt ----------
Alice20 Berta19 Haupt10 Alice21 Haupt11 Alice22 Haupt12 Haupt13 Alice23 Alice24
Alice25 Haupt14 Alice26 Alice27 Haupt15 Haupt16 Alice28 Alice29 Haupt17 Haupt18
Alice trifft Berta ----------
Alice30 Haupt19 Berta20 Berta21 Alice31 Alice32 Berta22 Berta23 Alice33 Alice34
Alice35 Berta24 Berta25 Alice36 Berta26 Berta27 Alice37 Berta28 Alice38 Alice39
Berta29
Alice trifft Haupt ----------
Alice40 Haupt20 Alice41 Haupt21 Alice42 Haupt22 Haupt23 Alice43 Alice44 Haupt24
Alice45 Haupt25 Alice46 Alice47 Alice48 Haupt26 Alice49
Alice trifft Berta ----------
Alice50 Alice51 Haupt27 Alice52 Haupt28 Haupt29 Alice53 Alice54 Alice55 Alice56
Alice57 Alice58 Alice59
Alice trifft Haupt ----------
```

Bei **dieser** Ausführung des Programms **TASKS_03** fand das **erste** Rendezvous zwischen ALICE und **BERTA** statt, und danach traf sich ALICE immer **abwechselnd** mit der Haupttask und mit BERTA. Eine erneute Ausführung des Programms könnte **ganz anders** ablaufen. Z.B. wird durch die Befehle des Programms **nicht** ausgeschlossen, daß die **ersten beiden** Rendezvous zwischen ALICE und der **Haupttask** stattfinden und sich ALICE erst danach mit BERTA trifft, etc.. **o**

Aufgabe 27.3.1.: Versuchen Sie, das Programm **TASKS_03** so abzuändern, daß **BERTA zweimal** ein Rendezvous mit **ALICE** hat, ehe **ALICE** sich zum ersten Mal mit der **Haupttask** trifft. Verändern Sie dazu nur die "Belastung" der Tasks durch Rechenarbeiten innerhalb der Prozedur **BELASTUNG**. **o**

Es kann vorkommen, daß eine Task **T1** einen Aufruf an einem ihrer Eingänge akzeptiert und dann feststellt, daß eigentlich ein **anderer Eingang** für diesen Aufruf zuständig ist. In diesem Fall kann die Task **T1** mit einer **requeue**-Anweisung den Aufruf zu diesem anderen Eingang "**umlenken**" oder "**weiterleiten**". Mit der Ausführung der **requeue**-Anweisung hat die Task **T1** den Aufruf fertig bearbeitet und kann "unabhängig weiterlaufen". Die aufrufende Task bleibt blockiert, bis ihr Aufruf am "neuen Eingang" akzeptiert und bearbeitet wurde. Die Task **T1** kann Aufrufe sowohl zu ihren **eigenen** Eingängen umleiten (internal requeue) als auch zu Eingängen von **anderen Tasks** (external requeue).

Das Programm **TASKS_04** ist ein Variante von **TASKS_03** und soll die Wirkung einer **requeue**-Anweisung deutlich machen. Die Task **BERTA** und die **Haupttask** verhalten sich **genauso** wie im vorigen Beispiel. Nur die Task **ALICE** ist in **TASKS_04** ein bißchen anders. Sie hat **zwei** Eingänge, einen öffentlichen **EINGANG1** und einen privaten **EINGANG2**. **ALICE** akzeptiert an ihrem **EINGANG1** zunächst Aufrufe von **BERTA** und der **Haupttask**. Den ersten Aufruf der Haupttask lenkt sie aber zum **EINGANG2** um und kümmert sich erstmal nicht darum. Die Aufrufe von **BERTA** bearbeitet **ALICE** wie gewöhnlich und endgültig. Erst nachdem sie alle Aufrufe von **BERTA** akzeptiert und erledigt hat, bearbeitet sie die Aufrufe der **Haupttask**.

Private Eingänge kann eine Task nur selbst in **requeue**-Anweisungen verwenden. Für andere Tasks sind solche Eingänge nicht sichtbar. Mit **requeue**-Anweisungen kann eine Task Aufrufe aber nicht nur zu **privaten**, sondern auch zu öffentlichen **Eingängen** umlenken.

Aufgabe 27.3.2.: Sehen Sie sich das Beispielprogramm **TASKS_04** genau an und lassen Sie es von einem maschinellen Ada-Ausführer ausführen. Versuchen Sie, anhand der Ausgabe des Programms genau zu erklären, wie das Programm im einzelnen ausgeführt wurde. **o**

Zusammenfassung 27.3.:

- Ein **Eingang** einer Task kann **formale Parameter** haben (wie eine Prozedur).
- Jeder formale Parameter hat einen **Namen**, einen **Untertyp** und einen **Modus** (**in**, **out** oder **in out**).
- Beim **Aufrufen** eines Eingangs muß man entsprechende **aktuelle Parameter** angeben.
- Mithilfe solcher Parameter können Tasks während eines Rendezvous **Daten austauschen**.
- Mit einer **requeue**-Anweisung kann eine Task einen Eingangsaufruf, den sie schon akzeptiert hat, zu einem anderen Eingang weiterleiten.

27.4. Tasktypen und Reihungen von Tasks

In den bisher vorgestellten Beispielprogrammen (**TASKS_01** bis **TASKS_03**) wurden die Tasks **ALICE** und **BERTA** jeweils "einzeln und von Hand" vereinbart, obwohl sie "fast gleich" aussehen. Wenn man **mehrere gleiche Tasks** braucht, ist es häufig günstiger, zuerst einen **Tasktyp** und die einzelnen Tasks dann als **Variablen dieses Typs** zu vereinbaren. Mithilfe eines Tasktyps kann man auch **Verbunde mit Taskkomponenten** oder ganze **Reihungen von Tasks** vereinbaren, wie das folgende Beispiel deutlich machen soll.

Ein **Tasktyp** kann **Diskriminanten** haben (wie ein Verbundtyp) und ist grundsätzlich ein **limitierter** Typ. Aus der Limitiertheit folgt, daß man eine Variable eines Tasktyps nicht **ausdrücklich initialisieren** kann und ihr auch keinen Wert **zuweisen** darf.

Beispiel 27.4.1.: Ein **Tasktyp** mit einer **Diskriminanten** und eine **Reihung von Tasks**:

```
01 with ada.text_io;
02 procedure TASKS_05 is
...   ...
...   -- Die Prozeduren BELASTUNG und PUT wie in TASKS_01
...   ...
31    ------------------------------------------------------------------
32    AKTUELLES_ZEICHEN: character := character'pred('A');
33    function NEUES_ZEICHEN return character is
34    -- Liefert beim ersten Aufruf das Zeichen 'A' und bei jedem
```

```
35    -- weiteren Aufruf das jeweils naechste Zeichen ('B', 'C' etc.).
36    -- Achtung: Greift auf die globale Variable AKTUELLES_ZEICHEN zu!
37    begin
38       AKTUELLES_ZEICHEN := character'succ(AKTUELLES_ZEICHEN);
39       return AKTUELLES_ZEICHEN;
40    end NEUES_ZEICHEN;
41    ------------------------------------------------------------------
42    task type AUSGABE(NAME: character := NEUES_ZEICHEN) is
43       entry START;
44    end       AUSGABE;
45    ------------------------------------------------------------------
46    task body AUSGABE is
47    begin
48       accept START;             -- Hier warten bis zum "Startzeichen"
49       for C1 in character range '0'..'5' loop
50          PUT(ITEM => NAME & C1);
51          BELASTUNG(WIE_STARK => 100);
52       end loop;
53    end AUSGABE;
54    ------------------------------------------------------------------
55    type   TASK_TAB is array(natural range <>) of AUSGABE;
56    TAB1 : TASK_TAB(1..3);       -- Eine Reihung von drei AUSGABE-Tasks
57    ------------------------------------------------------------------
58 begin
59    ada.text_io.put_line(item => "TASKS_05: Jetzt geht es los!");
60    for I in natural range TAB1'range loop
61       TAB1(I).START;            -- Startzeichen fuer die Task TAB1(I)
62    end loop;
63 end TASKS_05;
```

In den Zeilen 42 bis 53 wird ein **Tasktyp** namens **AUSGABE** vereinbart. Diese **Typvereinbarung** unterscheidet sich von der **Vereinbarung einer Task** nur durch das eine Wort **type** in Zeile 42. Die Vereinbarung des Tasktyps **AUSGABE** besteht aus der **Spezifikation** in Zeile 42 bis 44 und dem **Rumpf** in Zeile 46 bis 53.

Jede Task des Typs **!AUSGABE** besitzt eine **Diskriminante** namens **NAME** vom Untertyp **character**. Diese Diskriminante wird mit Hilfe der Funktion **NEUES_ZEICHEN** initialisiert. Die **erste** AUSGABE-Task gekommt das Zeichen 'A' als **NAME**, die **zweite** Task das Zeichen 'B', die **drittte** 'C' etc..

Der Typ **!TASK_TAB** (vereinbart in Zeile 55) ist ein **Reihungstyp**, dessen Komponenten **AUSGABE**-Tasks sind. Die Reihung **TAB1** (vereinbart in Zeile 56) besteht aus drei **AUSGABE**-Tasks. Da diese Tasks die ersten (und einzigen) Objekte des Typs **!AUSGABE** sind, werden ihre Diskriminanten von der Funktion **NEUES_ZEICHEN** mit den Werten 'A', 'B' bzw. 'C' initialisiert.

Jede **AUSGABE**-Task wartet zunächst auf ein Rendezvous an ihrem Eingang **START** (Zeile 48). Nach diesem Rendezvous gibt die Task ihre **NAME**-Diskriminante (mehrfach und jeweils mit zwei Ziffern verziert) zur aktuellen Ausgabe aus (Zeile 50).

Die **Haupttask** führt (in Zeile 61) mit jeder Task **TAB1(I)** ein Rendezvous an ihrem Eingang **TAB1(I).START** durch. Dadurch wird sichergestellt, daß der **put_line**-Befehl in Zeile 59 fertig ausgeführt ist, **bevor** die erste AUSGABE-Task ihren **Namen** ausgibt. Die Ausgabe des Programms **TASKS_05** kann z.B. so aussehen:

```
TASKS_05: Jetzt geht es los!
A00 A01 B00 B01 B02 A02 C00 C01 A03 C02 B03 A04 C03 B04 C04 B05 A05 C05 A06 C06
B06 C07 B07 B08 C08 B09 C09 B10 C10 C11 B11 C12 B12 B13 C13 B14 C14 B15 C15 C16
B16 C17 B17 B18 C18 B19 C19 B20 C20 B21 C21 C22 B22 B23 C23 B24 C24 B25 C25 B26
C26 C27 B27 A07 A08 C28 B28 B29 A09 C29 A10 B30 A11 C30 A12 C31 B31 C32 B32 A13
C33 C34 C35 B33 B34 A14 A15 B35 A16 B36 C36 B37 A17 C37 B38 A18 C38 B39 C39 C40
B40 A19 C41 B41 B42 C42 B43 A20 A21 C43 A22 C44 B44 A23 B45 B46 A24 C45 C46 B47
```

```
B48 A25 A26 C47 B49 A27 A28 C48 B50 B51 C49 B52 A29 A30 A31 C50 C51 B53 A32 B54
A33 C52 B55 B56 A34 C53 B57 C54 A35 A36 B58 C55 C56 B59 A37 A38 C57 C58 A39 C59
A40 A41 A42 A43 A44 A45 A46 A47 A48 A49 A50 A51 A52 A53 A54 A55 A56 A57 A58 A59
```

Diese Ausgabe widerspricht nicht der (korrekten) Vermutung, daß die drei Tasks **TAB1(1)**, **TAB1(2)** und **TAB1(3)** nebenläufig zueinander ("zeitlich unabhängig voneinander") ausgeführt werden (ähnlich wie **ALICE**, **BERTA** und die **Haupttask** oben im Beispiel 27.1.1.). **o**

Aufgabe 27.4.1.: Ändern Sie das Programm **TASKS_05** so ab, daß die **Haupttask** eine zweite Meldung ausgibt, **nachdem** die drei anderen Tasks (die in der Reihung **TAB1**) fertig sind, d.h. nachdem sie die Zeichenketten "A59", "B59" und "C59" ausgegeben haben. Beachten Sie dabei, daß der Ausführer **jede Task** (also auch die Haupttask) im Prinzip "**so schnell ausführen darf, wie er mag**". Sie müssen die vier Tasks ausdrücklich **synchronisieren**, wenn die zweite Meldung der Haupttask garantiert **nach** allen anderen Ausgaben ausgegeben werden soll. **o**

Wenn man einen **Zeigertyp** vereinbart, dessen Werte auf Objekte eines **Tasktyps** zeigen, kann man Tasks mit dem Allokator **new** erzeugen, statt sie zu vereinbaren. Der Ausführer beginnt mit der Ausführung einer allokierten Task unmittelbar nachdem er sie erzeugt hat.

Beispiel 27.4.2.: Zeiger auf Tasks vereinbaren und Tasks mit dem Allokator **new** erzeugen:

```
01 with ada.text_io;
02 procedure TASKS_06 is
...
03    task type AUSGABE is ... end AUSGABE;
...
04    task body AUSGABE is ... end AUSGABE;
...
05    type  Z_AUSGABE  is access AUSGABE; -- Zeiger auf AUSGABE-Tasks
06    Z1 :  Z_AUSGABE;
07    Z2 :  Z_AUSGABE;
...
08 begin
...
09    Z1 := new AUSGABE; -- Eine neue AUSGABE-Task wird erzeugt und gestartet
...
10    Z2 := new AUSGABE; -- Eine neue AUSGABE-Task wird erzeugt und gestartet
...
11 end TASKS_06;
```

In Zeile 05 wird ein Zeigertyp **!Z_AUSGABE** vereinbart, dessen (Zeiger-) Werte auf **AUSGABE**-Tasks zeigen. **Z1** und **Z2** sind Variablen dieses Zeigertyps.

In Zeile 09 wird mit dem Allokator **new** eine neue **AUSGABE**-Task erzeugt. Ein Zeigerwert, der auf diese neue Task zeigt, wird der Variablen **Z1** zugewiesen. Die **Ausführung** der neuen Task beginnt unmittelbar nach ihrer Erzeugung. Falls die neue Task **Eingänge** hat (z.B. Eingänge namens **EINGANG1**, **EINGANG2** etc.), kann man diese Eingänge mit zusammengesetzten Namen wie **Z1.all.EINGANG1**, **Z1.all.EINGANG2** etc. bezeichnen. **o**

Aufgabe 27.4.2.: Schreiben Sie ein Programm namens **TASKS_06**, welches eine natürliche Zahl **N** einliest und dann **N** Tasks eines Tasktyps **!AUSGABE** erzeugt und startet. Die Tasks sollen sich etwa so verhalten, wie **AUSGABE**-Tasks im Programm **TASKS_05**, d.h. sie sollen eine Diskriminante vomTyp **character** haben und den Wert dieser Diskriminanten wiederholt zur aktuellen Ausgabe ausgeben. **o**

Zusammenfassung 27.4.:

- **Tasks** kann man auch als Variablen eines **Tasktyps** vereinbaren.
- Die Vereinbarung eines **Tasktyps** unterscheidet sich nur durch das eine Wort **type** von der Vereinbarung einer **einzelnen Task**.
- Ein **Tasktyp** ist ein **limitierter** Typ und kann **Diskriminanten** haben.
- Ein Tasktyp kann der **Zieltyp eines Zeigertyps** sein.
- Mithilfe eines solchen **Zeigertyps**, dessen Werte auf Tasks zeigen, kann man Tasks mit dem Allokator **new** erzeugen, statt sie zu vereinbaren.

27.5. Ein Produzenten/Konsumenten-Problem, Lösung 1

Die bisher vorgestellten Beispiele für Tasks taten nichts weiter als um die Wette mit anderen Tasks ihren Namen auszugeben. Nur mit einiger Phantasie kann man sich vorstellen, daß die Tasks anstelle sinnloser Multiplikationen (innerhalb der Prozedur **BELASTUNG**) irgendwelche **sinnvollen Berechnungen** durchführen könnten. In diesem Abschnitt soll das Beispielprogramm **TASKS_07** erläutert werden, in dem drei etwas nützlichere Tasks miteinander kooperieren.

Viele Probleme, die man in der Praxis mit Hilfe von **nebenläufigen Befehlsfolgen** löst, sind Spezialfälle des allgemeinen **Produzenten-Konsumenten-Problems**. Ein solches Problem liegt vor, wenn ein **Produzent** und ein **Konsument** weitgehend unabhängig voneinander Produkte produzieren bzw. konsumieren. Z.B. produziert ein **Benutzer** mit Hilfe eines Textprogramms **Dokumente** und eine **Druckertask** soll diese Dokumente ausdrucken ("konsumieren").

Selbst wenn der Produzent und der Konsument **im Durchschnitt** ungefähr gleich schnell arbeiten, kann es durch Geschwindigkeitsschwankungen zu **gegenseitigen Behinderungen** kommen. Wenn der **Produzent** "eine schnelle Phase hat", wird er möglicherweise durch den Konsumenten gebremst, der die Produkte nicht schnell genug abnehmen kann. Wenn dagegen der **Konsument** eine schnelle Phase hat, kann der Produzent möglicherweise nicht schnell genug liefern. Deshalb verbindet man den **Produzenten** und **Konsumenten** häufig nicht direkt miteinander, sondern nur indirekt über ein **Lager**. Wenn der Produzent ein Produkt fertig produziert hat, legt er es in das **Lager**. Wenn der Konsument ein Produkt fertig konsumiert hat, holt er sich das nächste Produkt aus dem **Lager**. Wenn das Lager (ganz oder teilweise) **leer** ist, kann der **Produzent** ruhig **schneller** arbeiten als der **Konsument**, und allmählich das Lager **füllen**. Wenn das Lager (ganz oder teilweise) **gefüllt** ist, kann der **Konsument** ruhig schneller arbeiten als der **Produzent**, und allmählich das Lager **leeren**. So hilft das Lager, die negativen Folgen von Geschwindigkeitsschwankungen zu beseitigen oder zumindest zu mildern.

Auch das folgende Beispielprogramm **TASKS_07** ist nur "ein kleines Spielprogramm". Es modelliert aber (mit möglichst einfachen Mitteln) ein ernsthaftes Problem, welches in der Praxis häufig vorkommt.

Beispiel 27.5.1.: Eine **LAGER-Task** als Puffer zwischen einer Produzenten- und einer Konsumententask:

```
01 with ada.text_io;
02 procedure TASKS_07 is
...   -- Anfangskommentar
```

```
10    PRODUKT : string(1..10) := "ABCDEFGHIJ";
11    package NATU_EA is new ada.text_io.integer_io(num => natural);
12    ------------------------------------------------------------------------
13    task PRODUZENT;
14    task KONSUMENT;
15    task LAGER is
16       entry REIN(KISTE: in  character);
17       entry RAUS(KISTE: out character);
18    end LAGER;
19    ------------------------------------------------------------------------
20    task body PRODUZENT is
21       P_ZEIT : array(PRODUKT'range) of duration := --  Produktionszeiten
22       (0.1, 0.2, 0.1, 0.3, 0.2, 0.1, 0.0, 0.5, 0.1, 0.1);
23       I : natural := PRODUKT'first;             -- Index fuer PRODUKT und P_ZEIT
24    begin -- PRODUZENT
25       loop
26          delay P_ZEIT(I);                       -- Produktionszeit verbringen
27          LAGER.REIN(KISTE => PRODUKT(I));       -- Produkt ins Lager
28       exit when PRODUKT(I) = 'J';               -- letztes Produkt?
29          I := I + 1;
30       end loop;
31    end PRODUZENT;
32    ------------------------------------------------------------------------
33    task body KONSUMENT is
34       K_ZEIT : array(PRODUKT'range) of duration := -- Konsumptionszeiten
35       (0.5, 0.4, 0.0, 0.0, 0.0, 0.5, 0.5, 0.8, 0.8, 0.0);
36       I : natural := PRODUKT'first;             -- Index fuer PRODUKT und P_ZEIT
37    begin -- KONSUMENT
38       loop
39          LAGER.RAUS(KISTE => PRODUKT(I));       -- Produkt aus dem Lager
40          delay K_ZEIT(I);                       -- Konsumptionszeit verbringen
41       exit when PRODUKT(I) = 'J';               -- letztes Produkt?
42          I := I + 1;
43       end loop;
44    end KONSUMENT;
45    ------------------------------------------------------------------------
46    task body LAGER is
47       MAX_ANZAHL : constant natural := 3;       -- Kapazitaet des Lagers
48       REGAL      : string(1..MAX_ANZAHL);
49       AKT_ANZAHL : natural range 0..MAX_ANZAHL := 0;
50       REIN_PLATZ : natural range 1..MAX_ANZAHL := 1;
51       RAUS_PLATZ : natural range 1..MAX_ANZAHL := 1;
52    begin  -- LAGER
53       ada.text_io.put(item => "Lagerbestand: ");
54       loop
55          select
56             when AKT_ANZAHL < MAX_ANZAHL =>
57             accept REIN(KISTE: in character) do
58                REGAL(REIN_PLATZ) := KISTE;
59             end REIN;
60             REIN_PLATZ := REIN_PLATZ mod MAX_ANZAHL + 1;
61             AKT_ANZAHL := AKT_ANZAHL + 1;
62             NATU_EA.put(item => AKT_ANZAHL, width => 2);
63          or when AKT_ANZAHL > 0 =>
64             accept RAUS(KISTE: out character) do
65                KISTE := REGAL(RAUS_PLATZ);
66             end RAUS;
67             RAUS_PLATZ := RAUS_PLATZ mod MAX_ANZAHL + 1;
68             AKT_ANZAHL := AKT_ANZAHL - 1;
69             NATU_EA.put(item => AKT_ANZAHL, width => 2);
70          or
71             terminate;
72          end select;
73       end loop;
74    end LAGER;
75    ------------------------------------------------------------------------

76 begin -- TASKS_07
77    null;
78 end TASKS_07;
```

In diesem Beispielprogramm kooperieren drei Tasks namens **PRODUZENT**, **KONSUMENT** und **LAGER** miteinander. Als "Produkte" dienen die zehn Buchstaben 'A' bis 'J' (siehe Reihung **PRODUKT** in Zeile 10). Innerhalb der Task **PRODUZENT** ist jedem Produkt eine bestimmte **Produktionszeit** zugeordnet (in der Reihung **P_ZEIT**, siehe Zeile 21 bis 22. Die Angaben sind in **Sekunden**). Die Task **PRODUZENT** "verbringt diese Zeit" (mit einer **delay**-Anweisung in Zeile 26), bevor sie das Produkt an die **LAGER**-Task übergibt. Symmetrisch dazu ist in der Task **KONSUMENT** jedem Produkt eine **Konsumptionszeit** zugeordnet (in der Reihung **K_ZEIT**, siehe Zeile 34 bis 35). Die Task **KONSUMENT** "verbringt diese Zeit" (mit einer **delay**-Anweisung in Zeile 40), nachdem sie ein Produkt von der **LAGER**-Task übernommen hat. Die unterschiedlichen Produktions- und Konsumptionszeiten haben zur Folge, daß das **Lager** mal voller und mal leerer ist.

Die **LAGER**-Task spielt in diesem Beispiel eine zentrale Rolle. Ihre **Spezifikation** steht in Zeile 15 bis 18, ihr **Rumpf** in Zeile 46 bis 74. Nur die **LAGER**-Task hat **Eingänge**. Der Eingang **REIN** wird von der Task **PRODUZENT** aufgerufen, wenn sie eine Kiste ins Lager "**reintuen**" will. Der Eingang **RAUS** wird von der Task **KONSUMENT** aufgerufen, wenn sie eine Kiste aus dem Lager **rausholen** will.

In der **LAGER**-Task gibt es ein (relativ kleines) **REGAL** (Zeile 48), in welches drei "Produkte" (d.h. Buchstaben) hineinpassen. In der Variablen **AKT_ANZAHL** (Zeile 49) steht jeweils, **wieviele** Produkte sich gerade im **REGAL** des Lagers befinden (0, 1, 2 oder 3).

Die **LAGER**-Task enthält nur **zwei** Anweisungen, nämlich die **put**-Anweisung in Zeile 53 und die **Schleife** in den Zeilen 54 bis 73. Die Schleife enthält nur eine einzige Anweisung, nämlich eine **select**-Anweisung (Zeile 55 bis 72). Diese **select**-Anweisung hat **drei Alternativen**:

1. eine **accept**-Alternative in Zeile 56 bis 62
2. noch eine **accept**-Alternative in Zeile 63 bis 69
3. eine **terminate**-Alternative in Zeile 71.

Eine **select**-Anweisung wird dadurch ausgeführt, daß genau **eine** ihrer Alternativen **gewählt** und **ausgeführt** wird. Welche Alternative das ist, hängt "von den näheren Umständen" ab.

Der Befehl in Zeile 56 (**when AKT_ANZAHL < MAX_ANZAHL**) wird als **Wächter** der nachfolgenden **accept**-Alternative bezeichnet. Die **accept**-Alternative kann nur gewählt werden, wenn **ihr Wächter es erlaubt**. In diesem Beispiel bedeutet das: Wenn die aktuelle Anzahl der Produkte im Lager kleiner ist als die maximal mögliche Zahl, d.h. wenn noch Platz im Lager ist.

Ganz entsprechend kann die zweite **accept**-Alternative (in Zeile 63 bis 69) nur dann gewählt werden, wenn ihr Wächter in Zeile 63 (**when AKT_ANZAHL > 0**) es erlaubt, d.h. wenn sich noch mindestens **ein** Produkt im Lager befindet.

Außerdem kann eine **accept**-Alternative nur dann gewählt und ausgeführt werden, wenn eine andere Task den entsprechenden **Eingang aufruft** (im Beispiel ist das der Eingang **REIN** bzw. **RAUS** der **LAGER**-Task, weil in Zeile 57 **accept REIN** ... und in Zeile 64 **accept RAUS** ... steht).

Die **terminate**-Alternative (in der **select**-Anweisung der **LAGER**-Task) kann nur gewählt werden, wenn die folgende Bedingung erfüllt ist: Alle Tasks, die einen Eingang der **LAGER**-Task aufrufen **könnten**, haben sich schon beendet oder könnten selbst die **terminate**-Alternative einer **select**-Anweisung wählen und sich damit gemeinsam beenden. Damit wird sichergestellt, daß die **LAGER**-

Task sich nur **nach** oder **zusammen mit** den anderen Tasks beendet, die ihre Eingänge aufrufen können.

Solange noch mindestens eine der beiden Tasks **PRODUZENT** und **KONSUMENT** aktiv ist, gilt:
1. Die **LAGER**-Task kann **nicht** die **terminate**-Alternative ihrer **select**-Anweisung wählen, d.h. sie "bleibt in ihrer Schleife" und kann sich noch nicht beenden.
2. Die **select**-Anweisung besteht somit eigentlich nur aus den beiden **accept**-Alternativen (in Zeile 56 bis 62 bzw. 63 bis 69, das **or** in Zeile 63 gehört nicht dazu).

Die **LAGER**-Task, so kann man sich vorstellen, wartet "in ihrer **select**-Anweisung", bis sie eine der beiden **accept**-Alternativen wählen kann, d.h. bis der entsprechende Wächter es erlaubt und eine andere Task den entsprechenden Eingang (**REIN** bzw. **RAUS**) aufruft. Dann wird die gewählte **accept**-Alternative ausgeführt und damit ist dann die gesamte **select**-Anweisung fertig ausgeführt. Falls **beide accept**-Alternativen ausgewählt werden könnten, wählt die **LAGER**-Task "willkürlich" eine der beiden und führt sie aus.

Da die **select**-Anweisung im Rumpf einer **Schleife** steht, wird sie **wiederholt** ausgeführt. Die Schleife (in Zeile 54 bis 73) sieht auf den ersten Blick wie eine **Endlosschleife** aus. Ihre Ausführung wird aber beendet, wenn die sie umgebende **LAGER**-Task sich (über die **termniate**-Alternative ihrer **select**-Anweisung) **beendet**.

Wenn man das Programm **TASKS_07** ausführen läßt, passiert folgendes: Der Ausführer erzeugt zunächst die Reihung **PRODUKT**, das Paket **INTR_EA** und die drei Tasks **PRODUZENT**, **KONSUMENT** und **LAGER**. Dann startet er die drei Tasks und führt sie nebenläufig zueinander ("zeitlich unabhängig voneinander") aus. Die **Haupttask** kann man in diesem Beispiel vernachlässigen, weil sie nur die **null**-Anweisung in Zeile 77 auszuführen hat. Die Task **PRODUZENT** produziert der Reihe nach **zehn** Produkte ('A' bis 'J') und tut sie in das **LAGER** rein. Wenn das **LAGER** voll ist, muß die **PRODUZENT**en-Task warten, bis wieder Platz vorhanden ist. Die Task **KONSUMENT** holt der Reihe nach zehn Produkte aus dem **LAGER** raus und konsumiert sie. Wenn das **LAGER** leer ist, muß die **KONSUMENT**en-Task warten, bis wieder Produkte im **LAGER** sind. Wenn die beiden Tasks **PRODUZENT** und **KONSUMENT** fertig ausgeführt sind, beendet sich auch die **LAGER**-Task (über die **terminate**-Alternative ihrer **select**-Anweisung, siehe Zeile 71). Die **put**-Befehle in Zeile 53, 62 und 69 geben insgesamt folgende Zeichenketten zur aktuellen **Ausgabe** aus:

```
Lagerbestand:  1 0 1 2 1 2 3 2 1 0 1 0 1 0 1 2 3 2 1 0
```

Daß der Lagerbestand ausgerechnet **diese** Folge von Werten durchläuft, wird vor allem durch die **Produktionszeiten** und **Konsumptionszeiten** in den Reihungen **P_ZEIT** (Zeile 21) und **K_ZEIT** (Zeile 34) bestimmt.

Aufgabe 27.5.1.: Verändern Sie im Programm **TASKS_07** die **Produktionszeiten** und **Konsumptionszeiten** (in den Reihungen **P_ZEIT** bzw. **K_ZEIT** in Zeile 21 bzw. 34) so, daß das Lager meistens **voll** (bzw. meistens **leer**) ist. Lassen sie das Programm dann von einem maschinellen Ada-Ausführer ausführen und überprüfen Sie anhand der Ausgabe, ob Sie Ihr Ziel erreicht haben. **o**

Zusammenfassung 27.5.:

- Eine Task kann **mehrere Eingänge** haben

- Mit einer **select**-Anweisung kann sich eine Task gleichzeitig zu **mehreren** Rendezvous (an verschiedenen ihrer Eingänge) bereit erklären.
- Nur **das** Rendezvous, welches als **erstes** möglich wird, wird ausgewählt und findet statt.
- Deshalb steht eine **select**-Anweisung häufig innerhalb einer **Schleife** (damit mehrere Rendezvous nacheinander stattfinden können).
- Mit der **terminate**-Alternative einer **select**-Anweisung kann man erreichen, daß sich eine Task nur **nach** oder **zusammen mit** den Tasks beendet, die ihre Eingänge aufrufen können.

27.6. Verschiedene Formen der select-Anweisung

Die wichtigsten Ada-Befehle zum Kontrollieren von Tasks sind **accept**-Anweisungen, **Eingangsaufrufe** und **select**-Anweisungen. Wenn eine Task **T1** einen Eingang einer Task **T2** aufruft, wird **T1** als **rufende Task** und **T2** als **gerufene Task** bezeichnet. Es ist durchaus möglich, daß eine Task bei **einem** Rendezvous die Rolle der **rufenden Task** spielt, und beim **nächsten** Rendezvous die Rolle der **gerufenen Task**. Eine Task **T2** wird als **akzeptierende Task** bezeichnet, wenn sie eine **accept**-Anweisung (für einen ihrer Eingänge) ausführt.

Die **select**-Anweisung gehört zu den **zusammengesetzten** Anweisungen, d.h. sie besteht (ähnlich wie **if**-, **case**- und **Schleifen**-Anweisungen) unter anderem aus **Anweisungsfolgen**. Insbesondere kann eine **select**-Anweisung entweder eine oder mehrere **accept**-Anweisung(en) enthalten oder einen **Eingangsaufruf**. Eine **select**-Anweisung kann also sowohl in einer **rufenden** Task als auch in einer **akzeptierenden** Task vorkommen. Entsprechend unterscheiden wir hier zunächst einmal **akzeptierende** select-Anweisungen (mit mindestens einer **accept**-Anweisung darin) und **rufende** select-Anweisungen (mit einem **Eingangsaufruf** darin).

Jede **select**-Anweisung besteht aus **Alternativen**. Jede Alternative kann einen **Wächter** haben (z.B. **when X > Y =>**), der im wesentlichen aus einem booleschen **Ausdruck** besteht. Diesen Ausdruck bezeichnet man auch als **Wächterausdruck**. Eine **select**-Anweisung wird folgendermaßen ausgeführt:

1. Der Ausführer stellt fest, welche Alternativen **offen** sind. Eine Alternative ist offen, wenn sie **keinen** Wächter hat oder wenn ihr **Wächterausdruck** den Wert **true** ergibt.
2. Dann wartet der Ausführer, bis er eine der offenen Alternativen **auswählen** kann (Beispiele dazu weiter unten). Falls **mehrere** Alternativen **wählbar sind** (oder gleichzeitig **wählbar werden**) wählt der Ausführer **willkürlich** eine aus.
3. Schließlich führt der Ausführer die ausgewählte Alternative aus.

Man beachte, daß die Ausführung einer **select**-Anweisung also immer nur aus der Ausführung **einer** ihrer Alternativen besteht (ähnlich wie bei einer **case**-Anweisung). Wichtig ist außerdem, daß die **Wächterausdrücke** der Alternativen bei jeder Ausführung einer **select**-Anweisung nur **einmal** ausgewertet (und nicht etwa **wiederholt** getestet) werden.

Von der **select**-Anweisung gibt es mehrere Varianten. Diese Varianten sollen anhand typischer Beispiele vorgestellt werden. Eine vollständige und verbindliche Beschreibung der **select**-Anweisung findet man im (ARM 9.7).

Eine **akzeptierende select-Anweisung** muß mindestens eine **accept**-Anweisung enthalten. Außerdem **kann** sie enthalten:

1. entweder eine **terminate**-Alternative
2. oder eine oder mehrere **delay**-Alternative(n)
3. oder eine **else**-Alternative.

Diese zusätzlichen Alternativen schließen sich gegenseitig aus.

Beispiel 27.6.1.: Eine akzeptierende **select**-Anweisung nur mit **accept**-Alternativen:

Eine Task darf grundsätzlich nur für ihre eigenen Eingänge accept-Anweisungen ausführen. Von der folgenden select-Anweisung wird angenommen, daß sie in einer Task **TS** steht und daß **E1**, **E2** und **E3** Eingänge dieser Task **TS** sind.

```
01 select
02    accept E1(...) do         -- Alternative 1 ohne Waechter
03       ...
04    end E1;
05    ...
06 or
07    when A<B or C=D =>        -- Alternative 2 mit Waechter
08    accept E2(...) do
09       ...
10    end E2;
11    ...
12 or
13    when A>=B =>              -- Alternative 3 mit Waechter
14    accept E3(...) do
15       ...
16    end E2;
17    ...
18 end select;
```

Die Alternative 1 ist **wählbar**, wenn eine andere Task den Eingang **E1** der Task **TS** aufgerufen hat oder aufruft. Für die Alternativen 2 und 3 gilt entsprechendes. Möglicherweise muß die Task **TS** "in dieser **select**-Anweisung" warten, bis eine der Alternativen wählbar wird.

Beispiel 27.6.2.: Eine akzeptierende **select**-Anweisung mit einer **delay**-Alternative:

```
01 select
02    accept E1(...) do         -- Alternative 1 (accept-Alternative)
03       ...
04    end E1;
05    ...
06 or
07    accept E2(...) do         -- Alternative 2 (accept-Alternative)
08       ...
09    end E2;
10    ...
11 or
12    when X > Y =>             -- Alternative 3 (delay-Alternative mit Waechter)
13    delay 2.0 * (A + B);
14    ada.text_io.put(item => "Dann geh ich eben woanders hin!");
15    ada.text_io.new_line;
16 end select;
```

Der Ausdruck hinter **delay** muß zum (gewöhnlichen Festpunkt-) Untertyp **duration** gehören. Er wird ausgewertet, **nachdem** die offenen Alternativen bestimmt worden sind und nur dann, wenn die **delay**-Alternative **offen** ist. Als Beispiel sei angenommen, daß die Auswertung des **delay**-Ausdrucks den Wert **17.25** ergibt. Wenn innerhalb von **17.25** Sekunden keine der **accept**-Alternativen gewählt werden kann (weil kein entsprechender Eingangsaufruf vorliegt oder eintrifft), wird die **delay**-Alternative gewählt und die Anweisungen in Zeile 14 bis 15 werden ausgeführt. Ein negativer Wert des **delay**-Ausdrucks (z.B. -3.5) wird wie der Wert 0.0 behandelt.

Beispiel 27.6.3.: Eine akzeptierende **select**-Anweisung mit mehreren **delay**-Alternativen:

```
01 select
02    accept E1(...) do
03       ...
04    end E1;
05    ...
06 or
07    when ALLEGRO_MOLTO =>
08    delay 2.0;
09 or
10    delay 5.0;
11 end select;
```

Wenn die boolesche Variablen **ALLEGRO_MOLTO** den Wert **true** enthält, wird höchstens **2.0** Sekunden auf einen Aufruf für den Eingang **E1** gewartet. Sonst wird höchstens **5.0** Sekunden gewartet.

Beispiel 27.6.4.: Eine akzeptierende **select**-Anweisung mit einer **else**-Alternativen:

```
01 select
02    accept E1(...) do
03       ...
04    end E1;
05    ...
06 or
07    accept E2(...) do
08       ...
09    end E2;
10    ...
11 or
12    else
13       ada.text_io.put_line(item => "Keiner will was von mir!");
14       ada.text_io.put_line(item => "Schnueff!");
15 end select;
```

Wenn nicht **sofort** ein Aufruf für den Eingang **E1** oder für den Eingang **E2** akzeptiert werden kann, wird die **else**-Alternative gewählt und die Anweisungen in Zeile 13 bis 14 werden ausgeführt.

Die letzten drei Beispiele zeigen, daß eine **akzeptierende select-Anweisung** in bestimmten Fällen **keinen** Eingangsaufruf akzeptiert, sondern statt dessen andere Anweisungen ausführt.

Nach diesen Beispielen für **akzeptierende select-Anweisungen** folgen jetzt Beispiele für **rufende select-Anweisungen**.

Beispiel 27.6.5.: Eine rufende **select**-Anweisung mit einer **else**-Alternativen:

```
01 select
02    LAGER.REIN(...); -- Ein Eingangsaufruf
```

```
03    ada.text_io.put_line(item => "Einlagerung hat sofort geklappt!");
04    ada.text_io.put_line(item => "Gruss Elly");
05 else
06    ada.text_io.put_line(item => "Einlagerung war nicht sofort moeglich");
07    ada.text_io.put_line(item => "Werfe Produkt in den Muell!");
08 end select;
```

Wenn der Eingangsaufruf **LAGER.REIN(...)** nicht **sofort** akzeptiert wird (d.h. wenn nicht **sofort** ein Rendezvous mit der **LAGER**-Task möglich ist) wird die **else**-Alternative gewählt und ausgeführt. Eine **rufende select-Anweisung** mit einer **else**-Alternativen wird auch als **bedingter Eingangsaufruf** bezeichnet.

Beispiel 27.6.6.: Eine rufende **select**-Anweisung mit einer **delay**-Alternativen:

```
01 select
02    LAGER.RAUS(...); -- Ein Eingangsaufruf
03    ada.text_io.put_line(item => "Auslagerung hat innerhalb von ";
04    ada.text_io.put_line(item => "5.0 Sekunden geklappt!");
05 or
06    delay 5.0;
06    ada.text _io.put_line(item => "Lager zu troedelig! Nehme als Produkt");
07    ada.text _io.put_line(item => "einfach ein 'X'!");
08 end select;
```

Wenn der Eingangsaufruf **LAGER.RAUS(...)** nicht innerhalb von **5.0** Sekunden akzeptiert ist, wird er "zurückgezogen". An seiner Stelle wird die **delay**-Alternative gewählt und die Anweisungen in Zeile 06 bis 07 werden ausgeführt.

Die folgenden beiden Beispiele zeigen sogenannte **asynchrone select-Anweisungen**. Jede solche **select**-Anweisung enthält eine **auslösende Anweisung** (triggering statement). Diese auslösende Anweisung kann entweder eine **delay**-Anweisung sein (siehe unten Beispiel 27.6.7.) oder ein **Eingangsaufruf** (siehe Beispiel 27.6.8.).

Beispiel 27.6.7.: Eine **asynchrone select-Anweisung** mit **delay**-Anweisung:

```
01 declare
02    Z: natural := 0;
03 begin
04    select
05       delay 1.0;
06    then abort
07       loop              -- Eine Endlosschleife, die nach
08          Z := Z + 1;    -- 1.0 Sekunden
09       end loop;         -- abgebrochen wird.
10    end select;
11    ada.text_io.put_line("In einer Sekunde von 0 auf " & natural'image(Z));
...
```

Eine **asynchrone** select-Anweisung erkennt man an den reservierten Wörtern **then abort**. Die Anweisungsfolge zwischen **then abort** und **end select** bezeichnet man als den **abbrechbaren Teil** (abortable part) der **select**-Anweisung. Der Ausführer führt diesen abbrechbaren Teil aus, aber höchstens 1.0 Sekunden lang (siehe Zeile 05). Wenn er dann noch nicht fertig ist, bricht er die Ausführung des abbrechbaren Teils ab. In jedem Fall macht er dann hinter **end select** weiter. Die **put_line**-Anweisung in Zeile 11 gibt also aus, "wie weit der Ausführer in 1.0 Sekunden zählen kann".

Aufgabe 27.6.1.: Schauen Sie sich das Beispielprogramm **TASKS_08** genau an und lassen Sie es mehrmals von einem maschinellen Ada-Ausführer ausführen. Wie weit kann Ihr Ausführer in einer Sekunde zählen? **o**

Aufgabe 27.6.2.: Denken Sie sich eine Aufgabe aus, die man mit Hilfe einer **asynchronen select-Anweisung** (wie im vorigen Beispiel, mit einer **delay**-Anweisung darin) lösen kann. Lösen Sie die Aufgabe durch ein Programm namens **TASKS_09**. **o**

Beispiel 27.6.8.: Eine **asynchrone select-Anweisung**, die einen **Eingangsaufruf** als **auslösende Anweisung** besitzt: Die folgende select-Anweisung könnte z.B. in einer Task namens **BERTA** oder in der **Haupttask** (aber nicht in der Task **ALICE**) stehen:

```
01    select
02       ALICE.EINGANG1;
03    then abort
04       loop
05          ada.text_io.put_line(item => "Ich warte auf ein Rendezvous!");
06          delay 0.5;
07       end loop;
08    end select;
```

Wenn **ALICE** den Eingangsaufruf in Zeile 02 **sofort** akzeptiert, wird der **abbrechbare Teil** dieser **select**-Anweisung (die Schleife in Zeile 04 bis 07) **nicht** ausgeführt. Andernfalls (wenn der Eingangsaufruf in die Warteschlange vor dem **EINGANG1** eingeordnet wird) wird der abbrechbare Teil **ausgeführt**.

Wenn der **abbrechbare Teil** fertig ausgeführt ist, **bevor** der **Eingangsaufruf** akzeptiert und ausgeführt ist, wird der Eingangsaufruf **abgebrochen** und somit das Rendezvous mit **ALICE** abgesagt. Wenn umgekehrt der **Eingangsaufruf** akzeptiert und fertig ausgeführt ist, bevor der **abbrechbare Teil** fertig ausgeführt ist, wird die Ausführung des abbrechbaren Teils abgebrochen.

Aufgabe 27.6.3.: Schauen Sie sich das Beispielprogramm **TASKS_10** genau an und lassen Sie es von einem maschinellen Ada-Ausführer ausführen. Können Sie die Ausgabe des Programms genau erklären? **o**

Zusammenfassung 27.6.:

- Eine **select**-Anweisung besteht (ähnlich wie eine **case**-Anweisung) aus **Alternativen**.
- Davon wird jeweils nur **eine** ausgewählt und ausgeführt.
- Ausgewählt wird nur unter den **offenen** Alternativen einer **select**-Anweisung.
- Eine Alternative ist **offen**, wenn sie keinen Wächter hat oder wenn ihr Wächterausdruck (zu Beginn der Ausführung der **select**-Anweisung) den Wert **true** hat.
- Eine **akzeptierende** select-Anweisung enthält eine oder mehrere **accept**-Alternativen.
- Eine **rufende** select-Anweisung enthält einen **Eingangsaufruf** und andere Alternativen.
- Bei einer **asynchronen** select-Anweisung wird ein **abbrechbarer Teil** eine angegebene **Zeit** lang ausgeführt oder solange ausgeführt, wie ein Eingangsaufruf in einer Warteschlange warten muß.

27.7. Ein Produzenten/Konsumenten-Problem, Lösung 2

Nebenläufige Befehlsfolgen sind mit einer sehr allgemeinen und häufig auftretenden **Gefahr** verbunden. Wenn zwei nebenläufige Befehlsfolgen beide Zugriff auf eine **Variable** haben und **gleichzeitig** (oder fast gleichzeitig) versuchen, den Wert dieser Variablen zu **verändern**, können überraschende und **unerwünschte Effekte** ("Fehler") auftreten.

Beispiel 27.7.1.: Unerwünschte Effekte durch nebenläufige Befehlsfolgen:

Angenommen, zwei nebenläufige Befehlsfolgen **NB1** und **NB2** (z.B. zwei UNIX-Prozesse oder zwei Ada-Tasks etc.) haben Zugriff auf eine Ganzzahlvariable **X** und versuchen (fast) gleichzeitig, den Befehl **X := X + 1;** auszuführen. Es sei angenommen, daß **X** vor Ausführung dieser Befehle den Wert **17** hat. Dann können z.B. folgende Einzelaktionen in der angegebenen Reihenfolge ablaufen:

1. Die Befehlsfolge **NB1** liest den Wert der Variablen **X** und "sieht" den Wert **17**.
2. Die Befehlsfolge **NB2** liest den Wert der Variablen **X** und "sieht" ebenfalls den Wert **17**.
3. Die Befehlsfolge **NB1** addiert eine **1** zum gelesenen Wert **17** und schreibt das Ergebnis dieser Addition (den Wert **18**) zurück in die Variable **X**.
4. Die Befehlsfolge **NB2** addiert eine **1** zum gelesenen Wert **17** und schreibt das Ergebnis dieser Addition (den Wert **18**) zurück in die Variable **X**.

Nach diesen Aktionen hat die Variable **X** nicht den erwarteten Wert **19**, sondern den falschen Wert **18**. Die Veränderung von **X** durch die Befehlsfolge **NB1** ist vollständig "verlorengegangen". **o**

Die Gefahr solcher Fehler tritt immer dann auf, wenn zwei nebenläufige Befehlsfolgen gleichzeitig **Schreibzugriff** auf dieselbe Variable haben.

Um diese Gefahr zu vermeiden, wurde im Beispiel 27.5.1. die Variable **REGAL** (zusammen mit ein paar weiteren Hilfsvariablen) innerhalb einer **Task** (der **LAGER**-Task) vereinbart. Dadurch hat nur diese eine Task **direkten Zugriff** auf die Variable. Die anderen Tasks (**PRODUZENT** und **KONSUMENT**) können nur **indirekt** auf das **REGAL** zugreifen, indem sie die Eingänge **REIN** bzw. **RAUS** der **LAGER**-Task aufrufen. So ist grundsätzlich ausgeschlossen, daß zwei nebenläufige Befehlsfolgen gleichzeitig Schreibzugriff auf die Variable **REGAL** haben.

Mit **Tasks**, **Eingängen** und dem **Rendezvous**-Konzept kann man alle bekannte Nebenläufigkeits-probleme auf ziemlich elegante und relativ leicht verständliche Weise lösen. Es gibt aber Situationen, in denen Tasks nicht **effizient** genug sind. Bestimmte Probleme erfordern gar nicht alle Eigenschaften einer Task, sondern können mit **einfacheren Konstrukten** effizienter gelöst werden.

Zur Lösung von solchen "relativ einfachen Nebenläufigkeitsproblemen" hat man in Ada sogenannte **geschützte Objekte** und **geschützte Typen** (protected objects and types) eingeführt. Wenn man Variablen innerhalb eines geschützten Objekts vereinbart, garantiert der Ada-Ausführer einem, daß nie zwei nebenläufige Befehlsfolgen (d.h. zwei Tasks) gleichzeitig **Schreibzugriff** darauf haben werden. Einen ähnlichen Effekt kann man auch mit Hilfe einer **Task** erreichen, aber ein **geschütztes Objekt** ist ein einfacheres und effizienteres Konstrukt als eine Task.

Im folgenden Beispiel wird das gleiche **Produzenten/Konsumenten-Problem** gelöst wie oben mit dem Programm **TASKS_07** (im Beispile 27.5.1.), diesmal aber mit einem **geschützten Objekt** namens **LAGER**, statt mit einer LAGER-**Task**.

Beispiel 27.7.2.: Lösung eines Produzenten/Konsumenten-Problems mit einem **geschützten Objekt**:

```
01 with ada.text_io;
02 procedure TASKS_11 is
...
11    PRODUKT : string(1..10) := "ABCDEFGHIJ";
12    package NATU_EA is new ada.text_io.integer_io(num => natural);
13    ------------------------------------------------------------------------
14    task PRODUZENT;
15    task KONSUMENT;
16    MAX_ANZAHL : constant natural := 3; -- Kapazitaet des LAGERs
17    protected LAGER is    -- Spezifikation des geschuetzten Objekts LAGER
18       entry REIN(KISTE: in  character);
19       entry RAUS(KISTE: out character);
20    private
21       REGAL      : string(1..MAX_ANZAHL);
22       AKT_ANZAHL : natural range 0..MAX_ANZAHL := 0;
23       REIN_PLATZ : natural range 1..MAX_ANZAHL := 1;
24       RAUS_PLATZ : natural range 1..MAX_ANZAHL := 1;
25    end LAGER;
26    ------------------------------------------------------------------------
27    task body PRODUZENT is
28       P_ZEIT : array(PRODUKT'range) of duration := --  Produktionszeiten
29       (0.1, 0.2, 0.1, 0.3, 0.2, 0.1, 0.0, 0.5, 0.1, 0.1);
30       I : natural := PRODUKT'first;       -- Index fuer PRODUKT und P_ZEIT
31    begin -- PRODUZENT
32       loop
33          delay P_ZEIT(I);                 -- Produktionszeit verbringen
34          LAGER.REIN(KISTE => PRODUKT(I)); -- Produkt ins Lager
35       exit when PRODUKT(I) = 'J';         -- letztes Produkt?
36          I := I + 1;
37       end loop;
38    end PRODUZENT;
39    ------------------------------------------------------------------------
40    task body KONSUMENT is
41       K_ZEIT : array(PRODUKT'range) of duration := -- Konsumptionszeiten
42       (0.5, 0.4, 0.0, 0.0, 0.0, 0.5, 0.5, 0.8, 0.8, 0.0);
43       I : natural := PRODUKT'first;       -- Index fuer PRODUKT und K_ZEIT
44    begin -- KONSUMENT
45       loop
46          LAGER.RAUS(KISTE => PRODUKT(I)); -- Produkt aus dem Lager
47          delay K_ZEIT(I);                 -- Konsumptionszeit verbringen
48       exit when PRODUKT(I) = 'J';         -- letztes Produkt?
49          I := I + 1;
50       end loop;
51    end KONSUMENT;
52    ------------------------------------------------------------------------
53    protected body LAGER is  -- Rumpf des geschützten Objekts LAGER
54       entry REIN(KISTE: in character) when AKT_ANZAHL < MAX_ANZAHL is
55       begin
56          REGAL(REIN_PLATZ) := KISTE;
57          REIN_PLATZ        := REIN_PLATZ mod MAX_ANZAHL + 1;
58          AKT_ANZAHL        := AKT_ANZAHL + 1;
59          NATU_EA.put(item =>  AKT_ANZAHL, width => 2);
60       end REIN;
61       ---------------------------------------------------------------------
62       entry RAUS(KISTE: out character) when AKT_ANZAHL > 0 is
63       begin
64          KISTE             := REGAL(RAUS_PLATZ);
65          RAUS_PLATZ        := RAUS_PLATZ mod MAX_ANZAHL + 1;
66          AKT_ANZAHL        := AKT_ANZAHL - 1;
67          NATU_EA.put(item =>  AKT_ANZAHL, width => 2);
68       end RAUS;
69    end LAGER;
70    ------------------------------------------------------------------------
71 begin -- TASKS_11
72    ada.text_io.put(item => "Lagerbestand: ");
73 end TASKS_11;
```

Die beiden Tasks **PRODUZENT** und **KONSUMENT** funktionieren in diesem Beispielprogramm ganz genauso wie im Programm **TASKS_07** (siehe oben Beispiel 27.5.1.). Für diese beiden Tasks macht es also **keinen** Unterschied, ob sie über eine LAGER-**Task** oder über ein entsprechendes **geschütztes Objekt** namens **LAGER** miteinander verbunden sind.

Ein **geschütztes Objekt** hat Ähnlichkeit mit einem **Paket**. Es besteht aus einer **Spezifikation** (siehe Zeile 17 bis 25) und einem **Rumpf** (Zeile 53 bis 69). Die Spezifikation besteht aus einem **öffentlichen Teil** (Zeile 18 bis 19) und einem **privaten Teil** (Zeile 21 bis 24). Die Vereinbarungen im **privaten** Teil sind nur **innerhalb** des **LAGER**-Objekts **sichtbar**, die Vereinbarungen im **öffentlichen** Teil auch **außerhalb**. Von Stellen außerhalb des geschützten Objekts **LAGER** kann man also nur die Eingänge **REIN** und **RAUS** aufrufen, aber nicht direkt auf die privaten Komponenten **REGAL**, **AKT_ANZAHL**, **REIN_PLATZ** und **RAUS_PLATZ** zugreifen.

Im Rumpf des geschützten Objekts muß für jeden **Eingang** des Objekts ein entsprechender **Eingangsrumpf** (entry body) vereinbart werden. Der Rumpf des Eingangs **REIN** steht in Zeile 54 bis 60, der Rumpf des Eingangs **RAUS** in Zeile 62 bis 69.

Die **put**-Befehle in Zeile 72, 59 und 67 geben insgesamt folgende Zeichenketten zur aktuellen **Ausgabe** aus:

```
Lagerbestand:  1 0 1 2 1 2 3 2 1 0 1 0 1 0 1 2 3 2 1 0
```

Diese Ausgabe des Programms **TASKS_11** stimmt also genau mit der Ausgabe des Programms **TASKS_07** (siehe oben Beispiel 27.5.1.) überein. **o**

Allgemein gilt für **geschützte Objekte**:

Ein geschütztes Objekt besteht aus einer **Spezifikation** und einem **Rumpf**. Die Spezifikation besteht aus einem **öffentlichen Teil** und einem **privaten Teil**, zwischen denen das Wort **private** stehen muß (wie z.B. in Zeile 20 der Prozedur TASKS_11). Im **öffentlichen Teil** darf man nur **Operationen** spezifizieren, d.h. **Eingänge**, **Funktionen** und **Prozeduren** (wie z.B. in Zeile 18 bis 19 von TASKS_11). Im **privaten Teil** darf man **Operationen** spezifizieren und **Datenkomponenten** vereinbaren (wie z.B. in Zeile 21 bis 24 von TASKS_11). Die Datenkomponenten eines geschützten Objekts sind also grundsätzlich alle **privat** und nur die (öffentlichen oder privaten) **Operationen des Objekts** können auf diese Datenkomponenten zugreifen.

Im **Rumpf** eines geschützten Objekts muß man für jede **Operation** des Objekts einen passenden **Rumpf** vereinbaren (für jeden **Eingang** einen Eingangsrumpf, für jede **Funktion** einen Funktionsrumpf und für jede **Prozedur** einen Prozedurrumpf). Außerdem darf man dort weitere "interne" **Unterprogramme** vereinbaren. Andere Größen (z.B. weitere Datenkomponenten) darf man im Rumpf des Objekts **nicht** vereinbaren.

Jeder Eingangsrumpf **muß** mit einem **Wächter** versehen werden (wie z.B. in Zeile 54 bzw. 62 der Prozedur TASKS_11). Eine Prozedur **darf** grundsätzlich **nicht** mit einem Wächter versehen werden. Das ist der einzige wesentliche Unterschied zwischen **Eingängen** und **Prozeduren** eines geschützten Objekts.

Zum Vergleich: In einer **Task** sollte es für jeden **Eingang** eine oder mehrere **accept**-Anweisung(en) geben. **Eingangsrümpfe** sind in einer Task **nicht** erlaubt. In einem **geschützten Objekt**

sind keine **accept**-Anweisungen erlaubt. Dafür muß man für jeden Eingang einen (Eingangs-) **Rumpf** angeben. Eine **accept**-Anweisung in einer Task **kann** einen Wächter haben, ein **Eingangsrumpf** in einem geschützten Objekt **muß** einen Wächter haben.

Die **Eingänge** und **Prozeduren** eines geschützten Objekts **G** dürfen die Datenkomponenten von **G** verändern und gelten als **schreibende Operationen** von **G**. Die **Funktionen** von **G** dürfen die Datenkomponenten von **G nicht** verändern und gelten als **lesende Operationen**. Der Ausführer **garantiert**, daß eine **schreibende** Operation nur dann ausgeführt wird, wenn sie **exklusiven Zugriff** auf die Datenkomponenten des geschützten Objekts hat, d.h. wenn zur gleichen Zeit keine andere (lesende oder schreibende) Operation desselben Objekts ausgeführt wird. Wenn zwei Tasks versuchen, **gleichzeitig** auf ein geschütztes Objekt zuzugreifen und mindestens eine von beiden **schreibend** zugreifen will, läßt der Ausführer **eine** von beiden ("die später gekommene") warten, bis die **andere** Task mit ihrem Zugriff fertig ist. Gleichzeitige (oder zeitlich überlappende) Zugriffe durch mehrere **lesende** Operationen werden vom Ausführer zugelassen. Die **Datenkomponenten** eines geschützten Objekts werden also vom Ausführer vor gleichzeitigen **Schreibzugriffen** nebenläufiger Befehlsfolgen (Tasks) geschützt. Das ist die **wesentliche Eigenschaft** von **geschützten Objekten**.

Wenn man mehrere **gleiche** geschützte Objekte braucht, ist es häufig günstiger, zuerst einen **geschützten Typ** (protected type) und dann die einzelnen Objekte als **Variablen dieses Typs** zu vereinbaren. Ein geschützter Typ ist ein **limitierter** Typ und kann (ähnlich wie ein Tasktyp oder ein Verbundtyp) **Diskriminanten** besitzen. Mithilfe eines geschützten Typs kann man auch **Verbunde mit geschützten Komponenten** oder ganze **Reihungen von geschützten Objekten** vereinbaren, wie das folgende Beispiel andeuten soll:

Beispiel 27.7.2.: Einen **geschützten Typ** vereinbaren:

```
01 with ada.text_io;
02 procedure TASKS_12 is
... ...
08 ------------------------------------------------------------------------
09    protected type LAGER_TYP(MAX_ANZAHL: natural) is -- mit Diskriminante
10       entry REIN(KISTE: in character);
11       entry RAUS(KISTE: out character);
12    private
13       REGAL      : string(1..MAX_ANZAHL);
14       AKT_ANZAHL : natural := 0;
15       REIN_PLATZ : natural := 1;
16       RAUS_PLATZ : natural := 1;
17    end LAGER_TYP;
18    ------------------------------------------------------------------------
19    protected body LAGER_TYP is
20       entry REIN(KISTE: in character) when AKT_ANZAHL < MAX_ANZAHL is
21       begin
22          REGAL(REIN_PLATZ) := KISTE;
23          REIN_PLATZ        := REIN_PLATZ mod MAX_ANZAHL + 1;
24          AKT_ANZAHL        := AKT_ANZAHL + 1;
25       end REIN;
26       ------------------------------------------------------------------------
27       entry RAUS(KISTE: out character) when AKT_ANZAHL > 0 is
28       begin
29          KISTE             := REGAL(RAUS_PLATZ);
30          RAUS_PLATZ        := RAUS_PLATZ mod MAX_ANZAHL + 1;
31          AKT_ANZAHL        := AKT_ANZAHL - 1;
32       end RAUS;
33    end LAGER_TYP;
34    ------------------------------------------------------------------------
35    LAGER1 : LAGER_TYP(MAX_ANZAHL =>   3);
```

```
36    LAGER2 : LAGER_TYP(MAX_ANZAHL => 500);
37    subtype LAGER_TYP_25 is LAGER_TYP(MAX_ANZAHL => 25);
38    type    LAGER_TAB    is array(natural range <>) of LAGER_TYP_25;
39    TAB1: LAGER_TAB(1..3);
...    ...
```

In Zeile 09 bis 33 wird ein geschützter Typ **!LAGER_TYP** vereinbart. Dieser Typ hat eine **Diskriminante** namens **MAX_ANZAHL** vom Untertyp **natural** (siehe Zeile 09). Bei jeder Vereinbarung einer **LAGER_TYP**-Variablen muß ein Wert für diese Diskriminante angegeben werden. Z.B. hat das **LAGER1** (vereinbart in Zeile 35) Platz für **3** Kisten, dagegen passen in das **LAGER2** maximal **500** Kisten (Zeile 36). Zum Untertyp **LAGER_TYP_25** gehören nur die **LAGER_TYP**-Objekte, deren Diskriminante den Wert **25** hat. Die Reihung **TAB1** besteht aus drei solchen Objekten. **o**

Eine **Task** ist im Grunde genommen ein "**aktives** Wesen", welches "von sich aus" z.B. Unterprogramme und Eingänge aufrufen kann. Dagegen ist ein **geschütztes Objekt** ein eher "**passives** Gebilde", welches "nur herumliegt", bis eine **Task** eine seiner Operationen aufruft. Für heute übliche Computer ist die Verwaltung einer **Task** in aller Regel deutlich **aufwendiger** und **zeitraubender** als die Verwaltung eines **geschützten Objekts**. Deshalb sollte man Nebenläufigkeitsprobleme (z.B. Produzenten/Konsumenten-Probleme) wenn möglich mit "schlichten geschützten Objekten" lösen und "aufwendige Tasks" nur dann verwenden, wenn es nötig ist.

Zur Synchronisation von nebenläufigen Befehlsfolgen haben die Informatiker C.A.R. Hoare und P. Brinch Hanse schon 1974/75 ein Konstrukt vorgeschlagen, welches man als **Monitor** bezeichnet (welches aber nichts mit einem Bildschirm zu tun hat). Die geschützten Objekte in Ada sind eine Weiterentwicklung dieser Monitore.

Zusammenfassung 27.7.:

- Ein **geschütztes Objekt** besteht aus **Operationen** und aus **Datenkomponenten**.
- Auf die Datenkomponenten haben nur **die Operationen des Objekts** direkten Zugriff.
- Man unterscheidet **lesende Operationen** (Funktionen) und **schreibende Operationen** (Eingänge und Prozeduren).
- Der Ausführer sorgt dafür, daß eine **schreibende Operation** nur ausgeführt wird, wenn sie **exklusiven** Zugriff auf die Datenkomponenten des Objekts hat.
- Mehrere **gleichzeitige Lesezugriffe** auf ein geschütztes Objekt sind erlaubt.
- Jeder **Eingang** eines geschützten Objekts **muß** mit einem **Wächter** versehen werden. Eine **Prozedur** darf grundsätzlich **nicht** mit einem Wächter versehen werden.
- Geschützte Objekte können auch als Variablen eines **geschützten Typs** vereinbart werden.
- Ein **geschützter Typ** ist ein **limitierter** Typ und kann **Diskriminanten** haben. **o**

Zentraldokument: Nach Filialdokument A95-27-27