

Rendezvous mit Ada

Eine echtzeitige Annäherung

Johann Blieberger

8. November 2010

Vorwort

„ ... Also, wat is en Dampfmaschin?
Da stelle mer uns ganz dumm. Und da sage mer so:
En Dampfmaschin, dat is ene große schwarze Raum,
der hat hinten un vorn e Loch. Dat eine Loch, dat is de Feuerung.
Und dat andere Loch, dat krieje mer später.“ ...
„ Und wenn de große schwarze Raum Räder hat,
dann es et en Lokomotiv. Vielleicht aber auch en Lokomobil.“

Bömmel.

Heinrich Spoerl, „Die Feuerzangenbowle“.

Irgendwie spiegelt obiges Zitat meine Situation wider, als ich voll Elan daran ging, Vorbereitungen für die Vorlesung „*Entwurf von Automatisierungssystemen mit Ada*“ in Angriff zu nehmen. Was tut man in einem solchen Fall (außer sein eigenes Wissen zusammenzuraffen, zu ergänzen und zu ordnen)? Man sucht nach geeigneten unterrichtsbegleitenden schriftlichen Unterlagen. Hier nun werden die Parallelen zu obigem Zitat sichtbar:

1. Alle halbwegs brauchbaren Unterlagen wären für die meisten Studenten ein Greuel, da sie in einer mehr oder weniger unverständlichen Sprache verfaßt sind, nämlich Englisch. („*Da stelle mer uns ganz dumm.*“)
2. Die einzig wirklich profunde Quelle für die Programmiersprache Ada ist das *Reference Manual*, das nicht nur obigen Nachteil hat, in englischer Sprache geschrieben zu sein, sondern sich auch noch durch zahlreiche Vorgriffe auszeichnet („*Dat krieje mer später.*“). *Summa summarum*: Der, der Ada verstanden hat, wird auch das Reference Manual verstehen.

Wie aber führt man jemanden an eine so komplexe Programmiersprache wie Ada heran? Im besonderen jemanden, der bereits Programmiererfahrung hat? Am besten, indem man ihm, beginnend mit Bekanntem, sukzessive die Besonderheiten der neuen Sprache nahebringt. Dies kann jedoch nicht geschehen, indem man ihm das Reference Manual (oder eine deutsche Übersetzung) immer und immer wieder lesen läßt, bis er alle Vorwärtsreferenzen aufgelöst und ein Gesamtbild gewonnen hat.

Ein Ziel dieser Unterlagen war also, eine Einführung in die Programmiersprache Ada zu geben, die möglichst ohne Vorgriffe die wesentlichen Konzepte erläutert. Vom Leser wird erwartet, daß er Programmiererfahrung in einer Pascal-ähnlichen Sprache wie etwa MODULA-2 oder Pascal selbst besitzt. Hauptaugenmerk wird auf Ada-Sprachkonstrukte gelegt, die es in anderen Sprachen nicht oder nur zum Teil gibt.

Da Ada entwickelt wurde, um der in den Siebziger- und Achtzigerjahren allgegenwärtigen Software-Krise Herr zu werden, kann man die Sprache als solche eigentlich nicht getrennt von ihrem Anwendungsgebiet betrachten. Daher enthalten diese Unterlagen außer Information über die genormte Programmierungsumgebung und für Ada entworfene graphische Hilfsmittel auch Kapitel über den objekt-orientierten Entwurf von Ada-Programmen und den Entwurf von Echtzeit- und Automatisierungssystemen.

Altenmarkt, Oktober 1991

J. Blieberger

Mit der heuer vorliegenden Version des Ada-Skriptums wurden zum ersten Mal die durch den Revisionsprozeß des ISO-Standards bedingten Änderungen (Ada95) integriert. Die Änderungen in der Sprache stellen sich in den meisten Fällen als echte Erweiterungen dar. Daher ist auch der Umfang des Skriptums um einige Seiten gewachsen. Dies bezieht sich vor allem auf Teil I. Generell machte der Revisionsprozeß eine Änderung der Struktur des Skriptums erforderlich.

Oberwaltersdorf, Oktober 1995

J. Blieberger

Inhaltsverzeichnis

Ada – Ein Überblick	1
1 Lexikalische Einheiten und Namen	5
1.1 Der Zeichensatz	5
1.2 Lexikalische Einheiten und Namen	6
2 Typen, Klassen und Variablen	9
2.1 Skalare Typen	9
2.1.1 Ganze Zahlen	9
2.1.2 Aufzählungen	10
2.1.3 Reelle Zahlen	11
2.1.4 Universelle Typen	12
2.2 Zusammengesetzte Typen	12
2.2.1 Felder	12
2.2.2 Records	14
2.2.3 Aggregate	15
2.3 Subtypen	17
2.4 Pointer	17
2.4.1 Pointer auf Objekte	18
2.4.2 Anonyme Pointer	19
2.4.3 Pointer auf Unterprogramme	19
2.4.4 Alias-Objekte	19
2.4.5 Eine Variabilität konstanter Pointer	20
2.5 Private Typen	20
2.6 Limitierte Typen	21
2.7 Getaggte Typen und Typerweiterungen	21
2.7.1 Typerweiterungen	23
2.7.2 Erweiterungsaggregate	24
2.7.3 Abstrakte Typen	24
2.7.4 Access-Diskriminanten	24
3 Ausdrücke	25
3.1 Operatoren	25
3.2 Auswertung von Ausdrücken	26
3.3 Kurzauswertungen	26
3.4 Logische Operatoren	26
3.5 Vergleichsoperatoren	26
3.6 Elementtests	27
3.7 Operationen für ganze Zahlen	27
3.8 Konkatination	27
3.9 Potenzieren	28
4 Anweisungen	29

4.1	Die Null-Anweisung	29
4.2	Die Zuweisung	29
4.3	Die If-Anweisung	29
4.4	Die Case-Anweisung	30
4.5	Die Exit-Anweisung	30
4.6	Die Loop-Anweisung	30
4.6.1	Die Loop-Anweisung ohne Iterations-Schema	31
4.6.2	Die Loop-Anweisung mit einem While-Schema	31
4.6.3	Die Loop-Anweisung mit einem For-Schema	31
4.6.4	Benannte Loop-Anweisungen	31
4.7	Die Block-Anweisung	32
4.8	Die Return-Anweisung	32
4.9	Die Goto-Anweisung	33
5	Unterprogramme	35
5.1	Deklaration eines Unterprogrammes	35
5.2	Formale Parameter	35
5.3	Aufruf von Unterprogrammen	37
5.4	Rückkehr aus Unterprogrammen	37
5.5	Unterprogramm-Implementation	38
5.6	Das Parametertyp-Profil	38
5.6.1	Pointer auf Unterprogramme	39
5.6.2	Überladen von Unterprogrammen	39
5.6.3	Überladen von Operatoren	39
5.7	Abstrakte Unterprogramme	39
6	Exceptions	41
6.1	Die Deklaration von Exceptions	41
6.2	Das Auslösen von Exceptions	41
6.3	Die Behandlung von Exceptions	41
7	Pakete	45
7.1	Abstrakte Datentypen	45
7.2	Hinausgeschobene Konstanten-Definition	47
7.3	Zugriff auf in Paketen deklarierte Objekte	48
7.4	Gegenseitige Abhängigkeit	49
7.5	Getrennte Übersetzbarkeit	50
7.6	Benutzerdefinierte Initialisierung, Zuweisung und Finalisierung	50
8	Generische Einheiten	53
8.1	Ein einleitendes, umfangreicheres Beispiel	53
8.2	Generische formale Objekte	55
8.3	Generische formale Typen	56
8.4	Generische formale Unterprogramme	57
8.5	Generische formale Pakete	57
9	Tasks	61
9.1	Multitasking in Ada	61
9.2	Task-Typen und Task-Objekte	61
9.3	Die Aktivierung und die Ausführung von Tasks	62
9.4	Anweisungen zur Programmierung von Tasks	62
9.4.1	Die Delay-Anweisung	62
9.4.2	Die Delay-Until-Anweisung	63

9.4.3	Die Abort-Anweisung	63
9.5	Das Rendezvous-Konzept in Ada	63
9.5.1	Wie ein gerufener Task das Rendezvous beeinflussen kann	64
9.5.2	Die Requeue-Anweisung	67
9.5.3	Der Private-Teil von Tasks	67
9.5.4	Wie der rufende Task das Rendezvous beeinflussen kann	67
9.6	Familien von Task-Entries	68
9.7	Diskriminierte Tasks	70
9.8	Scheduling von Tasks	71
9.9	Rendezvous und Exceptions	71
9.10	Die Termination von Tasks	72
10	Geschützte Objekte	75
10.1	Spezifikation und Implementierung geschützter Objekte	75
10.2	Die unterschiedlichen Operationstypen geschützter Objekte	76
10.3	Deklaration geschützter Objekte	77
10.4	Diskriminierte geschützte Objekte	77
11	Programmstruktur und Übersetzungsvorgang	79
11.1	Getrennte Übersetzung	79
11.2	Übersetzungs- und Bibliothekseinheiten	79
11.3	Untereinheiten von Übersetzungseinheiten	80
11.4	Das Hauptprogramm eines Ada-Programmes	82
12	Objektorientierte Konzepte in Ada und ihre Umsetzung in die Praxis	85
12.1	Heterogene Listen	85
12.2	Präfix-Notation	87
12.3	Mehrfache Implementationen	87
12.4	Iteratoren	90
12.5	Ein allgemeines Mengenkonstrukt als Beispiel	91
12.5.1	Die Standard-Logik	95
12.5.2	Die Fuzzy-Logik	96
12.5.3	Ein Standard-Beispiel	97
12.5.4	Ein Fuzzy-Beispiel	99
13	Mehrfachvererbung in Ada	101
13.1	Kombination von Implementation und Abstraktion	101
13.2	Mixins	101
13.3	Interfaces	103
13.4	Mehrfache Sichtweisen eines Objektes	104
14	Objektorientierung und Tasking	109
15	Low-Level Features	111
15.1	Darstellungsklauseln	111
15.1.1	Die Spezifikation von Längen	111
15.1.2	Die interne Darstellung von Aufzählungstypen	111
15.1.3	Die interne Darstellung von Record-Typen	112
15.1.4	Die Festlegung von Adressen	113
15.2	Attribut-Definitionen	113
15.3	Unkontrollierte Deallokation und Unkontrollierte Typ-Konversion	113
15.4	Storage Pool Management	114
15.5	Maschinennahe Programmierung	115

16 I/O und das Calendar-Paket	117
16.1 Ein-/Ausgabe	117
16.1.1 Ein-/Ausgabe für binäre Daten	117
16.1.2 Ein-/Ausgabe für lesbare Daten	118
16.2 Das Paket Calendar	119
 Die Anhänge zum Sprachstandard	 121
C Systemprogrammierung	123
C.1 Unterstützung für Interrupts	123
C.2 Identifikation und Attributierung von Tasks	124
 D Echtzeitprogrammierung	 127
D.1 Prioritäten	127
D.2 Scheduling mit Prioritäten	127
D.3 Die Standard-Dispatching-Politik	128
D.4 Priority Ceiling Locking	129
D.5 Entry-Queuing-Politiken	130
D.6 Dynamische Prioritäten	130
D.7 Über die Zeit	130
D.8 Synchrone Task-Kontrolle	132
D.9 Asynchrone Task-Kontrolle	132
D.10 Execution Time	132
 E Verteilte Systeme	 135
E.1 Kategorisierung von Bibliothekseinheiten	135
E.1.1 Pure Library Units	135
E.1.2 Shared Passive Library Units	135
E.1.3 Remote Types Library Units	136
E.1.4 Remote Call Interface Library Units	136
E.1.5 Normal Library Units	136
E.2 Konfiguration von Partitionen	136

Ada – Ein Überblick

In diesem, ersten Abschnitt wird in den Kapiteln 1 bis 16 eine Einführung in die Programmiersprache Ada gegeben. Dabei wird versucht, mit wenigen Vorgriffen die wesentlichen Konzepte der Sprache zu erläutern. Leider ist es nicht möglich, ganz ohne Vorgriffe auszukommen, was nicht weiter verwunderlich ist, wenn man bedenkt, daß eine Programmiersprache diesen Umfangs ein äußerst komplexes System darstellt.

Nach Lektüre dieses Abschnittes sollte der Leser die wesentlichen Konzepte verstanden haben und in der Lage sein, das Reference Manual für die Programmiersprache Ada [Ada95][†] zu verstehen. Es ist übrigens sehr ratsam, das Reference Manual nicht nur zu Rate zu ziehen, wenn bei der Programmierung eines Programmes Probleme auftreten, sondern auch nach der Lektüre des ersten Abschnittes, einen Blick hinein zu werfen. Dadurch kann der Leser auch jene Lücken in der Beschreibung schließen, die dadurch erzwungen wurden, daß Vorgriffe in diesen Unterlagen – wie bereits gesagt – vermieden wurden.

Die Sprache Ada erzwingt eine bestimmte Art der Programmierung oder legt sie zumindest nahe. Diese Art der Programmierung wird normalerweise als *objekt-orientiertes Programmieren* bezeichnet. Eine sehr gut geschriebene Zusammenfassung von objekt-orientierten Konzepten und einen Vergleich der bekanntesten objekt-orientierten und objekt-basierenden Sprachen (darunter auch Ada) findet man in [Boo91].

Wir beginnen unsere Einführung mit der kurzen Darstellung der *lexikalischen Einheiten* und der Beschränkungen, der die vom Benutzer festlegbaren, benannten Einheiten unterliegen. Anschließend wenden wir uns den *Typen*, *Klassen* und *Variablen* zu. Die Kapitel über *Ausdrücke*, *Anweisungen* und *Unterprogramme* schließen den Teil ab, der auch von anderen höheren Programmiersprachen bekannt ist. Darauf erläutern wir, wie in Ada *Fehlermeldungen* behandelt werden, wie *Module* in Ada realisiert werden können und kommen dann zu den für Ada einzigartigen Sprachmitteln: *generische Einheiten*, *Tasks*, *geschützte Objekte* und Sprachkonstrukten, die *maschinennahes Programmieren* erleichtern. Außerdem wird die Anwendung *objektorientierter Konzepte* in Ada an Hand eines umfangreichen Beispiels vorgeführt und gezeigt wie *Mehrfachvererbung* in Ada realisiert werden kann.

Die Entstehung von Ada

Die Entwicklung der Programmiersprache Ada wurde vom Verteidigungsministerium der USA, dem DoD (Department of Defense), Mitte der Siebzigerjahre initiiert, als erkannt wurde, daß die Kosten für Software im Bereich des DoD die für Hardware bei weitem überstiegen. Außerdem war die Koordination der verschiedenen Software-Produkte ziemlich schwierig, da DoD-intern viele verschiedene höhere Programmiersprachen und zahlreiche Assemblersprachen in Verwendung waren. Eine Erhebung ergab, daß die meisten Software-Anwendungen den *Automatisierungssystemen (embedded systems)* als zugehörig anzusehen waren und folgende Eigenschaften hatten:

- Sie sind groß. Typischerweise enthalten sie mehrere Milliarden Zeilen Programm-Code.
- Sie sind lange in Verwendung. Zeiträume von 10 bis 15 Jahren sind zu erwarten.
- Sie verändern sich ständig aufgrund sich ändernder Anforderungen.
- Sie sind physikalischen Beschränkungen in Hinsicht auf Zeit- und Platzbedarf unterworfen.

[†]Eine Liste aller zitierten Bücher und Artikel findet sich auf Seite 136 (*Literatur*).

- Sie sollen sehr zuverlässig und fehlertolerant sein.

Für die Programmierung solcher Systeme sind unter anderen Sprachelemente erforderlich, die Unterstützung bieten für:

- Parallelverarbeitung,
- Echtzeitprogrammierung,
- Behandlung von Fehlern und
- einheitliche Ein-/Ausgabefunktionen.

Eine weitere Untersuchung ergab, daß keine der zu dieser Zeit existierenden Programmiersprachen in der Lage war, alle Anforderungen zu erfüllen, die aus den obigen Eigenschaften der zu implementierenden Systeme folgen. Daher wurde ein Projekt ins Leben gerufen, das den Entwurf einer geeigneten Sprache zum Ziel hatte. Da man darauf bedacht war, keine Sprache zu schaffen, die nur innerhalb des DoD Verwendung findet, wurden alle führenden Firmen und Wissenschaftler eingeladen, sich daran zu beteiligen. Das Result dieses mehrjährigen Prozesses war die Definition von *Ada*, so benannt nach *Augusta Ada Byron, Countess of Lovelace*, die



Abbildung 0.1: Lady Ada, Countess of Lovelace

Tochter des Dichters *Lord Byron* und Mitstreiterin von *Charles Babbage*, mit dem sie gemeinsam an der „difference and analytic engine“ arbeitete. Sie kann mit Fug und Recht als die erste Programmiererin gelten.

Es ist interessant anzumerken, daß die Programmiersprache Ada aus einem Entwurfsvorschlag hervorgegangen ist, der aus Europa kam. Ein Team der Firma *Honeywell/Honeywell Bull* unter der Leitung von *Jean Ichbiah* machte das Rennen.

Anfang 1995 trat eine Revision von Ada in Form eines neuen Standards in Kraft, die unter anderem objektorientierte Konzepte in Ada einführt.

Notationskonventionen

Wir werden in diesem Abschnitt manchmal eine halbformale Beschreibung der Sprachsyntax verwenden. Dabei werden wir alle nicht näher spezifizierten Teile in spitzen Klammern wiedergeben, z.B.:

```
if <Bedingung> then
  <Anweisungen>
else
  <Anweisungen>
end if;
```

Prinzipiell werden alle Programmbeispiele und Syntaxbeschreibungen in obigem Format angegeben.

1 Lexikalische Einheiten und Namen

*Juda Leon se dio a permutaciones
De letras y a complejas variaciones
Y al fin pronuncio el Nombre que es la Clave,
La Puerta, el Eco, el Huesped y el Palacio ...*

*(Juda Löw verlegte sich auf Permutationen
Von Lettern und auf komplexe Variationen
Und schließlich sprach er ihn aus, den Namen, welcher der Schlüssel ist,
Das Tor, das Echo, der Wirt und der Palast ...)*

Jorge Luis Borges, „El Golem“.

Eine der grundlegendsten Fragen, die es zu beantworten gilt, bevor man daran gehen kann, mit einer Programmiersprache zu arbeiten, ist die, aus welchen Zeichen ein Programm aufgebaut ist. Nicht weniger wichtig ist, wie vom Benutzer definierte Bezeichner auszusehen haben und welche Bezeichner von der Sprache reserviert sind. Diese beiden Themenkomplexe werden wir in diesem Kapitel behandeln.

1.1 Der Zeichensatz

*Die zweiundzwanzig elementaren Lettern schnitt er,
formte er, kombinierte er, wog er, stellte er um und
formte mit ihnen alles Geschaffene sowie alles,
was es in Zukunft zu formen gibt.*

Sefer Jezirah, 2.2

Jedes Ada-Programm besteht aus einem festen Satz von Zeichen (Basic Multilingual Plane von ISO 10646 Universal Multiple-Octet Coded Character Set). Dieser Satz beinhaltet

1. Großbuchstaben
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
2. Kleinbuchstaben
a b c d e f g h i j k l m n o p q r s t u v w x y z
3. Ziffern
0 1 2 3 4 5 6 7 8 9
4. spezielle Zeichen
" # ' () * + , - . / : ; < = > _ | & [] { }
5. das Abstandzeichen (englisch auch *space* oder *blank* genannt)
6. und *formatbeeinflussende Zeichen* wie Tabulatoren, CR, LF, FF, sowie andere sogenannte *Kontrollzeichen*, die aber nur innerhalb von Kommentaren erlaubt und implementationsabhängig sind.

Die Beschränkung auf diesen Zeichensatz soll die Portabilität von Ada-Programmen verbessern.

Darüberhinaus ist es nun etwa auch möglich, Umlautzeichen und Akzente in Ada-Programmen zu verwenden. Seit 2005 können auch griechische und cyrillische Zeichen genutzt werden.

1.2 Lexikalische Einheiten und Namen

Name ist Schall und Rauch.

Faust.
Johann Wolfgang von Goethe, „Faust“.
Der Tragödie erster Teil.

Maulaf:
*Pipfaf, Klammeraf, Ganzbaf, Paragraf,
Dauerschlaf, Pornograf, Dompfaf!*
An die Arbeit!

...
Obelix:
Und komische Namen haben die alle!
Hühihih!
Hinten alle mit „af“!
Majestix:
*Hahaha! Miraculix, Troubadix, Verleihnix,
Methusalix, Rührfix, Machtnix,
habt ihr das gehört?*

Renè Goscinny und Albert Uderzo,
„Asterix und die Normanen“.

Lexikalische Einheiten sind aus obigem Zeichensatz aufgebaut. Man unterscheidet zwischen *Bezeichnern* (*Identifiern*), *reservierten Worten*, *numerischen Literalen*, *Characters*, *Strings*, *Delimitern* und *Kommentaren*.

Identifier bestehen aus einem Buchstaben gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern und Underscores (um die Lesbarkeit zu erhöhen). Underscores dürfen jedoch weder am Ende eines Identifiers stehen, noch dürfen zwei von ihnen unmittelbar aufeinanderfolgen. Es gibt keine Beschränkung bezüglich der Länge von Identifiern, aber ein Identifier muß in eine Zeile passen. (Die Zeilenlänge wird im Sprachumfang als ≥ 200 festgelegt, aber im allgemeinen von Compilern auf einen festen Wert fixiert.) Ada unterscheidet nicht zwischen Groß- und Kleinschreibung. Die folgenden Beispiele sind korrekte Ada-Identifier:

Alphabet
Gamma_Strahl
SCSLInterface
F16

Reservierte Wörter haben in Ada-Programmen spezielle Bedeut. Es gibt genau die folgenden 72 reservierten Wörter.

abort	else	new	return
abs	elsif	not	reverse
abstract	end	null	
accept	entry		select
access	exception	of	separate
aliased	exit	or	subtype
all		others	synchronized
and	for	out	
array	function	overriding	tagged
at			task
	generic	package	terminate
begin	goto	pragma	then
body		private	type
	if	procedure	
case	in	protected	until
constant	interface		use
	is	raise	
declare		range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	
do	mod	requeue	xor

Numerische Literale stellen ganzzahlige oder reelle Zahlenwerte dar. Zahlen können zu jeder Basis von 2 bis 16 dargestellt werden. Außerdem können zwischen den einzelnen Ziffern *Underscores* zur besseren Lesbarkeit eingefügt werden. Es folgen einige korrekte numerische Literale:

```

7
1_000_000 = 1000000
1e6 = 1000000
2#1101# = 16#D# = 13

0.25
3.14_15_9
1.25e6 = 1250000.0
16#F.8# = 15.5
16#F.FF#E+2 = 4095.0

```

Reelle Zahlen erfordern zur syntaktischen Korrektheit mindestens eine Ziffer links und rechts vom Dezimalpunkt. Die Syntax der Darstellung von Zahlen zu einer bestimmten Basis sollte aus den Beispielen hervorgehen; wir ersparen uns daher eine genauere Erläuterung.

Ein *Character* besteht aus irgendeinem der Zeichen des oben angegebenen Zeichensatzes und ist von einfachen Hochkommata eingeschlossen, z.B.:

```

'A',
'*',
''' (Der Character ').

```

Ein *String* hingegen besteht aus null oder mehr Zeichen, die von doppelten Hochkommata eingeschlossen sind, z.B.:

```

"",
"Das ist ein String.",
"""" (Der String mit dem Wert ").

```

Delimiter bestehen aus den Zeichen:

```
' ( ) * + , - . / : ; < = > | &
```

Zusätzlich gibt es noch zusammengesetzte Delimiter:

```
=> .. ** := /= >= <= << >> <>
```

Zwischen zwei lexikalischen Einheiten darf eine beliebige Anzahl von *Blanks* stehen.

Kommentare beginnen mit -- und enden am Ende der Zeile.

Obwohl man sie für gewöhnlich nicht als lexikalische Einheiten auffaßt, gibt es noch eine Art von Konstrukten, die in Ada-Programmen vorkommt: die *Pragmas*. Ein *Pragma* ist zwar eine Anweisung an den Compiler, in Ada gibt es jedoch einige vordefinierte *Pragmas*, die wir noch behandeln werden.

2 Typen, Klassen und Variablen

*He, Leute!
Wieder so ne Type!*

Wachposten.
René Goscinny und Albert Uderzo,
„Obelix GmbH & Co.KG“.

In diesem Kapitel werden wir uns damit beschäftigen, welche vordefinierten *Typen* und *Klassen* in Ada existieren und wie man in Ada *Konstanten* und *Variablen* deklariert.

Ein *Typ* ist charakterisiert durch eine Menge von *Werten* und eine Menge von *primitiven Operationen*, die die grundlegenden Aspekte seiner Semantik implementieren. Ein *Objekt* eines bestimmten Typs existiert zur Laufzeit und beinhaltet einen Wert des Typs.

Typen werden zu *Klassen* zusammengefaßt, die die Ähnlichkeit ihrer Werte und primitiven Operationen widerspiegeln. Es gibt mehrere von der Sprache vordefinierte Klassen von Typen, die wir im folgenden behandeln werden.

2.1 Skalare Typen

Unter *skalaren Typen* versteht man alle Typen, die aus einem einzigen Wert bestehen, die also nicht zusammengesetzt sind. Im speziellen sind das

- ganze Zahlen,
- reelle Zahlen und
- Aufzählungen.

2.1.1 Ganze Zahlen

Primär kennt Ada den Typ `integer` zur Darstellung von ganzen Zahlen; es ist aber möglich, ganze Zahlen auf bestimmte Bereiche zu fixieren, etwa durch

```
subtype small_int is integer range -10 .. 10;
subtype index is integer range 1 .. 100;
subtype teilindex is index range 5 .. 17;
```

Die Werte, die bei der Definition eines Bereiches angegeben werden, sind sogenannte *statische Werte*, d.h. Werte oder Ausdrücke, die zur Compile-Zeit bekannt sind. Falls der Bereich leer ist, gibt es keinen regulären Wert für Variablen dieses Typs.

Obwohl Ada vordefinierte Typen für ganze Zahlen kennt, empfiehlt es sich dennoch, immer eigene Typen zu definieren, da die vordefinierten Typen maschinenabhängige Darstellung haben können. Wenn man seine eigenen Typen vereinbart, überläßt man dem Compiler die Darstellung und erhält so Programme, die sich viel leichter von einem Rechner auf einen anderen portieren lassen. Dieselbe Aussage gilt natürlich auch allgemeiner für alle vordefinierten Typen und nicht nur für ganzzahlige.

Andere vordefinierte Typen außer `integer` sind:

<code>positive</code>	alle positiven ganzen Zahlen
<code>natural</code>	alle positiven ganzen Zahlen und die Null

Außerdem kann man sogenannte *modulare* Integer-Typen definieren. Das sind ganze Zahlen, für die die Operationen modulo des spezifizierten Moduls ausgeführt werden. Eine Deklaration sieht etwa folgendermaßen aus:

```
type byte is mod 256;
type Hash_Index is mod 97;
```

2.1.2 Aufzählungen

Aufzählungen (Enumerations) können ebenfalls als Typen definiert werden, z.B.

```
type Tag is (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag);
```

Vordefinierte Aufzählungstypen sind `boolean` und `character`. Der Typ `boolean` umfaßt die beiden Werte `false` und `true`. Die Werte des Typs `character` sind die 256 Zeichen des Latin-1-Zeichensatzes von ISO 10646. Dabei können alle nicht graphisch darstellbaren Zeichen über einen vordefinierten Namen angesprochen werden.

Ada legt fest, daß erst dann, wenn der Typ von Objekten definiert ist, Variablen angelegt werden können, die solche Objekte bezeichnen. Variablen eines bestimmten Typs können etwa wie folgt deklariert werden:

```
s,t: small_int;
Wochentag: Tag;
```

Darüberhinaus ist es in Ada möglich, bei der Deklaration einer Variablen einen Anfangswert mitzugeben, z.B.:

```
Wochentag: Tag := Montag;
i,j: index := 1;
```

Fügt man bei der Deklaration einer „Variablen“ vor dem Typ-Indikator das Schlüsselwort `constant` ein, so kann der Wert der „Variablen“ nicht mehr geändert werden, sie verhält sich eben wie eine *Konstante*. Selbstverständlich muß man in diesem Fall eine explizite Initialisierung vornehmen. Folgendes Beispiel ist eine korrekte Konstanten-Deklarationen im Sinne von Ada:

```
Anzahl_der_Wochentage: constant integer := 7;
```

Typen, die Aufzählungen und ganze Zahlen darstellen, werden auch *diskrete Typen* genannt. Ada gestattet auch Informationen über Typen zu „erfragen“. Dies wird mit sogenannten *Attributen* durchgeführt. Die wichtigsten Attribute für diskrete Typen sind:

T'FIRST	die untere Schranke des Typs T.
T'LAST	die obere Schranke des Typs T.
T'POS(X)	gibt die Position des Wertes X im Typ T als integer zurück, z.B.: Tag'POS(Dienstag) = 2.
T'VAL(X)	gibt den Wert des Typs T zurück, der an der Position X steht, z.B.: Tag'VAL(3) = Mittwoch.
T'SUCC(X)	der Nachfolger von X, falls X nicht gleich der oberen Schranke von T ist, sonst erfolgt eine Fehlermeldung (<code>constraint_error</code>).
T'PRED(X)	der Vorgänger von X, falls X nicht gleich der unteren Schranke von T ist, sonst erfolgt eine Fehlermeldung (<code>constraint_error</code>).
T'IMAGE(X)	gibt einen String zurück, der den momentanen Wert von X enthält, z.B.: Tag'IMAGE(Montag) ergibt "MONTAG" oder small_index'IMAGE(10) ergibt "10".
T'VALUE(X)	wandelt den String X in den entsprechenden Wert des Typs T um, z.B.: Tag'VALUE("FREITAG") ergibt Freitag und small_index'VALUE("9") ergibt 9. Falls die Aufzählung einen entsprechenden Wert nicht enthält, die Syntax im Falle einer ganzen Zahl nicht korrekt ist oder der entsprechende Wert nicht im für den Typ spezifizierten Bereich liegt, wird <code>constraint_error</code> ausgelöst.

An dieser Stelle soll angemerkt werden, daß es auch möglich ist, bestimmte Attribute für selbst-definierte Typen zu implementieren. Näheres findet sich im Reference Manual ([Ada95]).

2.1.3 Reelle Zahlen

Die Darstellung *reeller Zahlen* ist in Ada weit flexibler als bei anderen Programmiersprachen, wie etwa Pascal oder Modula-2. Es gibt drei Arten von reellen Zahlen: *Gleitkommazahlen* (*floating point types*), *Festkommazahlen* (*fixed point types*) und *Dezimalzahlen*, wie sie etwa auch in COBOL Verwendung finden.

Bei der Definition eines Gleitkommazahltyps muß die Anzahl von Dezimalstellen angegeben werden, die die Mantisse umfaßt. Optional kann auch ein Bereich spezifiziert werden, in dem Werte des Types liegen sollen, z.B.:

```
type Masse is digits 10;
type real is digits 8;
type Wahrscheinlichkeit is digits 8 range 0.0 .. 1.0;
```

Der Typ einer Festkommazahl wird festgelegt, indem man die numerische Differenz zwischen zwei aufeinanderfolgenden Werten angibt. Wieder kann ein Bereich spezifiziert werden, z.B.:

```
type Volt is delta 0.01 range -12.0 .. +12.0;
```

Darüber hinaus kann man *Dezimalzahlen* verwenden, wie sie etwa in COBOL üblich sind. Ein Beispiel ist:

```
type Geld is delta 0.01 digits 10;
- maximal 10-stellige Geldbeträge auf Cent genau
```

In Ada besteht auch die Möglichkeit von *Typ-Konversionen*, so kann etwa die Zahl 1.2 durch `integer(1.2)` auf eine ganze Zahl (gerundet) abgebildet werden. Umgekehrt kann man mittels `Masse(2)` eine ganze Zahl auf eine Gleitkommazahl oder mit `Volt(2)` auf eine Festkommazahl abbilden.

2.1.4 Universelle Typen

Bei der bisherigen Darstellung von ganzen und reellen Zahlen haben wir ein Problem außer acht gelassen, nämlich: Welchen Typ haben eigentlich Literale?

Ada definiert zur Lösung dieses Problems die sogenannten *universellen Typen*. Um genau zu sein, gibt es vier universelle Typen, nämlich: `universal_integer`, `universal_real`, `universal_dezimal` und `universal_fixed`. Entsprechende Literale haben also den passenden Typ und bei der Zuweisung eines Literals zu einer Variablen erfolgt eine *implizite Typ-Konversion*. Das ist übrigens die einzige in Ada erlaubte implizite Typ-Konversion.

Abschließend sei noch darauf hingewiesen, daß neben numerischen Literalen auch gegebenenfalls der Rückgabewert von Attributen ein universeller Typ ist.

2.2 Zusammengesetzte Typen

Bisher haben wir nur Typen behandelt, die aus einer einzigen Komponente bestehen. Ada gestattet jedoch auch die Definition von Typen, die sich aus mehreren Komponenten zusammensetzen, nämlich *Arrays* und *Records*.

2.2.1 Felder

Felder (*arrays*) können in Ada sowohl in der – auch aus anderen Programmiersprachen wie etwa Pascal oder Modula-2 bekannten – Weise durch exakte Festlegung der Grenzen als auch durch deren Offenlassen definiert werden, z.B.:

```
type Acht_Bit_Vector is array(0 .. 7) of boolean;
type char_set is array(character) of boolean;
type Arbeitsstunden is array(Tag range Montag .. Freitag) of natural;

type Vector is array(integer range <>) of real;
type Matrix is array(integer range <>, integer range <>) of real;
type Bit_Vector is array(integer range <>) of boolean;
```

Das oben erstmals eingeführte <>-Zeichen wird auch als *Box* bezeichnet.

Ein Array-Typ, dessen Grenzen festgelegt sind, heißt *constrained*, sonst nennt man ihn *unconstrained*.

Auch für Array-Typen bzw. Array-Variable sind einige Attribute definiert:

A'FIRST	untere Schranke des Index-Typs
A'FIRST(N)	untere Schranke des N-ten Index-Typs
A'LAST	obere Schranke des Index-Typs
A'LAST(N)	obere Schranke des N-ten Index-Typs
A'RANGE	gibt einen Bereich zurück, nämlich A'FIRST .. A'LAST
A'RANGE(N)	gibt den Bereich A'FIRST(N) .. A'LAST(N) zurück*
A'LENGTH	gibt die Größe des Indexbereiches zurück, 0 für einen leeren Indexbereich
A'LENGTH(N)	gibt die Größe des N-ten Indexbereiches zurück, 0 für einen leeren Indexbereich

Der vordefinierte *String-Typ* kann aufgefaßt werden als

```
type string is array(positive range <>) of character;
```

Array-Variablen eines constrained Array-Typs können ohne weiteres etwa folgendermaßen definiert werden:

```
word: Acht_Bit_Vector;
```

Die Grenzen von Variablen eines unconstrained Array-Typs müssen bei ihrer Deklaration festgelegt werden. Dabei ist zu beachten, daß die angegebenen Grenzen keine statischen Werte sein müssen, sondern sich durchaus aus Werten rekrutieren können, die erst zur Laufzeit bekannt sind, z.B.:

```
Schachbrett: Matrix(1..8, 1..8);
Rechteck: Matrix(1..10, 1..20);
Quadrat: Matrix(1..N, 1..N);
- Dabei kann N auch eine Integer-Variable sein
mein_String: string (1..100);
```

Es bleibt noch die Frage zu klären, wie man auf die einzelnen Komponenten eines Arrays zugreifen kann. Dies geschieht in Ada durch runde Klammern, z.B.:

```
word(2)
Schachbrett(2,4)
Quadrat(1,1)
mein_String(5)
```

Abschließend soll noch darauf hingewiesen werden, daß in Ada ein wesentlicher Unterschied besteht zwischen zweidimensionalen Arrays und einem eindimensionalen Array, dessen Komponenten eindimensionale Arrays sind¹. Dieser Unterschied macht sich vor allem beim Zugriff auf Elemente solcher Arrays bemerkbar. Gerade haben wir den Zugriff auf Elemente von zweidimensionalen Arrays kennengelernt (z.B. Schachbrett(2,4)). Wäre Schachbrett als Array eines Array's definiert worden, z.B.:

*Das Range-Attribut ist speziell bei der Verwendung von For-Schleifen praktisch.

¹Entsprechende Aussagen gelten selbstverständlich auch für höhere Dimensionen.

```
type Zeile is array(1 .. 8) of real;
type Schachbrett is array(1 .. 8) of Zeile;
```

so lautete ein Zugriff auf ein entsprechendes Element:

```
Schachbrett(2)(4)
```

Zusätzlich zum Zugriff auf Array-Elemente kennt Ada sogenannte *Slices*. Damit kann man Teile eines eindimensionalen Arrays „herausschneiden“ und erhält so wieder ein Array. Beispiele sind:

```
word(2..6)
mein_String(1..8)
```

Dabei ist aber zu beachten, daß z.B. mein_String(1..1) ein Array der Länge 1 mit dem Komponenten-Typ Character ist, aber andererseits mein_String(1) ein Character.

2.2.2 Records

Gewöhnliche *Records* in Ada sind sehr ähnlich denen in Pascal oder Modula-2. Als Beispiel seien die folgenden Typen angegeben:

```
type Datum is
record
  Tag: integer range 1 .. 31;
  Monat: Monatsname;
  Jahr: integer range 0 .. 4000;
end record;

type komplex is
record
  re: real;
  im: real;
end record;
```

Beispiele für Record-Variablen sind:

```
Geburtstag: Datum;
a,b,c: komplex;
```

Ada bietet zusätzlich die Möglichkeit, Records mittels sogenannter *Diskriminanten* zu „parametrieren“. Ein wesentlicher Unterschied besteht in diesem Fall zu Pascal und Modula-2; in Ada ist nämlich die Diskriminante immer Teil des Records, in Pascal und Modula-2 muß das nicht so sein. Ein Beispiel ist:

```
type Quadrat(Seite: integer) is
record
  Mat: Matrix(1..Seite, 1..Seite);
end record;
```

Man kann auch Anfangswerte für Diskriminanten definieren:

```
type Rechteck(Laenge: integer := 10; Breite: integer := 5) is
record
  Mat: Matrix(1..Laenge, 1..Breite);
end record;
```

Interessant wird es, wenn man Variablen solchen Typs anlegt. So ist etwa

```
Q1: Quadrat;
```

nicht erlaubt, da die Seitenlänge nicht bekannt ist. Hingegen sind

```
Q2: Quadrat(5);
R1: Rechteck(11,2);
R2: Rechteck;
```

erlaubt. Dabei sind Q2 und R1 constrained, während R2 zwar eine Laenge von 10 und eine Breite von 5 hat, aber trotzdem unconstrained ist, d.h., man kann später Laenge und Breite von R2 verändern! Dabei ist aber gefordert, daß immer der gesamte Record einen neuen Wert erhält. Wie man das konkret bewerkstelligt, werden wir in Kapitel 2.2.3 kennenlernen.

*You behandle me as the last dreck:
as hampel-man of society,
as political hans-wurst,
as complet idiot,
as religious mama-kindl and
as social soup-kasper! Always the Black
are the beshittened! And the gelackmeiered!*

aus Armin Eichholz,
„Black and White – Nach moderner Negerlyrik“.

Diskriminierte Records, das sind Records, die abhängig von einer Diskriminante unterschiedliche Komponenten aufweisen, werden von Ada auch unterstützt, z.B.:

```
type Wochentag(T: Tag) is
  record
    Aufstehen: Uhrzeit;
  case T is
    when Samstag | Sonntag =>
      Freizeit: integer range 0 .. 24;
    when others =>
      Arbeit: integer range 8 .. 17;
  end case;
end record;
```

Das in obigem Beispiel angeführte *others* darf nur als letzte Alternative in der umschließenden Case-Anweisung verwendet werden. Das |-Zeichen dient zur Aufzählung mehrerer Werte, die in einer Alternative behandelt werden sollen.

Der Zugriff auf einzelne Record-Komponenten erfolgt in Ada durch einen sogenannten Punkt-Operator, z.B.:

```
Geburstag.Jahr
are
```

2.2.3 Aggregate

Ein *Aggregat* dient dazu, Werte von Komponenten zu Werten eines Arrays oder Records zusammenzufassen. Dabei unterscheidet man zwischen zwei Arten, wie die einzelnen Komponenten

angesprochen werden, nämlich einerseits über ihre *Position* innerhalb des übergeordneten Typs und andererseits über ihren *Namen*. Es ist auch erlaubt, beide Arten zu mischen, wobei jedoch immer die Festlegung über die Position vor der über den Namen zu verwenden ist, d.h., wenn einmal eine Komponente über ihren Namen angesprochen worden ist, so dürfen alle restlichen Komponenten auch nur mehr über Namen referenziert werden.

Betrachten wir zuerst *Record-Aggregate*. Ein Beispiel für die Festlegung durch die Positionen der Komponenten ist:

```
(29, September, 1967) -- siehe Typ Datum Seite 14
```

Derselbe Wert wird über benannte Komponentenfestlegung etwa ausgedrückt durch:

```
(Tag => 29, Monat => September, Jahr =>1967)
(Monat => September, Tag => 29, Jahr =>1967)
```

Mischformen sind:

```
(29, Jahr => 1967, Monat => September)
(29, September, Jahr => 1967)
```

Falls die restlichen Komponenten alle denselben Typ haben, kann man ihnen auch mit *others* geschlossen einen (gemeinsamen) Wert zuweisen, z.B.:

```
zero: constant komplex := (others => 0.0);
```

Beispiele für *Array-Aggregate* sind im eindimensionalen Fall:

```
word.1: Acht_Bit_Vector := (true, true, true, false, false, true, false, true);
word.2: Acht_Bit_Vector := (1 => true, 3 => false, 5 => true, 7 => true,
                           0 => true, 2 => true, 4 => false, 6 => false);
word.3: Acht_Bit_Vector := (3 .. 4 | 6 => false, others => true);
```

Das |-Zeichen dient wieder zur Aufzählung mehrerer Index-Bereiche, für die derselbe Wert zugewiesen werden soll. Das Schlüsselwort *others* darf nur verwendet werden, wenn aus dem Kontext klar ist, wie groß die Grenzen des Arrays wirklich sind. Wenn das nicht eindeutig bestimmt werden kann, darf man dem Compiler „unter die Arme greifen“. Dies geschieht durch sogenannte *qualifizierte Ausdrücke* (*qualified expressions*).

Das ist siche' ein T'ick!

Afrikanischer Pirat.
René Goscinny und Albert Uderzo,
„Asterix auf Korsika“.

Dabei wird dem Aggregat der Typ-Bezeichner eines constrained Arrays und außerdem ein ' (Tick) vorausgestellt, z.B.:

```
Acht_Bit_Vector'(3 .. 4 | 6 => false, others => true)
```

Im mehrdimensionalen Fall treten geschachtelte Aggregate auf, z.B.:

```
Schachbrett: Matrix(1..8, 1..8) := (1 .. 8 => (1 .. 8 => 0.0));
```

Auch hier gelten natürlich die obigen Ausführungen für `others`.

Seit 2005 kann man in einer `others`-Alternative auch die Box `<>` verwenden, die in diesem Fall für den Default-Wert steht. So bewirkt etwa

```
P: array (1 .. 1000) of integer := (1 => 2, 2 .. 1000 => <>);
```

dass die erste Komponente von `P` den Wert 2 erhält und alle anderen uninitialisiert bleiben.

Für Record-Aggregate gelten die Ausführungen ebenfalls.

Das ganze scheint hier nicht wirklich sinnvoll, bei anderen Typen macht es dann aber mehr Sinn.

2.3 Subtypen

Manchmal macht es Sinn, sich auf einen bestimmten Teilbereich eines Typs zu beziehen, ohne daß man einen gänzlich neuen Typ definieren will. Für solche Situationen bietet Ada das Konzept der *Subtypen* an. Ein Subtyp besteht aus einem Verweis auf seinen zugrundeliegenden Typ – den *Basis-Typ* – und einer *Bedingung*. Ein Wert gehört zu einem bestimmten Subtyp, wenn er zum zugrundeliegenden Typ gehört und die Bedingung erfüllt. Eine solche Bedingung kann sein:

- ein Bereich für ganze Zahlen, Gleitkommazahlen, Festkommazahlen oder Dezimalzahlen,
- eine Einschränkung eines Index oder
- eine Einschränkung einer Diskriminante.

Beispiele dafür sind die folgenden Subtyp-Definitionen:

```
subtype Arbeitstag is Tag range Montag .. Freitag;
subtype kleiner_index is index range 5 .. 25;
subtype positiv_Volt is Volt range 0.0 .. +12.0;
subtype kleine_Matrix is Matrix(1..5, 1..5);
subtype erster_Arbeitstag is Arbeitstag(Montag);
```

Für die im Kapitel 2.1.1 angegebenen Typen `positive` und `natural` gilt, daß sich der Typ `positive` wie ein Subtyp von `natural` verhält und daß sich beide Typen wie Subtypen von `integer` verhalten.

Es soll noch einmal darauf hingewiesen werden, daß durch einen Subtyp kein neuer Typ definiert wird, sondern nur eine Bedingung auf den Basis-Typ erzwungen wird. Zuweisungen von Variablen eines Subtyps auf solche des Basis-Typs (und umgekehrt) sind möglich, solange die zugehörige Bedingung erfüllt ist.

Seit 2005 kann zusätzlich bei einem Subtyp durch die Angabe von `not null` erreicht werden, dass Variablen dieses Subtyps nicht den Wert `null` annehmen dürfen, was aber erst im Zusammenhang mit Pointern relevant ist.

2.4 Pointer

Pointer-Typen werden von Ada ebenfalls unterstützt; sie werden häufig auch als *Access-Typen* bezeichnet. Es gibt *Pointer auf Objekte* und *Pointer auf Unterprogramme*¹.

¹Unterprogramme werden detailliert in Kapitel 5 behandelt.

2.4.1 Pointer auf Objekte

Ein Beispiel ist:

```
type Matrix_Zeiger is access Matrix;
```

Nach dem Schlüsselwort `access` kann auch eines der reservierten Wörter `all` oder `constant` stehen. Der erste Fall ist der defaultmäßige Fall und bedeutet, daß das designierte Objekt verändert werden darf. Im zweiten Fall ist das unterbunden.

Um verkettete Datenstrukturen zu realisieren, bietet Ada die Möglichkeit einer sogenannten *unvollständigen Typvereinbarung*, z.B.:

```
type Knoten;
```

```
type Zeiger_auf_Knoten is access Knoten;
```

```
type Knoten is
  record
    Wert: integer;
    links: Zeiger_auf_Knoten;
    rechts: Zeiger_auf_Knoten;
  end record;
```

```
Wurzel_des_Baumes: Zeiger_auf_Knoten;
```

Pointer-Variablen werden implizit mit dem Wert `null` initialisiert, der einem ungültigen Pointer entspricht.

Das Dereferenzieren eines Pointers geschieht in Ada implizit, falls man eine Komponente des dereferenzierten Objektes anspricht, d.h. also z.B. durch den Punkt-Operator:

```
Wurzel_des_Baumes.Wert;
```

Das Dereferenzieren eines Null-Pointers bewirkt allerdings eine Laufzeit-Fehlermeldung (`constraint_error`). Will man das gesamte dereferenzierte Objekt ansprechen, so kann man `all` verwenden, z.B. spricht man mit

```
Wurzel_des_Baumes.all
```

ein ganzes Objekt des Typs `Knoten` an.

Das dynamische Anlegen eines Objektes geschieht mittels `new`, z.B.:

```
new Knoten;
```

Man kann auch mit einem qualifizierten Ausdruck einen Anfangswert zuweisen, z.B.:

```
new Knoten'(Wert => 1, links => null, rechts => null);
```

Dabei muß auch darauf geachtet werden, daß das dynamisch angelegte Objekt `constrained` ist.

2.4.2 Anonyme Pointer

Die oben angeführte unvollständige Typvereinbarung ist nicht nur lästig, sondern man kann auch manchmal ohne sie auskommen. Das funktioniert über sogenannte anonyme Pointer, z.B.:

```
type Knoten is
  record
    Wert: integer;
    links: access Knoten;
    rechts: access Knoten;
  end record;

Wurzel_des_Baumes: access Knoten := ...;
```

Man kann solche anonyme Pointer auch mit einer sogenannten *Null-Exklusion* verbinden. So bewirkt etwa

```
Wurzel_des_Baumes: not null access Knoten := ...;
```

dass der Variablen niemals der Wert null zugewiesen werden darf. Widrigenfalls wird ein `constraint_error` ausgelöst.

Eine Null-Exklusion hat sehr viel Ähnlichkeit mit Bedingungen, die wir bei den Subtypen kennengelernt haben. Im Gegensatz zu diesen können jene aber bei Parameter-Definitionen von Unterprogrammen angegeben werden.

2.4.3 Pointer auf Unterprogramme

Es wird hier nur ein einfaches Beispiel angegeben:

```
type message_procedure is access procedure (m: in string := "Error!");
```

Man erkennt daraus, daß nur die Parameter spezifiziert werden. Es kann dann jedes Unterprogramm, das mit diesen Parametern konform ist, tatsächlich der Pointer-Variablen zugewiesen werden. Näheres in Kapitel 5 betreffend *Parametertypprofil*.

Pointer auf Unterprogramme können seit 2005 auch mit dem Konzept der anonymen Pointer kombiniert werden.

2.4.4 Alias-Objekte

Man kann auch bei einer Objektdекlaration angeben, daß es sich um ein Alias-Objekt handelt. Ein Beispiel ist:

```
Matrix_Ptr: Matrix_Zeiger;
Matrix_Array: array(1..2) of aliased Matrix;
...
Matrix_Ptr := Matrix_Array(1)'ACCESS;
```

Mit dem Schlüsselwort `aliased` werden also Alias-Objekte deklariert und mit dem Attribut `P'ACCESS` werden sie passenden Pointer-Variablen zugewiesen. `P` muß dabei ein Alias-Objekt sein.

Es sei hier explizit darauf hingewiesen, daß ein Alias-Objekt, auf das ein Pointer zeigt, auf dem Stack und nicht auf dem Heap liegt, d.h., daß es sich also um kein dynamisch angelegtes Objekt handelt.

2.4.5 Eine Variabilität konstanter Pointer

Anonyme Pointer können auch mit dem Schlüsselwort `constant` kombiniert werden. So kann man etwa schreiben:

```
ACT: access constant T := X1'ACCESS;
CAT: constant access T := X1'ACCESS;
CACT: constant access constant T := X1'ACCESS;
```

Dabei ist `ACT` eine Variable, die genutzt werden kann, auf verschiedene Objekte `X1` oder `X2` zuzugreifen. `CAT` ist eine Konstante, über die man nur auf das Objekt zugreifen kann, das bei seiner Initialisierung angegeben wurde. Allerdings kann der Wert dieses Objekts, etwa mittels `CAT.all := ...` geändert werden. `CACT` schließlich ist eine Konstante, die nur lesenden Zugriff auf `X1` erlaubt. Weder kann man `X1` durch `X2` ersetzen noch kann man den Wert von `X1` ändern.

2.5 Private Typen

Ada bietet, wie wir bisher gesehen haben, sehr flexible Möglichkeiten, um Datenstrukturen zu definieren. Das heißt, wir sind in der Lage, die Werte, die ein Objekt eines Typs annehmen kann, sehr exakt zu spezifizieren. Um jedoch die Operationen festzulegen, die auf ein Objekt ausgeübt werden können, bedarf es zusätzlicher Unterstützung. Meist besteht dabei in programmiersprachlicher Hinsicht das Problem vor allem darin, zu verhindern, daß hinterrücks Komponenten des Objektes (meist unabsichtlich) verändert werden, womit die Konsistenz des gesamten Objektes zerstört wird. Es ist daher erforderlich, auf Programmiersprachen-Niveau Sprachmittel zur Verfügung zu stellen, die solche „Untergriff-Methoden“ verhindern.

Ada kennt sogenannte *private Typen*, um dieses Prinzip des „*information hiding's*“ zu verwirklichen. Dabei werden die Einzelheiten der Implementierung verborgen und die Abstraktion im Problembereich in den Vordergrund gestellt. Wenn man zum Beispiel daran denkt, einen Stack zu realisieren, so sind vor allem die Operationen *Push* und *Pop* wichtig, nicht jedoch wie diese konkret implementiert werden. Der Stack kann dann konkret als Array oder als verkettete Liste realisiert werden, ohne die, davon unabhängig definierten Operationen zu beeinflussen.

Um ein vernünftiges Beispiel angeben zu können, müssen wir leider einen kleinen Vorgriff tätigen. Ein privater Typ muß nämlich in eine *Paket-Struktur* eingeschlossen werden. Wie solche Strukturen genauer aussehen, werden wir in Kapitel 7 sehen. Wir wählen als Beispiel unseren oben eingeführten Typ `Datum` (siehe Seite 14). Es ist leicht zu sehen, daß bei Beibehaltung des `Record`-Typs niemand daran gehindert werden kann, etwa das Datum 31. Februar 1991 darzustellen. In Ada kann das unterbunden werden, indem wir `Datum` als privaten Typ vereinbaren.

```
package Datum_Manager is
  type Datum is private;
...

private
  type Datum is
    record
      Tag: integer range 1 .. 31;
      Monat: Monatsname;
      Jahr: integer range 0 .. 4000;
    end record;
end Datum_Manager;
```

Die benötigten Operation, wie etwa Setzen und Lesen des Datums, sind im Anschluß an die Definition von Datum zu spezifizieren, die Implementation erfolgt dann in einem zugehörigen Package-Body.

Abschließend sei darauf hingewiesen, daß MODULA-2 ein ähnliches Sprachkonstrukt unterstützt, die sogenannten *opaquen Typen*.

2.6 Limitierte Typen

In Ada kann man einen Record-Typ auch als *limitiert* deklarieren. Damit erreicht man, daß es nicht möglich ist, einer Variablen dieses Typs einen Wert mittels der gewöhnlichen Zuweisung zuzuweisen. Außerdem ist dann die gewöhnliche Vergleichsoperation unterbunden, das heißt, man kann zwei Objekte eines limitierten Typs nicht mehr auf Gleichheit überprüfen. Vielmehr müssen dann Operationen vom Programmierer angeboten werden, mit denen Zuweisung und/oder Vergleich realisiert werden[‡]. Ein Beispiel ist

```
type t is limited
  record
    a: integer;
  end record;
```

```
v1, v2: t;
```

Dann sind etwa

```
v1 := v2;
```

und

```
if v1 = v2 then ...
```

nicht erlaubt und produzieren einen Fehler beim Compilieren.

Daher kennt Ada zusätzlich zu den *private types*, wie der oben angeführte, für die die für sie definierten Operationen und die vordefinierten Operationen der Zuweisung und der Gleichheit und Ungleichheit gelten, dann auch noch sogenannte *limited private types* (dabei würde limited in obigem Programm-Code zwischen den Schlüsselwörtern is und private stehen), bei denen die vordefinierten Operationen der Zuweisung und der Gleichheit und Ungleichheit ihre Gültigkeit verlieren. Näheres dazu findet sich im Kapitel 7.1.

2.7 Getaggte Typen und Typerweiterungen

In diesem Abschnitt behandeln wir die Kernpunkte von Ada, die die Objektorientierung betreffen, nämlich *Vererbung* und (*Laufzeit-*) *Polymorphismus*. Umfangreichere Beispiele zu diesem Thema finden sich in Kapitel 12.

Ein Record-Typ oder ein privater Typ, der das reservierte Wort tagged in seiner Deklaration beinhaltet ist ein *getaggtter Typ*. Ein Objekt eines getaggtten Typs trägt zur Laufzeit mit sich ein eindeutiges *Tag*.

[‡]Falls man sie überhaupt wünscht.

„Guten Tag!“, sagte der kleine Prinz.

Antoine de Saint-Exupéry,
„Der Kleine Prinz“

Nur von getaggtten Typen kann man andere Typen *ableiten*. Dabei können zusätzliche Komponenten und primitive Operationen definiert werden. Der abgeleitete Typ wird *Erweiterung* des Vorgängertyps oder einfach *Typerweiterung* genannt. Jede Typerweiterung ist wieder ein getaggtter Typ. Der Typ, von dem abgeleitet wird, heißt *Vatertyp*. Es gibt also nur *einfache Vererbung* und keine *mehrfache Vererbung*.

Man kann folgende Eigenschaften von abgeleiteten Typen festhalten:

- Jede Klasse von Typen, die den Vatertyp beinhaltet, beinhaltet auch den abgeleiteten Typ.
- Wenn der Vatertyp ein elementarer Typ oder ein Array-Typ ist, dann ist die Menge der Werte des abgeleiteten Typs eine Kopie der möglichen Werte des Vaternotyps. Für einen skalaren Typ ist der zugrunde liegende Bereich (range) des abgeleiteten Typs derselbe wie der des Vaternotyps.
- Wenn der Vatertyp ein zusammengesetzter Typ, aber kein Array ist, dann sehen die Komponenten und Operationen, die für den abgeleiteten Typ deklariert sind, folgendermaßen aus:
 - Wenn im Zuge der Ableitung eine Diskriminante festgelegt wird, dann ist diese Diskriminante eine Komponente des abgeleiteten Typs; sonst alle im Vatertyp festgelegten Diskriminanten; im letzteren Fall spricht man davon, daß die Diskriminanten geerbt wurden.
 - Alle Komponenten (keine Diskriminanten) und Operationen des Vaternotyps – man sagt dann, diese Komponenten wurden geerbt. Dabei werden bei den Operationen alle Parameter des Vaternotyps systematisch durch Parameter des abgeleiteten Typs ersetzt[‡].
 - Alle Komponenten, die im Erweiterungsteil festgelegt wurden.
- Der abgeleitete Typ ist limitiert genau dann, wenn der Vatertyp limitiert ist.
- Für jede vordefinierte Operation des Vaternotyps gibt es eine vordefinierte Operation des abgeleiteten Typs.

Man sagt ein abgeleiteter Typ ist *direkt* von seinem Vatertyp abgeleitet; er ist *indirekt* abgeleitet von jedem Typ, von dem sein Vatertyp abgeleitet ist. Durch Ableitung bildet sich somit ein *Ableitungsbaum* von getaggtten Typen. Man nennt so einen Ableitungsbaum mit Wurzel T eine *Ableitungsklasse von Typen für den Typ T*.

Klassenumfassende Typen (auch *klassenweite Typen* genannt) sind für alle Ableitungsklassen definiert, deren Wurzel ein getaggtter Typ ist. Man schreibt T'CLASS, um den klassenumfassenden Typ T zu bezeichnen. Der Name leitet sich her aus der Tatsache, daß ein formaler Parameter, dessen Typ T'CLASS ist, als aktuellen Wert ein Objekt eines beliebigen Typs aus der Ableitungsklasse von T annehmen kann. Wichtig ist, daß Operationen, die einen klassenweiten Typ als Parameter besitzen, keine primitiven Operationen sind.

Die Operationen, die auf einem getaggtten Typ definiert sind, nennt man *Dispatching-Operation*. Manchmal ist es möglich, zur Compile-Zeit zu entscheiden, welche Operation eines getaggtten Typs ausgeführt werden soll; meist jedoch kann diese Entscheidung erst zur Laufzeit gefällt werden. Das zur Laufzeit mit jedem Objekt eines getaggtten Typs mitgeführte Tag ermöglicht es,

[‡]Genauere Details kann man im Reference Manual [Ada95] nachlesen.

diese Entscheidung zu treffen[‡]. Dabei ist vor allem wichtig, daß eine Operation nicht von mehr als einem getaggten Typ abhängig sein darf.

Um Fehlern vorzubeugen, wurde mit Ada2005 die Möglichkeit geschaffen, Dispatching-Operationen als overriding und als not overriding zu klassifizieren.

Für alle Subtypen S eines getaggtten Typs T sind folgende Attribute definiert:

S'CLASS siehe oben

S'TAG gibt das Tag von T zurück

Vorgriff: *Objekte des Typs Tag können mittels des Packages Ada.Tags bearbeitet werden.*

```
package Ada.Tags is
  type Tag is private;

  function Expanded_Name(T: Tag) return string;
  function External_Tag(T: Tag) return string;
  function Internal_Tag(External: string) return Tag;

  Tag_Error: exception;
private
  ...
end Ada.Tags;
```

Beispiele für getaggte Record-Typen sind:

```
type point is tagged
  record
    x,y: real := 0.0;
  end record;
```

```
type expression is tagged null record;
  - hier wird erwartet, dass die Typerweiterungen Komponenten hinzufuegen
```

2.7.1 Typerweiterungen

Die Syntax von Typerweiterungen wird anhand einiger Beispiele illustriert:

```
type painted_point is new point with
  record
    paint: color := white;
  end record;
origin: constant painted_point := (x|y => 0.0, paint => black);
```

```
type literal is new expression with
  record
    value: real;
  end record;
```

```
type expr_ptr is access all expression'CLASS;
```

```
type binary_operation is new expression with
```

[‡]Details können im Reference Manual [Ada95] nachgelesen werden.

```
record
  left, right: expr_ptr;
end record;
```

```
type addition is new binary_operation with null record;
type subtraction is new binary_operation with null record;
```

Es ist immer möglich, einen abgeleiteten Typ auf seinen Vaternotyp zu konvertieren. Dies geschieht etwa durch `p := point(origin);`

2.7.2 Erweiterungsaggregate

Es gibt auch die Möglichkeit, für Typerweiterungen Aggregate zu verwenden. Beispiele sind:

```
painted_point'(point with red)
(point'(p) with paint => black)
```

2.7.3 Abstrakte Typen

Ein *abstrakter Typ* ist ein Typ, für den es *keine* Objekte dieses Typs geben darf.

Ein abstrakter Typ wird deklariert, indem das reservierte Wort `abstract` in seiner Deklaration verwendet wird. Nur getaggte Typen dürfen abstrakt sein!

Ein einfaches Beispiel ist:

```
type set is abstract tagged null record;
```

2.7.4 Access-Diskriminanten

Dieser Abschnitt versteht sich als Nachtrag zum Thema diskriminierte Records und Pointer.

Es ist nämlich auch möglich, einen Pointer-Typ als Diskriminante zu verwenden. Man kann damit einen Typ mit einem Verweis auf ein Objekt eines anderen Typs parametrieren. Allerdings muß der diskriminierte Typ ein limitierter Typ sein.

Diskriminierte Records bringen vor allem Vorteile bei der objektorientierten Programmierung und werden in Kapitel 13 eingehender erläutert.

3 Ausdrücke

ein Finger weist zum Mond

Setz den Ausdruck
ein Finger weist zum Mond, in Klammern
(ein Finger weist zum Mond)

Die Aussage:
'Ein Finger weist zum Mond ist in Klammern'
ist ein Versuch zu sagen, daß alles was in Klammern ist
(
im Verhältnis steht zu dem, was nicht in Klammern ist
wie ein Finger zum Mond
Setz alle möglichen Ausdrücke in Klammern
Setz alle möglichen Formen in Klammern
und setz die Klammern in Klammern
Jeder Ausdruck und jede Form
steht im Verhältnis zu dem, was ausdruckslos und formlos ist
wie ein Finger zum Mond
alle Ausdrücke und alle Formen
weisen zum Ausdruckslosen und Formlosen

die Behauptung
'Alle Formen weisen zum Formlosen'
ist selbst eine formale Behauptung

aus Ronald D. Laing, „Knoten“.

3.1 Operatoren

Ada kennt verschiedene *Operatoren* zum Aufbau von Ausdrücken. Es sind dies (die Priorität nimmt mit den Zeilen ab):

	** (<i>Potenzieren</i>) abs (<i>Absolutbetrag</i>) not
	* / mod (<i>Modulo-Operation</i>) rem (<i>Rest bei ganzzahliger Division</i>)
unäre Operatoren	+ -
binäre Operatoren	+ - &
Vergleichsoperatoren	= /= < <= > >=
logische Operatoren	and or xor

Zusätzlich gibt es noch die sogenannten *Kurzauswertungen* (*short-circuit expressions*) **and then** und **or else**, die dieselbe Priorität wie die logischen Operatoren, und die *Element-tests* (*membership tests*) **in** und **not in**, die dieselbe Priorität wie die Vergleichsoperatoren haben.

3.2 Auswertung von Ausdrücken

Die Auswertung von Ausdrücken, die die oben genannten Operatoren enthalten, geschieht so, daß zuerst jene Operanden verknüpft werden, deren Verknüpfungsoperator die höhere Priorität besitzt. Bei gleicher Priorität erfolgt die Auswertung von links nach rechts. Abweichungen von dieser Regel kann man mittels runder Klammern erzwingen.

Wenn bei der Auswertung eines Ausdrucks irgendein Fehler, wie z.B. Division durch Null, auftritt oder wenn das Resultat nicht innerhalb des geforderten Bereiches liegt, wird die Fehlermeldung `constraint_error` ausgelöst.

3.3 Kurzauswertungen

Technokratus: *Wenn du die Produktion nicht steigern kannst, befriedigt das Angebot nicht mehr die Nachfrage, was sich negativ auf den Kurs auswirken könnte.*

Obelix: *Häh?*

Technokratus: *Wenn-du-nicht-können-machen-mehr-Hinkelsteine-ich-dir-geben-weniger-Sesterze. Klar?*

Renè Goscinny und Albert Uderzo, „Obelix GmbH & Co.KG“.

Kurzauswertungen funktionieren wie folgt: In der Verknüpfung der booleschen Ausdrücke A und B in der Form A **and** then B wird der Ausdruck B nur dann ausgewertet, wenn die Auswertung von A `true` ergeben hat, und der Ausdruck B in A **or else** B wird nur dann ausgewertet, wenn die Auswertung von A `false` ergeben hat. Falls der rechte Ausdruck nicht ausgewertet wird, ist das Gesamtergebnis das Resultat des linken Ausdrucks, anderenfalls das des rechten.

3.4 Logische Operatoren

Die *logischen Operatoren* sind nicht nur für boolesche Operanden definiert, sondern auch für eindimensionale Arrays, deren Komponenten den Typ `boolean` haben. In diesem Fall wird die Operation komponentenweise durchgeführt und die Grenzen des Ergebnis-Arrays sind jene des linken Operanden. Entsprechendes gilt für den unären Operator `not`.

3.5 Vergleichsoperatoren

Kommt zurück! Wir sind mehr als die!
Kommt doch endlich zurück, beim Jupiter!
Die sind zahlreicher als ich!

Legat Volfgangamadeus.
Renè Goscinny und Albert Uderzo,
„Asterix bei den Belgiern“.

Die Vergleichsoperatoren `=` und `/=` sind für alle Typen außer `limited` definiert, die restlichen, die sogenannten *Ordnungsoperatoren*, sind nur für skalare Typen und eindimensionale Array-Typen mit skalaren Komponenten erlaubt.

Zwei Records sind dann gleich, wenn alle ihre Komponenten gleich sind, zwei Arrays sind es dann, wenn die entsprechenden Komponenten gleich sind und sie gleiche Länge haben, die Bereiche der Indizes sind dabei irrelevant. Ein Beispiel ist:

```
str_1: string (1 .. 3) := "AHA";
str_2: string (1 .. 5) := "AHAHA";
```

Dabei gilt dann: `str_1 = str_2(3 .. 5)`.

Die Ordnungsoperatoren für Arrays entsprechen der *lexikographischen Ordnung*. Das heißt, ein Array der Länge 0 ist kleiner als jedes Array mit einer größeren Länge. Für Arrays der Länge > 0 gilt: Der linke Operand ist lexikographisch kleiner der rechte, wenn entweder die erste Komponente des linken Operanden kleiner ist als die des rechten, oder, falls die ersten Komponenten übereinstimmen, der restliche linke Operand lexikographisch kleiner ist als der restliche rechte.

3.6 Elementtests

Die *Elementtests* sind für alle Typen vordefiniert. Beispiele sind:

```
N not in 1 .. 10
Heute in Montag .. Freitag
Heute in Arbeitstag                -- Elementtest fuer einen Subtyp
```

3.7 Operationen für ganze Zahlen

Die *ganzzahlige Division* und deren *Rest* ist definiert durch:

$$A = (A/B) * B + (A \text{ rem } B),$$

wobei $(A \text{ rem } B)$ das Vorzeichen von A und einen Absolutbetrag kleiner als der Absolutbetrag von B hat. Außerdem gilt

$$(-A)/B = -(A/B) = A/(-B).$$

Das Resultat der *Modulo-Operation* ist so beschaffen, daß $(A \bmod B)$ das Vorzeichen von B und einen Absolutbetrag kleiner als der Absolutbetrag von B hat. Zusätzlich gilt für eine (nicht näher bestimmte) ganze Zahl N :

$$A = B * N + (A \bmod B)$$

Die Unterscheidung zwischen `rem` und `mod` scheint künstlich, man benötigt jedoch in manchen Anwendungen beide.

3.8 Konkatination

Der binäre Operator `&` hängt zwei eindimensionale Arrays zusammen, im speziellen zwei Strings. Die untere Grenze des Ergebnisses ist die des linken Operanden, außer wenn dieser ein Null-Array ist, dann ist die untere Grenze des Resultats die des rechten Operanden.

3.9 Potenzieren

Die Potenzierung `**` ist für alle Integer-Typen und für alle Gleitkommazahlen definiert. In beiden Fällen muß aber der rechte Operand, der Exponent, vom Typ `integer` sein. Der Typ des Resultats ist gleich dem Typ des linken Operanden. Wenn man eine ganze Zahl mit einem negativen Exponenten verknüpft, wird die Fehlermeldung `constraint_error` ausgelöst.

Für dezimale Typen ist die Potenzierung nicht definiert.

4 Anweisungen

Nachdem wir in den bisherigen Kapiteln vor allem passive Programm-Elemente, wie Datenstrukturen und Ausdrücke, behandelt haben, werden wir uns nun aktiveren Teilen von Ada-Programmen zuwenden, den Anweisungen.

Als generelle Aussage, die für alle Anweisungen in Ada gilt, kann festgestellt werden, daß jede Anweisung in Ada mit einem Strichpunkt abgeschlossen wird.

4.1 Die Null-Anweisung

Die wohl einfachste Anweisung ist die *Null-Anweisung*. Sie ist definiert durch:

```
null;
```

und tut schlichtweg nichts. (Das braucht man manchmal und sei's nur zur vorübergehenden Implementation eines Dummy's).

4.2 Die Zuweisung

Als nächstes behandeln wir die *Zuweisung* (*assignment statement*). Sie dient dazu, einer Variablen einen neuen Wert zu geben. Dabei ist zu beachten, daß die Variable und der Ausdruck auf der rechten Seite denselben Typ haben. Dieser Typ darf kein limitierter Typ sein. Einfache Beispiele sind:

```
Wochentag := Freitag;
i := i*i + 1;
```

Sollte bei einer Zuweisung eine Verletzung des definierten Bereiches auftreten, so wird die Fehlermeldung `constraint_error` ausgelöst.

4.3 Die If-Anweisung

```
A FOISCH GEBBIS
in an wossaglas!
      kaunsd
mid an goidfisch
      ned fagleichen
```

aus Ernst Kein, „Wiener Grottenbahn“.

Eine *If-Anweisung* beeinflusst den Kontrollfluß des Programmes abhängig von einem logischen Ausdruck. Ada kennt außer der üblichen *If-then-else-Struktur* auch noch ein `elsif`. Eine typische If-Anweisung in Ada hat etwa folgendes Aussehen:

```
if <logischer_Ausdruck> then
  <Anweisungen>
elsif <logischer_Ausdruck> then
  <Anweisungen>
else
  <Anweisungen>
end if;
```

Dabei können 0 oder beliebig viele `Elsif-Zweige` vorhanden sein und der `Else-Zweig` kann, wenn nötig, ganz entfallen.

4.4 Die Case-Anweisung

Ein ähnliches Konstrukt ist uns schon bei der Definition von Varianten-Records begegnet. Eine typische Struktur in Ada ist etwa folgende:

```
case <Ausdruck> is
  when <Element_oder_Bereich_1> | <Element_oder_Bereich_2> =>
    <Anweisungen>
  when <Element_oder_Bereich_3> | <Element_oder_Bereich_4> =>
    <Anweisungen>
  when <Element_oder_Bereich_5> =>
    <Anweisungen>
end case;
```

Dabei können beliebig viele `<Element_oder_Bereich_x>`-Teile getrennt durch ein `|` hinter einem `when` stehen, es muß jedoch mindestens ein solcher Teil folgen. Ein `<Element_oder_Bereich_x>`-Teil kann dabei ein passender Ausdruck, ein Wert oder ein Bereich des durch `<Ausdruck>` definierten Typs sein. Als letzte Alternative darf auch `others` stehen.

4.5 Die Exit-Anweisung

Ada kennt eine bedingte und eine unbedingte *Exit-Anweisung*, um Schleifen zu verlassen. Letztere lautet:

```
exit;
```

Eine bedingte Exit-Anweisung sieht so aus:

```
exit when <logischer_Ausdruck>;
```

In diesem Fall wird die Schleife verlassen, wenn der logische Ausdruck wahr ist. Optional kann man unmittelbar anschließend an `exit` auch noch einen Schleifennamen angeben (siehe auch 4.6).

4.6 Die Loop-Anweisung

In Ada unterscheidet man drei verschiedene Arten von *Schleifen-Anweisungen*:

1. die Loop-Anweisung ohne Iterations-Schema

2. die Loop-Anweisung mit einem While-Schema
3. die Loop-Anweisung mit einem For-Schema

Zusätzlich kann man eine Schleifen-Anweisung auch noch benennen.

4.6.1 Die Loop-Anweisung ohne Iterations-Schema

Eine typische Loop-Anweisung ohne Iterations-Schema hat in Ada folgendes Aussehen:

```
loop
  <Anweisungen>
  exit when <logischer_Ausdruck>;
  <Anweisungen>
end loop;
```

Die Schleife wird verlassen, wenn bei der Abarbeitung der Exit-when-Anweisung der angegebene logische Ausdruck wahr ist.

4.6.2 Die Loop-Anweisung mit einem While-Schema

Eine solche Schleife sieht so aus:

```
while <logischer_Ausdruck> loop
  <Anweisungen>
end loop;
```

Die Schleife wird so lange ausgeführt, wie der logische Ausdruck wahr ist, d.h. gegebenenfalls auch gar nicht. Man kann eine While-Schleife auch mit einer Exit-Anweisung verlassen.

4.6.3 Die Loop-Anweisung mit einem For-Schema

Eine *For-Schleife* hat folgendes Aussehen:

```
for <index_variable> in <Bereich> loop
  <Anweisungen>
end loop;
```

Die *Index-Variable* muß nicht vorher deklariert werden, ihr Gültigkeitsbereich liegt innerhalb der durch die Loop-Anweisung definierte Klammer und ihr Wert ist nach Ausführung der Loop-Anweisung undefiniert. Als Bereich kann, wie bereits weiter vorne angekündigt, auch das Range-Attribut verwendet werden. Um die Reihenfolge des Bereichs umzukehren, kann das Schlüsselwort *reverse* unmittelbar vor dem Bereich angegeben werden. Auch eine For-Schleife kann man mit einer Exit-Anweisung verlassen.

4.6.4 Benannte Loop-Anweisungen

Man kann Schleifen auch benennen. Dies sieht beispielsweise so aus:

```
Aeussere_Schleife:
loop
  ...
  Innere_Schleife:
  loop
    ...
    exit Innere_Schleife when a=b;
    ...
    exit Aeussere_Schleife when a=c;
    ...
  end loop Innere_Schleife;
  ...
end loop Aeussere_Schleife;
```

Durch die Benennung der Schleifen und Benutzung dieser Namen bei den Exit-Anweisungen kann man genau spezifizieren, welche Schleife man verlassen will.

4.7 Die Block-Anweisung

She messed around with a block named Smokey.

Cab Calloway, „Minnie, the Moocher“.

Ein Beispiel für einen *Block* ist:

```
Smokey:
declare
  - hier kommen Typ-, Konstanten- und Variablendeklarationen
begin
  <Anweisungen>
end Smokey;
```

Dabei kann der Name auch entfallen:

```
declare
  - hier kommen Typ-, Konstanten- und Variablendeklarationen
begin
  <Anweisungen>
end;
```

4.8 Die Return-Anweisung

Hier ist wieder ein kleiner Vorgriff vonnöten: Die *Return-Anweisung* dient dazu, Unterprogramme zu verlassen. Eine Prozedur kann durch ein simples

```
return;
```

verlassen werden, eine Funktion wird durch

```
return <Ausdruck>;
```

verlassen, wobei der Ausdruck ausgewertet und an den Aufrufer übergeben wird (siehe auch Kapitel 5).

4.9 Die Goto-Anweisung

In Ada ist die Definition von *Sprungzielen* (*labels*) vor jeder Anweisung erlaubt; sie sieht folgendermaßen aus:

«Hier»

wobei die doppelten spitzen Klammern Teil der Syntax sind.

Eine *Sprung-Anweisung* hat folgendes Aussehen:

```
goto Hier;
```

5 Unterprogramme

Same procedure as every year, James!

Dinner for One. Freddie Frinton

In Ada gibt es zwei Arten von *Unterprogrammen*: *Prozeduren* und *Funktionen*. Der Aufruf einer Prozedur ist eine Anweisung; der Aufruf einer Funktion ist ein Ausdruck und liefert einen Wert zurück. Unterprogramme können aus zwei Teilen bestehen, einer *Unterprogramm-Deklaration*, die den Namen des Unterprogrammes, die Parameter und im Falle einer Funktion den Typ des Rückgabewertes festlegt, und einer *Unterprogramm-Implementation (body)*, die den funktionellen Ablauf des Unterprogrammes bestimmt. Prinzipiell sind alle Unterprogramme *reentrant* und können *rekursiv* verwendet werden.

5.1 Deklaration eines Unterprogrammes

Eine Prozedur wird deklariert durch:

```
procedure <Name>(<formale.Parameter>);
```

Eine Funktion hingegen wird festgelegt durch:

```
function <Name>(<formale.Parameter>) return <Typ>;
```

Als Rückgabewert einer Funktion kann jeder beliebige vordefinierte oder vom Benutzer definierte Typ verwendet werden. (Pascal und Modula-2 erlauben nur Rückgabewerte, die nicht größer sind als ein Maschinen-Wort.)

5.2 Formale Parameter

Es gibt drei Arten von *Parameterübergabe-Mechanismen*: *in*, *out* und *in out*. Bei *In-Parametern* wird der Wert des aktuellen Parameters beim Aufruf an das Unterprogramm übergeben. Innerhalb des Unterprogrammes verhält sich ein In-Parameter wie eine Konstante. *Out-Parameter* werden vom Unterprogramm mit einem Wert versorgt, der an den aufrufenden Agenden zurückgegeben wird. Ein Out-Parameter kann innerhalb eines Unterprogrammes sowohl geschrieben als auch gelesen werden, allerdings ist er zu Beginn nicht initialisiert. Der Wert eines *In-Out-Parameters* wird beim Aufruf an das Unterprogramm übergeben und bei der Rückkehr an den aufrufenden Agenden zurückgegeben. Ein In-Out-Parameter verhält sich innerhalb eines Unterprogrammes wie eine initialisierte Variable.

Ein formaler In-Parameter kann mit einer Access-Definition versehen werden. Man spricht dann von *Access-Parametern*.

Gibt man keines der Schlüsselwörter *in*, *out* oder *in out* an, so wird defaultmäßig angenommen, daß der betroffene Parameter ein In-Parameter ist.

Funktionen dürfen nur In-Parameter haben um Nebeneffekte zu vermeiden.

Unconstrained Arrays als formale Parameter erben ihre Grenzen beim Aufruf vom aktuellen Parameter. Dasselbe gilt für die Diskriminanten von unconstrained Records.

Bei einer Unterprogramm-Deklaration werden die formalen Parameter definiert, indem die Bezeichner der Parameter getrennt durch einen Doppelpunkt von ihrem Typ aufgelistet werden. Einzelne Parameter-Definitionen werden durch einen Strichpunkt voneinander getrennt. Parameter gleichen Typs können, bevor ihr Typ spezifiziert wird, mittels Beistrich getrennt aufgelistet werden. Ein einfaches Beispiel ist:

```
function Beispiel_1(
  integer_par_1, integer_par_2: integer;
  real_par: real;
  string_par: string)
  return boolean;
-- unconstrained Parameter
```

Es ist auch möglich, *Default-Werte* für Parameter anzugeben, z.B.:

```
procedure Beispiel_2(
  integer_par_1, integer_par_2: integer;
  real_par: real := 0.0;
  string_par: string := "");
-- 0.0 als Default
-- Leerstring als Default
```

Parameter, die bei der Definition mit einem Default-Wert versehen worden sind, müssen beim Aufruf des entsprechenden Unterprogrammes nicht versorgt werden.

Bezüglich des wirklichen Übergabemodus ist zu sagen, daß

- elementare Typen und private Typen, die nur aus elementaren Typen bestehen, als Wert-Parameter übergeben werden,
- getaggte Typen, Task-Typen, geschützte Typen* und nicht private limitierte Typen *by reference* übergeben werden¹ und
- für alle restlichen Typen im Sprachstandard keine Festlegung erfolgt, hier also dem Compiler die Entscheidung überlassen wird, welchen Übergabemodus er wählt.

Access-Parameter verhalten sich sehr ähnlich zu In-Out-Parametern. Es gibt allerdings wesentliche Unterschiede, die wir anhand eines Beispiels studieren wollen.

```
type T is tagged
  record ...

type Access_T is access T;

Obj: T;

Obj_Ptr: Access_T := new T(...);

procedure P(X: in out T);

procedure PA(XA: access T);
```

*Ein kleiner Vorgriff sei uns der Einfachheit halber gestattet.

¹Dasselbe gilt für zusammengesetzte Typen und private Typen, von denen mindestens ein Bestandteil *by reference* übergeben wird.

Innerhalb der Bodies von P und PA haben wir lesenden und schreibenden Zugriff zu den Komponenten des Records. Aufgrund der automatischen Dereferenzierung syntaktisch sogar auf dieselbe Weise. Und da getaggte Typen *by reference* übergeben werden, haben wir auch denselben Effekt in beiden Fällen. Zum Beispiel kann in beiden Fällen Dispatching stattfinden. Nun aber zu den Unterschieden.

Im Falle des In-Out-Parameters muß der aktuelle Parameter Obj oder Obj_Ptr.all sein. Daher

```
P(Obj); P(Obj_Ptr.all);
```

Im anderen Fall muß der aktuelle Parameter Obj'ACCESS oder Obj_Ptr sein. Außerdem muß die Variable Obj als aliased vermerkt sein.

```
Obj: aliased T;
PA(Obj'ACCESS); PA(Obj_Ptr);
```

Ein wesentlicher Unterschied ist nun, daß ein Access-Parameter nie null sein kann. Das wird vom Laufzeitsystem geprüft und man kann sich diesen Test innerhalb des Unterprogrammes also ersparen.

Außerdem sei noch einmal darauf hingewiesen, daß Funktionen keine In-Out-Parameter haben dürfen. In solchen Fällen ist also ein Access-Parameter sehr sinnvoll, besonders dann, wenn man keine Prozedur mit Out-Parameter verwenden kann, weil etwa der Typ des Resultats klassenweit oder sonst irgendwie unconstrained ist.

5.3 Aufruf von Unterprogrammen

Beim Aufruf eines Unterprogrammes müssen alle Parameter versorgt werden, die bei der Definition keinen Default-Wert bekommen haben, alle anderen können, müssen aber nicht, versorgt werden. Auch hier unterscheidet man, ähnlich wie bei den Aggregaten (siehe 2.2.3) zwischen Ansprechen durch *Position* und durch *Name*. Wieder gilt, daß, wenn einmal ein Parameter durch seinen Namen angesprochen worden ist, alle folgenden Parameter mit Namen angesprochen werden müssen. *Aktuelle Parameter* werden mittels Beistrich voneinander getrennt. Beispiele sind

```
Beispiel_2(1, 2);           -- Default-Werte fuer die restlichen Parameter
Beispiel_2(1, 2, string_par => "aha");
Beispiel_2(string_par => "aha", integer_par_2 => 2, integer_par_1 => 1);
Beispiel_2(1, 2, 3.0, "4");
```

5.4 Rückkehr aus Unterprogrammen

Die Rückkehr aus einer Prozedur erfolgt entweder dadurch, daß das Ende des Prozedur-Codes erreicht wird, oder durch eine *Return-Anweisung* an beliebiger Stelle innerhalb des Unterprogramm-Codes. Es kann auch mehrere Return-Anweisungen in einer Prozedur geben.

Die Rückkehr aus einer Funktion muß durch eine *Return-Anweisung* erfolgen, der der Rückgabewert mitgegeben wird, z.B.:

```
return true;
```

Es können mehr als eine einzige Return-Anweisung im Funktions-Code vorhanden sein. Wird jedoch eine Funktion dadurch verlassen, daß das Ende ihres Codes ohne eine Return-Anweisung erreicht wird, so wird die Fehlermeldung `program_error` ausgelöst.

Seit 2005 gibt es auch eine *erweiterte Return-Anweisung*.

So eine erweiterte Return-Anweisung kann etwa wie folgt aussehen:

```
return R: T := E do
  if ... then
    ...
    return;           -- gibt R zurück
  end if;
  ...
end return;
```

Es sind also beliebige Kontrollstrukturen innerhalb so einer Anweisung erlaubt. Ein darin vorkommendes return darf allerdings keinen Ausdruck haben, da es ja den Wert, den die angegebene Variable der erweiterten Return-Anweisung hat, zurückgibt.

Vorläufig können wir die erweiterte Return-Anweisung als alternative Syntax betrachten; sie erhält allerdings später, wie wir noch sehen werden, mehr Sinn.

5.5 Unterprogramm-Implementation

Bei der Implementation eines Unterprogrammes wird zuerst seine Deklaration ohne den letzten Strichpunkt wiederholt, dann kommt das Schlüsselwort `is` und abschließend folgt im wesentlichen eine Block-Anweisung, der nur das Schlüsselwort `declare` fehlt. Dem abschließenden `end` des Implementierungs-Blocks kann noch der Name des Unterprogrammes folgen, z.B.:

```
function Beispiel_1(
  integer_par_1, integer_par_2: integer;
  real_par: real;
  string_par: string)
return boolean is
  Zwischenwert: integer;
begin
  Zwischenwert := integer_par_1 + integer_par_2;
  if real_par > 0.0 then
    return true;
  elsif integer'IMAGE(Zwischenwert) = string_par then
    return false;
  else
    return integer_par_1 = integer_par_2;
  end if;
end Beispiel_1;
```

Abschließend soll noch bemerkt werden, daß die Deklaration eines Unterprogrammes auch weggelassen werden kann; wir werden aber noch sehen, wozu die Trennung von Deklaration und Implementation gut sein kann.

5.6 Das Parametertyp-Profil

Man sagt, zwei Unterprogramme haben dasselbe *Parametertyp-Profil* genau dann, wenn sie dieselbe Anzahl von Parametern haben und wenn die Parameter auf entsprechenden Positionen denselben Typ haben. Falls es sich um Funktionen handelt, muß auch der Typ des Rückgabewertes bei beiden derselbe sein.

5.6.1 Pointer auf Unterprogramme

Unterprogramme, die dafür geeignet sein sollen, daß ein Pointer darauf erlaubt ist, müssen das selbe Parametertyp-Profil haben wie das bei der Spezifikation des Pointers angegebene Profil (vgl. 2.4.3). Ein einfaches Beispiel ist

```
type Integrand is access function(X: Float_Type) return Float_Type;

function Integrate(
  F: Integrand;
  ...)
  return Float_Type;
```

5.6.2 Überladen von Unterprogrammen

Solange zwei Unterprogramme unterschiedliche Parametertyp-Profile haben, können sie denselben Namen haben. In diesem Fall spricht man von *Überladen* von Unterprogrammen.

5.6.3 Überladen von Operatoren

Ada erlaubt auch, daß Funktionen deklariert werden, deren Bezeichner mit einem Operator übereinstimmt, z.B.:

```
function "+" (left, right: Vektor)
  return Vektor;
```

Dabei muß der Bezeichner des Operators zwischen doppelten Hochkommata eingeschlossen werden. Als Operator-Bezeichner kommen alle Operatoren in Frage, die wir im Kapitel 3 kennengelernt haben.

Unter der Annahme, daß A, B und C Variablen vom selben (constrained) Typ Vektor sind, sind die folgenden Verwendungen gleichbedeutend:

```
A := "+"(left => B, right => C);
A := "+"(B, C);
A := B + C;
```

Der Gleichheitsoperator = kann für limitierte Typen explizit deklariert werden, sofern die beiden Parameter vom selben limitierten Typ sind und der Rückgabewert den Typ boolean hat. Dadurch wird implizit auch der Operator /= neu definiert, nämlich als not =. Explizites Überladen von /= ist verboten.

5.7 Abstrakte Unterprogramme

Abstrakte Unterprogramme treten zusammen mit abstrakten Typen (vgl. 2.7.3) auf. Ein abstraktes Unterprogramm ist ein Unterprogramm, das keinen Implementationsteil besitzt, von dem aber erwartet wird, das es „weiter unten“ im Ableitungsbaum einen solchen besitzen wird¹.

Wenn der Vaternotyp eines abgeleiteten Typs ein abstraktes Unterprogramm „besitzt“, dann

¹Vgl. die virtuellen Funktionen in C++ [Str86].

- muß das abgeleitete Unterprogramm abstrakt sein, wenn der abgeleitete Typ abstrakt oder nicht getaggt ist, oder
- es muß einen Body besitzen, d.h., es darf nicht abstrakt sein.

Beispiele für ein abstrakte Unterprogramme sind:

```
function is_empty(s: set) return boolean is abstract;
function union(left, right: set) return set is abstract;
procedure take(el: out element; from: in out set) is abstract;
```


6 Exceptions

Bisher haben wir uns um gewisse Dinge, die Programmierern wohlbekannt sind, geschickt herumgeschickt. Wir haben bislang, wenn es nötig war, von „Fehlermeldungen“ gesprochen, die das Ada-Laufzeitsystem auslöst, falls bestimmte Fehler aufgetreten sind. Wir erinnern uns z.B. daran, daß die Fehlermeldung `constraint_error` ausgelöst wird, wenn ein Null-Pointer dereferenziert wird. Dies sind Indizien, daß Ada eine vordefinierte Menge von solchen Fehlermeldungen kennt. In der Tat ist das so. Ada nennt solche Fehlermeldungen *Exceptions*. Darüberhinaus kann man in Ada aber auch als Anwender Exceptions deklarieren, diese im Programm auslösen und gegebenenfalls auf das Auftreten von Exceptions geeignet reagieren.

Die von der Sprache vordefinierten Exceptions sind:

```
constraint_error
program_error
storage_error
tasking_error
```

Einige davon kennen wir schon, andere werden wir noch kennenlernen. Es sei hier nur erwähnt, daß die Exception `storage_error` ausgelöst wird, wenn eine Operation, wie etwa `new`, zu wenig Speicherplatz vorfindet, um korrekt ablaufen zu können.

6.1 Die Deklaration von Exceptions

Exceptions können überall dort definiert werden, wo Typen, Konstanten oder Variablen deklariert werden können, z.B.:

```
zu_wenig_Speicher: exception;
Stack_list_leer: exception;
```

6.2 Das Auslösen von Exceptions

Exceptions werden im Programm mit der *Raise-Anweisung* ausgelöst, z.B.:

```
raise zu_wenig_Speicher;
raise Stack_list_leer;
raise constraint_error;
```

6.3 Die Behandlung von Exceptions

Eine vordefinierte Exception, die vom Laufzeitsystem ausgelöst worden ist, oder eine Exception, die mittels einer *Raise-Anweisung* ausgelöst worden ist, kann mit einem sogenannten *Exception-Handler* „gefangen“ werden. So ein Exception-Handler kann am Ende einer Block-Anweisung oder am Ende eines Unterprogrammes stehen. Eine derart gestaltete Block-Anweisung hat typischer Weise folgende Struktur

```
begin
  <Anweisungen>
exception
  when E.Bezeichner: Exception_1 | Exception_2 =>
    <Anweisungen>
  when Exception_3 =>
    <Anweisungen>
end;
```

Als letzte Alternative darf wieder `others` stehen.

Der oben angeführte *E.Bezeichner* ist optional und darf einmal pro *When-Anweisung* stehen. Er bezeichnet innerhalb der *When-Anweisung* die gerade behandelte Exception.

Vorgriff: *Der Exception-Bezeichner kann dazu verwendet werden, geeignete Operationen mit der behandelten Exception durchzuführen. Dazu gibt es ein passendes vordefiniertes Paket namens Ada.Exceptions:*

```
package Ada.Exceptions is
  type Exception_Id is private;
  Null_Id : constant Exception_Id;
  function Exception_Name(Id : Exception_Id) return String;

  type Exception_Occurrence is limited private;
  type Exception_Occurrence_Access is access all Exception_Occurrence;
  Null_Occurrence : constant Exception_Occurrence;

  procedure Raise_Exception(E : in Exception_Id; Message : in String := "");
  function Exception_Message(X : Exception_Occurrence) return String;
  procedure Reraise_Occurrence(X : in Exception_Occurrence);

  function Exception_Identity(X : Exception_Occurrence) return Exception_Id;
  function Exception_Name(X : Exception_Occurrence) return String;
  -- Same as Exception_Name(Exception_Identity(X)).
  function Exception_Information(X : Exception_Occurrence) return String;

  procedure Save_Occurrence(Target : out Exception_Occurrence;
                           Source : in Exception_Occurrence);
  function Save_Occurrence(Source : Exception_Occurrence)
    return Exception_Occurrence_Access;

private
  ... -- not specified by the language
end Ada.Exceptions;
```

Der Exception-Bezeichner ist in diesem Sinn vom Typ Exception_Occurrence.

Statt des Aufrufs

```
Raise_Exception(Problem'Identity, "Houston, wir haben ein Problem!");
```

kann man seit 2005 auch

```
raise Problem with "Houston, wir haben ein Problem!";
```

schreiben.

In einem Exception-Handler wird eine dort angeführte Exception genau dann gefangen, wenn sie innerhalb des darüber befindlichen Blocks ausgelöst wird. Mit `others` kann man alle anderen Exceptions, auch unbekannte, fangen.

Eine Exception, die in einem Unterprogramm nicht gefangen wird, wird an der Stelle wieder ausgelöst, wo dieses Unterprogramm aufgerufen wurde, eine Exception, die innerhalb eines Blocks nicht gefangen wird, wird am Ende des Blocks wieder ausgelöst.

Man kann natürlich auch innerhalb eines Exception-Handlers eine Exception mit einer Raise-Anweisung auslösen. Will man dieselbe Exception wieder auslösen, die man gerade gefangen hat, so genügt es

```
raise;
```

zu verwenden. Dies ist vor allem für Exception-Handler, die auf mehrere Exceptions gleich reagieren sollen, und für alle mit others gefangenen Exceptions sinnvoll. Mit dieser Art, Exception-Handler zu programmieren, kann man, insbesondere beim Auftreten von „unvorhergesehenen“ Fehlern, bevor die entsprechende Exception mit raise weitergereicht wird, noch „Clean up-Operationen“ durchführen.

7 Pakete

*Gebt ihr ein Stück, so gebt es gleich in Stücken!
Solch ein Ragout es muß euch glücken.*

Theaterdirektor.
Vorspiel auf dem Theater.
Johann Wolfgang von Goethe, „Faust“.
Der Tragödie erster Teil.

Module – in Ada *Pakete* (*packages*) genannt – erlauben, Unterprogramme auf eine vernünftige Art und Weise zu gruppieren. Es ist auch möglich, Pakete innerhalb von Paketen oder innerhalb von Unterprogrammen zu deklarieren. Ein Paket besteht aus einem *Spezifikations-* und einem *Implementationsteil* (*package specification* und *package body*). Für jeden Implementationsteil muß ein Spezifikationsteil existieren, aber nicht umgekehrt. Ein typisches Beispiel für den letztgenannten Fall ist ein Paket, das nur Definitionen von Typen, Konstanten und Variablen enthält.

Ein wesentlicher Vorteil dieser Trennung in Spezifikations- und Implementationsteil besteht darin, daß beispielsweise eine syntaktische Prüfung der Parameter eines Unterprogrammaufrufes für ein Unterprogramm, das in einem Spezifikationsteil definiert ist, auch dann möglich ist, wenn der zugehörige Implementationsteil noch nicht erstellt ist.

7.1 Abstrakte Datentypen

Gewöhnlicherweise werden Pakete aber verwendet, um *abstrakte Datentypen* zu realisieren. Der Spezifikationsteil beinhaltet dann die Deklaration des Typs als `private` oder `limited private`, darauf folgen die für diesen Typ definierten Operationen in Form von Prozedur- und Funktionsspezifikationen. Im anschließenden Private-Teil findet sich die tatsächliche Datenstruktur, die – für den Benutzer nicht sichtbar, d.h., les- und/oder modifizierbar – den privaten Typ repräsentiert. Im Implementationsteil (*body*) finden sich dann die Implementationsteile der Prozeduren und Funktionen. Alle in einem Body deklarierten Typen und Objekte sind nur innerhalb dieses Body's sichtbar.

Nun können wir unser Beispiel aus Kapitel 2.5 wiederaufnehmen. Eine geeignete Spezifikation lautet:

```
package Datum_Manager is

  subtype Tag_Typ is integer range 1 .. 31;

  type Monatsname is (Jaenner, Feber, Maerz, April, Mai, Juni, Juli,
    August, September, Oktober, November, Dezember);

  subtype Jahr_Typ is integer range 0 .. 4000;

  falsches_Datum: exception;

  type Datum is private;
```

```
function Setze(
  Tag: Tag_Typ;
  Monat: Monatsname;
  Jahr: Jahr_Typ)
return Datum;
```

```
function Tag(
  von_Datum: Datum)
return Tag_Typ;
```

```
function Monat(
  von_Datum: Datum)
return Monatsname;
```

```
function Jahr(
  von_Datum: Datum)
return Jahr_Typ;
```

`private`

```
type Datum is
record
  Tag: Tag_Typ;
  Monat: Monatsname;
  Jahr: Jahr_Typ;
end record;
```

`end Datum_Manager;`

Eine passende Implementierung umfaßt nun die Implementierung der Unterprogramme, die im Spezifikationsteil des Paketes angegeben worden sind:

`package body Datum_Manager is`

```
function Setze(
  Tag: Tag_Typ;
  Monat: Monatsname;
  Jahr: Jahr_Typ)
return Datum
is
  Dat: Datum := (Tag, Monat, Jahr);
  - hier kann man auf die Komponenten von Datum zugreifen
begin
  case Monat is
    when Jaenner | Maerz | Mai | Juli | August | Oktober | Dezember =>
      null; -- Anzahl der Tage passt
    when Feber =>
      if Jahr mod 4 = 0 and (Jahr mod 100 /= 0 or jahr mod 400 = 0) then
        - Schaltjahr
        if Tag not in 1 .. 29 then
          raise falsches_Datum;
        end if;
      else
        - kein Schaltjahr
```

```

        if Tag not in 1 .. 28 then
            raise falsches_Datum;
        end if;
    end if;
when others =>
    - hier kommen die Monate mit 30 Tagen
    if Tag not in 1 .. 30 then
        raise falsches_Datum;
    end if;
end case;
return Dat;
end Setze;

function Tag(
    von_Datum: Datum)
    return Tag_Typ
is
begin
    return von_Datum.Tag;
end Tag;

function Monat(
    von_Datum: Datum)
    return Monatsname
is
begin
    return von_Datum.Monat;
end Monat;

function Jahr(
    von_Datum: Datum)
    return Jahr_Typ
is
begin
    return von_Datum.Jahr;
end Jahr;

begin
    null;
end Datum_Manager;

```

Wenn wir Initialisierungen vornehmen wollten, müßten wir diese anstelle der Null-Anweisung vor dem Ende des Paketes einfügen. Da wir das nicht wollen, könnten wir die Anweisungen `begin null;` am Ende des Body's auch weglassen.

7.2 Hinausgeschobene Konstanten-Definition

Um in einem Spezifikationsteil eines Paketes Konstanten eines privaten oder limitierten privaten Typs vereinbaren zu können, gibt es in Ada sogenannte *hinausgeschobene Konstanten-Definitionen* (*deferred constants*). In diesem Fall ist es nicht möglich, die Initialisierung explizit vorzunehmen, da ja die interne Representation eines privaten Typs nicht nach außen bekannt werden soll. Daher entfällt die explizite Initialisierung, sie muß aber im Private-Teil der Spezifikation nachgeholt werden. Wir könnten etwa in unserem `Datum_Manager`

```
Geburtstag: constant Datum;
```

vereinbaren, müssen dann aber im Private-Teil etwas wie

```
Geburtstag: constant Datum := (29, September, 1967);
```

stehen haben.

7.3 Zugriff auf in Paketen deklarierte Objekte

Es bleibt die Frage zu beantworten, wie man auf jene Objekte zugreift, die im sichtbaren Teil eines Paketes deklariert worden sind. Innerhalb des Paketes, das die Deklaration enthält, und innerhalb des zugehörigen Body's kann man auf das deklarierte Objekt zugreifen, indem man den spezifizierten Namen verwendet.

Nehmen wir nun an, wir hätten im sichtbaren Teil des Paketes A ein Objekt O deklariert, und wir wollen vom Paket B auf dieses Objekt zugreifen. Ada erlaubt das durch explizites Nennen des Paketes, in dem sich das Objekt befindet, vor dem Objekt-Namen. Die Trennung der beiden Namen erfolgt durch einen Punkt. In unserem Beispiel heißt das

```
A.O
```

Das ist jedoch nur möglich, wenn man vor dem Spezifikations- oder Implementationsteil des Paketes B kund tut, daß man Objekte des Paketes A verwenden will. Das tut man mit

```
with A;
```

Es können sogar beliebig viele, durch Beistriche getrennte Pakete angegeben werden, auf die man sich beziehen will. Man sagt dann, alle in einer With-Anweisung angegebenen Pakete sind in dem entsprechenden Paket *sichtbar*.

Es sei explizit darauf hingewiesen, daß man sich vom Spezifikations- und Implementationsteil eines Paketes auf unterschiedliche Pakete beziehen kann. Dadurch ist die Spezifikation unabhängiger von der Implementierung. Ein Body erbt jedoch implizit die Pakete, auf die man sich im Spezifikationsteil bezogen hat.

Durch diesen Mechanismus können sehr lange Namen auftreten. Um diese zu vermeiden, gibt es in Ada zwei verschiedene Möglichkeiten. Die erste ist die, daß man unmittelbar nach der With-Anweisung eine *Use-Anweisung* angibt, z.B.:

```
with A;
use A;
```

Dies bewirkt, daß man auf Objekte, die im Paket A definiert sind ohne das Prefix A. zugreifen kann, und zwar innerhalb des gesamten betroffenen Paketes. Dabei kann es natürlich zu Namenskollisionen kommen, die entsprechend aufgelöst werden müssen.

Wenn man für einen selbstdefinierten Typ Operatoren wie etwa "+" oder "*" überlädt, würde man immer eine Use-Klausel benötigen, um die Infix-Notation (z.B. a+b) verwenden zu können*. Damit sind aber dann auch alle anderen in dem entsprechenden Paket deklarierten Typen, Objekte und Operationen ohne den Paketnamen-Prefix verwendbar. Um *nur* die Operatoren in Infix-Notation verwenden zu können, für alle anderen im Paket deklarierten Programmteile aber den Paketnamen-Prefix verwenden zu müssen, gibt es in Ada die *Use-Type-Klausel*. Man schreibt dann statt `use A;` zum Beispiel `use type A.My_Type;`

Die andere Möglichkeit ist die von *Renames-Anweisungen*.

*Ohne Use-Klausel müßte man etwa "+"(a,b) schreiben.

Asterix: *Aber Samson Himmelschorus,
Himmelschorus hat so eine
lateinische Endung?*
Himmelschorus: *Ich habe den Namen aus
kommerziellen Gründen angenommen!
In Wirklichkeit heie ich
Rosenblumenthalowitsch!*

Rene Goscinny und Albert Uderzo,
„Die Odyssee“.

Man kann sie einerseits verwenden, um Pakete umzubenennen, z.B.:

```
package DM renames Datum_Manager;
```

andererseits kann man damit einzelne Objekte eines Paketes umbenennen, z.B.:

```
function Set(  
  Tag: Tag_Typ;  
  Monat: Monatsname;  
  Jahr: Jahr_Typ)  
return Datum_Manager.Datum  
  renames Datum_Manager.Setze;
```

7.4 Gegenseitige Abhängigkeit

Oft treten bei der Implementierung von Datentypen gegenseitige Abhängigkeiten auf. Als (etwas künstliches) Beispiel sei gegeben:

```
type Punkt;  
type Gerade;  
  
type Punkt is  
  record  
    g,h: access Gerade;          -- Schnittpunkt zweier Geraden  
  end record;  
  
type Gerade is  
  record  
    p,q: access Punkt;          -- definiert durch zwei Punkte  
  end record;
```

Das würde so auch funktionieren.

Problematisch wird es, wenn man die beiden Typen Punkt und Gerade in unterschiedlichen Paketen haben möchte, was bei etwas komplizierteren Strukturen durchaus sinnvoll sein kann. Wir realisieren die beiden Typen nun als

```
limited with Gerade;  
  
package Punkt is  
  
  type punkt is  
    record
```

```
    g,h: access Gerade gerade;  
  end record;  
  
end Punkt;  
  
limited with Punkt;  
  
package Gerade is  
  
  type gerade is  
    record  
      p,q: access Punkt.punkt;  
    end record;  
  
end Gerade;
```

Würden wir statt `limited with` nur `with` schreiben, bekämen wir eine Compiler-Fehlermeldung, da Zirkular-Referenzen nicht erlaubt sind.

Mit der `limited with`-Anweisung aber sind alle Typen des angegebenen Pakets sichtbar, als wären sie unvollständige Typvereinbarungen. Damit können wir dann alles tun, was wir mit unvollständigen Typen tun können, also etwa Pointer darauf definieren.

Der Compiler überprüft dann natürlich später, dass alles passt.

Abschließend sei angemerkt, dass es im obigen Beispiel genügt hätte, in nur einem Paket eine `limited with`-Anweisung zu verwenden. Im anderen hätte man eine normale `with`-Anweisung nehmen können. Aus Symmetriegründen ist aber obige Lösung zu bevorzugen.

Das `limited with` wurde mit Ada2005 eingeführt und findet seine Anwendung auch beim Anbinden vorhandener APIs, die in anderen Sprachen geschrieben wurden.

7.5 Getrennte Übersetzbarkeit

Pakete sind die meistbenutzte Übersetzungseinheit in Ada (vgl. dazu auch Kapitel 11). Man kann etwa den Implementationsteil eines Paketes unabhängig von seinem Spezifikationsteil übersetzen. Durch die `With`-Anweisung wird jedoch einem Ada-Programm, das aus mehreren Paketen besteht, eine bestimmte *Übersetzungsreihenfolge* aufgezwungen. Es müssen alle in der `With`-Anweisung eines Paket-Teiles stehenden Pakete vor diesem Paket-Teil übersetzt werden. Wir müssen unser obiges Beispiel in der Reihenfolge:

```
A (Spez.), A (Body), B (Spez.), B (Body)
```

übersetzen. Die Übersetzung von A (Body) kann allerdings auch später erfolgen.

Allgemein gesprochen induzieren die `With`-Anweisungen eine *Ordnung* auf die einzelnen Paket-Teile (Spezifikationen und Bodies), die als gerichteter, azyklischer Graph dargestellt werden können muß. Falls ein Zyklus in den Abhängigkeiten auftaucht, ist das Ada-Programm ungültig.

7.6 Benutzerdefinierte Initialisierung, Zuweisung und Finalisierung

Für diese drei grundlegenden Operationen existieren von der Sprache vordefinierte Operationen, der Programmierer hat jedoch die Möglichkeit, sie selbst zu definieren. Dies geschieht durch

sogenannte *kontrollierte Typen*.

Der Programmierer kann drei Prozeduren definieren: `Initialize` wird nach der defaultmäßigen Initialisierung aufgerufen, `Adjust` als letzter Teil der defaultmäßigen Zuweisung (eines Objekts eines nicht-limitierten Typs) und `Finalize`, bevor die Finalisierung der Teile des Objekts durchgeführt wird.

Ein kontrollierter Typ ist eine Typerweiterung des Typs `Controlled`, der im Paket `Ada.Finalization` definiert ist:

```
package Ada.Finalization is
  type Controlled is abstract tagged private;

  procedure Initialize(Object: in out Controlled);
  procedure Adjust (Object: in out Controlled);
  procedure Finalize (Object: in out Controlled);

  type Limited_Controlled is abstract tagged limited private;

  procedure Initialize(Object: in out Limited_Controlled);
  procedure Finalize (Object: in out Limited_Controlled);
private
  ...
end Ada.Finalization;
```

Bei einer normalen Zuweisung eines privaten kontrollierten Typs der Form `a := b` werden folgende Operationen ablaufen:

1. `Finalize(a);`
2. `Initialize(a);`
3. Kopieren von `b` nach `a`
4. `Adjust(a);`

8 Generische Einheiten

Generische Einheiten in Ada sind in etwa vergleichbar den in manchen Assemblersprachen üblichen Makros. Eine generische Einheit ist eigentlich eine Art Schablone, die durch den Vorgang der *Instantiierung* funktionsfähig wird. In Ada wird zwischen *generischen Unterprogrammen* und *generischen Paketen* unterschieden. Wichtig ist, daß für solche Einheiten *generische Parameter* definiert werden können. Bevor wir fortfahren, ein Beispiel:

```
generic
  type Element is private;
  procedure Vertausche(a,b: in out Element);
```

Das ist die Deklaration einer *generischen Prozedur*, die die Werte der Parameter vertauschen soll. Da der Typ der Parameter uninteressant ist, wird er als generischer Parameter in Form eines privaten Typs vereinbart. Damit wird bewirkt, daß keinerlei Anforderungen an Operationen dieses Typs gestellt werden, außer daß es möglich sein muß, Werte zweier Variablen dieses Typs einander zuzuweisen und sie eventuell auf Gleich- und Ungleichheit zu testen.

Eine entsprechende Implementierung könnte etwa lauten:

```
procedure Vertausche(a,b: in out Element)
is
  h: Element;
begin
  h := a;
  a := b;
  b := h;
end Vertausche;
```

Die Implementierung erfolgt also wieder gänzlich unabhängig vom Typ der zu vertauschenden Werte in dem Sinne, daß nur der generische Parameter verwendet wird.

Beispiele für Instantiierungen lauten etwa:

```
procedure Vertausche_integer is new Vertausche(Element => integer);
procedure Vertausche_Datum is new Vertausche(Element => Datum);
```

8.1 Ein einleitendes, umfangreicheres Beispiel

Mit Hilfe von generischen Einheiten lassen sich wiederverwendbare Software-Komponenten besonders vorteilhaft gestalten (siehe z.B. [Boo87]). Damit ergibt sich auch die nicht unbedingt einfache Aufgabe, Bibliotheken von wiederverwendbaren und mehr oder minder öffentlich zugänglichen Software-Komponenten anzulegen und zu verwalten. Ein solcher Ansatz findet sich beispielsweise in [Con87].

Als Beispiel, wie solche Bibliothekseinträge aussehen könnten, betrachten wir einen Stack beliebiger Größe:

```
generic

  type Element is private;

package Stack is

  Stack_ist_leer: exception;

  procedure Push(E: in Element);

  procedure Pop (E: out Element);

end Stack;

package body Stack is

  type Listen_Element;

  type Zeiger_auf_Listen_Element is access Listen_Element;

  type Listen_Element is
    record
      Eintrag: Element;
      der_naechste: Zeiger_auf_Listen_Element;
    end record;

  Listen_Anfang: Zeiger_auf_Listen_Element;

  procedure Push(E: in Element)
  is
    Zeiger_auf_neues_Element: Zeiger_auf_Listen_Element :=
      new Listen_Element'(
        Eintrag => E,
        der_naechste => Listen_Anfang);
  begin
    Listen_Anfang := Zeiger_auf_neues_Element;
  end Push;

  procedure Pop (E: out Element)
  is
  begin
    if Listen_Anfang /= null then
      E := Listen_Anfang.Eintrag;
      Listen_Anfang := Listen_Anfang.der_naechste;
    else
      raise Stack_ist_leer;
    end if;
  end Pop;

end Stack;
```

Eine Instantiierung dieses generischen Paketes könnte etwa lauten:

```
package Integer_Stack is new Stack(Element => integer);
```

Eine mögliche Verwendung dieser Instanz könnte folgenderweise aussehen:

```
...

for i in 1 .. N loop
  Integer_Stack.Push (E => i);
end loop ;

...

declare
  i: integer;
begin
  loop
    Integer_Stack.Pop (E => i);
    - i verarbeiten
  end loop;
exception
  when Integer_Stack.Stack_Ist_leer =>
    null;
end;
```

Im zweiten Teil dieses Programmfragmentes ist es nicht notwendig zu wissen, wieviele Elemente tatsächlich auf dem Integer_Stack liegen, das Ende wird über die ausgelöste Exception ermittelt.

Es gibt nun verschiedene Arten von generischen Parametern, nämlich: *generische formale Objekte*, *generische formale Typen*, *generische formale Unterprogramme* und *generische formale Pakete*. Diese Parameter werden wir im folgenden erläutern. Für alle diese Parameter gilt, daß sie zwischen dem Schlüsselwort `generic` und einem der Schlüsselwörter `procedure`, `function` oder `package` stehen müssen.

8.2 Generische formale Objekte

Generische formale Objekte sind am ehesten mit normalen Parametern zu vergleichen. Sie können die Übergabemechanismen `in` oder `in out` haben. Erstere werden bei der Instantiierung versorgt und verhalten sich innerhalb der Instanz wie eine Konstante, zweiteere werden ebenfalls bei der Instantiierung versorgt und verhalten sich wie eine Variable. Wird kein Übergabemechanismus spezifiziert, so wird default-mäßig `in` angenommen. Im Falle des Übergabemechanismus `in` kann auch ein Default-Wert für den Parameter angegeben werden. Ein Beispiel könnte etwa sein:

```
generic
  size: in positive;
package Buffer is

...

end Buffer;
```

Hier wird die Größe eines Buffers als generisches formales Objekt festgelegt. Die für den Buffer definierten Operationen haben wir nicht explizit angeführt.

8.3 Generische formale Typen

Als *generische formale Typen* sind erlaubt:

- `private` Typen,
- limitierte `private` Typen,
- getaggte `private` Typen,
- getaggte limitierte `private` Typen,
- abstrakte Typen,
- `<>`,
- `range <>`,
- `mod <>`,
- `digits <>`,
- `delta <>`,
- `delta <> digits <>`,
- die Definition eines Array-Typs und
- die Definition eines Access-Typs.

Dabei steht `<>` für einen beliebigen diskreten Typ; bei der Implementierung sind für Variable dieses Typs alle Operationen erlaubt, die für diskrete Typen erlaubt sind. Der Ausdruck `range <>` steht für ganzzahlige Typen, `mod <>` für modulare Typen, `digits <>` für Gleitkommazahlen, `delta <>` für Festkommazahlen und `delta <> digits <>` für Dezimalzahlen; bei der Implementierung stehen die entsprechenden Operationen zur Verfügung.

Für Array-Typen sind z.B. die Operationen des Zugriffs auf Komponenten und Slices erlaubt. Für Access-Typen kann man z.B. den `New`-Operator verwenden. Für `private` Typen ist nur die Zuweisung und die Abfrage auf Gleichheit und Ungleichheit erlaubt. Für limitierte `private` Typen sind nicht einmal diese erlaubt, daher sind alle benötigten Operationen mit den im Abschnitt 8.4 näher behandelten generischen Unterprogrammen zu realisieren.

Die folgende Tabelle gibt einen Überblick, welche Typen durch welche formale generische Typen festgelegt werden.

Formaler generischer Typ	Festgelegter Typ
limited private	die Klasse aller Typen
private	die Klasse aller nicht limitierten Typen
tagged limited private	die Klasse aller getaggten Typen
tagged private	die Klasse aller nicht limitierten getaggten Typen

Wenn das Schlüsselwort `abstract` vorhanden ist, darf der aktuelle generische Parameter auch `abstract` sein.

Beispiele sind


```

type Element is private;
type Aufzaehlung is (<>);
type int is range <>;
type Volt is delta <>;
type Masse is digits <>;
type Tabelle is array (Aufzaehlung) of Element;
type Item is limited tagged private;

```

Weitere Beispiele finden sich in Kapitel 12 und 13.

8.4 Generische formale Unterprogramme

Legt man fest, daß man eine generische Einheit mittels privaten oder limitiert privaten generischen formalen Parameter realisieren will, so ergibt sich das Problem, daß die für diese Typen vordefinierten Operation im allgemeinen nicht ausreichen, um die Implementierung der generischen Einheit durchführen zu können. Man möchte sich vielmehr auf speziell für diesen (limitiert) privaten Typ deklarierte Operationen berufen können. Mit *generischen formalen Unterprogrammen* kann man festlegen, wie diese Operationen aussehen sollen. Bei der Instantiierung einer generischen Einheit muß dann sichergestellt sein, daß für den aktuellen generischen Parameter diese Operationen tatsächlich vorhanden sind.

Solche generische formale Unterprogramme werden mit dem Schlüsselwort `with` definiert, z.B.:

```

with function Summe (a,b: Element) return Element;

```

Es ist auch möglich, Default-Unterprogramme anzugeben. Dabei gibt es zwei verschiedene Arten:

1. Der Name eines bestimmten Unterprogrammes, z.B.:

```

with function IMAGE (x: Aufzaehlung) return string is Aufzaehlung'IMAGE;

```

2. Mit einer `<>`, was soviel bedeutet wie, daß bei der Instantiierung ein Unterprogramm als Default genommen wird, das denselben Namen hat wie das generische formale Unterprogramm und das zum Zeitpunkt der Instantiierung sichtbar ist, z.B.:

```

generic
  type Element is private;
  with function "*" (x,y: Element) return Element is <>;
function Quadrat(x: Element) return Element;

```

Sollte zum Zeitpunkt der Instantiierung für den aktuellen generischen Typen der Operator „*“ eindeutig definiert und sichtbar sein, so kann der aktuelle Wert für die generische Funktion „*“ beim Instantiieren weggelassen werden, z.B. wären die folgenden Instantiierungen gleichwertig:

```

function Quadriere is new Quadrat(Element => integer , "*" => "*");
function Quadriere is new Quadrat(integer , "*");
function Quadriere is new Quadrat(integer);

```

8.5 Generische Formale Pakete

Mittels solcher generischer Parameter können Instanzen von generischen Paketen übergeben werden. Die Syntax lautet

```

with package <Name_1> is new <Name_2>(<>);

```

Dabei steht `Name_2` für das generische Paket, von dem eine Instanz übergeben werden muß. Der interne Name dieses Paketes ist `Name_1`. (`<>`) bedeutet, daß die übergebene Instanz beliebige aktuelle generische Parameter besitzen darf. Man kann jedoch auch Einschränkungen vorschreiben.

Die Wahl des richtigen Parametertyps ist keine leichte Aufgabe; einerseits empfiehlt es sich, einen möglichst allgemeinen Typ zu wählen, andererseits sollte die Anzahl der generischen formalen Unterprogramme gering und die der implizit definierten Operationen groß sein.

Abschließend geben wir ein einfaches Beispiel für getaggte generische Parameter:

```

package Point is

```

```

  type point is tagged
    record
      x,y: float;
    end record;

```

```

  procedure Draw(p: point);

```

```

end Point;

```

```

with Point;

```

```

generic
  type derived_from_point is new Point.point with private;
package Graphics is

```

```

  procedure Draw_it(p: derived_from_point);

```

```

end Graphics;

```

```

package body Graphics is

```

```

  procedure Draw_it(p: derived_from_point) is
  begin
    Draw(p);
  end Draw_it;

```

```

end Graphics;

```

```

with Point, Graphics;

```

```

package Colored_Point is

```

```

  type color_t is (rot, grün, blau);

```

```

  type colored_point is new Point.point with
    record
      color: color_t;
    end record;

```

```
    procedure Draw(p: colored_point);  
  
end Colored_Point;  
  
with Colored_Point, Graphics;  
procedure tag_par is  
  
    package Color_Graphics is new Graphics(Colored_Point.colored_point);  
  
    col_pnt: Colored_Point.colored_point;  
  
begin  
    Color_Graphics.Draw_it(col_pnt);           -- ruft intern Colored_Point.Draw  
end tag_par;
```

9 Tasks

In diesem Kapitel wenden wir uns Programm-Komponenten zu, die Ada von anderen, bekannten höheren Programmiersprachen unterscheidet. Der auch heutzutage noch gängige Weg, Programme zu erstellen, die sich aus mehreren, parallel ablaufenden Prozessen zusammensetzen, besteht darin, die einzelnen Prozesse separat in einer Programmiersprache zu schreiben und die Kommunikation und Synchronisation der Prozesse durch Betriebssystem-Calls zu realisieren. Diese Methode erweist sich jedoch als äußerst unzureichend, wenn so ein Programm von einem Rechner auf einen anderen portiert werden muß, auf dem ein anderes Betriebssystem läuft.

Im Gegensatz dazu besitzt Ada auf Sprachniveau die erforderlichen Sprachmittel, um solche Programme zu schreiben, ohne auf Betriebssystem-Calls angewiesen zu sein. Daher können Programme, die aus mehreren Prozessen (*Tasks* in Ada-Terminologie) bestehen, ohne weiteres von einem Rechner auf einen anderen portiert werden*.

9.1 Multitasking in Ada

Tasks sind Objekte im Sinne von Ada, d.h., man kann unterschiedliche *Task-Typen* definieren und beliebig viele Objekte dieses Typs anlegen. So ist es zum Beispiel möglich, Arrays von Tasks anzulegen, einen Task als Komponente eines Records zu haben oder Pointer auf Tasks zu definieren und später Tasks dynamisch (mit *new*) anzulegen.

Wie schon von den anderen, höheren Sprachmitteln Adas gewohnt, besteht ein Task aus einem *Spezifikations-* und einem *Implementationsteil*, dem *Body*. Bei der Spezifikation eines Tasks definiert man seine *Entries*. Das sind jene Bestandteile, die man von außerhalb des Tasks aufrufen kann. Sie können auch Parameter haben. Der Body des Tasks legt dann das Verhalten beim Aufruf so eines Entry's fest. Die Sprache definiert, daß immer nur genau ein Entry eines Tasks gleichzeitig aktiviert werden kann. Gleichzeitig eintreffende Aufrufe werden in einer Warteschlange gereiht.

Ein einfaches Beispiel für die Spezifikation eines Tastatur-Treibers ist folgendes:

```
task type Tastatur_Treiber is
  entry read (c: out character);
  entry write(c: in character);
end Tastatur_Treiber;
```

Aufgrund der obigen Eigenschaften von Ada-Tasks ist durch die Spezifikation des Treibers als Task im Gegensatz zu einer Spezifikation als Paket gesichert, daß immer nur genau ein (anderer) Task Lese- und Schreiboperationen durchführen kann.

Falls nur ein Task gebraucht wird, kann das Schlüsselwort *type* nach *task* entfallen. Dann wird nur der Spezifikations- und Implementationsteil eines Tasks definiert, man kann jedoch keine weiteren Task-Objekte anlegen.

9.2 Task-Typen und Task-Objekte

Ein *Task-Typ* ist ein limitierter privater Typ, daher kann man Objekte dieses Typs weder einander zuweisen, noch kann man sie paarweise auf Gleichheit oder Ungleichheit überprüfen. Task-

*Natürlich nur, sofern es auf diesem Rechner ein Ada-Laufzeitsystem gibt.

Objekte deklariert man etwa so:

```
VT100_Tastatur_Treiber: Tastatur_Treiber;
```

Das dynamische Anlegen von Tasks kann folgendermaßen erfolgen:

```
type Zeiger_auf_Tastatur_Treiber is access Tastatur_Treiber;
VT200_Tastatur_Treiber: Zeiger_auf_Tastatur_Treiber := new Tastatur_Treiber;
```

9.3 Die Aktivierung und die Ausführung von Tasks

Die Aktivierung eines Tasks besteht im allgemeinen aus der Abarbeitung des einleitenden Definitionsteils des Body's (Typen, Konstanten, Variablen, usw.). Danach folgt die Ausführung des Tasks, diese hängt jedoch von den – die spezifizierten Entries rufenden – Tasks ab. Sollte während der Aktivierung eines Tasks ein Fehler auftreten, so wird die Fehlermeldung *tasking_error* ausgelöst. Die Ausführung von Tasks und schon ihre Aktivierung geschehen parallel.

Task-Objekte, die als Variable in einem Vereinbarungsteil definiert wurden, werden anschließend an den Vereinbarungsteil aktiviert (d.h. unmittelbar vor dem folgendem *begin*), Task-Objekte, die im Spezifikationsteil eines Paketes deklariert wurden, werden anschließend an die Abarbeitung des Vereinbarungsteiles des zugehörigen Paket-Bodys aktiviert, Task-Objekte, die mittels *new* dynamisch angelegt werden, werden am Ende der *New*-Anweisung aktiviert, d.h., nachdem alle anderen Initialisierungen vorgenommen worden sind.

9.4 Anweisungen zur Programmierung von Tasks

Da die Programmierung von parallelen Abläufen sich wesentlich von der sequentieller Abläufe unterscheidet, benötigt Ada dafür auch eigene Anweisungen, die wir im folgenden kennenlernen werden.

9.4.1 Die Delay-Anweisung

	<i>Sekunde</i>
	<i>Wie lange</i>
	<i>kann man warten</i>
	<i>Eine Sekunde</i>
	<i>Ewigkeit</i>
	<i>die nächste</i>
	<i>ist Zeit</i>
	Rose Ausländer

Einer der Entwurfsziele für Ada war es, eine Programmiersprache für Echtzeitanwendungen zu schaffen, daher gibt es in Ada auch Sprachmittel, die sich auf die Zeit beziehen. Die *Delay*-Anweisung versetzt den Task in einen Wartezustand, bis die angegebene Zeitspanne abgelaufen ist. Um einen Task für mindestens 10 Sekunden in den Wartezustand zu versetzen, schreibt man, z.B.:

```
delay 10.0;
```

Der Typ, der von der Delay-Anweisung als Argument erwartet wird, ist der vordefinierte Festkomma-Typ `duration`. Natürlich kann an Stelle einer Konstanten auch ein Ausdruck stehen.

Es soll jedoch explizit darauf hingewiesen werden, daß Ada nur garantiert, daß der betroffene Task *mindestens* für die Dauer des spezifizierten Intervalles in einen Wartezustand versetzt wird. Der Gesamtzustand des aus mehreren Tasks bestehenden Ada-Programmes kann jedoch bedingen, daß der Task erst später wieder aktiv wird.

9.4.2 Die Delay-Until-Anweisung

Zusätzlich zur Delay-Anweisung gibt es auch noch eine Delay-Until-Anweisung. Man schreibt dann etwa:

```
delay until <time>;
```

Als erlaubter Datentyp für `<time>` steht der vordefinierte Typ `Time` zur Verfügung.

9.4.3 Die Abort-Anweisung

Die *Abort-Anweisung* bewirkt, daß ein oder mehrere Tasks abgebrochen werden. Ein Beispiel ist:

```
abort VT100_Tastatur_Treiber;
```

Die Abort-Anweisung ist aber eher die abnormale Art, einen Task zu beenden, die sozusagen normale Art, einen Task zu beenden, werden wir in Kapitel 9.10 kennenlernen.

9.5 Das Rendezvous-Konzept in Ada

Tasks, die völlig unabhängig voneinander ablaufen, sind für die Programmierung eigentlich ohne Probleme, da sie auch als parallel ablaufende sequentielle Programme betrachtet werden können. Interessanter und vor allem schwieriger wird es, wenn wir in Betracht ziehen, daß Tasks untereinander kommunizieren und einander synchronisieren müssen. Diese beiden Aufgaben werden in Ada mit dem *Rendezvous-Konzept* erledigt. Ein dazu notwendiges Sprachmittel – die Entries eines Tasks – haben wir schon kennengelernt.

In Ada verläuft die Kommunikation zweier Tasks asymmetrisch in dem Sinn, daß ein Unterschied besteht zwischen dem einen Entry aufrufenden und dem aufgerufenen Task. Der Aufruf eines Task-Entry's unterscheidet sich syntaktisch in nichts vom Aufruf einer Prozedur. Semantisch jedoch gibt es wesentliche Unterschiede. Abgesehen von der bereits erwähnten Warteschlange, in der die Rufer abgelegt und mittels der sie – zufolge einer FIFO-Strategie – abgearbeitet werden, wird der rufende Task in einen Wartezustand versetzt, wenn der gerufene Task nicht bereit ist, die Kommunikation sofort zu bewerkstelligen. Auf seiten des gerufenen Tasks gibt es *Accept-Anweisungen* für die spezifizierten Entries. Falls ein solcher Task zu einer *Accept-Anweisung* eines Entry's kommt, aber momentan kein Aufruf dieses Entry's vorliegt, so geht dieser Task in den Wartezustand über. Kann die Kommunikation ablaufen, so werden alle nach dem entsprechenden *accept* angegebenen Anweisungen abgearbeitet und anschließend beide Tasks fortgesetzt. Diese Art der Kommunikation von parallelen Prozessen nennt man *Rendezvous*.

9.5.1 Wie ein gerufener Task das Rendezvous beeinflussen kann

Wir werden nun an Hand eines Beispiels die verschiedenen Möglichkeiten kennenlernen, wie die Task-Kommunikation in Ada ablaufen kann. Als realistisches Beispiel nehmen wir einen mehr oder weniger gestreßten Universitätsassistenten zur Hand. Zunächst soll seine Aufgabe nur darin bestehen, Studierende, die in seinem Kämmerchen auftauchen, zu betreuen. Wir spezifizieren dazu den Task Assistent folgendermaßen:

```
task Assistent is
  entry betreue;
end Assistent;
```

Jeden Studierenden modellieren wir als eigenen Task, der nötigenfalls durch

```
Assistent.betreue;
```

eine Betreuung durch den Assistenten anfordert. Sollte der Assistent gerade einen anderen Studierenden betreuen, muß der Student in einer Warteschlange warten, bis er an der Reihe ist.

Die „Innereien“ des Assistenten-Tasks können etwa folgenderweise aussehen:

```
task body Assistent is

begin
  loop
    accept betreue do
      -- betreue den Studierenden
    end betreue;
  end loop;
end Assistent;
```

Allerdings ist der Aufgabenbereich eines Universitätsassistenten damit bei weitem nicht erfaßt. Es sind nämlich nicht nur Studierende zu betreuen, sondern auch organisatorische Aufgaben zu erledigen und Forschungsarbeiten durchzuführen. Wir wollen annehmen, daß unser Assistent die zu erledigenden organisatorischen Aufgaben von der Sekretärin des Instituts oder dem Institutsvorstand zugeteilt bekommt. Die Forschungstätigkeiten sollen dem Assistenten obliegen. Wir wollen annehmen, daß unser Assistent immer dann forscht, wenn er nicht durch eine der beiden anderen Aufgaben beschäftigt ist. Daher modellieren wir die organisatorische Aufgabe als Entry des Assistenten-Tasks:

```
task Assistent is
  entry betreue;
  entry organisiere (Aufgabe: Aufgaben_Typ);
end Assistent;
```

Ein entsprechender Aufruf lautet dann z.B.:

```
Assistent.organisiere (Aufgabe => Bibliotheksverwaltung);
```

Der Ablauf des Assistenten-Tasks gestaltet sich nun schon etwas schwieriger. Der Assistent muß ja jeder Zeit bereit sein, entweder einen Studierenden zu betreuen oder eine organisatorische Aufgabe zu erledigen. In Ada realisiert man das mit einer *Select-Anweisung*. Man nennt dann den Vorgang der Kommunikation *selektives Rendezvous des Servers*. Unser Beispiel hat jetzt folgendes Aussehen:

```

task body Assistent is
begin
  loop
    select
      accept betreue do
        - betreue den Studierenden
      end betreue;
    or
      accept organisiere (Aufgabe: Aufgaben.Typ) do
        - organisiere
      end organisiere;
    end select;
  end loop;
end Assistent;

```

Diese Implementation bewirkt, daß unser Assistenten-Task zunächst überprüft, ob er einen Studierenden zu betreuen oder eine organisatorische Aufgabe zu erledigen hat (wobei die Reihenfolge durch die Sprachnorm nicht festgelegt ist), gegebenenfalls entsprechend agiert oder, falls keine Anforderung anliegt, wieder an den Beginn der Select-Anweisung zurückkehrt und dort auf einen Aufruf eines der beiden Entries wartet.

Wollen wir nun auch noch erreichen, daß unser Assistent seiner Forschungstätigkeit nachkommt, so erhalten wir folgende Implementation:

```

task body Assistent is
begin
  loop
    select
      accept betreue do
        - betreue den Studierenden
      end betreue;
    or
      accept organisiere (Aufgabe: Aufgaben.Typ) do
        - organisiere
      end organisiere;
    else
      - forsche
    end select;
  end loop;
end Assistent;

```

Damit haben wir erreicht, daß der Assistent immer dann, wenn er keine der spezifizierten Aufgaben zu erledigen hat, forscht, allerdings kann er in seiner Forschungstätigkeit nicht unterbrochen werden. Er muß daher immer in Zeitscheiben forschen, anderenfalls kann er sonst keine Aufgabe erledigen.

Nach einiger Zeit erkennt unser Assistent aber, daß er buchstäblich zu keinerlei Forschungstätigkeit kommt. Er beschließt daher, Sprechstunden einzuführen, innerhalb derer er den Studierenden für ihre Probleme zur Verfügung steht. Außerhalb dieser Zeiten ignoriert er Betreuungsansuchen. Unser Assistenten-Task sieht dann so aus:

```

task body Assistent is
begin
  loop
    select
      when Sprechstunde =>

```

```

      accept betreue do
        - betreue den Studierenden
      end betreue;
    or
      accept organisiere (Aufgabe: Aufgaben.Typ) do
        - organisiere
      end organisiere;
    else
      - forsche
    end select;
  end loop;
end Assistent;

```

Dabei steht Sprechstunde z.B. für eine boolesche Funktion, die true zurückliefert, wenn gerade eine Sprechstunde ist, sonst liefert sie false zurück. Man nennt so ein Konstrukt eine *Select-Anweisung mit Wächtern*. Die boolschen Ausdrücke vor den Alternativen werden immer nur am Anfang der Select-Anweisung ausgewertet. Alle Zweige, die false ergeben, sind dann für die momentane Ausführung der Select-Anweisung geschlossen, die anderen offen. Wächter dürfen für alle Accept-Anweisungen verwendet werden, nicht jedoch im Else-Zweig.

Die letzte Möglichkeit auf seiten des Assistenten, seinen Alltag zu gestalten, besteht darin, falls für eine bestimmte Zeit keine zu erledigende Aufgabe ansteht, eine Kaffeepause einzulegen. Nehmen wir an, unser Assistent wartet 1 Stunde vergebens auf zu verrichtende Aufgaben, dann lautet eine entsprechende Implementierung;

```

task body Assistent is
begin
  loop
    select
      when Sprechstunde =>
        accept betreue do
          - betreue den Studierenden
        end betreue;
    or
      accept organisiere (Aufgabe: Aufgaben.Typ) do
        - organisiere
      end organisiere;
    or
      delay 3600.0;
      Kaffeepause;
    end select;
  end loop;
end Assistent;
-- 1 Stunde

```

Wie bei der Else-Anweisung kann der Assistenten-Task auch während der Kaffeepause nicht unterbrochen werden, allerdings startet die Delay-Anweisung nach jedem eingetroffenen Aufruf eines Entry's neu.

Die Sprache erlaubt mehrere Delay-Anweisungen innerhalb einer Select-Anweisung. Klarerweise wird dann die Anweisung mit der kürzesten Zeitspanne zuerst ausgeführt. Außerdem kann eine Select-Anweisung mit einer Delay-Anweisung keinen Else-Zweig enthalten.

Klarerweise kann statt delay auch delay until stehen.

Ada erlaubt also die folgenden vier verschiedenen Möglichkeiten

1. die normale Select-Anweisung

2. die Select-Anweisung mit einem Else-Zweig
3. die Select-Anweisung mit einer Delay-Anweisung
4. die Select-Anweisung mit Wächtern

9.5.2 Die Requeue-Anweisung

Mit einer Requeue-Anweisung kann der gerufene Task eine Accept-Anweisung beenden und den rufenden Task zu der Warteschlange eines anderen Entry's „umleiten“. Die Syntax dazu lautet

```
requeue <entry_name>;
```

Sollte der rufende Task nicht mehr existieren, sollte der rufende Task mittlerweile seinen Entry-Call zurückgezogen haben oder sollte eine Zeitschranke überschritten worden sein, so macht so eine Requeue-Anweisung natürlich keinen Sinn mehr. Um solche Fälle zu handhaben, schreibt man

```
requeue <entry_name> with abort;
```

In beiden Fällen kann sich der Entry, zu dem umgeleitet wird, sowohl innerhalb als auch außerhalb des Tasks befinden, aus dem heraus umgeleitet wird. Der Ziel-Entry darf entweder gar keine Parameter haben oder sein Parametertyp-Profil muß in Übereinstimmung zum Parametertyp-Profil jenes Entry's sein, aus dessen Accept-Anweisung heraus das Requeue erfolgt.

In unserem Beispiel könnte der betreuende Assistent etwa den Studierenden an einen seiner Kollegen, d.h., an einen anderen Assistenten verweisen. Bei einem requeue with abort hätte der Studierende die Chance, sich nicht in die dort befindliche Warteschlange einzureihen, sondern er kann, wenn etwa die Zeit, die er für die Betreuungszeit reserviert hatte, abgelaufen ist, mit einer anderen Tätigkeit fortfahren. Bei einer normalen Requeue-Anweisung muß er sich auf jeden Fall in die dort befindliche Warteschlange einreihen und darauf warten, daß er betreut oder weitergeschickt wird. Näheres dazu im Unterabschnitt 9.5.4.

9.5.3 Der Private-Teil von Tasks

Man kann Tasks auch mit einem Private-Teil versehen. In diesem kann man etwa private Entries deklarieren, die den Vorteil besitzen, daß sie „von außen“ nicht aufgerufen werden können, sehr wohl aber etwa für eine Requeue-Anweisung aus dem Task-Body heraus zur Verfügung stehen.

9.5.4 Wie der rufende Task das Rendezvous beeinflussen kann

Abschließend wollen wir noch eingehend erörtern, wie ein Studierender auf unser Szenario reagieren kann. Viele Studenten können etwa nicht „endlos“ darauf warten, daß sie der Assistent betreuen kann, vielmehr haben sie nur beschränkt Zeit, da sie z.B. anschließend eine Vorlesung hören wollen. Eine solche Art der Problemstellung könnten wir wie folgt programmieren:

```
select
  Assistent.betreue;
or
  delay 15*60.0;                -- 15 Minuten
  besuche_vorlesung;
end select;
```

In diesem Fall wartet der Studierende maximal 15 Minuten auf die Möglichkeit des Rendezvous, dann bricht er das Warten ab und besucht eine Vorlesung. Das Laufzeitsystem nimmt dabei den wartenden Studenten-Task aus der zum Entry betreue gehörenden Warteschlange und setzt ihn fort. Auch hier kann anstatt der Delay-Anweisung eine Delay-Until-Anweisung stehen.

Ein äußerst ungeduldiger Student könnte etwa auch folgenderweise vorgehen:

```
select
  Assistent.betreue;
else
  lauf_weg;
end select;
```

In diesem Fall würde der Studenten-Task die im Else-Zweig angegebenen Anweisungen ausführen, wenn das Rendezvous nicht unmittelbar durchführbar ist oder sofort eine Requeue-Anweisung with abort durchgeführt wird.

Es gibt auch noch die sogenannte *asynchrone Select-Anweisung*. Bleiben wir bei unserem Beispiel:

```
select
  Assistent.betreue;
then abort
  lauf_weg;
end select;
```

Das bedeutet dann, daß der Studenten-Task die im „then abort“-Zweig stehenden Anweisung ausführt, wenn das Rendezvous nicht unmittelbar möglich ist oder wenn im Laufe des Rendezvous ein requeue with abort ausgeführt wird. Sollte das Rendezvous unmittelbar möglich sein und während des Rendezvous kein requeue with abort auftreten, wird der „then abort“-Zweig nicht ausgeführt.

Die letzte Möglichkeit, die dem Studierenden bleibt, seine Betreuung durch einen Assistenten zu gestalten, sieht so aus:

```
select
  delay 15*60.0;                -- 15 Minuten
  besuche_vorlesung;
then abort
  Assistent.betreue;
end select;
```

Hier wird nach Ablauf der angegebenen Zeit versucht, die im „then abort“-Zweig stehenden Anweisungen abubrechen¹. Danach werden die zwischen der Delay-Anweisung und den Schlüsselwörtern then abort stehenden Anweisungen ausgeführt.

Sollte die Berechnung im „then abort“-Zweig beendet sein, bevor die Zeit abgelaufen ist, so wird versucht, die Delay-Anweisung abubrechen. Wenn das gutgeht, ist die gesamte Anweisung korrekt beendet.

9.6 Familien von Task-Entries

Die Erörterung von Task-Entries wollen wir damit abschließen, indem wir noch erwähnen, daß es auch die Möglichkeit gibt, sogenannte *Familien von Task-Entries* zu bilden. Dazu benötigt

¹Näheres zum Thema „Abbrechen von Tasks und einer Folge von Anweisungen“ kann man im Ada Reference Manual [Ada95] nachlesen.

man einen endlichen Bereich eines diskreten Typs. Für jedes Element dieses Bereiches kann man dann einen Task-Entry definieren, z.B.:

```
type Wichtigkeit is (gering, mittel, gross);

task Nachricht is
  entry Empfang(Wichtigkeit) (Nachricht: out Nachrichtentyp);
  entry Sende(Wichtigkeit) (Nachricht: in Nachrichtentyp);
end Nachricht;
```

Entsprechende Aufrufe von solchen Entries sind etwa:

```
Nachricht.Sende(gross) (meine.Nachricht);
Nachricht.Empfange(gering) (deine.Nachricht);
```

Bevor wir ein Beispiel für einen Task-Body erarbeiten, wollen wir ein auf Task-Entries definiertes Attribut kennenlernen, nämlich E'COUNT. Dieses Attribut liefert die Anzahl der Tasks, die momentan in der Warteschlange des angegebenen Entry's stehen, wobei der rufende Task nicht mitgezählt wird, wenn der Aufruf des Attributes innerhalb einer Accept-Anweisung des Entry's E stattfindet.

Ein Beispiel für die Implementation eines Task-Body's für die obigen Angaben ist:

```
task body Nachricht is
begin
  loop
    select
      accept Empfang(gross) (Nachricht: out Nachrichtentyp) do
        - empfangen Nachricht
      end Empfang;
    or
      when Empfang(gross)'COUNT = 0 =>
        accept Empfang(mittel) (Nachricht: out Nachrichtentyp) do
          - empfangen Nachricht
        end Empfang;
    or
      when Empfang(gross)'COUNT = 0 and
        Empfang(mittel)'COUNT = 0 =>
        accept Empfang(gering) (Nachricht: out Nachrichtentyp) do
          - empfangen Nachricht
        end Empfang;
        - entsprechende Accept-Anweisungen für die Sende-Entries
    end select;
  end loop;
end Nachricht;
```

Diese Art der Programmierung ermöglicht, wie wir gerade gesehen haben, die Vergabe von *Prioritäten* für Task-Entries. Wie man mittels Prioritäten auf ganze Tasks einwirken kann, werden wir im folgenden Unterkapitel sehen.

Abschließend sei noch erwähnt, daß man bei der Implementierung von Familien von Task-Entries auch

```
accept Empfang( for N in Nachrichtentyp'RANGE) (Nachricht: out Nachrichtentyp) do
  - empfangen Nachricht
if N = gross then
```

```
- tu was
elsif N = mittel then
  - tu was anderes
else
  - tu was ganz anderes
end if;
end Empfang;
```

schreiben kann.

9.7 Diskriminierte Tasks

Task-Typen können auch eine Diskriminante besitzen. Typische Anwendungen sind zum Beispiel

- ein Semaphore, wobei mehrere Tasks parallel auf die Ressource zugreifen können (die Anzahl der möglichen parallelen Zugriffe wird als Wert der Diskriminante beim Anlegen des Task-Objektes übergeben) und
- Setzen der Priorität oder des maximalen Speicherbedarfs eines Tasks mit den passenden Pragmas `priority` und `storage_size`.

Diskriminanten können aber auch verwendet werden, um Tasks eine Art von Identifikation zu geben. Betrachten wir folgendes Beispiel:

```
subtype Task_Range is 1 .. 1000;

function Next_One return Task_Range;

task type Computer(Index: Task_Range := Next_One);

The_Tasks: array (Task_Range) of Computer;
```

Die Funktion `Next_One` liefert einen Wert der `Task_Range` nach dem anderen. Es ist zwar nicht gewährleistet, daß die ID eines Tasks mit dem Index im array übereinstimmt, aber da die Tasks nicht parallel gestartet werden, brauchen wir uns keinerlei Synchronisationsmechanismen im Zusammenhang mit der Funktion `Next_One` überlegen.

Ein anderes Beispiel, wie diskriminierte Tasks und Typerweiterung zusammenarbeiten können, sei wie folgt gegeben:

```
task type T(Job: access Job_Descriptor'CLASS);

task body T is
begin
  Start(Job);
  for I in 1 .. Iterations(Job) loop
    delay Interval(Job);
    Do_It(Job, I);
  end loop;
  Finish(Job);
exception
  when Event: others =>
    Handle_Failure(Job, Event);
end T;
```

Da die Diskriminante ein klassenweiter Typ ist, kann der Dispatching-Mechanismus voll greifen. Das heißt, die Unterprogramme `Start`, `Iterations`, `Interval`, `Do-It`, `Finish` und `Handle-Failure` sind alle Operationen des Typs `Job_Descriptor` und könnten wie folgt aussehen:

```
package Base_Job is
  type Job_Descriptor is abstract tagged null record;
  procedure Start(J: access Job_Descriptor);
  function Iterations(J: access Job_Descriptor) return integer is abstract;
  function Interval(J: access Job_Descriptor) return duration is abstract;
  procedure Do-It(J: access Job_Descriptor; I: integer) is abstract;
  procedure Finish(J: access Job_Descriptor);
  procedure Handle-Failure(J: access Job_Descriptor; E: Exception_Occurence);
end Base_Job;
```

Die meisten Unterprogramme sind abstrakt, um den Benutzer zu zwingen, seine eigenen Implementierungen zu schaffen. Nur `Start` und `Finish` haben als Default Null-Operationen, da das sinnvoll erscheint. `Handle-Failure` sollte ebenfalls eine sinnvolle Default-Operation besitzen.

Eine sinnvolle Implementierung eines Jobs ist dem Leser überlassen.

9.8 Scheduling von Tasks

Programme, die aus mehreren Prozessen bestehen, erfordern, daß man sich Gedanken darüber macht, wann welcher Prozeß nun tatsächlich auf dem Prozessor abläuft. Die Zuordnungsstrategie, die das festlegt, nennt man *Scheduling-Strategie*. Da Ada mit dem vordefinierten Typ *task* Prozesse aus der Verantwortlichkeit des Betriebssystems herausgenommen und dem Laufzeitsystem zugeordnet hat, bestimmt daher dieses das Scheduling[†].

Das Ada Reference Manual legt fest, daß das Scheduling von Tasks *indeterministisch* ist und stellt dem einzelnen Compiler frei, wie es zu realisieren ist. Das Scheduling von Ada-Tasks ist daher *compiler-* und *maschinenabhängig*. Der Programmierer kann und darf sich daher nicht darauf verlassen, in welcher Reihenfolge Tasks dem Prozessor zugeteilt werden. Das einzige reguläre Mittel zur Prozeßsynchronisation und -kommunikation ist und bleibt das Rendezvous.

Die einzige Aufweichung dieses Prinzips ist, daß Ada gestattet, Prioritäten für Tasks zu vergeben. Dieses Mittel sollte jedoch wieder nicht zur Prozeßsynchronisation oder Prozeßkommunikation verwendet werden. Man kann es z.B. benutzen, um sicherzustellen, daß eine *Interrupt-Service-Routine* vorrangig behandelt wird (siehe auch das Kapitel D).

9.9 Rendezvous und Exceptions

Bei der Besprechung der Exceptions im Kapitel 6 sind wir nicht auf parallelen Programmfluß eingegangen, da wir zu dieser Zeit noch zu wenig (oder gar nichts) über das Tasking-Konzept in Ada gewußt haben. Nun aber stellt sich die Frage, wie diese beiden Sprachmittel integriert werden können.

Da das einzige Mittel zur Prozeßkommunikation in Ada das Rendezvous ist, muß also nur festgelegt werden, was mit Exceptions passiert, die während eines Rendezvous, d.h., in einer `Accept`-Anweisung, ausgelöst werden.

[†]Das heißt natürlich nicht, daß ein bestimmtes Laufzeitsystem für Ada nicht Betriebssystemprozesse verwenden kann, um Ada-Tasks zu implementieren.

Ada legt fest, daß solche Exceptions sowohl an den gerufenen als auch an den aufrufenden Task propagiert werden. Das bedeutet, daß die Exception in beiden Tasks entsprechend behandelt werden muß.

9.10 Die Termination von Tasks

*So always look on the bright side of death!
Just before you draw your terminal breath.
Life's a piece of shit,
When you look at it.*

Aus Monty Python's „Das Leben des Brian“.

Jedem Task ist eine Vateinheit zugeordnet. Das kann jener Programmteil sein, in dem das entsprechende Task-Objekt deklariert ist (also z.B. eine Prozedur, ein Paket oder ein anderer Task), oder der Programmteil, in dem das entsprechende Task-Objekt dynamisch mit `new` angelegt worden ist. Man sagt in diesem Fall, daß der Task von seiner Vateinheit abhängt. In allen Fällen gilt, daß die Exekution der Vateinheit nicht beendet werden kann, bevor alle von der Vateinheit abhängigen Tasks sich beendet haben oder beendet worden sind.

Wie wird aber nun ein Task beendet oder wie kann er sich selbst beenden? Die erste Frage haben wir schon beantwortet: mit der `Abort`-Anweisung aus Kapitel 9.4.3. Für die zweite Frage gibt es zwei Antworten:

1. Der Task erreicht die letzte Anweisung seines Body's.
2. Wie wir bereits oben gesehen haben, bestehen Tasks gewöhnlich aus einer `Select`-Anweisung innerhalb einer Endlosschleife. Klarerweise kann so ein Task nie die letzte Anweisung seines Body's erreichen. Um trotzdem eine halbwegs annehmbare Möglichkeit zur Beendigung (Termination) eines solchen Tasks zu schaffen, erlaubt Ada eine *Terminate-Alternative* in einer `Select`-Anweisung, z.B.:

```
task body Assistent is
begin
  loop
    select
      when Sprechstunde =>
        accept betreue do
          - betreue den Studierenden
        end betreue;
    or
      accept organisiere(Aufgabe: Aufgaben_Typ) do
        - organisiere
      end organisiere;
    or
      terminate;
    end select;
  end loop;
end Assistent;
```

In diesem Fall kann sich der Task beenden, wenn alle von ihm abhängigen Tasks sich entweder bereits beendet haben, oder wenn sie eine *Terminate-Alternative* haben und gerade kein Rendezvous durchführen.

3. Die letzte Alternative besteht darin, dass ein Task einen eigenen Shutdown-Entry anbietet, der, wenn er aufgerufen wird, den Task regulär beendet, etwa indem er die Endlosschleife, die die Select-Anweisung umgibt, mittels einer Exit-Anweisung verlässt.

10 Geschützte Objekte

Mit den im vorigen Kapitel vorgestellten Tasks steht dem Ada-Programmierer ein unheimlich mächtiges Werkzeug zur Verfügung, um Multitasking-Anwendungen zu schreiben. Es gibt allerdings in solchen Anwendungen Probleme, für deren Lösung Tasks nicht das beste Hilfsmittel sind.

Ein illustratives Beispiel ist etwa, wenn mehrere parallel laufende Tasks auf ein und dasselbe Objekt lesend und/oder schreibend zugreifen wollen. Wenn es nur Tasks zur Lösung dieses Problems gäbe, müßte man um das vor parallelem Zugriff zu schützende Objekt eine „Task-Hülle“ errichten. Die Lese- und Schreiboperationen des Objekts werden als Entries realisiert und die korrekte Synchronisation ist durch die Semantik des „Hüllen-Tasks“ garantiert.

Allerdings hieße das, mit Kanonen auf Spatzen schießen. Wie hinreichend bekannt, kann dieses Problem nämlich auch durch einen einzigen Semaphore gelöst werden, der bei weitem nicht denselben Overhead mit sich bringt wie ein zusätzlicher, eigentlich mißbräuchlich verwendeter „Hüllen-Task“.

Semaphore sind nun aber Betriebssystemmittel, deren Verwendung die Portierung eines Programmes verschlechtert. Daher gibt es in Ada ein Programmkonstrukt, das die Realisierung von vor parallelem Zugriff geschützten Objekten ermöglicht.

10.1 Spezifikation und Implementierung geschützter Objekte

Ein geschütztes Objekt wird mittels

```
protected type <Name> is
  <Operationen>
private
  <Operationen und/oder Deklarationen>
end <Name>;
```

spezifiziert. Das Schlüsselwort `type` kann wie bei den Tasks weggelassen werden, wenn man nur ein Objekt dieses Typs benötigt. Ebenso kann der gesamte „private-Teil“ weggelassen werden, wenn man ihn nicht braucht.

Genauso wie Tasks besitzen geschützte Objekte auch einen Implementierungsteil. Dieser beinhaltet die Implementierung der definierten Operationen; er kann aber auch etwa zusätzliche, lokale Unterprogramme enthalten.

Ein einfaches Beispiel ist:

```
protected type V is
  function Lies return Wert;
  procedure Schreib(neuer_Wert: Wert);
private
  Datum: Wert;
end V;

protected body V is
```

```
function Lies return Wert is
begin
  return Datum;
end Lies;

procedure Schreib(neuer_Wert: Wert) is
begin
  Datum := neuer_Wert;
end Schreib;
end V;
```

Bei geschützten Records ist gewährleistet, daß die definierten Operationen serialisiert werden, d.h., daß nie gleichzeitig auf die geschützten Daten zugegriffen werden kann.

10.2 Die unterschiedlichen Operationstypen geschützter Objekte

Man kann Prozeduren, Funktionen und Entries für geschützte Records definieren. Mit Funktionen kann nur lesenderweise auf die geschützten Daten zugegriffen werden. Das heißt, daß gegebenenfalls Funktionen zur selben Zeit ausgeführt werden können. Aber nie eine Prozedur gleichzeitig mit einer anderen Operation. Dasselbe gilt für einen Entry.

Entries können ähnlich wie Entries von Tasks mit Wächtern versehen werden. Nur alle jene Entries, deren Wächter den Wert `true` ergeben, können sofort ausgeführt werden, alle anderen werden ähnlich wie bei Tasks in eine Warteschlange gestellt. Wächter bei geschützten Records unterscheiden sich von denen bei Tasks allerdings wesentlich darin, wann die entsprechenden booleschen Ausdrücke ausgewertet werden. Bei geschützten Records wird der Ausdruck eines Wächters evaluiert,

1. wenn der entsprechende Entry aufgerufen wird und
2. es werden die Ausdrücke aller jener Wächter ausgewertet, deren Warteschlange nicht leer ist, wenn die Abarbeitung eines Entry's oder einer Prozedur beendet ist.*

Ein Beispiel, das auch einen Entry beinhaltet:

```
protected type Resource is
  entry Seize;
  procedure Release;
private
  Busy: Boolean := False;
end Resource;

protected body Resource is
  entry Seize when not Busy is
  begin
    Busy := True;
  end Seize;

  procedure Release is
  begin
```

*Bei Funktionen ist das nicht notwendig, da diese ja nur lesen und daher den internen Zustand des geschützten Records nicht verändern können.

```

    Busy := False;
end Release;
end Resource;

```

Es bleibt noch zu sagen, daß die im Kapitel 9 vorgestellte Requeue-Anweisung auch bei der Implementation von Entries geschützter Objekte verwendet werden kann und daß wie bei den Tasks im Private-Teil auch private Entries deklariert werden dürfen.

10.3 Deklaration geschützter Objekte

Deklariert werden geschützte Objekte genau so wie andere Objekte, also z.B.:

```

Printer: Resource;
gemeinsames_Datum: V;

```

10.4 Diskriminierte geschützte Objekte

Ähnlich wie bei Tasks ist auch hier die Möglichkeit vorhanden, geschützte Objekte durch Diskriminanten zu parametrieren.

Wir geben als ein Beispiel eine geschützte Warteschlange. Angenommen wir haben ein Warteschlangenprotokoll wie folgt gegeben:

```

type Queue is abstract tagged null record;
function Is_Empty(Q: in Queue) return boolean is abstract;
function Is_Full(Q: in Queue) return boolean is abstract;
procedure Add_To_Queue(
    Q: access Queue;
    X: Queue_Data) is abstract;
procedure Remove_From_Queue(
    Q: access Queue;
    X: out Queue_Data) is abstract;

```

Eine allgemeine geschützte Warteschlange kann nun wie folgt realisiert werden:

```

protected type PQ(Q: access Queue'CLASS) is
    entry Put(X: in Queue_Data);
    entry Get(X: out Queue_Data);
end;

protected body PQ is
    entry Put(X: in Queue_Data) when not Is_Full(Q.all) is
        begin
            Add_To_Queue(Q,X);
        end Put;

    entry Get(X: out Queue_Data) when not Is_Empty(Q.all) is
        begin
            Remove_From_Queue(Q,X);
        end Get;
end PQ;

```

Die ursprünglich nur zu Informationszwecken vorhandenen gewesenen Funktionen Is_Empty und Is_Full werden nun als Wächter verwendet.

Ein Benutzer kann nun seine eigene Queue implementieren, zum Beispiel durch

```

type My_Queue is new Queue with private;
function Is_Empty(Q: My_Queue) return boolean;
...

```

und kann dann eine geschützte Warteschlange wie folgt deklarieren und benutzen

```

Raw_Queue: aliased My_Queue;
My_Protected_Queue: PQ(Raw_Queue'ACCESS);
...
My_Protected_Queue.Put(An_Item);

```

11 Programmstruktur und Übersetzungsvorgang

Ein *Ada-Programm* besteht aus einer Menge von *Partitions* (siehe auch Kapitel E), von denen jede in einem eigenen Adressraum oder sogar auf einem eigenen Computer ablaufen kann. Jede Partition besteht aus *Bibliothekseinheiten* (*library units*). Bibliothekseinheiten können hierarchisch organisiert werden.

11.1 Getrennte Übersetzung

Eine *Programmeinheit* ist entweder ein Paket, ein Task, eine geschützte Einheit, ein geschützter Entry, eine generische Einheit oder ein explizit deklariertes Unterprogramm. Bestimmte Arten von Programmeinheiten können getrennt übersetzt werden oder sie können geschachtelt innerhalb von anderen Programmeinheiten auftreten.

Der Programtext kann dem Compiler in einer oder mehreren *Übersetzungen* übergeben werden. Jeder Übersetzung besteht aus einer Folge von *Übersetzungseinheiten*. Eine Übersetzungseinheit enthält entweder die Spezifikation, die Implementierung oder die Umbenennung einer Programmeinheit.

Eine Bibliothekseinheit ist eine getrennt übersetzte Programmeinheit auf „höchster Ebene“ und ist immer ein Paket, ein Unterprogramm oder eine generische Einheit. Eine Bibliothekseinheit kann andere (logisch gesehen geschachtelte) Bibliothekseinheiten als Kinder besitzen und andere Programmeinheiten physikalisch geschachtelt beinhalten. Die Wurzel-Bibliothekseinheit einer solchen baumartigen Struktur zusammen mit ihren Kindern und Kindeskindern und so weiter nennt man *Subsystem*.

11.2 Übersetzungs- und Bibliothekseinheiten

Ein *Bibliothekseintrag* ist eine Übersetzungseinheit, die wiederum eine Spezifikation, eine Implementierung oder eine Umbenennung einer Bibliothekseinheit ist. Jede Bibliothekseinheit (außer der vordefinierten Bibliothekseinheit Standard) besitzt eine *Vatereinheit*, die ein Bibliothekspaket oder ein generisches Bibliothekspaket ist. Eine Bibliothekseinheit ist das *Kind* seiner Vater-einheit. Die Wurzel-Bibliothekseinheiten sind die Kinder des vordefinierten Bibliothekspaketes Standard.

Dieser Mechanismus paßt gut mit dem der getaggten Typen zusammen. So kann man etwa in einem Kindpaket einen vom Vaterpaket abgeleiteten Typ deklarieren.

Die Spezifikation und die Umbenennung können *privat* sein, wenn das Schlüsselwort *private* in deren Deklaration verwendet wird, sonst sind sie *public*. Diese privaten Einheiten sind dann nur innerhalb jener Einheiten sichtbar, die Kinder oder Kindeskindern usw. der Vater-einheit dieser privaten Einheit ist.

Ein Beispiel:

```
package Rational_Numbers is
  type Rational is
    record
      Numerator: integer;
      Denominator: positive;
    end record;
  function "="(X,Y: Rational) return Boolean;

  function "/"(X,Y: integer) return Rational;
    - um eine rationale Zahl zu erzeugen

  function "+"(X,Y: Rational) return Rational;
  function "-"(X,Y: Rational) return Rational;
  function "*" (X,Y: Rational) return Rational;
  function "/"(X,Y: Rational) return Rational;
end Rational_Numbers;

package Rational_Numbers.IO is
  procedure Put(R: in Rational);
  procedure Get(R: out Rational);
end Rational_Numbers.IO;

private procedure Rational_Numbers.Reduce(R: in out Rational);
  - privates Kind von Rational_Numbers

with Rational_Numbers.Reduce;
package body Rational_Numbers is
  ...
end Rational_Numbers;

with Rational_Numbers.IO; use Rational_Numbers;
procedure Main is
  R: Rational;
begin
  R := 5/3;
  IO.Put(R);
end Main;

with Rational_Numbers.IO;
package Rational.IO renames Rational_Numbers.IO;
  - eine Umbenennung einer Bibliothekseinheit
```

Jede der oben angeführten Bibliothekseinheiten kann getrennt übersetzt werden.

11.3 Untereinheiten von Übersetzungseinheiten

Ada bietet allerdings noch zusätzliche Unterstützung für getrennt übersetzbare Untereinheiten von Übersetzungseinheiten. Es ist beispielsweise erlaubt, anstelle der Implementation eines Unterprogrammes, eines Paketes, eines Tasks oder eines geschützten Objektes

is separate;

zu schreiben, und die tatsächliche Implementation erst in einer Untereinheit zu realisieren. So eine Untereinheit sieht dann so aus, daß man sie mit separate (<Vater.Einheit>) einleitet

(wobei die entsprechende Vater-Einheit, wie Paket oder Unterprogramm, anzuführen ist) und danach eine korrekte Implementierung folgen läßt. Etwaig notwendige With-Anweisungen sind nötigenfalls vor dem Schlüsselwort `separate` anzugeben. Klarerweise kann auch innerhalb einer Untereinheit mit `is separate` auf weitere Untereinheiten verwiesen werden. Dabei verlängert sich der Name der Vater-Einheit entsprechend um die neu hinzugekommene Untereinheit, die, mit einem Punkt getrennt, hinten anzuhängen ist, z.B. (vgl. das Beispiel aus Kapitel 7):

```
separate (Datum_Manager.Setze)
```

falls wir ein in `Datum_Manager.Setze` lokal definiertes Unterprogramm getrennt übersetzen wollen.

Die Implementierung unseres Paketes `Datum_Manager` könnte also auch folgendermaßen lauten:

```
package body Datum_Manager is
```

```
function Setze(
  Tag: Tag_Typ;
  Monat: Monatsname;
  Jahr: Jahr_Typ)
return Datum
is separate;
```

```
function Tag(
  von_Datum: Datum)
return Tag_Typ
is
begin
  return von_Datum.Tag;
end Tag;
```

```
function Monat(
  von_Datum: Datum)
return Monatsname
is
begin
  return von_Datum.Monat;
end Monat;
```

```
function Jahr(
  von_Datum: Datum)
return Jahr_Typ
is
begin
  return von_Datum.Jahr;
end Jahr;
```

```
begin
  null;
end Datum_Manager;
```


```
separate(Datum_Manager)
function Setze(
  Tag: Tag_Typ;
```

```
  Monat: Monatsname;
  Jahr: Jahr_Typ)
return Datum
is
Dat: Datum := (Tag, Monat, Jahr);
- hier kann man auf die Komponenten von Datum zugreifen
begin
case Monat is
when Jaenner | Maerz | Mai | Juli | August | Oktober | Dezember =>
  null; -- Anzahl der Tage passt
when Feber =>
  if Jahr mod 4 = 0 and jahr mod 1000 ≠ 0 then
    - Schaltjahr
    if Tag not in 1 .. 29 then
      raise falsches_Datum;
    end if;
  else
    - kein Schaltjahr
    if Tag not in 1 .. 28 then
      raise falsches_Datum;
    end if;
  end if;
when others =>
  - hier kommen die Monate mit 30 Tagen
  if Tag not in 1 .. 30 then
    raise falsches_Datum;
  end if;
end case;
return Dat;
end Setze;
```

Dabei können der Body von `Datum_Manager` und der `separate` Teil getrennt übersetzt werden.

11.4 Das Hauptprogramm eines Ada-Programmes

Kapitän: *Wer hat hier eigentlich das Kommando?!?*

Tennisplatzis: 

Dolmetscher: *Er sagt, Ihr sollt die Finger davon lassen!*

Renè Goscinny und Albert Uderzo,
„Asterix als Legionär“.

In Ada wird ein *Hauptprogramm* im herkömmlichen Sinne nicht definiert. Ein Ada-Programm besteht gewöhnlich aus sehr vielen Paketen und Untereinheiten, die irgendwie voneinander abhängen. Meist erlaubt dann der *Linker*, der die einzelnen Einheiten zusammenbindet, den Namen einer Prozedur anzugeben. Diese Prozedur wird dann beim Start des (gebundenen) Ada-Programmes als erste ausgeführt, nachdem alle sonstigen Initialisierungen vorgenommen worden sind.

Genauso wird die Festlegung, welche Bibliothekseinheiten sich in welcher Partition befinden, nicht vom Sprachstandard festgelegt. Jede Partition kann dann natürlich ein eigenes Hauptprogramm beinhalten.

12 Objektorientierte Konzepte in Ada und ihre Umsetzung in die Praxis

In diesem Kapitel wird anhand einiger Beispiele illustriert, wie die objektorientierten Sprachkonstrukte von Ada verwendet werden können, um Standard-Anwendungen der objekt-orientierten Programmierung zu realisieren. Dabei wird nur die von Ada angebotene einfache Vererbung Verwendung finden; mehrfache Vererbung wird in Kapitel 13 behandelt.

12.1 Heterogene Listen

Zunächst geben wir ein Beispiel für eine doppelt verkettete Liste.

```
package Doubly_Linked is

  type Node_Type is tagged limited private;
  type Node_Ptr is access all Node_Type'CLASS;

  procedure Add(Item: Node_Ptr; Head: in out Node_Ptr);
  procedure Remove(Item: Node_Ptr; Head: in out Node_Ptr);

  function Next(Item: Node_Ptr) return Node_Ptr;
  function Prev(Item: Node_Ptr) return Node_Ptr;

private
  type Node_Type is tagged limited
    record
      Prev: Node_Ptr := null;
      Next: Node_Ptr := null;
    end record;
end Doubly_Linked;
```

Diese Spezifikation einer doppelt verketteten Liste kann später durch Typerweiterung mit zusätzlichen Komponenten und Operationen versehen werden.

Wir werden nun dieses Paket benutzen, um eine Assoziationstabelle zu realisieren. Das generische Paket Association hat als Parameter einen Key_Type, einen darauf definierten Gleichheitsoperator und eine darauf definierte Hash-Funktion.

```
with Doubly_Linked;
generic
  type Key_Type is limited private;
  with function "="(Left, Right: Key_Type) return Boolean is <>;
  with function Hash(Key: Key_Type) return Integer is <>;
package Association is

  type Element_Type is new Doubly_Linked.Node_Type with
    record
      Key: Key_Type;
```

```
    end record;
  type Element_Ptr is new Doubly_Linked.Node_Ptr;

  function Key(E: Element_Ptr) return Key_Type;

  type Association_Table(Size: Positive) is limited private;

  procedure Enter(
    Table: in out Association_Table;
    Element: in Element_Ptr);

  function Lookup(
    Table: in Association_Table;
    Key: in Key_Type)
    return Element_Ptr;

  -- andere Operationen ...

private
  type Element_Ptr_Array is array (Integer range <>) of Element_Ptr;
  type Association_Table(Size: Positive) is
    record
      Buckets: Element_Ptr_Array(1..Size);
    end record;
end Association;
```

Eine Association_Table ist also eine Hash-Tabelle, wobei jeder Eintrag in der Tabelle eine doppelt verkettete Liste von Elementen besitzt. Die Elemente können irgend welche Typen sein, die von Element_Type abgeleitet sind. Der Listenkopf ist vom Typ Element_Ptr, der wiederum von Node_Ptr abgeleitet ist (ein nicht getaggter, abgeleiteter Typ). Alle primitiven Operationen von Node_Ptr wie zum Beispiel Add oder Remove werden also von Element_Ptr geerbt.

Wir werden nun diese Hash-Tabelle benutzen, um eine Symboltabelle für einen Compiler einer einfachen Sprache zu definieren. Die Sprache soll Typen, Objekte und Funktionen kennen und die Symboltabelle soll dafür jeweils unterschiedliche Einträge haben.

```
with Association;
package Symbol_Table_Pkg is

  type Identifier is access string;

  function Equal(Left,Right: identifier) return Boolean;
  function Hash(Key: Identifier) return Integer;

  -- Instantiierung
  package Symbol_Association is
    new Association(Identifier, Equal, Hash);
  subtype Symbol_Table is
    Symbol_Association.Association_Table;

  type Type_Symbol is new Symbol_Association.Element_Type with
    record
      Category: Type_Category;
      Size: Natural;
    end record;
  type Type_Ptr is access Type_Symbol;
```

```

type Object_Symbol is new Symbol_Association.Element_Type with
record
  Object_Type: Type_Ptr;
  Stack_Offset: Integer;
end record;

type Function_Symbol is new Symbol_Association.Element_Type with
record
  Return_Type: Type_Ptr;
  Formals: Symbol_Table(5);           -- eine kleine Hash-Tabelle
  Locals: Symbol_Table(19);          -- eine etwas groessere Hash-Tabelle
  Function_Body: Statement_List;
end record;

end Symbol_Table_Pkg;

```

12.2 Präfix-Notation

Die ursprüngliche (und auch bis jetzt verwendete) Notation von Ada für primitive Operationen ist prozedural. Mit Ada2005 wurde auch die von anderen objekt-orientierten Sprachen gewohnte Präfix-Notation erlaubt. D.h., man darf jetzt sowohl `Add(Item, Head)` als auch `Item.Add(Head)` schreiben, wobei wir uns auf das Paket `Doubly_Linked` beziehen.

12.3 Mehrfache Implementationen

Obwohl schon häufig erkannt worden ist, daß es sehr sinnvoll ist, zu einer Abstraktion mehrere Implementationen schaffen zu können, die auf unterschiedliche Eigenschaften der Objekte eingehen, wie zum Beispiel dicht oder dünn besetzte Matrizen, war ein solches Vorgehen erstmals in C++ möglich. Nun bietet auch Ada diese Möglichkeit.

Eine Spezifikation eines abstrakten Mengen-Paketes könnte etwa wie folgt aussehen.

```

package Abstract_Sets is

  type Set is abstract tagged private;

  -- leere Menge
  function Empty return Set is abstract;

  -- Aufbauen einer Menge mit einem Element
  function Unit(Element: Set_Element) return Set is abstract;

  -- Vereinigung von Mengen
  function Union(Left, Right: Set) return Set is abstract;

  -- Durchschnitt zweier Mengen
  function Intersection(Left, Right: Set) return Set is abstract;

  -- Entfernen irgendeines Elements aus der Menge
  procedure Take(
    From: in out Set;

```

```

  Element: out Set_Element) is abstract;

  Element_Too_Large: exception;

private

  type Set is abstract tagged null record;

end Abstract_Sets;

```

Der `Set`-Typ ist ein abstrakter getaggtter privater Typ, dessen interne Darstellung ein Null-Record ist. Für ihn ist eine Reihe von primitiven Operationen in Form von abstrakten Unterprogrammen definiert. Diese können natürlich nicht direkt aufgerufen werden, aber sie können sehr wohl vererbt und umdefiniert werden. Abgeleitete Typen von `Set` müssen dann ihre eigenen Implementierungen schaffen.

Ein Beispiel für so einen abgeleiteten Typ ist folgendes.

```

with Abstract_Sets;
package Bit_Vector_Sets is

  type Bit_Set is new Abstract_Sets.Set with private;

  -- neue Definition der Operationen
  function Empty return Bit_Set;
  function Unit(Element: Set_Element) return Bit_Set;
  function Union(Left, Right: Bit_Set) return Bit_Set;
  function Intersection(Left, Right: Bit_Set) return Bit_Set;
  procedure Take(From: in out Bit_Set;
    Element: out Set_Element);

private
  Bit_Set_Size: constant integer := 64;
  type Bit_Vector is
    array (Set_Element range 0 .. Bit_Set_Size-1) of Boolean;
  pragma Pack(Bit_Vector);

  type Bit_Set is new Abstract_Sets.Set with
  record
    Data: Bit_Vector;
  end record;
end Bit_Vector_Sets;

package body Bit_Vector_Sets is

  function Empty return Bit_Set is
  begin
    return (Data => (others => False));
  end;

  function Unit(Element: Set_Element) return Bit_Set is
  S: Bit_Set := Empty;
  begin
    S.Data(Element) := True;

```



```

    return S;
end;

function Union(Left, Right: Bit_Set) return Bit_Set is
begin
    return (Data => Left.Data or Right.Data);
end;

...

end Bit_Vector_Sets;

```

Eine alternative Implementierung für dünn besetzte Mengen könnte dann eine Listenverwaltung der Elemente verwenden.

```

with Abstract_Sets;
package Linked_Sets is

    type Linked_Set is new Abstract_Sets.Set with private;

    -- neue Definition der Operationen
    function Empty return Linked_Set;
    function Unit(Element: Set_Element) return Linked_Set;
    function Union(Left, Right: Linked_Set) return Linked_Set;
    function Intersection(Left, Right: Linked_Set) return Linked_Set;
    procedure Take(From: in out Linked_Set;
                  Element: out Set_Element);

private
    type Element;
    type Element_Ptr is access all Element;
    type Element is
        record
            the_Element: Set_Element;
            the_next: Element_Ptr;
        end record;
    type Linked_Set is new Abstract_Sets.Set with
        record
            first_Element: Element_Ptr;
        end record;
end Linked_Sets;

```

```

package body Linked_Sets is

    function Empty return Linked_Set is
        S: Linked_Set := (first_Element => null);
    begin
        return S;
    end Empty;
    function Unit(Element: Set_Element) return Linked_Set is
        S: Linked_Set;
    begin
        S.first_Element := new Element'(Element,null);
        return S;
    end Unit;

```

```

end Unit;

...

end Linked_Sets;

```

Wir können dann ein Programm schreiben, das beide Formen von Mengen gleichzeitig verwendet, und wir können eine Konversion zwischen den beiden Darstellungen wie folgt durchführen.

```

procedure Convert(From: in Set'Class; To: out Set'Class) is
    Temp: Set'Class := From;
    Elem: Set_Element;
begin
    -- build up target set, one element at a time
    To := Empty;
    while Temp /= Empty loop
        Take(Temp, Elem);
        To := Union(To, Unit(Elem));
    end loop;
end Convert;

```

Dabei ist zu beachten, daß der Dispatching-Mechanismus voll zur Wirksamkeit gelangt.

Abschließend sei noch darauf hingewiesen, daß das abstrakte Set-Paket auch generisch sein könnte, was die Wiederverwendbarkeit des Paketes noch steigern würde.

12.4 Iteratoren

Eine der grundlegenden Operationen in der objektorientierten Programmierung ist es, eine Operation auf eine Menge von unterschiedlichen Elementen einer Menge anzuwenden. Die Elemente müssen dazu nur einen gemeinsamen Vaternotyp haben und besagte Operation muß diesem Vaternotyp bekannt sein.

Als Beispiel betrachten wir:

```

type Element is ...

package Sets is
    type Set is limited private;
    -- mehrere Operationen des Typs Set

    type Iterator is abstract tagged null record;
    procedure Iterate(S: Set; IC: Iterator'CLASS);
    procedure Action(
        E: in out Element;
        I: in out Iterator) is abstract;
private
    type Node;
    type Ptr is access Node;
    type Node is
        record
            E: Element;
            Next: Ptr;
        end record;

```

```

    type Set is new Ptr;
end Sets;

package body Sets is
  -- Bodies der Set-Operationen

  procedure Iterate(S: Set; IC: Iterator'CLASS) is
    This: Ptr := Ptr(S);
  begin
    while This ≠ null loop
      Action(This.E, IC);
      This := This.Next;
    end loop;
  end Iterate;

end Sets;

```

Wir haben also einen abstrakten Typ Iterator mit einer primitiven Operation Action eingeführt. Die Prozedur Iterate vollführt eine Schleife über die ganze Menge und ruft unter Verwendung des Dispatching-Mechanismus die Prozedur Action auf, die zu den Typen der einzelnen Mengenelementen gehört.

Eine einfache Anwendung dieses Iterators, nämlich das Zählen der Elemente einer Menge könnte wie folgt aussehen.

```

package Sets.Stuff is
  function Count(S: Set) return Natural;
end Sets.Stuff;

package body Sets.Stuff is

  type Count_Iterator is new Iterator with
    record
      Result: Natural := 0;
    end record;

  procedure Action(
    E: in out Element;
    I: in out Count_Iterator) is
  begin
    I.Result := I.Result + 1;
  end Action;

  function Count(S: Set) return Natural is
    I: Count_Iterator;
  begin
    Iterate(S, I);
    return I.Result;
  end Count;
end Sets.Stuff;

```

12.5 Ein allgemeines Mengenkonstrukt als Beispiel

In diesem Abschnitt wollen wir ein Paket realisieren, das den abstrakten Datentyp „Menge“ dermaßen darstellt, daß basierend auf einer „allgemeinen Logik“ unterschiedliche Ausprägungen

von Mengen möglich sind. Im speziellen werden wir Standard-Mengen und Fuzzy-Mengen als Beispiele angeben.

Da der Typ Boolean nicht getaggt ist, können wir ihn nicht einfach durch Typerweiterung für unsere Zwecke anpassen. Es ist daher zunächst notwendig, ein allgemeines Logik-Paket zu erstellen, von dem dann spezielle Logiken wie etwa die Standard-Logik (zweiwertige Logik) oder die Fuzzy-Logik abgeleitet werden können. Wir werden uns dabei auf die Operationen „und“, „oder“ und „not“ beschränken. Unser Paket hat daher folgendes Aussehen:

```

package General_Boolean is

  type gen_bool is abstract tagged private;

  function "and"(left,right: gen_bool) return gen_bool is abstract;
  function "or" (left,right: gen_bool) return gen_bool is abstract;
  function "not"(val: gen_bool) return gen_bool is abstract;

private

  type gen_bool is abstract tagged null record;

end General_Boolean;

Der Typ gen_bool ist also abstrakt genauso wie seine Operationen. Von den „speziellen“ Logiken wird dann erwartet, daß sie diesen Typ erweitern und die Operationen implementieren.

Damit als Basis schaffen wir eine Spezifikation für unser allgemeines Mengen-Paket.

with General_Boolean;

generic
  type item is private;
  type set_bool is new General_Boolean.gen_bool with private;
package General_Set is

  type set is tagged private;

  function Is_Member(I: item; s: set) return set_bool'CLASS;
  function Union(left,right: set) return set;
  function Intersection(left,right: set) return set;

  type Member_Function_Pointer is access
    function(I: item) return set_bool'CLASS;

  function Create(Member_Function: Member_Function_Pointer) return set;

private

  type node;
  type node_pointer is access all node'CLASS;

  type set is tagged
    record
      root: node_pointer;
    end record;

  type node is abstract tagged null record;

```

```

function Process(litem; np: access node) return set_bool'CLASS is abstract;

end General_Set;

```

Dabei steht der generische Parameter `item` für den Typ der Mengenelemente, und `set_bool` ist der generische Parameter, der die Logik festlegt, mit der innerhalb dieses Pakets „gerechnet“ werden soll. Als Operationen beschränken wir uns auf `Is_Member`, um zu erfragen ob ein gegebenes Objekt Element der Menge ist, `Union` zur Vereinigung zweier Mengen und `Intersection` zur Bildung des Durchschnitts zweier Mengen. Da es sehr schwierig ist, für die Schaffung von Mengen eine allgemein passende Operation anzugeben, überlassen wir die Realisierung dieser Operation dem Benutzer des Paketes, indem wir eine Funktion `Create` spezifizieren, die eine Menge zurückgibt und deren Parameter ein Pointer auf eine Funktion ist. Mit Hilfe dieser Funktion kann der Benutzer beliebige (Grund-)Mengen generieren, die er dann mit den anderen Operationen kombinieren kann.

Im Private-Teil des Paketes sehen wir bereits den ersten Teil der Implementierung des Datentyps `set`. Alle Grund-Mengen werden als Blätter eines Baumes repräsentiert, die Operationen `Union` und `Intersection` werden als interne, binäre Knoten eines Baumes dargestellt. Der gesamte Baum ist die interne Darstellung einer Menge. Der Typ `node` ist der abstrakte Vartyp der Knoten des Baumes. Für ihn ist auch eine abstrakte Operation `Process` definiert, die uns für die Traversierung des Baumes gute Dienste leisten wird.

```

package body General_Set is

  type binary_operator is (and_op, or_op);

  type binary_node is new node with
    record
      left, right: node_pointer;
      op: binary_operator;
    end record;

  function Process(litem; np: access binary_node) return set_bool'CLASS;

  function Process(litem; np: access binary_node) return set_bool'CLASS
  is
    left_result: set_bool'CLASS := Process(l,np.left);
    right_result: set_bool'CLASS := Process(l,np.right);
  begin
    case np.op is
      when and_op =>
        return (left_result and right_result);
      when or_op =>
        return (left_result or right_result);
    end case;
  end Process;

  type leaf is new node with
    record
      m: Member_Function_Pointer;
    end record;

  function Process(litem; np: access leaf) return set_bool'CLASS;

  function Process(litem; np: access leaf) return set_bool'CLASS

```

```

is
begin
  return np.m(l);
end Process;

function Is_Member(l: item; s: set) return set_bool'CLASS
is
begin
  return Process(l,s.root);
end Is_Member;

function Union(left,right: set) return set
is
  h: set;
  bn: binary_node;
begin
  bn.op := or_op;
  bn.left := left.root;
  bn.right := right.root;
  h.root := new binary_node'(bn);
  return h;
end Union;

function Intersection(left,right: set) return set
is
  h: set;
  bn: binary_node;
begin
  bn.op := and_op;
  bn.left := left.root;
  bn.right := right.root;
  h.root := new binary_node'(bn);
  return h;
end Intersection;

function Create(Member_Function: Member_Function_Pointer) return set
is
  h: set;
  l: leaf;
begin
  l.m := Member_Function;
  h.root := new leaf'(l);
  return h;
end Create;

end General_Set;

```

In der Implementierung werden von diesem Vartyp die Typen `leaf` und `binary_node` abgeleitet und die Operation `Process` für diese Knoten definiert. Die Funktion `Is_Member` ruft nur `Process` für die Wurzel des Baumes auf. Der Baum wird traversiert und die entsprechende Antwort zurückgeliefert. Dabei werden in den Blättern die vom Benutzer als Funktion definierten Grund-Mengen ausgeführt und deren Ergebnisse über die internen Knoten unter Verwendung der logischen Operationen „and“ und „or“ nach oben weiter gereicht. Die logischen Operationen sind natürlich die der allgemeinen Logik, die erst später festgelegt werden muß.

12.5.1 Die Standard-Logik

Im folgenden Paket wird eine Instanz der zweiwertigen Standard-Logik geschaffen.

```
with General_Boolean;

package Standard_Boolean is

    type stand_bool is new General_Boolean.gen_bool with private;

    true_value: constant stand_bool;
    false_value: constant stand_bool;

    function "and"(left,right: stand_bool) return stand_bool;
    function "or" (left,right: stand_bool) return stand_bool;
    function "not"(val: stand_bool) return stand_bool;

private

    type stand_bool is new General_Boolean.gen_bool with
        record
            b: boolean;
        end record;

    true_value: constant stand_bool := (General_Boolean.gen_bool with true);
    false_value: constant stand_bool := (General_Boolean.gen_bool with false);

end Standard_Boolean;

package body Standard_Boolean is

    function "and"(left,right: stand_bool) return stand_bool
    is
        h: stand_bool;
    begin
        h.b := left.b and right.b;
        return h;
    end "and";

    function "or" (left,right: stand_bool) return stand_bool
    is
        h: stand_bool;
    begin
        h.b := left.b or right.b;
        return h;
    end "or";

    function "not"(val: stand_bool) return stand_bool
    is
        h: stand_bool;
    begin
        h.b := not val.b;
        return h;
    end "not";

end Standard_Boolean;
```

Zusätzlich zu den Operationen wurden auch noch Konstanten für true und false definiert. Die Implementierung folgt den bekannten booleschen Operationen.

12.5.2 Die Fuzzy-Logik

Fuzzy-Logik lässt sich wie folgt aus unserem allgemeinen Paket ableiten:

```
with General_Boolean;

generic
    type fuzzy_float is digits <>;
package Fuzzy_Boolean is

    type fuzzy_bool is new General_Boolean.gen_bool with private;

    zero: constant fuzzy_bool;
    one: constant fuzzy_bool;

    function Convert(float: fuzzy_float) return fuzzy_bool;
    function Convert(fuzzy: fuzzy_bool) return fuzzy_float;

    function "and"(left,right: fuzzy_bool) return fuzzy_bool;
    function "or" (left,right: fuzzy_bool) return fuzzy_bool;
    function "not"(val: fuzzy_bool) return fuzzy_bool;

private

    type fuzzy_bool is new General_Boolean.gen_bool with
        record
            f: fuzzy_float;
        end record;

    zero: constant fuzzy_bool := (General_Boolean.gen_bool with 0.0);
    one: constant fuzzy_bool := (General_Boolean.gen_bool with 1.0);

end Fuzzy_Boolean;
```

Diesmal haben wir ein generisches Paket vorliegen, deren Parameter den reellen Typ der Fuzzy-Logik einstellbar gestaltet. Zusätzlich wurden Konstanten für zero und one deklariert.

```
package body Fuzzy_Boolean is

    function Convert(float: fuzzy_float) return fuzzy_bool
    is
    begin
        return (General_Boolean.gen_bool with float);
    end Convert;

    function Convert(fuzzy: fuzzy_bool) return fuzzy_float
    is
    begin
        return fuzzy.f;
    end Convert;

    function "and"(left,right: fuzzy_bool) return fuzzy_bool
```

```

is
  h: fuzzy_bool;
begin
  h.f := left.f * right.f;
  return h;
end "and";

function "or" (left,right: fuzzy_bool) return fuzzy_bool
is
  h: fuzzy_bool;
begin
  h.f := 1.0 - (1.0 - left.f) * (1.0 - right.f);
  return h;
end "or";

function "not"(val: fuzzy_bool) return fuzzy_bool
is
  h: fuzzy_bool;
begin
  h.f := 1.0 - val.f;
  return h;
end "not";

begin
  if fuzzy_float'First  $\neq$  0.0 or else fuzzy_float'Last  $\neq$  1.0 then
    raise constraint_error;
  end if;
end Fuzzy_Boolean;

```

In der Implementierung wurde für „und“ die Multiplikation der beiden Operanden gewählt. Man könnte natürlich auch das Minimum der beiden Operanden wählen oder gar einen Typ schaffen, wo die Operationen „and“ und „or“ abstrakt sind, und dann etwa in Kindpaketen die beiden unterschiedlichen Realisierungsmöglichkeiten implementieren.

Da bei der Definition des generischen Parameters für den reellen Typ der Fuzzy-Logik syntaktisch nicht berücksichtigt werden kann, daß der reelle Typ nur Werte zwischen 0.0 und 1.0 annehmen kann, geschieht diese Überprüfung im Paket-Body zum Zeitpunkt der Instantiierung.

12.5.3 Ein Standard-Beispiel

Wir benötigen zunächst ein Paket, das allgemeine Definitionen enthält.

```

package Defs is
  type fuzzy_real is digits 3 range 0.0 .. 1.0;
  type real is digits 10;
end Defs;

```

Der Typ `real` soll der Typ der Elemente unserer Mengen sein; der Typ `fuzzy_real` wird erst bei der Fuzzy-Logik interessant, er stellt nämlich den reellen Typ der Fuzzy-Logik dar.

Sodann instantiieren wir eine Standard-Menge mit dem Elementtyp `Defs.real`.

```

with General_Set, Standard_Boolean, Defs;

package Standard_Set is new General_Set(Defs.real,Standard_Boolean.stand_bool);

```

Die generische Funktion `Closed.Interval` gestattet uns, später Grund-Mengen zu instantiieren.

```

with Standard_Boolean;
generic
  type real is digits <>;
  l,u: real;
function Closed.Interval(I: real)
  return Standard_Boolean.stand_bool'CLASS;

function Closed.Interval(I: real)
  return Standard_Boolean.stand_bool'CLASS
is
begin
  if I  $\leq$  u and I  $\geq$  l then
    return Standard_Boolean.true_value;
  else
    return Standard_Boolean.false_value;
  end if;
end Closed.Interval;

```

Solche Intervalle werden mit

```

with Closed.Interval, Defs;

function Interval_1 is new Closed.Interval(Defs.real,0.0,10.0);

with Closed.Interval, Defs;

function Interval_2 is new Closed.Interval(Defs.real,20.0,30.0);

```

geschaffen und in der procedure `main_stand` weiter verarbeitet.

```

with Defs, Closed.Interval, Standard_Boolean, Standard_Set, Interval_1, Interval_2;
use Standard_Boolean;

procedure main_stand
is
  int_1_ptr: Standard_Set.Member_Function_Pointer := Interval_1'Access;
  int_2_ptr: Standard_Set.Member_Function_Pointer := Interval_2'Access;

  my_set_1, my_set_2, my_set_3: Standard_Set.set;

begin
  my_set_1 :=
    Standard_Set.Create(
      Member_Function => int_1_ptr);

  my_set_2 :=
    Standard_Set.Create(
      Member_Function => int_2_ptr);

  my_set_3 := Standard_Set.Union(my_set_1, my_set_2);

  if Standard_Set.Is_Member(5.0, my_set_3) = Standard_Boolean.true_value then

```

```

    ...
end if;

end main_stand;

```

12.5.4 Ein Fuzzy-Beispiel

Als erstes schaffen wir eine geeignete Fuzzy-Logik.

```

with Fuzzy_Boolean, Defs;

package My_Fuzzy_Boolean is new Fuzzy_Boolean(Defs.fuzzy_real);

```

Als nächstes generieren wir eine passende Fuzzy-Menge.

```

with My_Fuzzy_Boolean, General_Set, Defs;

package Fuzzy_Set is new General_Set(Defs.real, My_Fuzzy_Boolean.fuzzy_bool);

```

Die generische Funktion Triangle erlaubt uns, später die für Fuzzy-Mengen typischen dreiecksförmigen Verteilungen der Membership-Funktionen zu bilden (vgl. Abb. 12.1).

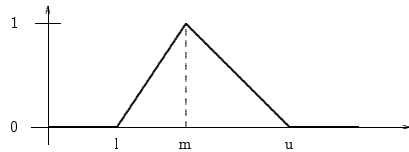


Abbildung 12.1: Eine dreiecksförmige Verteilung einer Membership-Funktion

```

with Fuzzy_Boolean;
generic
  type real is digits <>;
  l,m,u: real;
  type fuzzy_real is digits <>;
  with package This_Fuzzy_Boolean is new Fuzzy_Boolean(fuzzy_real);
  function Triangle(l: real)
    return This_Fuzzy_Boolean.fuzzy_bool'CLASS;

  function Triangle(l: real)
    return This_Fuzzy_Boolean.fuzzy_bool'CLASS
is
begin
  if l<=l or else l>=u then
    return This_Fuzzy_Boolean.zero;
  elsif l=m then
    return This_Fuzzy_Boolean.one;
  elsif l>l and l<m then
    return This_Fuzzy_Boolean.Convert(
      fuzzy_real((l-l)/(m-l)));
  else
    -- l>m and l<u then
    return This_Fuzzy_Boolean.Convert(

```

```

      fuzzy_real((l-u)/(m-u)));
    end if;
  end Triangle;

```

Solche Instanzen finden wir in

```

with Triangle, My_Fuzzy_Boolean, Defs;

function Triangle_1 is new Triangle(
  Defs.real,0.0,1.0,2.0,Defs.fuzzy_real,My_Fuzzy_Boolean);

with Triangle, Defs, My_Fuzzy_Boolean;

function Triangle_2 is new Triangle(
  Defs.real,1.0,2.0,3.0,Defs.fuzzy_real,My_Fuzzy_Boolean);

```

In der Prozedur main_fuzzy werden diese Mengen entsprechend weiterbehandelt.

```

with Defs, Triangle, My_Fuzzy_Boolean, Fuzzy_Set, Ada.Text_IO, Triangle_1, Triangle_2;

procedure main_fuzzy
is
  tr_1_ptr: Fuzzy_Set.Member_Function_Pointer := Triangle_1'Access;
  tr_2_ptr: Fuzzy_Set.Member_Function_Pointer := Triangle_2'Access;

  my_set_1, my_set_2, my_set_3: Fuzzy_Set.set;

  package Real_IO is new Ada.Text_IO.Float_IO(Defs.fuzzy_real);

begin
  my_set_1 :=
    Fuzzy_Set.Create(
      Member_Function => tr_1_ptr);

  my_set_2 :=
    Fuzzy_Set.Create(
      Member_Function => tr_2_ptr);

  my_set_3 := Fuzzy_Set.Union(my_set_1, my_set_2);

  Real_IO.Put(My_Fuzzy_Boolean.Convert(Fuzzy_Set.Is_Member(1.5, my_set_3)));

end main_fuzzy;

```

13 Mehrfachvererbung in Ada

*Immer, wenn du glaubst,
es geht nicht mehr,
kommt von irgendwo ein Lichtlein her.*

Alte Volksweisheit.

Wie aus den bisherigen Erläuterungen klar sein sollte, unterstützt Ada nur einfache Vererbung. Man kann jedoch durch geschicktes Kombinieren verschiedener Sprachmittel mehrere Arten von mehrfacher Vererbung realisieren.

13.1 Kombination von Implementation und Abstraktion

In Sprachen wie C++ ([Str86]) und Eiffel ([Mey88]), in denen Klassen die einzige Art von Modul sind, ist Vererbung die gewöhnliche Methode, um Abstraktionen zu kombinieren. Um zum Beispiel in Eiffel eine Klasse Bounded.Stack[T] zu realisieren, könnte man von einer abstrakten Klasse Stack[T] und von einer zweiten Klasse Array[T] erben. Die Klasse Array[T] würde man dann benutzen, um die abstrakten Operationen zu implementieren, die nicht von der Klasse Stack[T] definiert sind. Idealerweise sollten dabei die Array-Operationen vor dem Benutzer verborgen werden.

Diese Art von Vererbung kann in Ada bereits ohne getaggte Typen, nur unter Zuhilfenahme von Paketen erreicht werden.

```
package Bounded is
  type Bounded.Stack(Size: Natural := 0) is private;
  procedure Push(S: in out Bounded.Stack; Element: T);
  procedure Pop(S: in out Bounded.Stack);
  function Top(S: Bounded.Stack) return T;
private
  type T.Array is array (Integer range <>) of T;
  type Bounded.Stack(Size: Natural := 0) is
    record
      Data: T.Array(1..Size);
    end record;
end Bounded;
```

13.2 Mixins

Eine andere Art von Mehrfach-Vererbung, sogenannte *Mixins*, wurde in [BC90] eingeführt. Dabei kann einer der Vater-Klassen keine eigenen Instanzen haben, vielmehr existiert sie nur, damit eine Menge von Eigenschaften von ihr geerbt werden kann. Typischer Weise wird diese abstrakte Mixin-Klasse für genau diese Zwecke geschaffen.

In Ada kann man Mixins mit Hilfe von Typerverweiterung und generischen Einheiten realisieren. Dabei definiert die generische Einheit die Mixin-Klasse und der Typ, der als aktueller generischer Parameter übergeben wird fungiert als „Vater“.

Als Beispiel diene die Möglichkeit, beliebige Objekte mit einer Version zu versehen.

```
with OM;                                -- Objekt-Manager bietet eindeutige Objekt-IDs an
with VM;                                -- Versionsmanager bietet Versionskontrolle an
generic
  type Parent is abstract tagged private;
package Versioned is
  - Ein Versionsobjekt hat eine ID, die eine Menge von
  - Versionen dieses Objekts identifiziert, und eine
  - Versionsnummer, die mit der ID kombiniert das Objekt
  - eindeutig identifiziert.
  type Versioned.Object is abstract new Parent with private;

  procedure Create_New_Version(
    O: in Versioned.Object;
    New_O: out Versioned.Object);
  - gibt eine neue Version des uebergebenen Objekts zurueck

  function Version_Number(
    O: Versioned.Object)
  return VM.Version_Number;
  - gibt die Version eines gegebenen Objekts zurueck

  procedure Get_Version(
    ID_From: in Versioned.Object;
    Version: in VM.Version_Number;
    Object: out Versioned.Object);
  - fuer ein gegebenes Objekt und eine gegebene Versionsnummer
  - wird diese Version des Objekts zurueckgeliefert

private

  type Versioned.Object is abstract new Parent with
    record
      ID: OM.Object_ID := OM.Unique_ID;
      Version: VM.Version_Number := VM.Initial_Version;
    end record;

end Versioned;
```

Ein typisches Beispiel für kaskadierende Mixins ist ein Window-System. Wir beginnen mit einem einfachen Basis-Window und verschiedenen Operationen

```
type Basic.Window is tagged limited private;
procedure Display(W: in Basic.Window);
procedure Mouse_Click(
  W: in out Basic.Window;
  Where: in Mouse.Coords);
```

Dann definieren wir eine Menge von Mixins wie zum Beispiel

```
generic
  type Some_Window is abstract new Basic.Window with private;
package Label_Mixin is
  type Window_With_Label is abstract new Some_Window with private;
  - einige Operationen werden über definiert
```

```

procedure Display(W: in Window_With_Label);

-- einige neue kommen dazu
procedure Set_Label(
    W: in out Window_With_Label;
    S: in String);
function Label(
    W: Window_With_Label)
    return String;
private
type Window_With_Label is abstract new Some_Window with
    record
        Label: String_Quark := Null_Quark;
        -- eine X-Window-artige eindeutige ID fuer einen String
    end record;
end Label_Mixin;

```

Im Body dieses generischen Pakets implementieren wir die überdefinierten und neu hinzu gekommenen Operationen. Zum Beispiel kann die neue Version von Display wie folgt aussehen

```

procedure Display(W: Window_With_Label) is
begin
    Display(Some_Window(W));
    -- das zugrundeliegende Window wird angezeigt,
    -- indem man die Operation des Vaternotyps verwendet
    if W.Label ≠ Null_Quark then
        -- jetzt wird der Label ausgegeben, wenn er vorhanden ist
        Display_On_Screen(XCoord(W), YCoord(W)-5, Value(W.Label));
    end if;
end Display;

```

wobei XCoord und YCoord von Basic_Window geerbt sind und die Koordinaten implizieren, wo der Label ausgegeben wird.

Wir können nun eine ganze Reihe solcher Pakete deklarieren und schließlich schreiben.

```

package Frame is
    type My_Window is new Basic_Window with private;
    ...
private
    package Add_Label is new Label_Mixin(Basic_Window);
    package Add_Border is new Border_Mixin(Add_Label.Window_With_Label);
    package Add_Menu_Bar is new Menu_Bar_Mixin(
        Add_Border.Window_With_Border);

    type My_Window is new
        Add_Menu_Bar.Window_With_Menu_Bar with null record;
end Frame;

```

Die verschiedenen Operationen der unterschiedlichen Mixins können durch Renames-Anweisungen im Paket-Body selektiv auch vom Paket Frame angeboten werden.

13.3 Interfaces

Ein weiteres Konzept von Mehrfachvererbung, das erstmals in der weit verbreiteten Sprache Java eingesetzt wurde, sind die *Interfaces*. Grundlage dieses Konzepts ist, dass Mehrfachvererbung

keine Probleme macht, wenn ein Vater keine Komponenten und keine konkret implementierten Operationen besitzt. So ein Vater wird *Interface* genannt.

In Ada2005 kann ein getaggtter Typ erweitert werden, indem zu diesem Vaternotyp eine beliebige Anzahl von Interfaces hinzugefügt werden. Diese Interfaces werden *Progenitoren* genannt. Der Vaternotyp darf auch ein Interface sein.

Zum Beispiel ist in

```

type NT is new T and Int1 and Int2 with ...;

```

T der Vaternotyp und Int1 sowie Int2 sind Progenitoren.

Ein Interface wird angelegt wie ein getaggtter Typ, nur dass es etwa

```

type Int1 is interface;

```

heißen muss. Darüber hinaus müssen alle primitiven Operationen entweder abstrakt sein oder eine Null-Implementierung haben.

Interfaces können auch von anderen Interfaces komponiert werden. Zum Beispiel

```

type Int2 is interface;
...
type Int3 is interface and Int1;
...
type Int4 is interface and Int1 and Int2;

```

Interfaces können auch limited sein.

Die Implementierung eines von Interfaces abgeleiteten Typs muss natürlich auch die primitiven Operationen der Progenitoren beinhalten.

13.4 Mehrfache Sichtweisen eines Objektes

Schließlich gibt es auch Mehrfachvererbung, wo die Nachfolgerklasse wirklich von mehreren Klassen abgeleitet wird und wo Anwender dieser Klasse sie als irgendeine ihrer Vorgängerklassen „sehen“ wollen. In Ada ist es möglich, das zu erreichen, indem man Access-Diskriminanten einsetzt.

Mit Access-Diskriminanten kann man erreichen, daß eine Komponente eines Records auf den Record verweist, der ihn umschließt. Damit kann man komplexe, verkettete Strukturen errichten und mehrfache Sichtweisen einer Struktur ermöglichen. Betrachten wir beispielsweise folgendes Programmfragment

```

type Outer is limited private;

private

type Inner(Ptr: access Outer) is limited ...

type Outer is limited
    record
        ...
        Component: Inner(Outer'ACCESS);
        ...
    end record;

```

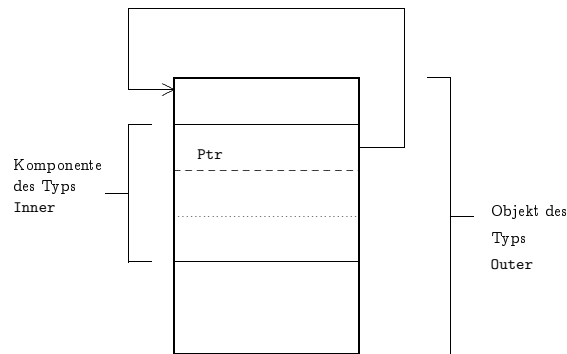



Abbildung 13.1: Eine selbstreferenzierende Struktur

Component des Typs Inner hat eine Access-Diskriminante Ptr, die zur umschließenden Instanz des Records Outer weist. Und das deshalb, weil das Attribut ACCESS, wenn es auf den Namen eines Record-Typs innerhalb seiner Deklaration angewandt wird, auf die aktuelle Instanz des Typs verweist. Wenn wir nun ein Objekt des Typs Outer deklarieren

```
Obj: Outer;
```

dann entsteht eine selbstreferenzierende Struktur einer Gestalt, wie sie in Abbildung 13.1 dargestellt ist. Allerdings kann man diese Struktur nicht damit vergleichen, wenn etwa eine Komponente sich selbst aufgrund einer Zuweisung referenziert. Alle Instanzen des Typs Outer referenzieren sich selbst und Ptr kann seinen Wert nicht ändern, da Diskriminanten konstant sind.

So ein einfaches Beispiel bringt nicht viel; wir werden daher ein etwas komplizierteres studieren. Wir nehmen an, wir hätten ein komplexes organisches Molekül, das wir über verschiedene Monitore darstellen wollen. Unsere Datenstruktur wird so beschaffen sein, daß die einzelnen Monitore linear verkettet sind, jeder Monitor aber direkten Zugriff auf das Molekül besitzt, um es darstellen zu können.

```
type Monitor;
type Monitor_Ptr is access all Monitor'CLASS;

type Monitored_Object is abstract tagged limited
  record
    First: Monitor_Ptr;           -- der erste der Liste der Monitoren
    -- andere Komponenten koennen durch Typerweiterung
    -- spaeter hinzugefuegt werden
  end record;

type Monitored_Object_Ptr is access all Monitored_Object'CLASS;

type Monitor is abstract tagged limited
  record
    Next: Monitor_Ptr;
    Obj: Monitored_Object_Ptr;
    -- weiter Komponenten koennen entsprechend den
    -- Beduerfnissen der einzelnen Monitore
    -- ueber Typerweiterung hinzugefuegt werden
```

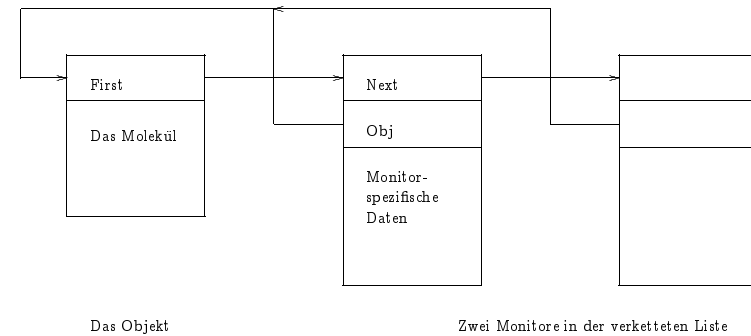


Abbildung 13.2: Eine verkettete Liste von Monitoren

```
end record;

procedure Update(M: in out Monitor) is abstract;
...
procedure Notify(MO: Monitored_Object'CLASS) is
  This_Mon: Monnitor_Ptr := MO.First;
begin
  while This_Mon ≠ null loop
    Update(This_Mon.all);           -- Dispatching-Mechanismus!
    This_Mon := This_Mon.Next;
  end loop;
end Notify;
```

Dabei ist Notify eine klassenweite Operation des Typs Monitored_Object, die alle Update-Operationen der Monitore in der verketteten Liste aufruft.

Unser Objekt, das das Molekül repräsentiert, habe den Typ Molecule.

```
type Monitored_Molecule is new Monitored_Object with
  record
    M: Molecule;
  end record;
...
Medikament: Monitored_Molecule;
```

Dann können wir das Objekt beliebig modifizieren und von Zeit zu Zeit das veränderte Molekül mittels

```
Notify(Medikament);
```

ausgeben. Eine mögliche interne Konfiguration könnte wie in Abbildung 13.2 aussehen.

Jetzt wollen wir annehmen, daß wir eines unserer Windows von vorhin verwenden wollen, um unser Molekül darzustellen. Wir benötigen also ein Objekt, das sich einerseits wie ein Window verhält und andererseits wie ein Monitor. Zunächst definieren wir ein Mixin, das ein Monitor ist und definieren seine Update-Operation neu.

```
type Monitor_Mixin(Win: access Basic_Window'CLASS) is
```

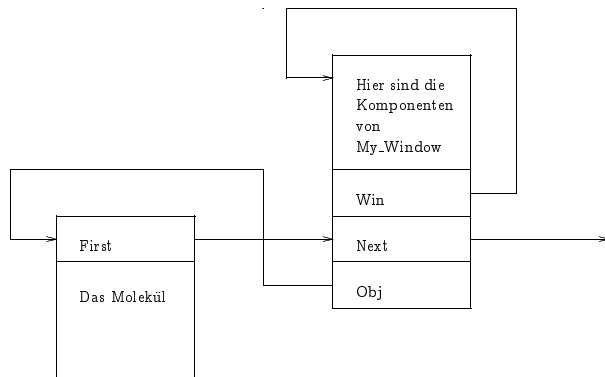


Abbildung 13.3: Ein Window in der Monitor-Liste

```
new Monitor with null record;
procedure Update(M: in out Monitor_Mixin);
```

Die Implementation könnte wie folgt aussehen:

```
procedure Update(M: in out Monitor_Mixin) is
  -- wir geben einfach das Window neu aus
begin
  Display(M.Win.all);
end Update;
```

Und jetzt mixen wir diesen Monitor_Mixin in einen beliebigen Window-Typ, indem wir schreiben:

```
type Window_That_Monitors is new My_Window with
  record
    Mon: Monitor_Mixin(Window_That_Monitors'ACCESS);
  end record;
```

Die innere Komponente Mon hat eine Access-Diskriminante, die zum äußeren Typ verweist. Seine Monitor-Komponente wiederum kann nun, wie in Abbildung 13.3 gezeigt, in die Liste eingehängt werden. Wenn nun Notify mit dem Molekül aufgerufen wird, bewirkt das, daß verschiedene Update-Operationen angestoßen werden. Die Update-Operation des Typs Monitor_Mixin ruft Display für den Typ Window_That_Monitors auf, dem es als Teil angehört, und dieser hat alle Information über das Window und darüber, ein Monitor zu sein.

Die vorgeführten Beispiele dieses Abschnittes belegen, daß Ada auch für viele wichtige Anwendungen geeignet ist, die mehrfache Vererbung voraussetzen.

14 Objektorientierung und Tasking

Ada2005 kennt mehrere Arten von Interfaces, die man sich wie folgt angeordnet denken kann: Zunächst unterscheidet man zwischen *limited* und *nonlimited* Interfaces. Limited Interfaces können *synchronized* sein. Diese können entweder durch einen protected Type oder durch einen Task (Type) implementiert werden.

Es gibt dann noch *task interfaces*, die nur von einem Task, und *protected interfaces*, die nur von einem protected object implementiert werden können.

Wir können also schreiben:

```
type LI is limited interface;
type LI2 is limited interface;
type SI is synchronized interface;
type TI is task interface;
type PI is protected interface;
```

Zusätzlich können wir auch noch Operationen definieren, die allerdings alle entweder abstract oder null sein müssen.

Diese Interfaces können wir auch kombinieren, sofern keine Konflikte auftreten. Also etwa:

```
type TI2 is task interface and LI and TI;
type LI3 is limited interface and LI and LI2;
type TI3 is task interface and LI and LI2;
type SI2 is synchronized interface and LI and SI;
```

Konkrete Tasks oder Protected Objects können dann passende Interfaces implementieren, wobei die definierten Operationen entweder durch Entries oder durch Prozeduren realisiert werden können.

Zum Beispiel kann das Interface

```
package Pkg is
  type TI is task interface;
  procedure P(X: in TI) is abstract;
  procedure Q(X: in TI; I: in integer) is null;
end Pkg;
```

wie folgt implementiert werden

```
package PT1 is
  task type TT1 is new TI with
  entry P;
  entry Q(I: in integer);
  end TT1;
end PT1;
```

oder wie

```
package PT2 is
  task type TT2 is new TI with
  entry P;
  end TT2;
  procedure Q(X: in TT2; I: in integer);
end PT2;
```

oder sogar wie

```
package PT3 is
  task type TT3 is new TI with end;
  procedure P(X: in TT3);
  -- Q als Null-Procedure geerbt
end PT2;
```

Klarerweise kann man die Schlüsselwörter overriding und not overriding in diesen Fällen auch für Task Entries und für protected Operationen verwenden.

15 Low-Level Features

Für Anwendungen, die maschinennahes Programmieren erfordern, bietet Ada verschiedene Arten der Unterstützung an. Die wichtigsten werden wir in den folgenden Kapiteln kennenlernen.

15.1 Darstellungsklauseln

Die *Darstellungsklausel* (*representation clause*) erlaubt es, die Art und Weise festzulegen, in der Daten intern dargestellt werden sollen. Es gibt vier solche Darstellungsklauseln, nämlich zur Festlegung

1. der Länge,
2. der internen Darstellung von Aufzählungstypen,
3. der internen Darstellung von Record-Typen und
4. von Adressen.

15.1.1 Die Spezifikation von Längen

Man kann in Ada den Speicherbedarf von Objekten explizit festlegen, indem man etwa definiert:

```
Bits: constant := 1;
type mein_integer is range -100 .. 100;
for mein_integer'SIZE use 8*Bits;
```

Das Attribut T'SIZE bezieht sich auf den Speicherplatz den eine Variabale des Typs T braucht. Die Konstante Bits wurde nur eingeführt um die Lesbarkeit zu erhöhen.

Ein anderes Beispiel ist:

```
Bytes: constant := 8*Bits;
kB: constant := 1024*Bytes;
task type Tastatur_Treiber is
...
end Tastatur_Treiber;
for Tastatur_Treiber'SORAGE_SIZE use 3*kB;
```

Dabei bezieht sich das Attribut T'SORAGE.SIZE auf den Speicherplatz den der entsprechende Task benötigt, wenn er aktiviert ist (die Daten, nicht den Code). Falls nicht genug Speicher vorhanden ist, wird die Exception storage_error ausgelöst.

15.1.2 Die interne Darstellung von Aufzählungstypen

Angenommen wir haben einen einfachen Festplatten-Controller, dessen Befehlsvorrat gegeben ist durch:

```
type command is (home, seek, step, where, read, write);
```

Falls jedes dieser Kommandos eigentlich eine mnemonische Darstellung eines Bit-Musters ist, kann man mittels einer Darstellungsklausel dieses Muster festlegen:

```
for command'SIZE use 6*Bits;
for command use (
  home => 8#00#,
  seek => 8#04#,
  step => 8#06#,
  where => 8#10#,
  read => 8#50#,
  write => 8#70#);
```

wobei zu beachten ist, dass die Zuordnung in aufsteigender Reihenfolge der Zahlen rechts von => zu erfolgen hat.

Jedenfalls erlaubt uns die Verwendung solcher Darstellungsklauseln, gut lesbare Programme zu schreiben, obwohl wir eigentlich mit Bit-Mustern hantieren.

15.1.3 Die interne Darstellung von Record-Typen

Auch die interne Darstellung von Records können wir exakt festlegen, z.B.:

```
type IO_Port is
record
  data: integer range 0 .. 255;
  ready: boolean;
  interrupt_enabled: boolean;
end record;
```

Eine mögliche interne Darstellung ist:

```
word: constant := 1*Bytes;
for IO_Port use
  record at mod 2;
    data at 0*word range 0..7;
    ready at 1*word range 3..3;
    interrupt_enabled at 1*word range 7..7;
  end record;
```

Das at mod-Konstrukt legt fest, daß das Alignment des Records in jeweils zwei ganzen Speichereinheiten erfolgt. Mit at legt man fest, in welcher Speichereinheit sich die Record-Komponente befindet, und mit range legt man die Bit-Position der Komponente fest.

In Ada95 hat sich die Syntax des at mod-Konstruktes geändert; sie lautet nun

```
for IO_Port'ALIGNMENT use 2;
```

Allerdings ist die alte Notation noch gültig.

Neu ist auch, daß man nun die Ordnung der Bits festlegen kann. Zum Beispiel

```
for R'BIT_ORDER use Low_Order_First;
```

15.1.4 Die Festlegung von Adressen

In Ada ist auch möglich, genau festzulegen, wo im Speicher eine Variable oder eine Konstante liegt, z.B.:

```
DA_Konverter: IO_Port;
for DA_Konverter use at 16#17A4#;
```

Außerdem kann man die Startadresse eines Unterprogrammes, eines Paketes oder eines Tasks exakt festlegen, z.B.:

```
procedure Shut_down;
for Shut_down use at 16#0815#;
```

In diesem Fall bewirkt der Aufruf der Prozedur Shut_down die Ausführung der unter der angegebenen Adresse stehenden Maschinensprachen-Routine.

Interrupt-Service-Routinen können in Ada realisiert werden, indem die Adresse einer geschützten Prozedur festgelegt wird, z.B.:

```
protected power_failure is
  procedure fail;
  for fail use at 16#AFFE#;
end power_failure;

protected body power_failure is
  procedure fail;
  begin
    - rette sich, wer kann!
  end power_failure;
```

Wenn der Hardware-Interrupt auftritt und die Exekution zur Adresse 16#AFFE# springt, so ist dies genauso, als ob der Entry fail aufgerufen worden wäre.

Auch hier hat sich in Ada95 die Syntax geändert; man schreibt jetzt

```
for DA_Konverter'ADDRESS use 16#17A4#;
```

Auch hier ist die alte Notation noch gültig.

15.2 Attribut-Definitionen

Für manche Attribute ist es möglich, sie für eigene Typen selbst zu definieren. Ein Beispiel ist

```
type Matrix is ...
for Matrix'Read use My_Matrix_Reader;
for Matrix'Write use My_Matrix_Writer;
```

15.3 Unkontrollierte Deallokation und Unkontrollierte Typ-Konversion

Wie wollen noch zwei generische Unterprogramme erwähnen. Diese dienen zur unkontrollierten *Deallokation* von Speicher und zur unkontrollierten *Typ-Konversion*. Ihre Deklaration lauten:

```
generic
  type object is limited private;
  type name is access object;
  procedure unchecked_deallocation (x: in out name);

generic
  type source is limited private;
  type target is limited private;
  function unchecked_conversions (s: source) return target;
```

15.4 Storage Pool Management

In Ada ist es möglich, die Default-Einstellung für die Speicher-Allokation und -Deallokation auf dem Heap durch eigene Algorithmen zu ersetzen. Für jeden Access-Typ gibt es einen eigenen *Storage Pool*, der für jenen Speicherbereich steht, wo Objekte dieses Typs angelegt werden. Ein Storage Pool kann aber auch dazu genutzt werden, daß mehrere Access-Typen auf ihm angelegt werden. So sind zum Beispiel alle von einem Vartyp abgeleiteten Typen ein und demselben Storage Pool zugeordnet.

Das Paket System.Storage_Pools bietet die Möglichkeit, von dem abstrakten Typ Root_Storage_Pool eigene Storage Pools abzuleiten. Ein Beispiel ist:

```
with System.Storage_Pools;
with System.Storage_Elements; use System;
package Mark_Release_Storage is

  type Mark_Release_Pool(Size: Storage_Elements.Storage_Count) is
    new Storage_Pools.Root_Storage_Pool with private;

  type Pool_Mark is limited private;

  - controlled operations
  procedure Initialize(Pool: in out Mark_Release_Pool);

  procedure Finalize(Pool: in out Mark_Release_Pool);

  - storage pool operations
  procedure Allocate(
    Pool: in out Mark_Release_Pool;
    Storage_Address: out Address;
    Size_In_Storage_Elements: in Storage_Elements.Storage_Counts;
    Alignment: in Storage_Elements.Storage_Counts);

  procedure Deallocate(
    Pool: in out Mark_Release_Pool;
    Storage_Address: in Address;
    Size_In_Storage_Elements: in Storage_Elements.Storage_Counts;
    Alignment: in Storage_Elements.Storage_Counts);

  function Storage_Size(Pool: Mark_Release_Pool)
    return Storage_Elements.Storage_Count;

  - Mark_Release_Pool-specific operations
  procedure Set_Mark(
    Pool: in Mark_Release_Pool;
```

```

    Mark: out Pool.Mark);
- markiert den momentanen Zustand des Pools, um ihn
- spaeter wiederherstellen zu koennen

procedure Release.To_Mark(
  Pool: in out Mark_Release.Pool;
  Mark: in Pool.Mark);
- der gesamte Speicher, der seit dem angegebenen Mark
- belegt wurde, wird freigegeben

private
  ...
end Mark_Release.Storage;
```

Es ist zu beachten, daß die Prozeduren `Allocate` und `Deallocate` implizit aufgerufen werden, wenn ein entsprechendes Objekt angelegt werden (`new ...`) oder zerstört werden soll (`Unchecked_Deallocation`).

Um nun dieses Paket zu verwenden, müssen wir zunächst so einen Pool anlegen und ihn dem entsprechenden Access-Typ zuordnen. Das geschieht etwa so:

```

use Mark_Release.Storage;
Big_Pool: Mark_Release.Pool(50_000);

type Some_Type is ...;
type Some_Access is access Some_Type;
for Some_Access'SORAGE_POOL use Big_Pool;
```

15.5 Andere Arten maschinennaher Programmierung

Ada bietet noch andere Möglichkeiten, maschinennahe zu programmieren z.B. können Teile des Programmes in Maschinen-Code geschrieben und inkludiert werden. Maschinenabhängige Teile des Ada-Systems findet man im Paket `System`.

16 I/O und das Calendar-Paket

Wir haben jetzt den gesamten Sprachumfang von Ada behandelt, der geneigte Leser wird jedoch feststellen, daß wir immer noch nicht in der Lage sind, Daten ein- oder auszugeben. In diesem Kapitel werden wir uns sowohl mit Themen betreffend die Ein- und Ausgabe beschäftigen, als auch noch einmal auf den Begriff Zeit und seine Behandlung in Ada eingehen.

16.1 Ein-/Ausgabe

In Ada gibt es zwei generische Pakete zur Ein-/Ausgabe auf Files:

1. `Sequential_IO` für sequentielle Files und
2. `Direct_IO` für Files mit wahlfreiem Zugriff (random-access files).

Darüberhinaus gibt es auch noch Unterstützung für die Ausgabe von Texten z.B. auf Ein-/Ausgabegeräten wie Drucker oder Terminals.

16.1.1 Ein-/Ausgabe für binäre Daten

Die beiden generischen Pakete `Sequential_IO` und `Direct_IO` dienen zur Ausgabe von beliebigen Daten auf Files. Beide enthalten die folgenden Operationen:

- `close`,
- `create`,
- `delete`,
- `open`,
- `read`,
- `reset` und
- `write`.

Außerdem enthalten sie die Funktionen:

- `end_of_file`,
- `form`,
- `is_open`,
- `mode` und
- `name`.

Zusätzlich enthält `Direct_IO` die Prozedur `Set_Index` und die Funktionen `Size` und `Index`, um den direkten Zugriff auf Records zu ermöglichen.

Da beide Pakete generisch sind, müssen vor ihrer Verwendung Instanzen geschaffen werden, z.B.:

```
package integer_IO is new Sequential_IO(Element_Type => integer);
```

Es können aber auch Records als `Element_Type` angegeben werden. Manche Compiler erlauben nicht, daß Pointer im `Element_Type` enthalten sind.

Alle File-Operationen werden sequentiell ausgeführt, d.h., wenn mehrere Prozesse auf ein File zugreifen, muß die Anwendung sicherstellen, daß keine Probleme auftreten.

Alle Operationen sind auf einem *File* definiert. Files müssen aber vorher deklariert werden und sind vom Typ `file_Type`, z.B.:

```
integer_file: integer_IO.file_Type;
```

Die Zugriffsart, die ein File erlaubt, wird festgelegt, wenn das File geöffnet oder kreiert wird. Es gibt die folgenden drei *Zugriffsarten* (*modes*):

- `in_file`,
- `out_file` und
- `inout_file`.

Nähere Information findet man im Ada Reference Manual.

16.1.2 Ein-/Ausgabe für lesbare Daten

Das Paket `Text_IO` ermöglicht die Ausgabe von lesbarem Text sowohl auf Files als auch auf Ein-/Ausgabegeräte. *Standard-Input* und *Standard-Output* sind die Defaults für die Ein-/Ausgabegeräte. Man kann diese jedoch mittels der Funktionen `Set_input` und `Set_output` umdefinieren. Es gibt die folgenden Operationen:

- `close`,
- `create`,
- `delete`,
- `open`,
- `reset`,
- `is_open`,
- `end_of_file`,
- `mode`,
- `name` und
- `form`.

Text_IO arbeitet auf zeilenorientierten Geräten, daher gibt es zur Unterstützung die folgenden Operationen:

- `set_line_length`,
- `set_page_length`,
- `new_line`,
- `skip_line` (für Eingabe),
- `col` (die laufende Spalte),
- `set_col`,
- `line` (die laufende Zeile),
- `set_line`,
- `page` (die laufende Seitennummer),
- `new_page`,
- `skip_page` (für Eingabe) und
- `end_of_page`.

Ein-/Ausgabe für Strings

Für die Ein-/Ausgabe von Strings existieren die Operationen:

- `Put`, die einen String ausgibt, und
- `Get`, die einen String einliest.

Ein-/Ausgabe für andere Typen

Für ganze Zahlen, reelle Zahlen und für Aufzählungen gibt es generische Unterpakete von Text_IO, die entsprechende Get- und Put-Operationen zur Verfügung stellen. Instantiierungen sind z.B.:

```
package small_int_IO is new Text_IO.Integer_IO(small_int);
package mod_IO is new Text_IO.Modular_IO(mod_int);
package real_IO is new Text_IO.Float_IO(real);
package Tag_IO is new Text_IO.Enumeration_IO(Tag);
```

Details findet man im Ada Reference Manual.

16.2 Das Paket Calendar

Das Paket Calendar enthält Definitionen von Typen und Operationen, die für den Zeitbegriff in Ada nötig sind. Die Spezifikation lautet:

```
package Ada.Calendar is
  type Time is private;

  subtype Year_Number is Integer range 1901 .. 2099;
  subtype Month_Number is Integer range 1 .. 12;
  subtype Day_Number is Integer range 1 .. 31;
  subtype Day_Duration is Duration range 0.0 .. 86.400.0;

  function Clock return Time;

  function Year (Date : Time) return Year_Number;
  function Month (Date : Time) return Month_Number;
  function Day (Date : Time) return Day_Number;
  function Seconds (Date : Time) return Day_Duration;

  procedure Split (Date : in Time;
                  Year : out Year_Number;
                  Month : out Month_Number;
                  Day : out Day_Number;
                  Seconds : out Day_Duration);

  function Time_Of (Year : Year_Number;
                  Month : Month_Number;
                  Day : Day_Number;
                  Seconds : Day_Duration := 0.0)
    return Time;

  function "+" (Left : Time; Right : Duration) return Time;
  function "+" (Left : Duration; Right : Time) return Time;
  function "-" (Left : Time; Right : Duration) return Time;
  function "-" (Left : Time; Right : Time) return Duration;

  function "<" (Left, Right : Time) return Boolean;
  function "<=" (Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">=" (Left, Right : Time) return Boolean;

  Time_Error : exception;

private
  ...
end Ada.Calendar;
```

-- not specified by the language

Mit diesen Operationen kann man nun einigermaßen bequem mit dem Begriff Zeit innerhalb von Ada-Programmen hantieren. Außer den Vergleichsoperationen und den Operationen zum „Rechnen“ mit Variablen des Typs `time` und `duration` beinhaltet das Paket Calendar auch Unterprogramme zum Aufbau von Variablen des Typs `time` (`time.of`) und zum Konvertieren von solchen Variablen in Einheiten von Jahr, Monat, Tag und Sekunden (`split`, `year`, `month`, `day` und `seconds`). Die Funktion `clock` gibt die aktuelle Zeit zurück.

Die Anhänge zum Sprachstandard

An den Sprachstandard von Ada [Ada95] angefügt sind mehrere Anhänge, nämlich

Anhang A: Vordefinierte Sprachumgebung: Dieser Anhang enthält alle vordefinierten Pakete.

Anhang B: Schnittstellen zu anderen Sprachen: Behandelt werden die Sprachen C, COBOL und Fortran.

Anhang C: Systemprogrammierung: Beschreibt unter anderem Unterstützung für Interrupts und zur eindeutigen Identifizierung von Tasks.

Anhang D: Echtzeitsysteme: Beinhaltet unter anderem die Problematik des Scheduling und der Zeitmessung.

Anhang E: Verteilte Systeme: Behandelt werden Partitionen, Konsistenzprobleme verteilter Systeme und Remote Procedure Calls.

Anhang F: Informationssysteme: Beinhaltet die Dezimalzahlendarstellung und Picture-String-Formate (vgl. COBOL).

Anhang G: Numerik: Behandelt werden komplexe Arithmetik und Gleitkommaarithmetik.

Anhang H: Sicherheitsrelevante Systeme: Beinhaltet unter anderem Instrumentierungskonstrukte.

Anhang I: Veraltete Konstrukte: Stellt den Bezug zu Ada83 her.

Anhang J: Sprachdefinierte Attribute

Anhang K: Sprachdefinierte Pragmas

Wir werden uns im folgenden mit den Anhängen C, D und E auseinandersetzen.

C Systemprogrammierung

C.1 Unterstützung für Interrupts

Interrupts werden gewöhnlich von der Hardware oder von System-Software ausgelöst. Jener Teil der Software, der abläuft, wenn ein Interrupt auftritt, heißt *Interrupt-Handler*. Wenn ein bestimmter Interrupt den Ablauf eines bestimmten Interrupt-Handlers aktiviert, so sagt man dieser Interrupt-Handler ist mit dem Interrupt *verbunden* (*attached*). Als Interrupt-Handler kommen nur parameterlose Prozeduren geschützter Objekte in Frage.

Um das alles zu bewerkstelligen, gibt es zwei reservierte Pragmas: Mit

```
pragma Interrupt_Handler(<Name>);
```

wird festgelegt, daß die geschützte Prozedur <Name> als Interrupt-Handler in Frage kommt. Die entsprechende Spezifikation des geschützten Objekts muß eine Bibliothekseinheit sein.

Mit

```
pragma Attach_Handler(<Name>,expression);
```

wird ein Interrupt zu einem Interrupt-Handler gebunden, dabei muß *expression* vom Typ `Interrupts.Interrupt_ID` sein.

Mit dem Paket `Ada.Interrupts` können Interrupt-Handler dynamisch verbunden oder entbunden werden. Genaueres findet sich in der folgenden Spezifikation.

```
with System;

package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is
    access protected procedure;

  function Is_Reserved (Interrupt : Interrupt_ID)
    return Boolean;

  function Is_Attached (Interrupt : Interrupt_ID)
    return Boolean;

  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;

  procedure Attach_Handler(
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);

  procedure Exchange_Handler(
    Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
```

```
  Interrupt : in Interrupt_ID);

  procedure Detach_Handler(Interrupt : in Interrupt_ID);

  function Reference(Interrupt : Interrupt_ID)
    return System.Address;

private
  ... -- not specified by the language
end Ada.Interrupts;
```

C.2 Identifikation und Attributierung von Tasks

Es gibt das folgende sprachdefinierte Paket:

```
package Ada.Task_Identification is

  type Task_ID is private;
  Null_Task_ID : constant Task_ID;
  function "=" (Left, Right : Task_ID) return Boolean;

  function Image (T : Task_ID) return String;
  function Current_Task return Task_ID;
  procedure Abort_Task (T : in out Task_ID);

  function Is_Terminated(T : Task_ID) return Boolean;
  function Is_Callable (T : Task_ID) return Boolean;

private
  ... -- not specified by the language
end Ada.Task_Identification;
```

Mit den Operationen dieses Pakets ist es möglich, jeden Task eindeutig zu identifizieren. Darüberhinaus existieren zwei Attribute, die sehr nützlich sind:

<code>T'IDENTITY</code>	gibt die <code>Task_ID</code> des Tasks <code>T</code> zurück.
<code>E'CALLER</code>	gibt die <code>Task_ID</code> des Tasks zurück, dessen Entry-Call gerade behandelt wird; nur erlaubt innerhalb einer <code>Accept</code> -Anweisung des Entry's <code>E</code> oder eines Entry-Body's eines Entry's <code>E</code> eines geschützten Objektes.

Wenn eine Instanz des generischen Pakets `Ada.Task_Attributes` in einer Partition geschaffen wird, so wird jeder Task, der in dieser Partition kreiert wird, implizit mit einem Objekt des generischen `Parameters` Attribut versehen. Dieses Objekt fungiert als benutzerdefiniertes Attribut des Tasks. Der Benutzer kann dann damit anstellen, was immer er will.

```
with Ada.Task_Identification; use Ada.Task_Identification;
```

```
generic
  type Attribute is private;
  Initial_Value : in Attribute;
package Ada.Task_Attributes is

  type Attribute_Handle is access all Attribute;

  function Value(T : Task_ID := Current_Task)
    return Attribute;

  function Reference(T : Task_ID := Current_Task)
    return Attribute_Handle;

  procedure Set_Value(
    Val : in Attribute;
    T : in Task_ID := Current_Task);

  procedure Reinitialize(T : in Task_ID := Current_Task);

end Ada.Task_Attributes;
```

D Echtzeitprogrammierung

D.1 Prioritäten

Es gibt zwei zusätzliche Pragmas: Mit dem ersten wird die Priorität von Interrupt-Handlern festgelegt und zwar mit

```
pragma Interrupt_Priority(<expression>);
```

wobei die *expression* auch weggelassen werden kann, wenn es nur eine Interrupt-Priorität gibt. So ein *Pragma* ist nur unmittelbar in der Spezifikation eines Tasks oder eines geschützten Objektes erlaubt.

Sonst verwendet man

```
pragma Priority(<expression>);
```

um Prioritäten festzulegen. So ein *Pragma* ist nur unmittelbar in der Spezifikation eines Tasks, eines geschützten Objektes oder im Deklarationsteil eines Unterprogrammbody's erlaubt.

Es muß mindestens eine Interrupt-Priorität und mindestens 30 normale Prioritätsstufen geben.

- Wenn ein Task aktiviert wird, erbt er die Priorität des Tasks, der ihn aktiviert.
- Während eines Rendezvous erbt der Task, der den Entry-Call bearbeitet (Server) die Priorität des Tasks, der den Entry-Call abgesetzt hat (Client).
- Während ein Task eine Operation eines geschützten Objekts ausführt, erbt er die Ceiling-Priorität des geschützten Objektes (vgl. Abschnitt D.4), das ist das Maximum aller Prioritäten aller Tasks, die dieses geschützte Objekt verwenden.

D.2 Scheduling mit Prioritäten

In diesem Abschnitt wird festgelegt, welcher Task ablauffähig wird, wenn mehr als nur ein Task dazu bereit ist. Dazu ist es nötig ein *Dispatching-Modell* und eine *Dispatching-Politik* zu haben.

Das Task-Dispatching-Modell

Unter *Dispatching* versteht man den Prozeß, der aus allen Tasks, die im Zustand *ready* sind, einen auswählt, der in den Zustand *running* übergeht.

Das geschieht zu bestimmten *Dispatching-Zeitpunkten*. Solche Zeitpunkte sind,

- wenn ein Task blockiert,
- wenn er *ready* wird,

- wenn eine Accept-Anweisung fertig wird und
- wenn ein Task terminiert.

Weitere Dispatching-Zeitpunkte werden im folgenden eingeführt.

Wichtige Begriffe für Dispatching-Politiken sind *Ready-Warteschlangen*, Task-Zustände und *Task-Preemption*. Jeder Prozessor besitzt eine Ready-Warteschlange für jede Prioritätsebene. Zu jedem Zeitpunkt beinhaltet so eine Ready-Warteschlange alle Tasks, die gerade dazu bereit sind, auf dem entsprechenden Prozessor abzulaufen.

Ein Task kann sich in mehreren Ready-Warteschlangen unterschiedlicher Prozessoren befinden. Wenn er jedoch auf einem bestimmten Prozessor ablauffähig wird, wird er aus allen Ready-Warteschlangen, in denen er gewartet hat, entfernt.

Für jeden Prozessor gibt es auch einen *running* Task, das ist der Task, der gerade abläuft. Es wird zu einem Dispatching-Zeitpunkt immer derjenige Task *running*, der an erster Stelle der Ready-Warteschlange mit höchster Priorität steht.

Wenn ein Task mit höherer Priorität ankommt, als der, der gerade abläuft, oder wenn die Dispatching-Politik es bestimmt, tritt ebenfalls ein Dispatching-Zeitpunkt ein.

D.3 Die Standard-Dispatching-Politik

Mit dem *Pragma*

```
pragma Task_Dispatching_Policy(<Name>);
```

wird die Dispatching-Politik festgelegt. Dabei darf *<Name>* entweder *FIFO_Within_Priorities* oder ein implementationsabhängiger Name sein.

Wenn *FIFO_Within_Priorities* angegeben wird, so sollte auch die *Ceiling_Locking-Politik* (vgl. Abschnitt D.4) gewählt werden.

Die *FIFO_Within_Priorities*-Politik funktioniert folgendermaßen:

- Wenn ein blockierter Task *ready* wird, wird er am Ende der Ready-Warteschlange seiner Priorität eingereiht.
- Wenn sich die Priorität eines *ready* Tasks, der nicht läuft, ändert (vgl. D.6), so wird er aus seiner derzeitigen Ready-Warteschlange entfernt und am Ende der seiner neuen Priorität entsprechende Ready-Warteschlange eingereiht. Sollte sich die Priorität aufgrund der Aufhebung einer geerbten Priorität vermindern, wird der Task jedoch am Anfang der seiner neuen Priorität entsprechende Ready-Warteschlange eingereiht.
- Wenn sich die Priorität eines laufenden Tasks ändert, wird er am Ende der Ready-Warteschlange seiner neuen Priorität eingereiht.
- Wenn ein Task eine Delay-Anweisung ausführt, die ihn nicht blockiert, wird er am Ende der Ready-Warteschlange seiner Priorität eingereiht.

Alle oben angeführten Zeitpunkte sind Dispatching-Zeitpunkte.

Wenn ein Task preempted wird, wird er am Anfang der Ready-Warteschlange seiner Priorität eingereiht.

Seit 2005 gibt es darüber hinaus noch die folgenden Task-Dispatching-Politiken:

FIFO_Within_Priorities: Innerhalb einer Prioritätsstufe: First In First Out, wobei ein Task Tasks niedrigerer Priorität unterbrechen kann.

Non_Preemptive_FIFO_Within_Priorities: wie oben, nur dass ein Task nicht von einem höher prioren Task unterbrochen werden kann, außer er ist geblockt oder führt eine Delay-Anweisung aus.

Round_Robin_Within_Priorities: Innerhalb einer Prioritätsstufe werden die Tasks in Zeitscheiben abgearbeitet.

EDF_Across_Priorities: Earliest Deadline First Scheduling sorgt dafür, dass immer der Task mit der nächstgelegenen Deadline abgearbeitet wird.

Mit dem Pragma `Priority_Specific_Dispatching` kann man verschiedene Politiken über verschiedene Prioritätsstufen mischen. Z.B. setzt

```
pragma Priority_Specific_Dispatching(Round_Robin_Within_Priority, 1, 1);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
pragma Priority_Specific_Dispatching(FIFO_Within_Priority, 11, 24);
```

Round Robin für Prioritätsstufe 1, EDF für die Ebenen 2 bis 10 und FIFO für 11 bis 24.

Interessant ist, dass

```
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 5);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 6, 10);
```

nicht dasselbe ist wie

```
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
```

da im ersten Fall alle Tasks im Bereich 6 bis 10 unabhängig von den Deadlines den Vorzug vor denen im Bereich 2 bis 5 erhalten.

Mit dem Pragma `Relative_Deadline(RD)` kann man die Deadline eines Tasks zu der Zeit setzen, wenn er kreiert wird. Das Paket `Ada.Dispatching.EDF` beinhaltet Operationen, mit denen man die Deadline eines Tasks dynamisch setzen und verändern kann.

D.4 Priority Ceiling Locking

In diesem Abschnitt wird das Zusammenspiel von Scheduling von mit Prioritäten behafteten Tasks und geschützten Objekten behandelt. Wieder gibt es ein Pragma

```
pragma Locking_Policy(<Name>);
```

wobei `<Name>` entweder `Ceiling_Locking` oder ein implementationsabhängiger Name ist.

Ceiling-Locking funktioniert folgendermaßen:

- Jedes geschützte Objekt hat einen sogenannten *Ceiling*. Dieser ist das Maximum aller Prioritäten, die Tasks haben können, die das geschützte Objekt verwenden.
- Während ein Task eine Operation eines geschützten Objektes ausführt, erbt er diese Ceiling-Priorität.
- Vor dem Aufruf einer Operation eines geschützten Objekts wird die Priorität des Tasks mit dem Ceiling des Objektes verglichen; sollte die Priorität des Tasks höher sein als das Ceiling, wird die Exception `programm_error` ausgelöst.

D.5 Entry-Queuing-Politiken

Mittels des Pragmas

```
pragma Queuing_Policy(<Name>);
```

wird festgelegt, wie Tasks in die Entry-Warteschlange eingereiht werden. Es gibt zwei vordefinierte Strategien, eine Implementation kann aber wieder eigene definieren. Die beiden vordefinierten sind: `FIFO_Queueing` und `Priority_Queueing`. `FIFO_Queueing` ist die für Tasks aus dem Kapitel 9 bekannte Strategie. Die `Priority_Queueing`-Strategie funktioniert so:

- Die rufenden Tasks werden entsprechend ihrer Priorität eingereiht; innerhalb einer Prioritätsebene heißt das FIFO-mäßig.
- Verübergende Veränderungen der Priorität eines eingereihten Tasks wirken sich nicht aus.

D.6 Dynamische Prioritäten

Mit dem folgenden Paket kann man Prioritäten von Tasks zur Laufzeit, d.h. dynamisch ändern:

```
with System;
with Ada.Task_Identification;                                -- See C.7.1

package Ada.Dynamic_Priorities is

  procedure Set_Priority(
    Priority : in System.Any_Priority;
    T : in Ada.Task_Identification.Task_ID :=
      Ada.Task_Identification.Current_Task);

  function Get_Priority(
    T : Ada.Task_Identification.Task_ID :=
      Ada.Task_Identification.Current_Task)
    return System.Any_Priority;

end Ada.Dynamic_Priorities;
```

D.7 Über die Zeit

*Die Uhr vertreibt
meine Zeit
ins Nirgends
ich ringe um Raum
mit der zwölffingrigen
Null*

Rose Ausländer
Aus: „Die Uhr“

*Ausnahmen sind im Ada Reference Manual [Ada95] nachzulesen.

Das Paket `Ada.Real.Time` definiert eine Echtzeit:

```
package Ada.Real.Time is

  type Time is private;
  Time_First : constant Time;
  Time_Last : constant Time;
  Time_Unit : constant := implementation-defined-real-number;

  type Time.Span is private;
  Time.Span_First : constant Time.Span;
  Time.Span_Last : constant Time.Span;
  Time.Span_Zero : constant Time.Span;
  Time.Span_Unit : constant Time.Span;

  Tick : constant Time.Span;
  function Clock return Time;

  function "+" (Left : Time; Right : Time.Span) return Time;
  function "+" (Left : Time.Span; Right : Time) return Time;
  function "-" (Left : Time; Right : Time.Span) return Time;
  function "-" (Left : Time; Right : Time) return Time.Span;

  function "<" (Left, Right : Time) return Boolean;
  function "<=" (Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">=" (Left, Right : Time) return Boolean;

  function "+" (Left, Right : Time.Span) return Time.Span;
  function "-" (Left, Right : Time.Span) return Time.Span;
  function "-" (Right : Time.Span) return Time.Span;

  function "*" (Left : Time.Span; Right : Integer) return Time.Span;
  function "*" (Left : Integer; Right : Time.Span) return Time.Span;
  function "/" (Left, Right : Time.Span) return Integer;
  function "/" (Left : Time.Span; Right : Integer) return Time.Span;

  function "abs"(Right : Time.Span) return Time.Span;

  function "<" (Left, Right : Time.Span) return Boolean;
  function "<=" (Left, Right : Time.Span) return Boolean;
  function ">" (Left, Right : Time.Span) return Boolean;
  function ">=" (Left, Right : Time.Span) return Boolean;

  function To_Duration (TS : Time.Span) return Duration;

  function To_Time.Span (D : Duration) return Time.Span;

  function Nanoseconds (NS : Integer) return Time.Span;
  function Microseconds (US : Integer) return Time.Span;
  function Milliseconds (MS : Integer) return Time.Span;

  type Seconds_Count is range implementation-defined;

  procedure Split(T : in Time; SC : out Seconds_Count; TS : out Time.Span);
```

```
function Time.Of(SC : Seconds_Count; TS : Time.Span) return Time;

private
  ...                               -- not specified by the language
end Ada.Real.Time;
```

Von einer Implementation wird erwartet, daß sie genaue Angaben über etwaige Gangungenauigkeiten dieser Uhr und über die Genauigkeit einer Delay-Anweisung macht. Details finden sich im Ada Reference Manual [Ada95].

D.8 Synchrone Task-Kontrolle

Hier wird ein semaphor-artiges Objekt definiert, das die Möglichkeit zur Realisierung von komplexen Warteschlangen bietet.

```
package Ada.Synchronous_Task_Control is

  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
  ...                               -- not specified by the language
end Ada.Synchronous_Task_Control;
```

Die Operationen `Set_True` und `Set_False` verhalten sich atomar zueinander und zur Operation `Suspend_Until_True`. Wenn `Suspend_Until_True` aufgerufen wird, wird die Exception `Program_Error` ausgelöst, falls schon ein anderer Task auf das `Suspension_Object` wartet.

D.9 Asynchrone Task-Kontrolle

Mit folgendem Paket kann man Tasks asynchron unterbrechen und fortsetzen:

```
with Ada.Task_Identification;

package Ada.Asynchronous_Task_Control is

  procedure Hold(T : in Ada.Task_Identification.Task_ID);
  procedure Continue(T : in Ada.Task_Identification.Task_ID);
  function Is_Held(T : Ada.Task_Identification.Task_ID)
    return Boolean;
end Ada.Asynchronous_Task_Control;
```

D.10 Execution Time

Das Paket `Ada.Execution.Time` hat Operationen mit denen man die tatsächliche Ausführungszeit eines Tasks erfragen kann und Operationen, um damit Berechnungen durchzuführen.

Das Paket `Ada.Execution.Time.Timers` erlaubt es, zeitabhängige Handler-Routinen zu definieren. Damit kann auf Überläufe von oberen Schranken von Laufzeiten reagiert werden.

Mit dem Paket `Ada.Execution.Time.Group_Budgets` kann man Tasks zu Gruppen zusammenfügen, für die dann ein gemeinsames Zeit-Budget gilt.

Das Paket `Ada.Real.Time.Timing_Events` schließlich erlaubt, Handler für Zeitereignisse aufzusetzen.

E Verteilte Systeme

DE DARM
 san in de katakombm
 es heaz
 is in da augustinakiachn
 da keapa
 in da kapuzinagruf
 so haums die keisarin
 mariaderesia fadeut
 damid ma dreimoi
 zoen mus
 waumas seng wi

aus Ernst Kein, „Wiener Grottenbahn“.

Die Einheit der Verteilung in Ada ist die *Partition*. Eine Partition besteht aus einer oder mehreren Bibliothekseinheiten. Typischerweise befindet sich eine Partition auf einem Rechner im Netzwerk und alle seine Bibliothekseinheiten befinden sich im selben logischen Adreßraum. Die Schnittstelle zwischen Partitionen ist eine oder mehrere Spezifikationsteile von Paketen.

Dieser Anhang legt zwar fest, wie Partitionen aufgebaut sein müssen, wie sie elaboriert werden, wie sie sich bei der Abarbeitung verhalten und wie sie untereinander kommunizieren, jedoch nicht, wie die Konfiguration von Partitions syntaktisch erfolgt und wie der zugrundeliegende Kommunikationsmechanismus im Detail aussieht. Diese Teile sind implementationsabhängig.

Wichtig ist vor allem, daß die Unterschiede zwischen der Entwicklung eines verteilten und eines nicht-verteilten Systems minimal sind und daß das bekannte strenge Typenkonzept von Ada erhalten bleibt.

E.1 Kategorisierung von Bibliothekseinheiten

Prinzipiell wird zwischen *aktiven* und *passiven* Partitionen unterschieden. Aktive Partitionen haben einen eigenen Kontrollfluß, passive nicht. Es gibt zwei Arten von passiven Partitionen, die durch Pragmas festgelegt werden.

E.1.1 Pure Library Units

Solche „reine Bibliothekseinheiten“ beinhalten z.B. nur Typen und Konstante, sie dürfen keinen sich ändernden internen Zustand besitzen und sind passiv. Eine Kopie einer Pure Library Unit wird zu jeder Partition dazugegeben, die eine Bibliothekseinheit beinhaltet, die sich auf diese Pure Library Unit bezieht (with). Eine Pure Library Unit darf sich selbst nur auf andere Pure Library Units beziehen. Das zugehörige Pragma lautet pragma Pure(Name).

E.1.2 Shared Passive Library Units

Shared Passive Library Units werden verwendet, um globale Daten zu verwalten, auf die von mehreren aktiven Partitionen zugegriffen wird. Auch sie sind passiv, allerdings können sie nur

genau einer Partition zugeordnet werden. Bei solchen Einheiten ist sichergestellt, daß es nicht möglich ist, auf Daten oder Tasks einer Partition über die in einer Shared Passive Library Unit deklarierten Objekte von einer anderen Partition aus zuzugreifen. So eine Einheit darf daher keinen Access-Typ, der auf einen klassenweiten Typ zeigt, keinen Task-Typ und kein geschütztes Objekt mit Entries auf oberster Bibliotheksebene beinhalten. Sie darf sich nur auf andere Shared Passive Library Units und auf Pure Library Units beziehen. Das zugehörige Pragma lautet pragma Shared.Passive(Name).

E.1.3 Remote Types Library Units

Diese dienen dazu, Pointerwerte zwischen Partitionen austauschen zu können. Normalerweise macht das keinen Sinn, da eine Adresse außerhalb ihres Adreßraumes keine wirkliche Bedeutung mehr besitzt. Manchmal aber kann so etwas durchaus sinnvoll sein, etwa bei einem Pointer, der auf eine systemweite Ressource zeigt. Details kann man im Ada Reference Manual nachlesen. Das zugehörige Pragma lautet pragma Remote.Types(Name).

E.1.4 Remote Call Interface Library Units

Das sind jene Einheiten einer Partition, die mit *Remote Procedure Calls* von einer anderen Partition aus aufgerufen werden können. Dabei befindet sich der Body in genau einer Partition. Die Spezifikation befindet sich in allen jenen Partitionen, die eine Einheit beinhalten, die sich auf diese Einheit beziehen; statt des Body's hält sich dort aber nur ein *Body Stub* auf. Dieser besteht aus den notwendigen Aufrufen, um einen RPC (Remote Procedure Call) auszuführen.

Es gibt sowohl synchrone RPCs als auch asynchrone RPCs. Bei synchronen RPCs wartet der Rufer, bis das Resultat der Berechnung von der anderen Partition eintrifft. Bei asynchronen RPCs setzt der Rufer nach Absetzen des RPC's seine Arbeit fort, ohne auf das Resultat zu warten. Klarerweise dürfen nur Prozeduren, die ausschließlich In-Parameter besitzen, asynchrone RPCs sein. Die Asynchronität einer solchen Prozedur wird mit dem Pragma Asynchronous(Name) festgelegt.

Tritt während eines RPC's eine Exception auf, wird sie im synchronen Fall an den Rufer weitergereicht, im asynchronen geht sie schlichtweg verloren.

Das zugehörige Pragma lautet pragma Remote.Call.Interface(Name).

E.1.5 Normal Library Units

Wenn eine Bibliothekseinheit durch kein Pragma einer der obigen vier Kategorien zugeordnet ist, nennt man sie Normal Library Unit. Sie können in beliebig vielen Partitionen vorhanden sein, allerdings kann jede dieser Einheiten einen eigenen internen Zustand haben, der sich von Partition zu Partition unterscheidet.

E.2 Konfiguration von Partitionen

Die Konfiguration von Partitionen, also welche Bibliothekseinheiten in welcher Partition und welche Partition auf welchem Rechner liegt, wird nach dem Compilieren als Teil des Link-Vorganges vorstatten gehen. Details legt, wie schon gesagt, der Sprachstandard hier nicht fest.

Abschließend sei noch darauf hingewiesen, daß natürlich alle objektorientierten Möglichkeiten auch in verteilten Systemen erhalten bleiben. So wird etwa dafür gesorgt, daß Tags in einem verteilten System eindeutig sind. Details kann man im Reference Manual nachlesen.

Literaturverzeichnis

- [Ada95] ISO/IEC 8652. *Ada Reference manual*, 1995.
- [BC90] G. Bracha und W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA Proceedings*, 1990.
- [Boo87] Grady Booch. *Software Components with Ada (Structures, Tools, and Subsystems)*. Benjamin/Cummings, Menlo Park, CA, 1987.
- [Boo91] Grady Booch. *Object-oriented design with applications*. Benjamin/Cummings, Redwood City, CA, 1991.
- [Con87] Richard Conn. *The Ada Software Repository and the Defense Data Network (A Resource Handbook)*. Zoetrope, New York, 1987.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, NY, 1988.
- [Str86] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, 1986.