LynxOS und die Programmiersprache ADA

Andreas Faul

30. Januar 1995

Inhaltsverzeichnis

1. Das Betriebssystem LynxOS

1.1 Was sind Echtzeitbetriebssysteme

1.1.1 Was bedeutet Echtzeit

Echtzeit bedeutet, daß ein System innerhalb einer vorgegebenen maximalen Zeit auf Anforderungen einer Anwendung reagiert. Die Anforderungen an Echtzeitsysteme können sehr unterschiedlich sein; z.B. ist bei der Temperaturregelung eine Reaktionszeit von einer Sekunde durchaus noch ausreichend, wobei bei Antriebsregelungen Reaktionszeiten von 1 ms erreicht werden müssen. Typische Reaktionszeiten liegen im Bereich von 10 bis 1000 ms. Bei den Reaktionszeiten ist auch auf die zu verarbeitende Datenmenge zu achten.

In einem Echtzeitsystem ist das zeitliche Verhalten des gesamten Systems deterministisch, um eine maximale Reaktionszeit auf bestimmte Ereignisse zu garantieren (Worst-Case-Analyse).

Man findet Echtzeitsysteme in den verschiedensten Formen

- embedded Controller: Spezielle Hardware, z.B. zur Steuerung von Waschmaschinen, große Stückzahlen und minimale Kosten, nur für sehr spezielle Aufgaben. Die Peripherie besteht aus Sensoren und Steuerungen.
- Echtzeitsysteme / Echtzeitrechner: Normale Rechner mit der üblichen Peripherie (meist mit mehreren angeschlossenen Terminals) werden als übergeordnete Einheiten betrachtet, die mehrere Systeme überwachen und regeln. Sie sind flexibel und erweiterbar.

1.1.2 Anforderungen an ein Echtzeitbetriebssystem

Ein EZ-System hat folgende Anforderungen zu erfüllen:

- Garantierte maximale Reaktionszeit auf äußere Ereignisse.
- Beendigung von Aufgaben innerhalb einer bestimmten Zeit
- Verwaltung unterschiedlicher Prioritäten von Ereignissen und Aufgaben

- Diverse Verküpfungsmöglichkeiten zwischen verschiedenen Aufgaben (z. B. Kommunikation zwischen verschiedenen Prozessen)
- Hohes Maß an Zuverlässigkeit

Der Rechner erfährt durch Interrupt-Kanäle von äußeren Ereignissen. Interrupts treten zu unbestimmten Zeiten auf, müssen aber jederzeit erkannt und behandelt werden. Der Rechner hat dann unverzüglich seine augenblickliche Tätigkeit zu unterbrechen und die vorgesehene Interruptbehandlung durchzuführen (Interrupt Service Routine). In der Realität kommen Ereignisse meist gleichzeitig und mit verschiedenen Prioritäten an. Ereignisse mit höherer Priorität verdrängen Prozesse mit niedrigerer Priorität.

1.1.3 Harte und weiche Echtzeitsysteme

Man unterscheidet harte und weiche EZ-Betriebssysteme

Ein EZ-BS hat in der Regel nicht nur eine Aufgabe, sondern mehrere zu erledigen, die jeweils bis zu einem bestimmten Zeitpunkt beendet sein müssen (deadline). Je nach Bedeutung dieser deadline spricht man von einer weichen oder harten Echtzeit. Harte Echtzeit garantiert, daß die Aufgabe zu einem festen Zeitpunkt beendet ist. Ein Überschreiten dieses Zeitpunktes kann nicht toleriert werden. Bei weicher Echtzeit kann die Überschreitung der Frist bis zu einem Grenzwert geduldet werden (z.B. Temperaturregelung)

1.1.4 Prozeß- und Speicherverwaltung in Echtzeitbetriebssystemen

EZ-Systeme sind durchwegs Multitasking-Betriebsysteme. Die Tasks können unabhängig voneinander gestartet und ausgeführt werden, gesteuert durch Interrupts, Zeitabläufe und Eingangsdaten. Sie müssen miteinander kommunizieren und sich koordinieren, um gemeinsame Aufgaben lösen zu können.

Der Scheduler ist der zentrale Teil des EZ-Betriebssystems. Er entscheidet, wann ein Task den Prozessor zugesprochen bekommt, und wann er ihn wieder abgeben muß. Das eigentliche Umschalten von einer Task zur anderen (Context Switch) verrichtet der Dispatcher (Verteiler).

Im normalen Mehrbenutzerbetriebssystem (auch Timesharing-System), verwendet der Scheduler meist das Round-Robin-Verfahren, bei dem jede Task einen gleichmäßigen Anteil an Rechenzeit erhält. Die Zeitscheibe (Time Slice) ist dabei so kurz gewählt, daß keine bemerkbaren Verzögerungen auftreten. Dies mag bei den üblichen Mehrbenutzerbetriebssystemen ausreichend sein, aber nicht bei EZ-Betriebssystemen. Manche Aufgaben und Ereignisse sind wichtiger und daher vorrangig zu behandeln. EZ-Betriebsysteme arbeiten daher mit einem anderen Verfahren, nämlich der Prioritätensteuerung. Der Programmierer gibt jeder Task eine feste Priorität und der Scheduler wählt jeweils die lauffähige Task mit der höchsten Priorität aus. Jetzt werden wichtigere Aufgaben vorrangig erledigt, und die weniger wichtigen unterbrochen (Preemption). Nur wenn mehrere Tasks gleicher Priorität vorliegen, steht der Scheduler wieder vor dem Auswahlproblem und greift in der Regel auf das Round-Robin Verfahren zurück.

Dieses preemptive Multitasking ist eines der wesentlichen Kennzeichen von EZ-Betriebssystemen.

Das klassische UNIX als Mehrbenutzerbetriebssystem erlaubt Preemption nur dann, wenn die Task gerade keinen Systemaufruf durchführt. Der Kernel des Standard-UNIX ist non-preemptive, also nicht unterbrechbar. Für Echtzeitanwendungen sind diese Systeme daher unbrauchbar.

Alle gleichzeitig laufenden, "gescheduleden" Tasks brauchen Hauptspeicher. Auch hier können die Verfahren anderer Betriebsysteme nicht ohne weiteres übernommen werden. EZ-Tasks können es sich nicht leisten, erst zur Laufzeit Programmcode oder Daten von der Platte nachzuladen. Swapping oder virtuelles Demand-Paging, wie es z.B. bei UNIX oder OS/2 üblich ist, fällt daher im allgemeinen für EZ-Tasks aus. EZ-Betriebssysteme, die Swapping oder Demand-Paging anbieten, sehen dies nur für zeitunkritische Tasks vor.

Im EZ-Betriebssystem müssen alle wirklichen EZ-Tasks im Speicher gehalten werden. Damit kommt der Speicherverwaltung eine eminent wichtige Rolle zu. Man sollte dabei bedenken, daß jederzeit eine Task mit einer Speicheranforderung oder -freigabe an den Speicherverwalter herantreten kann, daß rekursive Strukturen mit lokalen Datenbereichen möglich und üblich sind, daß gleiche Tasks auch mehrmals parallel laufen können (gegebenfalls mit Code-Sharing), daß sogar während des Allozierens von Speicher eine höherpriorisierte Task dazwischenfunken kann... Nebenbei muß der Speichermanager noch Lücken im Speicher optimal zusammenfassen und nicht mehr benötigte Daten entfernen (Carbage Collection).

Der Speichermanager bestimmt nicht zuletzt die Reaktionszeit auf ein Ereignis. Bei EZ-Betriebssystemen wird die Reaktionszeit länger, je mehr Tasks sich im Speicher befinden, weil der Speichermanager beim Allozieren mehr Zeit benötigt, um freien Speicher zu finden (auch hier gibt es ausgeklügelte Methoden, um die Suchzeit so klein wie möglich zu halten).

1.1.5 Polling und Interrupts

Alle Rechner kennen gemeinhin zwei Verfahren, um auf äußere Ereignisse zu reagieren. Das Polling und die schon erwähnten Interrupts. Beim Polling fragt das Programm in regelmäßigen Abständen alle in Frage kommenden Eingänge nach gewünschten Meßwerten und Zuständen ab. Die Grenzen dieses Verfahrens sind jedoch bald erreicht. Um schnell reagieren zu können, muß das System oft abfragen - und das meist umsonst, da das gewünschte Ereignis noch nicht eingetreten ist. Das kostet unnötig viel Rechenzeit.

Dem gegenüber bieten Interrups dem Programmierer ein wesentlich leistungsfähigeres und komfortableres Hilfsmittel für die Ereignisbehandlung. Über die

Prozeßperipherie führt das Ereignis zum Interrupt, der dann den Programmablauf unterbricht und eine zugeordnete Behndlungsroutine startet - es sei denn, es ist gerade ein höherpriorisierter Task in Behandlung, dann muß er warten. Die Interrupt Service Routine kann je nach Dringlichkeit sofort auf das Ereignis reagieren oder aber eine spätere Bearbeitung veranlassen (Messages oder Signale aussenden, Tasks anstoßen oder beim Scheduler einhängen et cetera). Nach Ende dieser Routine wird der unterbrochene Programmablauf automatisch fortgesetzt. Die Reaktionszeit auf Interrupts (Interrupt Latency) liegt bei den gängigen Prozessoren normalerweise im Bereich von 10 bis 100 Mikrosekunden. Anzahl und Prioritäten der Interrupt-Kanäle hängen von der Rechnerarchitektur ab (so unterstützt der PC/AT 11 externe IRQs mit üblicherweise festgelegter Priorität). Reichen die Kanäle nicht aus, so gibt es die Möglichkeit, daß sich mehrere Ereignisse einen Kanal teilen (Interrupt-Sharing). In diesem Fall muß die Interrupt Routine erst mal per Polling feststellen, wer überhaupt den Interrupt ausgelöst hat. Das geht wiederum auf Kosten der Reaktionszeit.

1.1.6 Prozeßkommunikation

Durch Verwendung von Tasks und Interrupt Service Routinen kann der Programmierer seine Anwendung in kleinere Einheiten aufteilen. Diese Einheiten müssen miteinander in Verbindung treten können. Der sonst übliche Weg über Dateien ist für EZ-Anwendungen viel zu aufwendig. Unter dem Begriff IPC (Inter Process Communication) gibt es auch hier mehrere Verfahren. Tasks müssen sich synchronisieren, wenn sie auf gemeinsame Resourcen zugreifen (Semaphore, Mutual Exclusion). Der Datenaustausch und die Verwaltung gemeinsamer Datenbestände erfolgt meist über Shared Memory. Aus Performance-Gründen müssen die kritischen Bereiche (critical Regions), auf die durch Mutual Exclusion zugegriffen wird, möglichst klein sein. Die Prozeßkommunikation findet meistens innerhalb eines Programmkontextes (Adreßraums) zwischen verschieden Threads statt. Das hat den Vorteil, daß bei der Prozeßkommunikation die Prozeßgrenzen nicht überschritten werden müssen (sehr schnell).

Messages (Nachrichten) sind ein Verfahren, das den Datenaustausch zwischen verschiedenen Objekten (Prozessen) ermöglicht. Nachrichten sind immer gerichtet, der Absender muß den Empfänger angeben.

Signals (Signale) sind schließlich eine Art Software-Interrupt. Ihre Information besteht allein aus ihrer Bezeichnung und Nummer. Sie werden von einer Task an eine andere geschickt und lösen dort ähnlich wie Interrupts bestimmte Reaktionen aus.

1.1.7 Treiber in Echtzeitbetriebssystemen

Eines der wesentlichen Merkmale von EZ-Systemen ist, daß sie intensiv mit ihrer Umwelt kommunizieren. Für diese Peripherie sind Gerätetreiber notwendig. Betriebssysteme bieten meist nur Treiber für die wichtigste Standardperipherie

(Terminals, Festplatten). Bei zusätzlicher Hardware ist man darauf angewiesen, daß der Hersteller entsprechende Treiber zur Verfügung stellt. Sind keine Treiber erhältlich, bleibt nur der Weg der eigenhändigen Programmierung. EZ-Betriebsysteme bieten daher auch in diesem Bereich zusätzliche und bessere Möglichkeiten in der Treiberprogrammierung. Sie verfügen über eine leistungsfähige Programmierschnittstelle zur direkten Ansteuerung der Hardware und der Interrupts. Programmierung in höheren Sprachen, wie zum Beispiel C, und bessere Testmöglichkeiten erleichtern die Arbeit deutlich. Dynamische Konfiguration, d.h. die Auswahl von Treibern erst zur Laufzeit, erleichtert den Test, aber auch den Betrieb bei wechselnden Anforderungen.

Alle EZ-Betriebssysteme stellen diese grundlegenden Dienstleistungen zur Verfügung. Für einen Einsatz in einem Embedded Controller sind damit alle Voraussetzungen erfüllt. Größere Echtzeit-Anwendungen benötigen noch weitere Betriebssystemdienste wie Dateisysteme, Netzwerke, allgemeine Hilfsprogramme usw. Für den Programmierer ist weiter eine gute Entwicklungsumgebung unverzichtbar.

Die wesentlichen Grundfunktionen sind im Kernel implementiert (auch Nucleus oder Core genannt). Bei embedded Controllern ist dieser Kernel oft nur wenige KB groß und enthält nur die unbedingt notwendigen Funktionen. Auch größere EZ-Betriebssysteme haben oft nur einen Mikrokernel, um den sich je nach Anwendung diverse konfigurierbare Zusätze gruppieren.

1.1.8 Dateisysteme in Echtzeitbetriebssystemen

Sobald EZ-Systeme mit einer Festplatte ausgestattet sind, benötigt man ein Filesystem. Das simple FAT-System von DOS ist sehr beliebt, aber auch im Leistungsumfang beschränkt (große Platten werden schlecht unterstützt; Fragmentierung der Dateien; sichert Daten unzureichend ab; kennt keinerlei Zugriffsrechte). UNIX-Dateisysteme lösen die meisten dieser Probleme wesentlich besser, sind aber höchst sensibel gegenüber Systemabstürzen. Aufwendige Rekonstruktionen von Datenbeständen sind aber in EZ-Systemen nicht akzeptabel. Die dort zum Einsatz kommenden Dateisysteme sind meistens UNIX-Varianten, die im Hinblick auf Sicherheit und Performance optimiert wurden.

1.2 Das Echtzeitbetriebssystem LynxOS

1.2.1 Was ist LynxOS?

LynxOS ist ein UNIX-ähnliches Echtzeitbetriebssystem. Gegenüber dem Benutzer verhält es sich wie ein UNIX-Betriebssystem. LynxOS wurde von Grund auf unter dem Aspekt der Echtzeitprogrammierung entwickelt. Dabei wurde besonders auf die Antwortzeiten geachtet. LynxOS ist mit dem POSIX 1003.1 Standard kompatibel, enthält aber nicht den AT&T/USL/Novell Source Code.

Der Betriebssystemkern wurde völlig neu unter dem Echtzeit-Aspekt programmiert. LynxOS unterstützt POSIX.1, POSIX.4, POSIX.4a, Berkeley (BSD) und System V APIs. Weiter gibt es unter LynxOS TCP/IP, NFS, Streams, X11, Motif und GNU-Programme.

1.2.2 Die Entwicklung von LynxOS

Lynx Real-Time Systems wurde 1985 in Dallas von den Brüdern Mitch und Mike Brunnell gegründet. Ziel war es, ein Echtzeitbetriebssystem der Industrie zu Verfügung zu stellen, das dem Anwender und Programmierer eine standardisierte, an Unix angelehnte Oberfläche bietet. Nach einigen Jahren Entwicklung entstand das Echtzeitbetriebssystem LynxOS. LynxOS wurde im Laufe der Zeit auf die verschiedensten Architekturen portiert. und ist heute auf den meisten Plattformen verfügbar. Firmen wie IBM, HP, CDC haben LynxOS direkt lizensiert.

1.2.3 Die Hauptmerkmale von LynxOS

- Preemptives Design: Das gesamte Design (Kernel, Networking, Graphical Interface) von LynxOS ist "preemtive", d.h. zu keinem Zeitpunkt wird die Antwortzeit (Real-Time Responsiveness) überschritten. Jede Operation wird strikt nach der Priorität des jeweiligen Tasks/Threads ausgeführt. Die "real-time nature" von LynxOS ist für den Anwender oder Programmierer völlig transparent.
- In LynxOs ist es möglich, eine maximale Reaktionszeit anzugeben. Diese Zeit bezeichnet man als "worst case task response time"; d.h. es kann angegeben werden, wie lange es dauert, bis der Task mit der höchsten Priorität aktiviert wird, wenn er ein Signal von außen erhalten hat. (Real Time Benchmarking)
- Full Memory Protection LynxOS unterstützt die virtuelle Adressierung des Systemspeichers. Außerdem hat jeder Prozeß seine eigene geschützte Umgebung, die es ermöglicht, daß kein Prozeß einen anderen oder gar den Kernel überschreibt, bzw. in seiner Integrität verletzen kann (Absturzsicherheit). Weiter wird eine Fragmentierung des Hauptspeichers verhindert (Memory Fragmentation), da keine "Löcher" im Speicher entstehen können.
- Advanced Filesystem Support. Ein I-node-basiertes Filesystem mit der Berkley Namenskonvention, mit symbolischen und harten Links und eine einfache Implementierung von Devices (/dev), die vor allem nützlich für die einfache Implementierung von Treibern für I/O Karten und anderen Erweiterungen sind.
- Connectivity: Da sich das System-Call-Interface von LynxOS kaum von Unix unterscheidet, ist es relativ einfach, die Netzwerkfähigkeiten von

UNIX zu implementieren. Der TCP/IP Code, der die Netzwerkanbindung ermöglicht, beruht in LynxOS auf dem Sourcecode von Berkeley, mit den Änderungen, die notwendig sind, um die Echtzeit-Fähigkeiten von LynxOS aufrechtzuerhalten Das NFS von LynxOS basiert auf dem NFS RPC Code von SUN Microsystems. Das STREAMS Interface folgt dem System V Standard. Weiter werden die Netzwerkutilities wie rsh, rlogin,rcp, telnet, usw. unterstützt. Standard Unix-Protokolle oder neue Protokolle können leicht durch das STREAM Interface oder Ethernet Interface für TCP/IP realisiert werden. Fremdfirmen stellen ATM, OSI und weitere Netzwerkprotokolle für LynxOS zur Verfügung.

- Entwicklungsumgebung unter LynxOS (self-hosted Development):

  Es können Anwendungen direkt unter LynxOS mit zahlreichen Hilfsmitteln, wie rcs, lex, awk, entwickelt und getestet werden. Dies hat den Vorteil, daß man sofort testen kann, wie sich ein Programm in der jeweiligen Umgebung verhält.

- X und Motif: Unter LynxOS gibt es die komplette X11 Release 5 Umgebung mit dem Motif Toolkit und dem Window Manager von OSF. Weitere Developmenttools, GUI Builders und Widgets wurden von verschiedenen Herstellern auf LynxOS portiert, um die grafische Entwicklungsumgebung zu verbessern.

- ROMability: Es ist unter LynxOS möglich, ausführbare Module zu erzeugen, die in Verbindung mit einem LynxOS Kernel in ein EPROM gebrannt werden, über ein Netzwerk geladen oder direkt von einer Systemdiskette geladen werden können (nützlich für embbeded Controller).

- Kompatibel zu den Standards LynxOS erfüllt den NIST POSIX 1003.1 Standard. Weiter ist es zum System V.4 und BSD 4.3 Standard kompatibel.

1.2.4 Auf welchen Plattformen läuft LynxOS heute?

- Intel 80386/80486/Pentium (neueste Version ist 2.2.1)
- Motorola MVME 167/MVME166
- Motorola MVME 162
- Motorola MVME 147
- Sparc-2/SPARC-IPX
- microSPARC
- R3000 (source code only)
- PowerPC 601/603/604

Von anderen Firmen wurde LynxOS auf ihre Plattformen portiert

- PA-RISC 7100 (Hewlett Packard)
- PowerPC 601 (Cetia)

1.2.5 Wesentliche Unterschiede zwischen UNIX und LynxOS

Ein UNIX Kernel ist nicht preemptiv; das heißt, daß die Tasks des Kernels nicht unterbrochen, bzw. nach einer Unterbrechung nicht wieder an derselben Stelle fortgesetzt werden können. Führt ein Prozeß einen Systemaufruf durch (Betriebssystemmodus), dann kann dieser nicht von einem Task außerhalb des Betriebssystems unterbrochen werden.

Alle Prozesse laufen unter UNIX gleichberechtigt in einem time-sharing-Verfahren. Wann genau ein Prozeß aktiviert wird, ist zeitlich nicht exakt festgelegt (not predictable). Die Abarbeitung der Prozesse ist nicht deterministisch.

Der Kernel von LynxOS wurde dagegen komplett neu unter dem Echtzeitaspekt entwickelt. LynxOS ist deterministisch, d.h. die zeitliche Abfolge der Prozesse ist bekannt. Die Prozesse können in Echtzeit agieren und reagiern. LynxOS ist ein "priority based Operating System", das heißt, daß das Scheduling nur durch die Priorität eines Tasks bestimmt wird. Der Task mit der höchsten Priorität wird stets aktiviert.

Der LynxOS Kernel ist so implementiert, daß Interrupts und Data Streams sofort in mehreren priorisierten und preemptiven Kernel-Tasks behandelt werden, und User-Tasks und Kernel-Tasks gemeinsam mit verschiedenen Prioritäten nebeneinander laufen. Insgesamt ist der Kernel preemptiv und mit einem sehr schnellen deterministischen harten Echtzeitverhalten ausgestattet.

2. Die Programmiersprache ADA

2.1 Einleitung

Im Jahre 1974 erkannte das amerikanische Verteidigungsministerium, daß die Kosten für die Entwicklung von Software in integrierten Systemen viel zu hoch waren. So setzten alle drei Armeegruppen verschiedene Sprachen in verschiedenen Varianten ein. Diese waren zum Teil systemabhängig und verursachten zusätzliche Kosten in Übersetzung, Wartung und Schulung.

2.1.1 Der Weg zur Standardisierung

Es wurde beschlossen, einen einheitlichen Standard für integierte Systeme zu schaffen. Dazu wurden die Anforderungen umrissen. Bereits existierende Sprachen (Cobol, SPL/1 u.a.) wurden verglichen und neue Sprachvorschläge eingeholt.

Stufen der Entwicklung:

1. Sprachvorschläge:

   1975: Strawman und Woodenman

   1976: Tinman (schon sehr präzise), Ironman (ca. 17 versch. Vorschläge)

   1978: Steelman

   1979: Vorschlag von CII Honeywell Bull wird zum Sieger erklärt

2. Bekanntgabe der Sprache

   Die neue Sprache sollte ADA heißen. Zu Ehren von Augusta Bayron, Gräfin von Lovelace (1816-1851). ADA, die Tochter von Loyd Bayron, war eine Assistentin und Gönnerin von Charles Babbage und arbeitete mit ihm an einer mechanisch analytischen Maschine. Sie wird daher als die erste Programmierin der Welt angesehen.

3. Standardardisierung

Die Sprache wurde nach der offizellen Bekanntgabe weltweit getestet. Verschiedene Verbesserungsvorschläge und Korrekturen wurden in die Sprache eingebracht. 1980 bzw. 1981 wurde sie mit dem LRM (Language Reference Manual) standardisiert.

2.2 Wichtige Merkmale der Programmiersprache ADA

ADA ist eine wesentlich umfangreichere Sprache als Pascal, an das sich ADA zum Teil anlehnt. Die wichtigsten Aspekte in ADA sind die folgenden.

- Lesbarkeit
  Die Programme sind sehr leicht zu lesen und damit besser verständlich.
- Starke Typprägung
  Sie garantiert, daß jedes Objekt eine klar definierte Menge von Werten besitzt und verhindert eine Verwechslung logisch sich unterscheidender Typen. Dadurch können viele Fehler durch den Übersetzer festgestellt werden, welche in vielen anderen Sprachen (z.B. C) zu einem durchführbaren, aber unrichtigen Programm geführt hätten.
- Programmieren in Paketen
  Mechanismen für die Kapselung, getrennte Übersetzung und für das Bibliotheksmangement sind für das Entwerfen übertragbarer und wartbarer großer Programme unbedingt notwendig.
- Behandlung von Ausnahmebedingungen
  Es ist notwendig, ein Mittel bereitzustellen, mit dem ein Programm schichtweise und modular aufgebaut werden kann, so daß die Folgen von Fehlern in einem Programmteil auf diesen beschränkt gehalten werden können.
- Datenabstraktion
  Gesteigerte Übertragbarkeit und Wartbarkeit werden erreicht, wenn die Einzelheiten der Datendarstellung von der Beschreibung der logischen Operationen für die Daten getrennt gehalten werden.
- Prozesse
  Für viele Anwendungen ist es wichtig, daß ein Programm eher parallel ablaufende Aktivitäten zeigt als sequentielle. Man fügt der Sprache passende Hilfsmittel zu und realisiert die Prozeßkommunikation nicht über das Betriebssystem, sondern durch die Programmiersprache. Damit erhält man eine bessere Übertragbarkeit und Sicherheit der Sprache.
- Generische Einheiten (Templates)
  In vielen Fällen ist die Logik eines Programmteils von den Typen der Werte, die manipuliert werden, unabhängig. Es ist daher ein Mechanismus für die Schaffung verwandter Programmteile von einer einzigen Schablone her notwendig. Das ist besonders für die Errichtung von Bibliotheken nützlich.

Da die Beschreibung der gesamten Programmiersprache ADA den Rahmen eines einzigen Seminarvortrages sprengen würde, beschränke ich mich auf die Beschreibung der wichtigsten Bestandteile und besonderen Charakteristika der Sprache.

2.3 Das ausgefeilte Typkonzept in ADA und Überladungen in ADA

Definition laut Languages Reference Manual 3.3:

Ein Typ charakterisiert eine Menge von Werten und eine Menge von Operationen, die auf diesen Werten anwendbar sind.

Jeder Typ erhält einen eigenen Namen, der mittels einer Typvereinbarung eingeführt wird. (Von vordefinierten Typen wie beispielsweise INTEGER nimmt man an, daß sie im Paket Standard vereinbart sind.)

Die Mengen der Werte, die zu zwei verschiedenen Typen gehören, sind unterschiedlich, obwohl in einigen Fällen die tatsächliche lexikalische Form der Werte identisch sein kann - welche Menge jeweils gemeint ist, wird durch den Zusammenhang bestimmt.

Die Idee, daß eine lexikalische Form zwei oder mehrere verschiedene Objekte darstellt, wird als Überladung bezeichnet.

Werte eines Typs können nicht Variablen eines anderen Typs zugewiesen werden.

Das ist die fundamentale Regel der starken Typprägung in ADA. Sie ist ein große Hilfe für die schnelle Entwicklung richtiger Programme, weil sie gewährleistet, daß viele Fehler bei der Übersetzung der Programme bereits entdeckt werden können. Die Syntax der Typvereinbarung besteht aus dem reservierten Wort **type**, dem Bezeichner, der mit dem Typ verküpft wird, dem reservierten Wort **is** und der anschließenden Definition des Typs mit abschließendem Strichpunkt.

Beispiel:

```
type COLOUR is (RED, AMBER, GREEN);
```

Diese Anweisung führt einen neuen Typ mit der Bezeichnung COLOUR ein. Außerdem gibt sie an, daß es nur drei Werte dieses Typs gibt, nämlich RED, AMBER, GREEN.

Ein Objekt des Typs COLOUR wird wie folgt vereinbart:

```
C: COLOUR:=RED;                 -- Mit Anfangswert RED vorbelegt
C: constant COLOUR:=RED;        -- Als Konstante vereinbart
```

2.3.1 Untertypen

Ein Untertyp charakterisiert eine Menge von Werten, die nur eine Teilmenge der Werte eines anderen Typs ist, der als Basistyp bezeichnet wird. Diese Teilmenge wird mit Hilfe von Einschränkungen definiert. Eine Teilmenge kann auch die vollständige Menge des Basistyps sein. Es gibt jedoch keinen Weg die Menge der Operationen des Basistyps zu beschränken. Der Untertyp erbt alle Operationen. Das Bilden von Teilmengen ist nur bei Werten erlaubt.

Beispiel:

```
subtype DAY_NUMBER is INTEGER range} 1..31;

D: DAY_NUMBER;
```

Es ist nun sichergestellt, daß D nur Werte von 1 bis 31 annehmen kann. Der Übersetzer wird, wenn notwendig, Laufzeitüberprüfungen durchführen. Wenn eine Prüfung eine Typverletzung liefert, wird die entsprechende Ausnahmebedingung aktiviert (CONSTRAINT_ERROR).

Es ist wichtig festzustellen, daß eine Untertypvereinbarung keinen neuen unterschiedlichen Typ einführt. Ein Objekt wie D ist vom Typ INTEGER und deshalb sind Zuweisungen von Objekten des Typs Integer und DAY_NUMBER zulässig, solange sie nicht den zulässigen Bereich verletzen (sonst wird CONTRAINT_ERROR aktiviert). Ein Untertyp kann auch mit Hilfe eines vorhergehenden Untertyps definiert werden.

Beispiel:

```
subtype FEB_DAY is DAY_NUMBER range 1..29;
```

Fazit: Ein Untertyp definiert keinen neuen Typ, sondern erzeugt ein Duplikat für einen bestehenden Typ mit einer möglichen Einschränkung.

2.3.2 Attribute und Operationen der Typen INTEGER und REAL

Der Typ INTEGER ist in ADA fest implementiert. Dagegen ist der Typ REAL ein Gleitpunkttyp, der von der jeweiligen Implementierung der Sprache abhängig ist (dazu später mehr). Der Minimalwert des Typs INTEGER wird durch INTEGER'FIRST, der Maximalwert durch INTEGER'LAST bereitgestellt.

In einer 16Bit Zweierkomplement-Implementierung ist beispielsweise INTEGER'FIRST = -32768 und INTEGER'LAST = +32767.

Beispiel für die Definition der natürlichen Zahlen:

```
subtype NATURAL is INTEGER range 1..INTEGER'LAST;
```

Es gibt bereits vordefinierte Operationen auf den Typen INTEGER und REAL.

+,-	dyadisch oder monadische Operatoren
*	Multiplikation
/	Division
rem	Rest
mod	Modulo
abs	absoluter Wert
**	Potenzierung

2.3.3 Aufzählungstypen

Beispiele für Aufzählungstypen sind:

```
type COLOUR is (RED,AMBER,GREEN);
type STONE is (AMBER,BERYL,QUARZ);
```

In diesem Beispiel sind die Typen überladen. Die Konstante AMBER kann entweder vom Typ COLOUR oder vom Typ STONE sein. Beide Bedeutungen desselben Namens sind gleichzeitig sichtbar und die zweite Vereinbarung versteckt die erste nicht, ganz gleich, ob sie im selben Vereinbarungsteil vorkommen, oder ob sie in einem inneren Vereinbarungsteil vereinbart werden. Gewöhnlich kann man aus dem Zusammenhang erkennen, welche Bedeutung gemeint ist. Aber in jenen Fällen, in denen das nicht möglich ist, kann man auf die Konstante immer zugreifen, indem man sie in Klammern setzt, ihren Typnamen (bzw. Untertypnamen) und ein einzelnes Anführungszeichen davorschreibt.

Beispiel:

```
COLOUR'(AMBER)
STONE'(AMBER)
```

Obwohl AMBER in zwei verschiedenen Aufzählungstypen verwendet werden kann, kann man sie nicht gleichzeitig als Aufzählungskonstante und als Bezeichner einer Variablen verwenden. Die Vereinbarung von dem einen würde das andere verbergen, und sie könnten nicht beide im selben Vereinbarungsteil vereinbart werden. Es gibt keine obere Grenze für die Anzahl von Werten in einem Aufzählungstyp, aber es muß mindestens einen geben. Ein leerer Aufzählungstyp ist nicht zugelassen. Einschränkungen für Aufzählungstypen gibt es genauso wie für ganze Zahlen:

Beispiel:

```
subtype COLOURLESS is COLOUR range RED..AMBER;
```

Wichtige Attribute bei Aufzählungstypen sind:

Das Attribut FIRST liefert das erste Element in der Aufzählung, LAST das letzte.

Beispiel:

```
COLOUR'FIRST = RED
COLOUR'LAST = GREEN
```

Es gibt eingebaute funktionelle Attribute, um den Nachfolger oder den Vorgänger eines Aufzählungswertes zu bestimmen. Es sind dies die Attribute SUCC und PRED, die nach dem Typnamen und einem einzelnen Anführungszeichen stehen.

Beispiel:

```
COLOUR'SUCC(AMBER) = GREEN
COLOUR'PRED(GREEN) = AMBER
T'SUCC(T'PRED(X)) = X
```

Ein anderes funktionelles Attribut heißt POS. Es gibt die Positionsnummer des Aufzählungswertes an. Die Positionsnummer bezieht sich auf die Vereinbarung, wobei die erste Positionsnummer immer Null ist.

Beispiel:

```
COLOUR'POS(RED) = 0
COLOUR'POS(AMBER) = 1
```

Das Gegenteil von POS ist VAL. Es erhält die Positionsnummer und gibt den entsprechenden Aufzählungswert zurück.

Beispiel:

```
COLOUR'VAL(0) = RED
```

Wird ein Positionswert außerhalb des Wertebereichs angegeben COLOUR'VAL-(3), dann wird CONSTRAINT_ERROR aktiviert. Es sollte beachtet werden, daß diese vier Attribute SUCC, PRED, POS und VAL auch auf Untertypen anwendbar sind.

Es ist kein sehr guter Programmierstil, POS und VAL zu verwenden. Wenn der Programmierer dies trotzdem tut, denkt er eher in Zahlen, als in Aufzählungstypen und zerstört daher die Abstraktion. Die Operatoren =, /=, <, <=, > und >= sind auch für Aufzählungstypen anwendbar. Das Ergebnis wird durch die Reihenfolge der Werte in der Typvereinbarung bestimmt.

Beispiel:

RED < GREEN liefert TRUE

2.3.4 Der Boolesche Typ

Der Boolesche Typ ist ein vordefinierter Aufzählungstyp, dessen Vereinbarung als

```
type BOOLEAN is (FALSE,TRUE);
```

angenommen werden kann.

Der Boolesche Typ weist alle üblichen Merkmale eines Aufzählunstyps auf, wie zum Beispiel:

```
FALSE < TRUE = TRUE
BOOLEAN'VAL(TRUE) = 1
```

Der Boolesche Typ hat die gewöhnlichen Operatoren, =, /=, <, <=, >, >=, and, or, not und xor. Die Verwendung ist analog zu Pascal.

2.3.5 Typumwandlung zwischen INTEGER und REAL

Es kann eine Typumwandlung zwischen INTEGER und REAL Typen vorgenommen werden. Soll ein INTEGER in REAL umgewandelt werden, dann schreibt man für

```
I:INTEGER
REAL(I)      -- conversion
```

Soll nur der Typ angeführt werden, so wählt man die attributierte Schreibweise:

```
INTEGER'(I)  -- qualification
```

2.3.6 Die speziellen Operatoren in und not in

Es gibt zwei Gültigkeitstests, die für alle skalaren Typen gültig sind. Diese heißen **in** und **not in**. Sie sind eigentlich im strengen Sinne keine Operatoren wie = oder <=. Sie ermöglichen uns eine Überprüfung, ob ein Wert innerhalb eines bestimmten Wertebereichs liegt (einschließlich der Endwerte), oder ob er einer Einschränkung genügt, die durch einen Untertyp hervorgerufen wird. Der erste Operand ist daher ein skalarer Ausdruck, der zweite ein Bereich oder Untertyp, und das Ergebnis ist ein boolescher Ausdruck.

Beispiele:

```
I not in range 1..10;
I in NATURAL;
TODAY in DAY range MON.. WED;
```

2.3.7 Die Konstrukte and then und or else

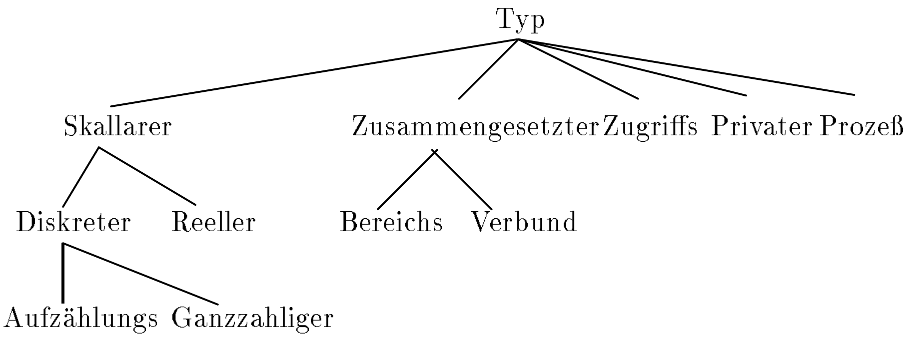
Es gibt weiter noch die zwei Befehle **and then** und **or else**. Die Form **and then** ist eng mit dem Operator **and** verwandt, während **or else** eng mit dem Operator **or** verwandt ist. Sie können in Ausdrücken vorkommen und haben die gleiche Priorität in der Auswertung wie **and**, **or** und **xor**. Der Unterschied liegt in den Regeln, die die Auswertung ihrer Operanden betreffen.

Im Falle **and** und **or** werden beide Operanden ausgewertet, aber die Reihenfolge ist nicht bestimmt. Im Falle von **and then** und **or else** wird der linke Operand immer zuerst ausgewertet und der rechte Operand wird nur bearbeitet, wenn es notwendig ist, um das Ergebnis zu bestimmen. So wird bei X **and then** Y zuerst X ausgewertet. Wenn X falsch ist, ist das Ergebnis falsch, egal welchen Wert Y hat, deshalb wird Y nicht ausgewertet. Wenn X richtig ist, muß Y ausgewertet werden und der Wert von Y ist das Ergebnis. In gleicher Weise wird bei X **or else** Y zuerst X ausgewertet. Wenn X richtig ist, ist das Ergebnis richtig, egal welchen Wert Y hat. Wenn X falsch ist, muß Y ausgewertet werden, und der Wert von Y ist das Ergebnis.

Die Formen **and then** und **or else** sollten in solchen Fällen verwendet werden, in denen es auf die Reihenfolge der Auswertung ankommt.

2.4 Typenklassifikation

Die Typen in ADA lassen sich wie folgt klassifizieren:



2.5 Kontrollstrukturen

In diesem Abschnitt sollen kurz die drei Kontrollstrukturen von ADA behandelt werden, die im sequentiellen Programmlauf Verwendung finden. Das sind die IF-Anweisung, die CASE-Anweisung und Schleifenanweisungen. Weiter gibt es in ADA auch die GOTO-Anweisung. Sie soll in diesem Kapitel wenigstens erwähnt werden.

2.5.1 IF-Anweisung

Die IF-Anweisung ist analog zu Pascal und ähnlichen Sprachen definiert:

```
if <boolean> then <expressions>; else <expressions>; end if;
```

Eine Besonderheit ist die Verschachtelung von IF-Anweisung mit **elsif** (nur ein **end if** wird zum Abschluß der IF-Anweisung benötigt).

Beispiel:

```
if <boolean> then
  <expressions>;
elsif <boolean>
  <expressions>;
else
  <expressions>;
end if;
```

2.5.2 CASE-Anweisung

Die Case-Anweisung erlaubt es, entsprechend dem Wert eines Ausdrucks, eine oder mehrere Anweisungen auszuführen.

Beispiel:

```
case <expression> is
  when <expression> => <sequence of statements>;
  when others => null;
end case};
```

Mit **others** erreicht man, daß die darauffolgenden Anweisungen alternativ ausgeführt werden.

2.5.3 Schleifenanweisungen

Die einfachste Schleife ist wohl:

```
loop <sequence_of_statements>; end loop;
```

Die Anweisungen werden so lange ausgeführt, bis die Endlosschleife abgebrochen wird. Eine Schleife kann sofort mit der Anweisung **exit** verlassen werden. Die Kontrolle geht unmittelbar auf die Stelle nach **end loop** über. Eine Erweiterung der exit-Anweisung ist

```
{\bf exit when} <condition>
```

Dabei wird die Schleife verlassen, sobald <boolean> den Wert true annimmt.

Elegantere Formen der Schleifenbildung sind:

while-Schleife

Die while-Schleife wird so lange ausgeführt bis die Auswertung der Bedingung false ergibt.

```
while <condition> loop
  <sequence_of_statements>;
end loop;
```

for-Schleife

Die for-Schleife erlaubt eine spezielle Anzahl von sequentiellen Wiederholungen. Dazu wird ein Schleifenindex eingeführt, der alle Werte des diskreten Wertebereichs in aufsteigender, bzw. absteigender Reihenfolge durchläuft.

```
for I in reverse 1..N loop
  <sequence_of_statements>;
end loop;
```

Der Schleifenindex ist innerhalb der Anweisungsfolge eine Konstante und darf nicht durch Wertzuweisungen oder Argumentübergaben verändert werden.

goto-Anweisung

Die Verwendung der goto-Anweisung zeugt von einem schlechten Programmierstil. Sie wird nur der Vollständigkeit halber hier erwähnt. Ihre sinnvollste Anwendung scheint in der Behandlung von Ausnahmebedingungen zu liegen. Eine Sprungmarke springt einem in ADA förmlich ins Auge. Sie wird durch jeweils zwei spitze Klammern eingerahmt.

```
<<THE_DEVIL>>
```

Angesprungen wird die Marke mit

```
goto THE_DEVIL;
```

2.6 Zusammengesetzte Typen

In diesem Abschnitt werden die zusammengesetzten Typen, nämlich Felder und Verbunde, beschrieben.

2.6.1 Felder

Ein Feld ist ein zusammengesetztes Objekt, das aus einer Anzahl von Komponenten besteht, die alle vom selben Typ sind. Ein Feld kann eine, zwei oder mehrere Dimensionen haben. Eine typische Feldvereinbarung ist beispielsweise

```
A: array (INTEGER range 1..6) of REAL;
```

Diese Anweisung vereinbart A als ein Objekt, das 6 Komponenten besitzt, von denen jede vom Typ REAL ist. Auf die einzelnen Komponenten kann man sich beziehen, indem man nach dem Feldnamen, einen Ausdruck in Klammern schreibt, der einen ganzzahligen Wert im diskreten Wertebereich 1..6 angibt. Wenn dieser Ausdruck (Indexwert) ein Wert außerhalb des Indexbereichs ist, wird die Ausnahmebedingung CONSTRAINT_ERROR aktiviert. Wir könnten jede Komponente von A Null setzen, indem wir schreiben.

```
for I in 1..6 loop
  A(I):=0.0;
end loop;
```

Ein zweidimensionales Feld kann wie folgt definiert werden.

```
AA: array (INTEGER range 0..2, INTEGER range 0..3) of REAL;
```

Die diskreten Indexbereiche müssen nicht statisch sein. Man könnte

```
N: INTEGER:=...;
B: array (INTEGER range 1..N) of BOOLEAN;
```

schreiben, und der Wert von N würde an der Stelle, an der die Vereinbarung von B durchgeführt wird, die Anzahl der Elemente in B bestimmen. Natürlich könnte die Vereinbarung von B mehrmals während eines Programmlaufs durchgeführt werden, wobei N jedesmal verschieden sein könnte. Ein Feldindex könnte jedoch jedem beliebigen diskreten Typ angehören.

```
HOURS_WORKED: array (DAY range MON..FRI) of REAL;
```

Dieses Feld besitzt fünf Komponenten, auf die man sich durch HOURS_WORKED(MON), ..., HOURS_WORKED(FRI) beziehen kann. Felder haben verschiedene Attribute, die sich auf ihre Indizes beziehen. A'FIRST und A'LAST geben die untere und obere Grenze des ersten (oder einzigen) Index von A an.

```
HOURS_WORKED'FIRST = MON
HOURS_WORKED'LAST = FRI
```

A'LENGTH gibt die Anzahl von Werten des ersten (einzigen) Index an:

```
HOURS_WORKED'LENGTH = 5
```

A'RANGE ist eine Kurzform für A'FIRST..A'LAST. So entspricht HOURS_WORKED'RANGE dem Indexbereich MON..FRI. Für die verschiedenen Dimensionen eines mehrdimensionalen Feldes können dieselben Attribute verwendet werden, indem man die Dimensionszahl in Klammern hinzufügt. Es muß ein statischer Ausdruck sein. So erhalten wir im Falle unseres zweidimensionalen Feldes AA

```
AA'FIRST(1) = 0
AA'FIRST(2) = 0
```

```
AA'LAST(1) = 2
AA'LAST(2) = 3
AA'LENGTH(1) = 3
AA'LENGTH(2) = 4
```

und

```
AA'RANGE(1) ist 0..2
AA'RANGE(2) ist 0..3
```

Es ist immer von Vorteil, Attribute zu verwenden, um Beziehungen zwischen den Objekten im Programm auszudrücken, weil dies im allgemeinen bedeutet, daß die Änderungen lokal bleiben.

Das Attribut RANGE ist bei Schleifen besonders praktisch:

```
for I in AA'RANGE(1) loop
  for J in AA'RANGE(2) loop
    AA(I,J):=0.0;
  end loop;
end loop;
```

Wie bei anderen veränderlichen Objekten kann auch für Felder ein Anfangswert festgesetzt werden. Dieser wird durch ein Aggregat realisiert, das die Literalform eines Feldwertes ist.

Die einfachste Form eines Aggregats besteht aus einer Liste von Ausdrücken, die die Werte der Komponenten der Reihe nach angibt.

```
AA: array (0..2,0..3) of REAL:=((0.0,0.0,0.0,0.0),
                                (0.0,0.0,0.0,0.0),
                                (0.0,0.0,0.0,0.0));
```

Ein Aggregat muß vollständig sein. Soll eine Komponente initialisiert werden, dann müssen alle anderen auch initialisiert werden. Ein Feld kann als Konstante vereinbart werden, wobei ein Anfangswert obligatorisch ist.

```
WORK_DAY : constant array (DAY)
             of BOOLEAN:=(TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,FALSE);
```

Zum Abschluß ein extremes Beispiel:

```
STRANGE : array (COLOUR,2..7, DAY range MON.. THU)
            of PLANET range MARS..SATURN;
```

2.6.2 Feldtypen

Die Felder, die im letzten Abschnitt vorgestellt wurden, hatten tatsächlich keinen explizit definierten Typnamen. Sie gehörten zum anonymen Typ. Hierbei handelt es sich um einen der wenigen Fälle, in denen bei ADA ein Objekt vereinbart werden kann, ohne den Typ zu nennen - ein weiterer Fall sind die Prozesse.

Beispiel von oben:

```
type VECTOR_6 is array (1..6) of REAL;
A: VECTOR_6;B:VECTOR_6;
```

Dies ermöglicht Zuweisungen B:=A;

Andererseits, wenn wir definiert hätten

```
C: array (1..6) of REAL;
D: array (1..6) of REAL;
```

wäre die Zuweisung C:=D unzulässig, weil C und D nicht vom selben Typ sind. Sie gehören zu verschiedenen Typen, die beide anonym sind. Dagegen ist C:=D zulässig, wenn C,D: array (1..6) of REAL definiert wurden. Dann ist der Typ anonym, aber C und D gehören demselben Typ an.

Das bisher vorgestellte Modell der Feldtypen ist noch nicht zufriedenstellend. Es erlaubt uns nicht, eine abstrakte Sicht darzustellen, die das Gemeinsame zwischen Feldern umfaßt, die zwar verschiedene Grenzen haben, aber sonst dem gleichen Typ angehören. Im besonderen dürften keine Unterprogramme geschrieben werden, die ein Feld mit willkürlichen Grenzen als aktuelle Parameter haben könnten. Das wird allgemein bei Pascal als erhebliche Schwierigkeit angesehen. Daher wird das Konzept eines uneingeschränkten Feldtyps eingeführt, in dem die Wertebereichseinschränkungen für die Indizes nicht definiert sind.

```
type VECTOR is array (INTEGER range <>) of REAL;
```

(das zusammengesetzte Symbol <> wird als “Box” bezeichnet.)

Das bedeutet, daß VECTOR der Name eines Typs ist, der ein eindimensionales Feld mit Komponenten des Typs REAL mit einem INDEX vom Typ INTEGER definiert. Die obere und untere Grenze sind nicht definiert. Der Ausdruck **range** <> soll die Vorstellung vermitteln, daß später an dieser Stelle weitere Informationen hinzugefügt werden. Wenn wir Objekte des Typs VECTOR vereinbaren, müssen wir die Grenzen bereitstellen. Wir können das auf verschiedene Art und Weise tun. Man könnte einen Untertyp einführen und dann die Objekte vereinbaren

```
subtype VECTOR_5 is VECTOR(1..5);
V:VECTOR_5;
```

oder wir vereinbaren die Objekte direkt

```
V: VECTOR(1..5);
```

In beiden Fällen sind die Grenzen durch einen diskreten Wertebereich in Klammern definiert. Alle Formen diskreter Wertebereiche können verwendet werden.

Es gibt noch eine weitere Form des Aggregats, die als benanntes Aggregat bezeichnet wird, in der den Komponenten ihr Indexwert und => vorausgeht. Ein einfaches Beispiel wäre

```
(1 => 0.0,2 =>0.0,3=>0.0,4=>0.0,5=>0.0,6=>0.0)
```

Die Regeln für benannte Ausdrücke benannter Aggregate sind den Regeln für die Alternativen in einer case-Anweisung sehr ähnlich.

Ein benanntes Aggregat kann als eine Reihe von Alternativen betrachtet werden, wobei es sich bei jeder um einen einzelnen Wert oder einen diskreten Wertebereich handeln kann. Wir können daher einige vorangegangene Beispiele neu schreiben.

```
A: array (1..6) of REAL:=(1..6 => 0.0);
WORK_DAY : constant array (DAY)
             of BOOLEAN:=(MON..FRI => TRUE, SAT|SUN => FALSE);
```

Im Gegensatz zu einem Stellungsaggregat müssen die Indexwerte nicht in einer bestimmten Reihenfolge auftreten. Wir können auch **others** verwenden, aber dann muß es wie bei der case-Anweisung als letztes und allein stehen. Die Verwendung von **others** wirft einige Probleme auf, weil klar sein muß, was man unter der Gesamtheit der Werte versteht. Man sollte auch erkennnen, daß Aggregate, obwohl wir sie bisher als Anfangswerte dargestellt haben, ganz allgemein überall verwendet werden können, wo ein Ausdruck eines Feldtyps benötigt wird. Es ist daher notwendig, daß der Kontext die Grenzen bestimmt, wenn **others** verwendet wird. Eine Art, den richtigen Kontext bereitzustellen, ist die, daß man das Aggregat qualifiziert, um zwischen überladenen Aufzählungsliteralen zu unterscheiden. Um das durchzuführen, müssen wir einen passenden Typ- oder Untertyp zur Verfügung haben. Vielleicht ist es überraschend, daß der Kontext eines Anfangswertes nicht ausreicht, auch wenn die Grenzen des Typs bekannt sind. Wir können nicht

```
WORK_DAY : constant array (DAY) of

             BOOLEAN:=(MON..FRI => TRUE, others => FALSE);
```

schreiben, sondern müssen statt dessen

```
WORK_DAY : constant array (DAY) of
             BOOLEAN:=DAY'(MON..FRI => TRUE, others => FALSE);
```

schreiben.

Feldaggregate sind ziemlich kompliziert, und man sollte noch einige Punkte erwähnen. In einem Aggregat müssen alle Wertebereichsaufzählungen und Werte vor => statisch sein (wie bei der case-Anweisung). Gibt es allerdings nur eine Alternative, so könnte es sich auch um einen dynamischen Indexbereich oder einen einzelnen dynamischen Wert handeln.

```
A: array (1..N) of REAL:=(1..N => 0.0);
```

In diesem Fall könnte N sogar Null sein, wodurch ein leeres Feld erzeugt würde.

Ein weiterer Punkt ist der, daß bei Feldaggregaten die Stellungsschreibweise und die benannte Schreibweise nicht miteinander vermischt werden dürfen, mit der Ausnahme, daß others am Ende eines Stellungsaggregats verwendet werden darf. Die Verwendung von others in einem Stellungsaggregat bedeutet, daß die untere Grenze durch den Indextyp bestimmt werden kann, die obere Grenze aber nicht festgesetzt werden kann, weil die Anzahl der Komponenten nicht bekannt ist. Es gelten dann dieselben Regeln wie bei einem benannten Aggregat mit others - die Grenzen müssen durch den Kontext bestimmt werden.

Eine sinnvolle Anwendung für das RANG-Attribut ist folgende:

```
A: array (1..N) of REAL:=(A'RANGE => 0.0);
```

Das ist besser, als 1..N zu wiederholen, weil es die Abhängigkeit von N lokalisiert. Das Auftauchen von A in der Definition von A ist in diesem Fall zulässig, da zuerst A definiert wird und darauf eine Zuweisung erfolgt.

Zum Thema der Feldzuweisungen wäre noch zu sagen, daß es zwar notwendig ist, daß der Feldwert und das Feld denselben Typ haben und die Komponenten zusammenpassen, aber nicht, daß die Grenzen gleich sein müssen, sondern nur, daß die Anzahl der Komponenten und Dimensionen die gleiche sein muß.

```
V: VECTOR(1..5);
W: VECTOR(0..4);
...
V:=W;
```

2.6.3 Zeichen und Zeichenketten

Um die Aufzählungstypen zu vervollständigen, führen wir die Zeichentypen ein. Ein Zeichenliteral ist eine weitere Form einer lexikalischen Einheit. Es besteht aus einem einzelnen Zeichen zwischen zwei einfachen Anführungszeichen. Das Zeichen muß eines der druckfähigen Zeichen sein. Es darf kein Kontrollzeichen sein, wie horizontaler Tabulatorsprung oder Neue Zeile.

'A' oder 'a'

Es gibt einen vordefinierten Aufzählungstyp CHARACTER, der folgendermassen implementiert sein könnte.

```
type CHARACTER is ('A','B',...);
```

Neben dem vordefinierten Typ CHARACTER gibt es auch den vordefinierten Typ STRING.

```
type STRING is array (NATURAL range <>) of CHARACTER;
```

Das ist ein gewöhnlicher Feldtyp und folgt allen Regeln des vorhergehenden Abschnitts. Folglich kann man schreiben:

```
S: STRING(1..7);
```

um ein Feld mit dem Indexbereich 1..7 zu vereinbaren. Im Falle eines konstanten Feldes können die Grenzen von Anfangswerten abgeleitet werden

```
G: constant STRING:=('P','I','G');
```

wobei der Anfangswert die Form eines normalen Stellungsaggregats einnimmt. Die untere Grenze von G (d.h. G'FIRST) ist 1, weil der Indextyp von STRING NATURAL ist, und NATURAL'FIRST 1 ist.

Eine andere Schreibweise ist für ein Stellungsaggregat vorgesehen, dessen Komponenten Zeichenliterale sind:

```
G: STRING:="PIG";
```

Eine Zeichenkette kann auch nur aus einem Zeichen bestehen oder leer sein. Die äquivalenten entsprechenden Aggregate, die Zeichenliterale verwenden, müssen in benannter Schreibweise geschrieben werden.

```
(1 => 'A') = "A"
(1..0 => 'A') = ""
```

Eine wesentliche Anwendung von Zeichenketten ist die Ausgabe von Zeichen auf den Bildschirm oder in eine Datei mit der Anweisung put.

```
put("Hello");
```

2.7 Verbunde

Ein Verbund ist ein zusammengesetztes Objekt, das aus benannten Komponenten besteht, die zu verschiedenen Typen gehören können. Im Gegensatz zu den Feldern gibt es keine anonymen Verbundtypen - sie müssen alle benannt sein.

Betrachtet man folgendes Beispiel

```
type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,
                     AUG,SEP,OCT,NOV,DEC);
type DATE is
  record
    DAY: INTEGER range 1..31;
    MONTH: MONTH_NAME;
    YEAR: INTEGER;
  end record;
```

so vereinbart es den Typ DATE als einen Verbund, der drei benannte Komponenten enthält, nämlich DAY, MONTH und YEAR.

```
D: DATE;
```

vereinbart ein Objekt D vom Typ DATE. Die einzelnen Komponenten von D können gekennzeichnet werden, indem man D mit einem Punkt und dem Komponentennamen schreibt.

```
D.DAY:=4;
D.MONTH:=JUL;
D.YEAR:=1776;
```

oder `D:=(4,JUL,1776);`
oder `D:=(MONTH => JUL, DAY => 4, YEAR => 1776);`

Verbunde sind bei weitem nicht so kompliziert wie Feldaggregate, da die Anzahl der Komponenten immer bekannt ist. In einem Stellungsaggregat treten die Komponenten immer in der definierten Reihenfolge auf, wogegen in einem benannten Aggregat die Reihenfolge beliebig ist.

Es gibt eine zusätzliche Möglichkeit für Verbunde, die positionelle und die benannte Schreibweise zu vermischen. Dabei müssen die positionellen Komponenten zuerst kommen. Und zwar in der Reihenfolge (ohne Löcher) wie gewohnt. D.h. wir können auf die benannte Bezeichnung überall im Aggregat umsteigen, müssen aber dann dabei bleiben. Man könnte dann schreiben

```
(4,YEAR => 1776, MONTH => JUL);
```

Es ist möglich, einige oder alle Komponenten in der Typvereinbarung zu initialisieren.

```
type COMPLEX is
  record
    RL,IM:REAL:=0.0;
  end record;
```

Man könnte vereinbaren:

```
C1: COMPLEX;
C2: COMPLEX:=(1.0,0.0);
```

Im Fall C2 werden die Initialisierungswerte mit 1.0 und 0.0 überschrieben. Man beachte, daß ein Aggregat immer vollständig angegeben werden muß. Eine weitere Zuweisung wäre `(RL|IM => 1.0)`

Eine genauere Behandlung befindet sich im Abschnitt Höhere Typen.

2.8 Unterprogramme

2.8.1 Funktionen

Beginnen wir mit einem Beispiel: Der Funktion SQRT

```
function SQRT(X:REAL) return REAl is
  R: REAL;
begin
  -- compute value of SQRT(X) in R
  return R;
end SQRT;
```

Jeder Funktionsrumpf beginnt mit dem reservierten Wort **function** und dem Namen der Funktion, die definiert wird. Wenn die Funktion Parameter hat, folgt dem Namen eine Liste von Parametervereinbarungen in Klammern. Falls es mehrere Vereinbarungen gibt, sind sie durch Strichpunkte voneinander getrennt. Jede Vereinbarung gibt die Bezeichner von einem oder mehreren Parametern an, denen ein Doppelpunkt und ihr Typ oder Untertyp folgen. Nach der Parameterliste, falls sie existiert, steht das reservierte Wort **return** und der Typ oder Untertyp des Ergebnisses der Funktion.

Der bisher beschriebene Teil wird als Spezifikation der Funktion bezeichnet. Er spezifiziert die Funktion hinsichtlich ihrer Aufrufumgebung. Nach der Spezifikation kommt **is** und dann der entsprechende Rumpf, der ein begin-Block mit einem abschließendem **end** ist. Zwischen **end** und dem abschließenden Strichpunkt kann man den Namen der Funktion wiederholen. Ein Funktionsrumpf kann mehrere returns enthalten. Es ist aber darauf zu achten, daß man nicht in das abschließende end hineinläuft, da sonst die Funktion einen undefinierten Wert zurück erhält. Funktionen können, wie in anderen Programmiersprachen (Pascal), rekursiv aufgerufen werden. Ein Vorteil gegenüber Pascal ist die Übergabe von uneingeschränkten Feldtypen, die durch folgendes Beispiel deutlich wird.

```
type VECTOR is array (INTEGER range <>) of REAL;
```

In einem solchen Fall werden als Grenzen des formalen Parameters jene des aktuellen Parameters gesetzt.

```
function SUM(A:VECTOR) return REAL is
  RESULT: REAL :=0.0;
begin
  for I in A'RANGE loop
    RESULT:=RESULT+A(I);
  end loop;
return RESULT;
end SUM;
```

Die Funktion SUM kann verwendet werden, um die Komponenten eines Vektors mit beliebigen Grenzen zu summieren. ADA überwindet damit ein Problem von Pascal, das nur Feldparameter mit statischen Grenzen besitzt.

2.8.2 Operatoren und Funktionen

Im letzten Abschnitt folgte auf das reservierte Wort **function** immer der Bezeichner der Funktion. Er kann jedoch auch eine Zeichenkette sein, vorausgesetzt, daß die Zeichenkette aus einem der folgenden Sprachoperatoren zwischen doppelten Anführungszeichen besteht.

`and, or, xor, =, <,<=,>, >=, +, -, \&, not, *, /, mod, rem, **`

In einem solchen Fall definiert die Funktion eine neue Bedeutung des betreffenden Operators.

```
function "*" (A,B: VECTOR) return REAL is
  RESULT: REAL:=0.0;
begin
  for I in A'RANGE loop
    RESULT:=RESULT+A(I)*B(I);
  end loop;
  return RESULT;
end "*";
```

Man ruft diese Funktion mit der normalen Syntax auf, die im Zusammenhang mit dem Operator * verwendet wird.

```
V,W: VECTOR;
R:=V*W;
```

Die Bedeutung von * wird von den bestehenden Bedeutungen der ganzzahligen und reellen Multipliktion durch den Kontext unterschieden, der durch die Typen der aktuellen Parameter bestimmt wird.

Einem Operator mehrere Bedeutungen zu geben, ist ein weiteres Beispiel für Überladung (siehe Aufzählungsliterale). Man kann den Operatoren zwar neue Bedeutungen hinzufügen, aber man kann die Syntax des Aufrufs nicht ändern. Folglich muß die Anzahl der Parameter von * immer zwei sein. Die Operatoren +, - neu zu definieren, ist nicht üblich, da sie entweder dyadischer oder monadischer Natur sind.

Beispiel SUM:

```
function "+" (A: VECTOR) return REAL is
  RESULT: REAL:=0.0;
begin
  for I in A'RANG loop
    RESULT:=RESULT+A(I);
  end loop;
  return RESULT;
end "+";
```

Aufruf ist dann s:=+V; und nicht S:=SUM(V);

2.8.3 Prozeduren

Der Rumpf einer Prozedur ist dem einer Funktion sehr ähnlich. Die Unterschiede sind:

- eine Prozedur beginnt mit **procedure**
- ihr Name muß ein Bezeichner sein
- sie gibt kein Ergebnis zurück
- es gibt drei verschiedene Arten von Parametern: **in**, **out** oder **in out**.

Die Art des Parameters wird dadurch spezifiziert, daß nach dem Doppelpunkt in der Parametervereinbarung **in** oder **out** oder **in out** steht. Wird die Art ausgelassen, dann wird **in** angenommen. Im Falle von Funktionen ist **in** die einzig erlaubte Art (**function** SQRT(X: **in** REAL) **return** REAL; siehe auch LRM Abschnitt 6.2)

in: Der Parameter agiert als lokale Konstante, deren Wert durch den entsprechenden aktuellen Parameter bereitgestellt wird.

out: Der Parameter agiert als lokale Variable, deren Wert dem entsprechenden aktuellen Parameter als Ergebnis der Durchführung der Prozedur zugewiesen wird.

in out: Der Parameter agiert als lokale Variable und gestattet Zuweisungen an den aktuellen Parameter.

2.8.4 Benannte und default Parameter

Bis jetzt wurden die aktuellen Parameter immer in positioneller Reihenfolge angegeben. Wie bei den Aggregaten kann man auch die benannte Schreibweise verwenden.

```
procedure INCREMENT(X: in out INTEGER) is
begin
  X:=X+1;
end ;
```

Aufruf in benannter Schreibweise: `INCREMENT( X => X);`

Wie bei den Verbundaggregaten dürfen die benannten und positionellen Bezeichnungen gemischt werden , und positionelle Parameter müssen zuerst in der richtigen Reihenfolge kommen. Anders als bei den Verbundaggregaten muß jeder Parameter einzeln geschrieben werden, **others** darf nicht verwendet werden.

vorbesetzte Parameter

Es kommt machmal vor, daß ein oder mehrere in-Parameter gewöhnlich denselben Wert bei jedem Aufruf haben. Man kann für sie einen Vorbesetzungswert in der Prozedurspezifikation angeben und kann sie dann beim Aufruf weglassen.

Prozentiges Beispiel:

```
type SPIRIT is (GIN,VODKA);
type STYLE is (ON_THE_ROCKS,STRAIGHT_UP);
type TRIMMING is (OLIVE,TWIST);
```

vorbesetzte Prozedurspezifikation

```
procedure DRY_MARTINI(BASE: SPIRIT:=GIN;
                     HOW: STYlE:=ON_THE_ROCKS;
                     PLUS: TRIMMING:=OLIVE);
```

Typische Aufrufe wären:

```
DRY_MARTINI;
DRY_MARTINI(VODKA,PLUS => TWIST);
```

2.9 Modulkonzept (Programmieren im großen)

In diesem Kapitel sollen Pakete und ansatzweise die Mechanismen für eine getrennte Übersetzung der einzelnen Pakete beschrieben werden.

2.9.1 Pakete

Eines der Hauptprobleme der traditionellen blockstrukturierten Sprachen, wie z.B. Pascal, besteht darin, daß sie nicht genug Kontrolle über die Sichtbarkeit bieten. Nehmen wir als Beispiel einen Kellerspeicher an, der dargestelt wird durch ein Feld und eine Variable, die das oberste Element anzeigt, eine Prozedur PUSH und eine Funktion POP.

Man könnte schreiben:

```
MAX: constant:=100;
S: array(1..MAX) of INTEGER;
TOP: INTEGER range 0..MAX;

procedure PUSH(X: INTEGER) is
begin
  TOP:=TOP+1;
  S(TOP):=X;
```

```
end PUSH;

function POP return INTEGER is
begin
  TOP:=TOP-1;
  return S(TOP+1);
end POP;
```

In einer einfachen blockstrukturierten Sprache gibt es keine Möglichkeit, zu den Unterprogrammen PUSH und POP Zugriff zu bekommen, ohne gleichzeitig auch direkten Zugriff zu den Variablen S und TOP zu haben. Als Folge davon ist es nicht möglich, einerseits die für die Benutzung der Unterprogramme notwendigen Details bereitzustellen, ohne andererseits Details der Implementierung des Kellers zu verraten.

Das Paket in ADA erlaubt es, diese Problem zu lösen, indem es einen Schutzwall um eine Gruppe von Vereinbarungen errichtet und nur Zugriff auf jene gestattet, die wir sichtbar machen. Ein Paket besteht im wesentlichen aus zwei Teilen: Der Spezifikation, die die Schnittstelle zur Außenwelt darstellt, und dem Rumpf, der die verborgenen Einzelheiten enthält.

```
package STACK is                 -- specification
  procedure PUSH(X: INTEGER);
  function POP return INTEGER;
end STACK;

package body STACK is            -- body
  MAX: constant:=100;
  S: array(1..MAX) of INTEGER;
  TOP: INTEGER range 0..MAX;

  procedure PUSH(X: INTEGER) is
  begin
    TOP:=TOP+1;
    S(TOP):=X;
  end PUSH;

  function POP return INTEGER is
  begin
    TOP:=TOP-1;
    return S(TOP+1);
  end POP;

begin                            -- initialisation
  TOP:=0;
end STACK;
```

Die Paketspezifikation beginnt mit dem reservierten Wort **package**, dem Bezeichner des Paketes und **is**. Dem folgen dann Vereinbarungen der Objekte, die nach außen sichtbar sein sollen. Diese werden mit **end**, dem Bezeichner (fakultativ) und dem Strichpunkt abgeschlossen.

In unserem Beispiel enthält die Spezifikation nur die Vereinbarungen der zwei Unterprogramme PUSH und POP.

Der Paketrumpf beginnt ebenfalls mit **package**, aber darauf folgen das reservierte Wort **body**, der Bezeichner und **is**. Man hat dann einen normalen Vereinbarungsteil, **begin**, eine Folge von Anweisungen, **end**, einen fakultativen Bezeichner und einen Strichpunkt, genauso wie in einem Block.

In unserem Beispiel enthält der Vereinbarungsteil die Variablen, die den Kellerspeicher und die Rümpfe von PUSH und POP darstellen.

Die Anweisungsfolge zwischen **begin** und **end** wird ausgeführt, wenn das Paket vereinbart wird und kann für die Initialisierung verwendet werden. Wenn eine Initialisierung nicht nötig ist, kann **begin** weggelassen werden.

Man beachte, daß ein Paket selbst eine Vereinbarung ist und deshalb nur eine der Einheiten in einem äußeren Vereinbarungsteil ist (wenn es sich nicht um eine Bibliothekseinheit handelt, die ohnedies die äußerste Schicht darstellt).

Die Abarbeitung eines Paketrumpfes besteht einfach aus der Abarbeitung der inneren Vereinbarungen, gefolgt von der Durchführung des Initialisierungsteils, falls einer vorhanden ist. Das Paket existiert bis zum Ende des Gültigkeitsbereichs, indem es vereinbart wurde. Objekte, die innerhalb eines Paketes vereinbart wurden, haben dieselbe Lebenszeit wie das Paket selbst.

Pakete können in jedem beliebigen Vereinbarungsteil, z.B. in einem Block, Unterprogramm oder in einem anderen Paket vereinbart werden. Wenn eine Paketspezifikation in einer anderen Paketspezifikation vereinbart wird, dann muß der Rumpf der einen im Rumpf der anderen vereinbart werden.

Das Paket selbst hat einen Namen und man kann sich die Objekte in einem sichtbaren Teil als Komponenten des Paketes vorstellen. Daher ist es notwendig, wenn man die Funktion PUSH aufruft, daß man auch den Namen STACK erwähnt.

```
declare
  package STACK is
    ...
  end STACK;
begin
  ...
  STACK.PUSH(M);
  ...
  N:=STACK.POP;
  ...
end;
```

Innerhalb des Paketes ruft man die Funktion PUSH mit PUSH oder STACK.-PUSH auf. Außerhalb des Pakete muß STACK.PUSH verwendet werden. Im allgemeinen ist es mühsam, immer STACK.PUSH zu schreiben, wenn man PUSH von außen aufrufen möchte.

Statt dessen könnte man

```
use STACK;
```

als eine Art Abkürzung schreiben.

```
declare
  use STACK;
begin
  ...
  PUSH(M);
  ...
  N:=POP;
  ...
end;
```

Der use-Satz wirkt wie eine Vereinbarung und hat auch einen Gültigkeitsbereich, der bis zum Ende des Blocks reicht. Außerhalb müßte man zur punktierten Schreibweise zurückkehren (STACK.POP bzw. STACK.PUSH).

Allgemein können wir zuerst alle Spezifikationen zusammen anordnen und dahinter alle Rümpfe, oder wir können die entsprechenden Spezifikationen und Rümpfe zusammen anordnen.

Die Regeln für die Anordnung sind:

- lineare Ausführung der Vereinbarungen,
- die Spezifikation muß vor dem Rumpf desselben Paketes stehen

2.9.2 Bibliothekseinheiten

In vielen Sprachen wurde in der Vergangenheit die Tatsache übersehen, daß Programme in Teilen entwickelt werden, die getrennt übersetzt und anschließend integriert werden. In ADA wurde die Notwendigkeit getrennter Übersetzung erkannt, und diese Sprache bietet daher zwei Mechanismen an - eine top-down-Übersetzung und eine bottom-up-Übersetzung.

Der top-down-Mechanismus eignet sich für die Entwicklung eines großen zusammenhängenden Programmsystems, das trotzdem aus verschiedenen Gründen in Untereinheiten zerlegt wird, die getrennt übersetzt werden können. Die Untereinheiten werden nach der Einheit übersetzt, in der sie logisch enthalten sind.

Der bottom up-Mechanismus eignet sich für den Aufbau einer Programmbibliothek, in der Einheiten für allgemeine Verwendungszwecke und folglich vor den Programmen entwickelt werden, die sie verwenden. Eine Bibliothekseinheit kann eine Unterprogrammspezifikation, ein Unterprogrammrumpf, eine Paketspezifikation oder ein Paketrumpf sein.

Wenn eine Bibliothekseinheit übersetzt ist, wird sie in einer Programmbibliothek abgelegt. Es wird klarerweise je nach Verwender, Projekt usw. mehrere solcher Bibliotheken geben.

Das Anlegen und die Handhabung von Bibliotheken werden später genauer erklärt.

Eine Einheit, die sich einmal in der Bibliothek befindet, kann von jeder nachfolgenden zu übersetzenden Einheit verwendet werden, aber die verwendete Einheit muß die Abhängigkeit mit einem with-Satz anzeigen.

Nehmen wir an, wir übersetzen das Paket STACK. Dieses Paket hängt von keiner anderen Einheit ab und braucht daher keinen with-Satz. Wir übersetzen Spezifikation und Rumpf miteinander, so daß der einzugebende Text so lautet.

```
package STACK is
  ...
end STACK;

package body STACK is
  ...
end STACK;
```

Nehmen wir jetzt an, daß wir eine Prozedur MAIN schreiben, die das Paket STACK verwendet. Unser Programm MAIN stellt das Hauptprogramm im üblichen Sinne da.(Anmerkung: Die ADA-Definition schreibt nicht vor, daß das Hauptprogramm den Bezeichner MAIN erhalten muß. Es ist aber allgemein üblich.

```
with STACK;
procedure MAIN is
  use STACK;
  M,N: INTEGER;
begin
  ...
  PUSH(M);
  ...
  N:=POP;
  ...
end MAIN;
```

Der with-Satz kommt vor der Einheit, so daß die Abhängigkeit der Einheit von der anderen klar ist. Ein with-Satz darf nicht in einem inneren Gültigkeitsbereich eingebettet sein. Wenn die Einheit von mehreren anderen Einheiten abhängig ist, so ist es zulässig zu schreiben

```
with STACK,STACK_NEW
```

Weiter kann der use-Satz direkt hinter die with-Anweisung geschrieben werden

```
with STACK; use STACK;
```

Wenn eine Paketspezifikation und ein Paketrumpf getrennt übersetzt werden, dann muß der Rumpf nach der Spezifikation übersetzt werden. Man sagt, daß der Rumpf von der Spezifikation abhängig ist.

Jede Einheit, die das Paket verwendet, ist aber nur von der Spezifikation und nicht vom Rumpf abhängig. Wenn der Rumpf auf irgendeine Art geändert wird, die im Einklang mit der unveränderten Spezifikation steht, muß jede beliebige Einheit, die das Paket verwendet, nicht nochmals übersetzt werden. Dies vereinfacht die Programmwartung erheblich.

Die allgemeine Regel für die Reihenfolge der Übersetzung ist, daß eine Einheit nach allen jenen Einheiten übersetzt werden muß, von denen sie abhängt.

Es gibt das STANDARD-Paket, das in einem with-Satz nicht erwähnt werden muß. Es enthält alle vordefinierten Typen, wie INTEGER und BOOLEAN. Intern ist ein Paket ASCII definiert, das die Zeichen zu Verfügung stellt (siehe auch jeweilige Implementation).

2.9.3 Untereinheiten (Subunits)

Der Rumpf eines Paketes, Unterprogramms (oder Prozesses) kann aus seiner unmittelbar umschließenden Einheit "herausgenommen" und getrennt übersetzt werden. Der Rumpf wird dann durch einen Rumpfstumpf (stub) ersetzt.

```
package body STACK is                           -- body
  MAX: constant:=100;
  S: array(1..MAX) of INTEGER;
  TOP: INTEGER range 0..MAX;
  procedure PUSH(X: INTEGER) separate;          -- stub
  function POP return INTEGER separate;         -- stub
begin                                           -- initialisation
  TOP:=0;
end STACK;
```

Die entfernten Einheiten werden als Untereinheiten bezeichnet und können getrennt übersetzt werden. Vor den Untereinheiten muß **separate**, gefolgt von dem Namen der Stammeinheit in Klammern, stehen.

Damit ergibt sich die Untereinheit von PUSH zu

```
separate (STACK)
procedure PUSH(X: INTEGER) is
begin
  TOP:=TOP+1;
  S(TOP):=X;
end PUSH;
```

Benötigen Untereinheiten weitere Zugriffe auf Bibliothekseinheiten, dann müssen die notwendigen with- und use-Sätze vor **separate** stehen.

2.10 Umbenennung

Gewisse Objekte können mit **renames** umbenannt werden. Dazu gibt man die Spezifikation der Funktion oder Prozedur mit dem neuen Bezeichner an, fügt **renames** und den Namen der alten Funktion, bzw. Prozedur hinzu (evtl. punktierte Schreibweise). Man kann z.B. schreiben:

```
declare
  procedure SPUSH(X: INTEGER) renames STACK.PUSH;
  function SPOP return INTEGER renames STACK.POP;
begin
  ...
  SPUSH(M);
  ...
  N:=SPOP;
  ...
end;
```

Ein möglicher Grund, das zu tun, ist Mehrdeutigkeiten zu beseitigen und dennoch die Verwendung der vollständigen punktierten Schreibweise zu vermeiden. Wenn man zwei Pakete mit einer Prozedur PUSH hätte (beide mit INTEGER Parametern), dann wäre der use-Satz kein Vorteil, weil der volle Name dennoch notwendig wäre, um den Doppelsinn zu beseitigen. Weiter kann man das Umbenennen von Funktionen dazu verwenden, Überladungen zu beseitigen.

```
function INNER(X;Y: VECTOR) return REAL renames "*";
```

Man beachte, daß die Umbenennung nicht einer Textsubstitution entspricht - die Identität des Objektes wird bestimmt, wenn die Umbenennung vorkommt. Wenn sich in der Folge der Name einer Variablen ändert , ändert sich die Identität des Objekts nicht.

Die Umbenennung kann für Objekte (Variablen und Konstanten), Ausnahmebedingungen, Unterprogramme und Pakete verwendet werden.

Die wahllose Verwendung von Umbenennungen sollte vermieden werden, weil dadurch eingeführte Decknamen das Prüfen eines Programms viel schwieriger machen.

2.11 Private Typen

Wir haben gesehen, wie wir mit Hilfe von Paketen dem Paketbenutzer innere Objekte eines Paketes verbergen können. Private Typen ermöglichen es uns, einem Benutzer Details der Konstruktion eines Typs zu verbergen.

2.11.1 Normale private Typen

Beginnen wir mit einem Beispiel: Paket COMPLEX_NUMBERS

```
package COMPLEX_NUMBERS is
  type COMPLEX is
    record
      RL,IM: REAL;
    end record;
  I: constant COMPLEX:=(0.0,1.0);

  function "+" (X,Y: COMPLEX) return COMPLEX;
  function "-" (X,Y: COMPLEX) return COMPLEX;
  function "*" (X,Y: COMPLEX) return COMPLEX;
  function "/" (X,Y: COMPLEX) return COMPLEX;
end;
```

Das Problem bei dieser Formulierung besteht darin, daß der Benutzer die kartesische Darstellung der komplexen Zahlen willkürlich verwenden kann.

Anstatt immer den complexen Operator + zu verwenden, könnte er auch

`C.IM:=C.IM+1.0;`

schreiben, statt des abstrakteren

`C:=C+I;`

Es könnte aber sein, daß wir verhindern wollen, daß der Benutzer sein Wissen über die Darstellung verwendet, so daß wir die Darstellung zu einem späteren Zeitpunkt vielleicht auf die polare Form ändern können und dann sicher sein wollen ,daß das verwendete Programm noch richtig ist. Mit Hilfe eines privaten Typs können wir das tun.

```
package COMPLEX_Numbers is
  type COMPLEX is private;
  I: constant COMPLEX;
  function "+" (X,Y: COMPLEX) return COMPLEX;
  function "-" (X,Y: COMPLEX) return COMPLEX;
  function "*" (X,Y: COMPLEX) return COMPLEX;
  function "/" (X,Y: COMPLEX) return COMPLEX;
  function CONS(R,I: REAL) return COMPLEX;
  function RL_PART(X: COMPLEX) return REAL;
  function IM_PART(X: COMPLEX) return REAL;
private
  type COMPLEX is
    record
      RL,IM: REAL;
    end record;
  I: constant COMPLEX:=(0.0,1.0);
```

```
end;
```

Der Teil der Paketspezifikation vor dem reservierten Wort **private** ist der sichtbare Teil und gibt die Information an, die außerhalb des Paketes verfügbar ist. Der Typ COMPLEX wird als privat vereinbart. Das bedeutet, daß außerhalb des Paketes nichts über die Details des Typs bekannt ist. Wir können auch Konstanten eines privaten Typs, wie z.B. I, im sichtbaren Teil vereinbaren. Der Anfangswert kann nicht im sichtbaren Teil festgesetzt werden, weil die Details des Typs noch nicht bekannt sind. Daher stellt man nur fest, daß I eine Konstante ist. Nach **private** muß man die Details der als privat vereinbarten Typen und die Anfangswerte der entsprechenden Konstanten angeben. Ein privater Typ kann auf jede beliebige Art implementiert werden, die im Einklang mit den Operationen steht, die dem Benutzer sichtbar sind.

2.11.2 Begrenzte private Typen (limited private types)

Die Operationen, die für einen privaten Typ zur Verfügung stehen, können vollkommen auf jene eingeschränkt werden, die im sichtbaren Teil des Paketes genannt werden. Das wird dadurch erreicht, daß man den Typ als begrenzt privat vereinbart.

```
type T is limited private;
```

In einem solchen Fall sind die Wertzuweisung, die Gleichheits (=) und Ungleichheitsabfrage (/=) außerhalb des Paketes unzulässig. Im Paket kann aber eine Funktion = definiert werden, wenn die zwei Parameter vom selben begrenzten privaten Typ sind. Sie muß einen Wert des Typs Boolean zurückliefern. Der Operator /= erhält seine Bedeutung immer von = und kann deswegen nicht explizit definiert werden.

Daß es für begrenzt private Typen keine Zuweisung gibt, hat einige natürliche Konsequenzen. Die Vereinbarung einer Variablen kann keine Initialisierung enthalten, Konstanten können nicht außerhalb des definierten Paketes vereinbart werden, und Parameter können keine Vorbesetzungswerte besitzen. Trotzdem ist es aber weiterhin möglich eigene Unterprogramme (Funktionen, Prozeduren) mit begrenzten privaten Typen als Parameter zu vereinbaren.

Der Vorteil, den privaten Typ zu begrenzen, liegt darin, daß der Paketentwickler damit die vollständige Kontrolle über die Objekte des Typs besitzt.

Als einfaches Beispiel betrachten wir folgendes:

```
package STACK is
  type STACK is limited private;
  procedure  PUSH(S: in out STACK; X: in INTEGER);
  procedure POP(S: in out STACK; X: out INTEGER);
  function "="(S,T:STACK) return BOOLEAN;
private
  MAX: constant:=100;
```

```
  type STACK is
    record
      S:array(1..MAX) of INTEGER;
      TOP: INTEGER range 0..MAX:=0;
    end record;
end;
```

Jedes Objekt des Typs STACK ist ein Verbund, der ein Feld S und eine ganze Zahl TOP umfaßt. Man beachte, daß TOP den Anfangswert Null besitzt. Das garantiert bei der Vereinbarung eines Stack-Objekts, daß dieses mit Null initialisiert wird.

Der Rumpf des Paketes könnte sein:

```
package body STACK is

  procedure PUSH(S: in out STACK; X: in INTEGER) is
  begin
    S.TOP:=S.TOP+1;
    S.S(TOP):=X;
  end PUSH;
  procedure POP(S: in out STACK; X: INTEGER) is
  begin
    X:=S.S(TOP);
    S.TOP:=S.TOP-1;
  end POP;
  function "="(S,T: STACK) return BOOLEAN is
  begin
    if S.TOP /= T.TOP then
      return FALSE;
    end if;
    for I in 1..S.TOP loop
      if S.S(I) /= T.S(I) then
        return FALSE;
      end if;
    end loop;
    return TRUE;
  end "=";

end STACK;
```

Dieses Beispiel enthält einige schöne Details:

Der Parameter S von PUSH hat die Übergabeart in out, weil wir sowohl vom Kellerspeicher lesen als auch auf den Kellerspeicher schreiben müssen. Weiter beachte man, daß POP keine Funktion sein darf, da S die Übergabeparameter in out haben muß und in Funktionen nur in-Parameter zugelassen sind. = kann eine Funktion sein, da vom Kellerspeicher nur gelesen werden muß. Die Funktion = liefert TRUE, wenn zwei Kellerspeicher gleich sind, d.h. wenn sie dieselbe

Größe und dieselben Elemente enthalten. Es ist falsch, den gesamten Verbund zu vergleichen, weil die nicht verwendeten Felder auch verglichen würden. Aus diesem Grund ist es sinnvoll, den Typ STACK nicht nur als privat, sondern als begrenzt privat zu vereinbaren. Wenn er nur privat wäre, dann könnte man = nicht neu definieren, um die richtige Bedeutung zu erhalten.

2.12 Ausnahmebedingungen (exceptions)

In den vorangegagenen Kapiteln haben wir an verschiedenen Stellen die Behandlung von Ausnahmebedingungen (exceptions) erwähnt. In diesem Kapitel beschreiben wir den Ausnahmebedingungsmechanismus und zeigen, was für Maßnahmen getroffen werden müssen, wenn eine Ausnahmebedingung auftritt. Weiter soll noch kurz auf die Definition und Behandlung eigener Ausnahmebedingungen eingegangen werden.

2.12.1 Behandlung von Ausnahmebedingungen

Wenn wir gegen eine der verschiedenen Sprachregeln verstoßen, dann wird eine Ausnahmebedingung bei der Programmausführung aktiviert.

Es gibt fünf vordefinierte Ausnahmebedingungen:

- CONSTRAINT_ERROR: diese bezieht sich im allgemeinen auf die Unter- und Überschreitung von Wertebereichen,
- NUMERIC_ERROR: tritt immer auf, wenn mit der Arithmetik etwas schief läuft, z.B. der Division durch Null
- STORAGE_ERROR: tritt auf , wenn der vorgesehene Speicherplatz überschritten wird, z.B. bei rekursiven Funktionen.

Die beiden anderen Ausnahmebedingungen sind SELECT_ERROR und TASKING_ERROR. Sie betreffen Prozesse und werden später behandelt.

Wenn wir erwarten, daß eine Ausnahmebedingung in einem Teil unseres Programms vorkommen kann, dann können wir einen Programmteil zur Behandlung dieser Ausnahmebedingung vorsehen.

```
begin
  --- sequence of statements
exeption
  when CONSTRAINT_ERROR =>
  -- do something
end;
```

Wenn CONSTRAINT_ERROR aktiviert wird, während die Anweisungsfolge zwischen **begin** und **exeption** ausgeführt wird, wird der Kontrollfluß unterbrochen und sofort zur Anweisungsfolge verzweigt, die auf => folgt. Der when-Satz wird als Programmteil zur Behandlung von Ausnahmebedingungen bezeichnet.

Ausnahmebedingungen sollten für selten vorkommende Fälle verwendet werden, oder für solche, die zum Zeitpunkt ihres Auftretens schwer zu überprüfen sind. Zwischen **exeption** und **end** können mehrere Programmabschnitte zur Behandlung von Ausnahmebedingungen angeführt werden.

```
begin
  -- sequence of statements
exception
  when NUMERIC_ERROR|CONSTRAINT_ERROR =>
    PUT("Numeric or Contraint error occurred")
    ...
  when STORAGE_ERROR =>
    PUT("Run out of Space");
    ...
  when others =>
    PUT("Something else went wrong");
    ...
end;
```

In diesem Beispiel wird für jede definierte Ausnahmebedingung eine Mitteilung ausgegeben. Man beachte die Ähnlichkeit zur case-Anweisung. Jedem **when** folgen ein oder mehrere Namen für Ausnahmebedingungen, die durch senkrechte Striche getrennt werden. Wie gewöhnlich können wir **others** verwenden, aber es muß zum Schluß und allein stehen. Es behandelt jede weitere Ausnahmebedingung, die nicht in den vorangegangenen Programmteilen zur Behandlung von Ausnahmebedingungen angeführt wurde. Programmteile zur Behandlung von Ausnahmebedingungen können am Ende eines Blockes, Unterprogrammrumpfes, Paketrumpfes (oder Prozeßrumpfes) auftreten und haben Zugriff auf alle Objekte, die in der Einheit vereinbart wurden. Es ist wichtig zu beachten, daß die Kontrolle niemals an die Einheit zurückgegeben werden kann, in der die Ausnahmebedingung aktiviert wurde. Die Anweisungsfolge, die auf => folgt, ersetzt den Rest der Einheit, die den Programmteil zur Behandlung der Ausnahmebedingung enthält, und beendet damit die Ausführung der Einheit. Ist dieser Programmteil eine Funktion, so muß die Ausnahmebehandlung eine return-Anweisung enthalten, um das Ergebnis der Ausnahmesituation zurückzugeben. Eine Ausnahmebehandlung am Ende eines Paketrumpfes gilt nur für die Initialisierungsfolge des Pakets und nicht für die Unterprogramme im Paket. Solche Unterprogramme müssen eigene Ausnahmebehandlungen besitzen, wenn sie Ausnahmebedingungen behandeln sollen. Wenn eine Einheit keine Ausnahmebehandlung für eine bestimmte Ausnahmebedingung bereitstellt, dann wird die Ausnahmebedingung dynamisch ausgebreitet. Das bedeutet, daß die Einheit beendet wird und die Ausnahmebedingung an jenem Punkt aktiviert wird, von dem aus die Einheit aufgerufen wurde. Im Falle eines Unterprogramms wird der Aufruf beendet, und wir suchen eine Ausnahmebehandlung in der Einheit, die das Unterprogramm aufgerufen hat. Dieser Vorgang wird wiederholt, bis wir entweder eine Einheit erreichen, in der eine Ausnahmebehandlung für die Ausnahmebedingung enthalten ist, oder bis wir auf die äußerste Ebene kommen. Haben wir das Hauptprogramm erreicht und noch keine Ausnahmebehandlung gefunden, dann wird das Hauptprogramm abgebrochen und die Laufzeitumgebung gibt eine entsprechende Meldung aus.

2.12.2 Vereinbarung und Aktivierung von Ausnahmebedingungen

Es ist kein guter Progammierstil, sich auf vordefinierte Ausnahmebedingungen zu verlassen. Es ist eine wesentlich bessere Lösung, eigene Ausnahmebedingungen zu definieren.

```
package STACK is                                 -- specification
  ERROR: exception                               -- exception
  procedure PUSH(X: INTEGER);
  function POP return INTEGER;
end STACK;

package body STACK is                            -- body
  MAX: constant:=100;
  S: array(1..MAX) of INTEGER;
  TOP: INTEGER range 0..MAX;

  procedure PUSH(X: INTEGER) is
  begin
    if TOP = MAX then
      raise ERROR;
    end if;
    TOP:=TOP+1;
    S(TOP):=X;
  end PUSH;

  function POP return INTEGER is
  begin
    if TOP = 0 then
      raise ERROR;
    end if;
    TOP:=TOP-1;
    return S(TOP+1);
  end POP;

begin                                            -- initialisation
  TOP:=0;
end STACK;
```

Eine Ausnahmebedingung wird auf ähnliche Art und Weise vereinbart wie eine Variable. Sie wird durch eine eigene Anweisung aktiviert, die den Namen der

Ausnahmebedingung enthält. Die Abwicklungs- und Abarbeitungsregeln sind dieselben wie für die vordefinierten Ausnahmebedingungen.

Man könnte jetzt schreiben.

```
declare
  use STACK;
begin
  ...
  PUSH(M);
  ...
  N:=POP();
  ...
exception
  when ERROR =>
    -- stack manipulation incorrect
  when others =>
    -- something else wrong
end;
```

Wir haben damit die Ausnahmebehandlung für eine fehlerhafte Verwendung des Kellerspeichers von den Ausnahmebehandlungen für andere Ausnahmebedingungen getrennt.

Für weitere Informationen und Besonderheiten sollten die entsprechende Literatur herangezogen werden.

2.13 Höhere Typen

In diesem Kapitel sollen die meisten der noch übriggebliebenen Typen beschrieben werde. Das sind die Verbundtypen mit Diskriminanten, die Zugriffstypen und die abgeleiteten Typen.

Die Prozeßtypen werden bei der Einführung der Prozesse besprochen.

2.13.1 Verbundtypen mit Diskriminanten

Bei den Verbundtypen, die wir bis jetzt kennengelernt haben, gab es keine formale Sprachabängigkeit zwischen den Komponenten. Eine etwaige Abhängigkeit existierte ausschließlich in der Vorstellungswelt des Programmierers, wie z.B. im Falle des begrenzten privaten Typs STACK, bei dem die Interpretation des Feldes S vom Wert der ganzen Zahl TOP abhing. Im Falle des Verbundtyps mit Diskriminanten werden einige Komponenten als Diskriminanten bezeichnet, und die übrigen Komponenten können von diesen abhängen. Diese Diskriminanten, die diskrete Typen sein müssen , kann man sich als eine Parametrisierung des Typs vorstellen.

Als einfaches Beispiel möchten wir ein Paket schreiben, das verschiedene Operationen auf quadratischen Matrizen ausführt. Insbesondere soll eine Funktion TRACE geschrieben werden, die die Spur einer Matrix berechnet.

Man könnte den Typ einer Matrix wie folgt definieren

```
type MATRIX is array (INTEGER range <>,INTEGER range <>) of REAL;
```

Aber unsere Funktion müßte dann überprüfen, ob die Matrix, die als aktueller Parameter übergeben wurde, wirklich quadratisch ist.

Da dies unbefriedigend ist, würde man eine Formulierung vorziehen, die garantiert, daß die Matrix immer quadratisch ist. Das kann man erreichen, indem wir einen Typ mit einer Diskriminante verwenden.

```
type SQUARE(ORDER: NATURAL) is
  record
  MATRIX: array (1..ORDER,1..ORDER) of REAL;
end record;
```

Das ist ein Verbundtyp mit zwei Komponenten: die erste, ORDER, ist eine Diskriminante des diskreten Typs NATURAL, und die zweite, ist ein Feld, dessen Grenzen vom Wert von ORDER abhängen. Variablen des Typs Square können wie gewöhnlich vereinbart werden, aber der Wert der Diskriminanten muß als Einschränkung gegeben sein.

```
M: SQUARE(3);
```

oder in der benannten Form

```
M: SQUARE(ORDER => 3);
```

Der Wert, der als Einschränkung vorgesehen ist, könnte jeder beliebige dynamische Ausdruck sein. Wenn die Variable einmal vereinbart ist, kann ihre Einschränkung nicht mehr geändert werden. Ein Anfangswert für M könnte durch ein Aggregat bereitgestellt werden. Dieses müßte aber vollständig sein und die passende Einschränkung wiederholen.

```
M: SQUARE(3):=(3,(1..3 => (1..3 => 0.0)));
```

Man könnte sogar schreiben:

```
M: SQUARE(N):=(M.ORDER,(M.MATRIX'RANGE(1) =>
                       (M.MATRIX'RANGE(2) => 0.0)));
```

Man kann natürlich auch Untertypen einführen:

```
subtype SQUARE_3 is SQUARE(3);
M: SQUARE_3;
```

Man könnte die Funktion TRACE nun wie folgt schreiben.

```
function TRACE(M:SQUARE) return REAL is
  SUM: REAL:=0.0;
begin
  for I in M.MATRIX'RANGE loop
    SUM:=SUM+M.MATRIX(I,I);
  end loop;
  return SUM;
end TRACE;
```

Es gibt jetzt keine Möglichkeit mehr, daß ein Aufruf von TRACE mit einer nicht quadratischen Matrix versorgt wird.

Es ist möglich, die Diskriminante mit einem vorbesetzten Wert zu vereinbaren.

```
type SQUARE(ORDER: NATURAL := 3) is
  record
  MATRIX: array (1..ORDER,1..ORDER) of REAL;
end record;
```

Die vorbesetzte Diskriminante kann bei der Definition später durch eine Zuweisung geändert werden. Wird keine Diskriminante angegeben, dann wird der vorbesetzte Wert verwendet.

Das Attribut CONSTRAINED kann für ein Objekt eines Typs mit Diskriminanten verwendet werden und liefert einen Booleschen Wert, der anzeigt, ob das Objekt eingeschränkt ist oder nicht.

Für jedes Objekt von Typen wie SQUARE, die keine Vorbesetzungswerte für die Diskriminanten haben, ist das Attribut TRUE.

2.13.2 Variante Teile

Es ist manchmal angenehm, einen Verbundtyp zu haben, in dem ein Teil der Struktur für alle Objekte des Typs fest ist, bei dem aber der Rest eine Form aus einer Menge verschiedener Möglichkeiten sein kann. Dies kann so realisiert werden, in dem man einen varianten Teil verwendet und die Wahl zwischen den Alternativen durch den Wert einer Diskriminante bestimmt.

```
type GENDER is (MALE, FEMALE);

type PERSON(SEX: GENDER) is
  record
    BIRTH: DATE;
    case SEX is
      when MALE =>
        BEARDED: BOOLEAN;
      when FEMALE =>
        CHILDREN: INTEGER;
    end case;
```

```
  end record;
```

Dieses Beispiel vereinbart einen Verbundtyp PERSON mit einer Diskriminante SEX. Die Komponente BIRTH des Typs DATE gilt für alle Objekte des Typs. Die übrigen Komponenten hängen jedoch von der Diskriminate SEX ab und werden als varianter Teil vereinbart. Wenn der Wert von SEX MALE ist, dann gibt es eine weitere Komponente BEARDED, während wenn SEX FEMALE ist, es eine Komponente CHILDREN gibt. Da kein Vorbesetzungswert für die Diskriminante gegeben ist, müssen alle Objekte des Typs eingeschränkt werden.

```
JOHN: PERSON(MALE);
BABARA: PERSON(FEMALE);
```

Aggregate haben die übliche Form. Sie geben nur die Komponenten für die entsprechende Alternative des varianten Teils an. Der Wert einer Diskriminante, die eine Variante bestimmt, muß statisch sein, d.h. Variablen sind als Diskriminanten nicht zulässig.

```
JOHN:=(MALE,(19,AUG,1937),FALSE);
BARBARA:=(FEMALE,(13,MAY,1943),2);
```

Auf die Komponenten einer Variante kann wie üblich zugegriffen werden.

```
JOHN.BERDED:=TRUE;
BARBARA.CHILDREN:=BARBARA.CHILDREN+1;
```

Der Zugriff auf eine Komponente der falschen Alternative (z.B. JOHN.CHILDREN) würde CONSTRAINT_ERROR aktivieren. Man beachte, daß das Geschlecht von Objekten des Typs PERSON zur Übersetzungszeit nicht bekannt sein muß, obwohl es nicht verändert werden kann.

```
S: GENDER:=...;
CHRIS: PERSON(S);
```

Dabei ist das Geschlecht von CHRIS nicht bestimmt, solange er oder sie nicht vereinbart sind. Daß eine Diskriminante statisch sein muß, gilt nur für Aggregate. Die Variablen des Typs PERSON sind notwendigerweise eingeschränkt, weil der Typ für die Diskriminante keinen Vorbesetzungswert hatte. Es ist daher nicht möglich, einen Wert zuzuweisen, der das Geschlecht ändern würde; ein Versuch, das zu tun, würde CONSTRAINT_ERROR aktivieren. Wir können jedoch für die Diskriminante einen vorbesetzten Anfangswert und in der Folge uneingeschränkte Variablen vereinbaren. Solchen uneingeschränkten Variablen können dann Werte mit verschiedenen Diskriminanten zugewiesen werden, aber nur durch vollständige Verbundzuweisungen.

2.13.3 Zugriffstypen (Zeiger)

Bei den bisher genannten Typen war der Name eines Objekts permanent mit dem Objekt verbunden. Objekte eines Zugriffstyps geben Zugriff auf andere

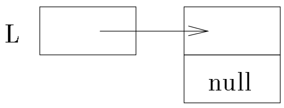
Objekte, und diese anderen Objekte können auf eine Art bereitgestellt werden, die unabhängig von der Blockstruktur ist. Die einfachste Verwendung für einen Zugriffstyp ist die Listenbearbeitung:

```
type CELL;                    -- vorlaeufige Vereinbarung

type LINK is access CELL;     -- Zugriffstyp
type CELL is
  record
    VALUE: INTEGER;
    NEXT: LINK;
  end record;

L: LINK:=null;
```

Diese Vereinbarungen führen einen Typ LINK ein, der auf CELL verweist (Verweistyp, Zeigertyp). Die Variable L kann man sich als Bezugsvariable vorstellen, die nur auf Objekte des Typs CELL hinzeigt. Das sind Verbunde mit zwei Komponenten, VALUE und NEXT, das wiederum auf Objekte des Typs CELL zeigt. Die Verbunde können daher in einer verketteten Liste angeordnet werden. Am Anfang gibt es keine Verbundobjekte, nur den einzelnen Zeiger L, der den Standardwert null hat. Die Zugriffsobjekte werden durch die Ausführung eines Objektkonstruktors (Allokators) geschaffen, der einen Anfangswert bereitstellen kann, aber nicht muß. Ein Objektkonstruktor besteht aus dem reservierten Wort **new** und dem Typ des neuen Objekts. Darauf kann noch der Anfangswert des Objekts folgen. Das Ergebnis eines Objektkonstruktors ist ein Zeigerwert, der dann einer Variablen des Zugriffstyps zugewiesen werden kann.

```
L:= new CELL;
```

Legt einen Verbund des Typs CELL an und weist dann L einen Zeiger auf das Objekt zu.



Die NEXT-Komponente erhält den Wert null, wogegen die VALUE-Komponente undefiniert bleibt. Auf die Komponente des Objekts, auf das L hinweist, kann mit der normalen punktierten Schreibweise zugegriffen werden.

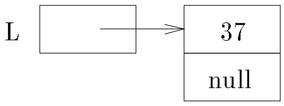
```
L.VALUE:=37;
```

oder auch

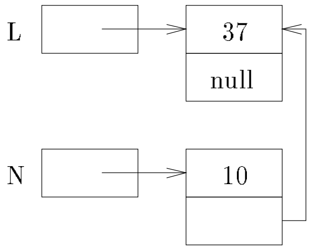
```
L:=new CELL(37,null);
```

Der Anfangswert nimmt hier die Form eines Aggregats an und muß Werte für alle Komponenten enthalten. Nehmen wir an wir möchten einen weiteren Verbund anlegen und mit dem bisherigen verbinden.

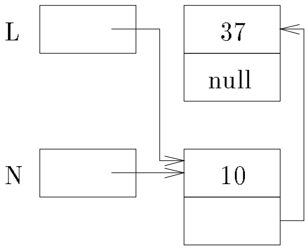
```
N: LINK;
N: new CELL(10,L);
L:=N;
```



(a) N: LINK;



(b) N: new CELL(10,L);



(c) L:=N;

Man beachte, daß durch L:=N; die Zugriffswerte (d.h. die Zeiger) und nicht die Objekte kopiert werden. Wenn wir die Objekte kopieren wollen, können wir das Komponente für Komponente tun, oder indem wir **all** verwenden.

`L.all` bezieht sich auf das ganze Objekt, auf das duch L hingewiesen wird.

`L = N` ist richtig, wenn sich L und N auf dasselbe Objekt beziehen, während `L.all = N.all` richtig ist, wenn die betreffenden Objekte den gleichen Wert haben.

Wir können eine Konstante des Zugriffstyps vereinbaren.

```
C: constant LINK:= new CELL(0,null);
```

Das bedeutet, daß sich die Variable C immer auf das gleiche Objekt beziehen muß. Der Wert des Objekts könnte jedoch geändert werden.

```
C.all:=L.all;
```

ist erlaubt, wogegen

```
C:=L;
```

verboten ist.

Dies soll eine Einführung in die Arbeit mit Zeigern in ADA sein. Weiter können sich Zugriffstypen natürlich auch auf private Typen beziehen oder auf Typen mit Diskriminanten. Für weitere Informationen verweise ich auf die Literatur.

2.13.4 Abgeleitete Typen (Vererbung)

Manchmal ist es sinnvoll, einen neuen Typ einzuführen, der in vielerlei Hinsicht einem vorhandenen Typ ähnlich ist, der aber trotzdem ein anderer Typ ist. Wenn T ein Typ ist, dann ist

```
type S is new T
```

S ist ein abgeleiteter Typ und T ist Stammtyp von S. Ein abgeleiteter Typ gehört zur selben Typengruppe wie sein Stamm. Die Wertemenge eines abgeleiteten Typs ist eine Kopie der Wertemenge des Stamms. Die Schreibweise für Literale und Aggregate und vorbesetzte Anfangswerte für den Typ und seine Komponenten sind gleich.

Folgende Operationen sind für einen abgeleiteten Typ möglich:

Er hat dieselben Attribute wie der Stamm.

Wenn der Stammtyp und deshalb auch der abgeleitete Typ nicht begrenzt privat sind, sind die Zuweisung, vordefinierte Gleichheit und Ungleichheit auch anwendbar.

Weiter wird ein abgeleiteterTyp gewisse Unterprogramme vom Stammtyp ableiten oder erben. Solche abgeleiteten Unterprogramme werden implizit an der Stelle der Definition des abgeleiteten Typs vereinbart, können aber später in der gleichen Vereinbarungsliste neu definiert werden. Die abgeleiteten Unterprogramme sollen an Beispielen erklärt werden.

Der einfachste Fall liegt vor, wenn der Stammtyp ein vordefinierter Typ ist. In diesem Fall sind die abgeleiteten Unterprogramme einfach die vordefinierten Unterprogramme.

```
type INTEGER_A is new INTEGER;
```

Dann erbt INTEGER_A alle vordefinierten Unterprogramme, wie z.B. +, -, * und ABS, sowie die Attribute FIRST und LAST. Wenn wir von INTEGER_A einen weiteren Typ ableiten, dann werden diese vererbten Unterprogramme weitergegeben.

Es sei das Paket P mit dem Typ INTEGER_A gegeben.

```
package P is
  type INTEGER_A is new INTEGER;
  procedure INCREMENT(I: in out INTEGER_A);
  function "&"(I,J: INTEGER_A) return INTEGER_A;
end;
```

Wenn wir jetzt

```
type INTEGER_B is new INTEGER_A;
```

nach dem Ende der Spezifikation vereinbart haben, dann erbt INTEGER_B die neuen Unterprogramme INCREMENT und & genauso wie die vordefinierten

Unterprogramme. Wenn wir eines der vererbten Unterprogramme nicht brauchen, dann können wir es einfach neu definieren.

```
package Q is
  type INTEGER_X is new INTEGER;
  function ABS(X: INTEGER_X) return INTEGER_X; -- neu
end;
```

Hier wird die vordefinierte Funktion ABS durch eine neue Version ersetzt. Wenn wir jetzt

```
type INTEGER_Y is new INTEGER_X;
```

nach dem Ende der Spezifikation schreiben, dann wird eine neue Version von ABS und nicht die ursprüngliche vererbt. Obwohl abgeleitete Typen voneinander verschieden sind, können die Werte des einen Typs in den anderen umgewandelt werden. Wenn

```
type S in new T;
TX: T;
SX: S;
```

gegeben ist, können wir

```
TX:=T(SX);
```

schreiben,oder

```
SX:=S(TX);
```

aber nicht

```
TX:=SX; -- illegal
SX:=TX; -- illegal
```

Wenn mehrfache Ableitungen vorhanden sind, dann muß man jede Umwandlung Schritt für Schritt nachvollziehen.

2.14 Numerische Typen

Es gibt in ADA zwei Gruppen numerischer Typen: ganzzahlige und reelle Typen. Hinsichtlich der Darstellung numerischer Typen in einem Computer gibt es zwei Probleme. Erstens kann der Wertebereich durch die Hardware (Speicherplatz) eingeschränkt sein. Zweitens könnte es unmöglich sein, alle möglichen Werte eines Typs, auf verschiedenen Maschinen, genau darzustellen. In ADA wurden daher die numerischen Typen mit einer Mindestmenge von Eigenschaften versehen. Damit wird die Portierbarkeit von Programmen erleichtert.

2.14.1 Ganzzahlige Typen

Alle Implementierungen von ADA weisen den vordefinierten Typ INTEGER auf. Zusätzlich können noch andere vordefinierte Typen vorkommen, wie z.B. LONG_INTEGER oder SHORT_INTEGER usw. Der Wertebereich dieser vordefinierten Tyen ist symmetrisch um Null angeordnet. Der vollständige Wertebereich kann mit den Attributen INTEGER'BASE'FIRST und INTEGER'-BASE'LAST ausgegeben werden.

z.B. Wertebereich für INTEGER (16Bit) -32768...+32767

z.B. Wertebereich für LONG_INTEGER (32Bit) -2147483648...+2147483647

Das Attribut BASE gilt für jeden Typ oder Untertyp und gibt den entsprechenden Basistyp an.

Abschließend sollte noch gesagt werden, daß der kleinste und größte positive Wert, der durch vordefinierte ganzzahlige Typen dargestellt werden kann, durch SYSTEM.MIN_INT und SYSTEM.MAX_INT bereitgestellt wird (diese Zahlen sind im Paket SYSTEM fest vereinbart).

2.14.2 Gleitpunkttypen

Alle Implementierungen von ADA haben einen vordefinierten Gleitpunkttyp FLOAT. Es gibt auch weitere vordefinierte Gleitpunkttypen wie SHORT_-FLOAT oder LONG_FLOAT. Wir können unseren eigenen Typ direkt ableiten durch

```
type REAL is new FLOAT;
```

In vielen Fällen ist es aber besser, die verlangte Genauigkeit auszudrücken und es der Implementierung zu überlassen, die passende Genauigkeit auszuwählen.

```
type REAL is digits 7; -- Genauigkeit 7 Dezimalstellen.
```

Verschiedene Attribute stehen zur Verfügung und sie können verwendet werden, um besser übertragbare Programme zu schreiben. Für jeden Typ F sind dies:

F'DIGITS	die Anzahl der Dezimalziffern D
F'MANTISSA	die entsprechende Anzahl von binären Ziffern B
F'EMAX	die größte Mantisse 4*F'MANTISSA
F'SMALL	die kleinste positive Modellzahl, 2.0**(- F'EMAX-1)
F'LARGE	die größte positive Modellzahl, 2.0**F'EMAX*(1.0-2.0**(- F'MANTISSA))
F'EPSILON	der Unterschied zwischen 1 und der nächsten Modellzahl, 2.0**(- F'MANTISSA+1)

Weiter gibt es noch die Attribute F'FIRST undF'LAST. Das Attribut BASE kann verwendet werden, um die Basis des implementierten Typ herauszufinden.

2.14.3 Festpunkttypen

Festpunkttypen verhalten sich im wesentlichen wie Gleitpunkttypen. Allerdings haben sie einen absoluten Fehler und keinen relativen, wie Gleitpunkttypen.

2.15 Generische Programmeinheiten

In diesem Kapitel soll der Mechanismus für generische Programmeinheiten beschrieben werden, der es ermöglicht Unterprogramme und Pakete zu parametrisieren, wobei die Parameter Typen, Unterprogramme, Werte und Objekte sein können.

2.15.1 Vereinbarungen und Inkarnation

Eines der Probleme einer typgebundenen Sprache wie ADA ist es, daß alle Typen zur Übersetzungszeit bestimmt werden müssen. Das bedeutet, daß Typen nicht als Laufzeitparameter übergeben werden können. Oft ergibt sich jedoch eine Situation, in der die Logik eines Programmstücks von den enthaltenen Typen unabhängig ist und es daher nötig erscheint, es für all die verschiedenen Typen zu wiederholen, für die wir es anwenden möchten.

Beispiel:

```
procedure SWAP(X;Y: in out REAL) is
  T: REAL;
begin
  T:=X;X:=Y;Y:=T;
end;
```

Es ist klar, daß der Sinn des Programms vom Typ der Werte, die ausgetauscht werden, unabhängig ist. Wenn wir auch ganze oder Boolesche Werte vertauschen möchten, könnte man natürlich andere Prozeduren schreiben. Der Mechanismus für generische Programmeinheiten ermöglicht es uns, dieses Problem eleganter zu lösen.

```
generic
  type ITEM is private;
procedure EXCHANGE(X,Y: in out ITEM);

procedure EXCHANGE(X,Y: in out ITEM) is
  T: ITEM;
begin
  T:=X;X:=Y;Y:=T;
end;
```

Das Unterprogramm EXCHANGE ist ein generisches Unterprogramm und stellt eine Art Schablone dar. Vor der Unterprogrammspezifikation steht der generische Teil, der aus dem reservierten Wort **generic** besteht, dem eine (möglicherweise leere) Liste von generischen Formalparametern folgt. Der Rumpf des Unterprogramms ist genau wie sonst geschrieben. Man beachte aber, daß im Falle eines generischen Unterprogramms sowohl der Rumpf , als auch die Spezifikation getrennt angegeben werden muß.

Die generische Prozedur kann nicht direkt aufgerufen werden. Wir müssen von ihr eine aktuelle Prozedur mit Hilfe eines Mechanismus erzeugen, der als generische Inkarnation bezeichnet wird.

```
procedure SWAP is new EXCHANGE(REAL);
```

Das ist eine Vereinbarung, die festlegt, daß SWAP von der Schablone, die durch EXCHANGE beschrieben wird, generiert werden soll. Aktuelle generische Parameter werden auf die übliche Art und Weise in einer Parameterliste bereitgestellt. Der aktuelle Parameter ist in diesem Fall vom Typ REAL.

In benannter Schreibweise:

```
procedure SWAP is new EXCHANGE(ITEM => REAL);
```

Man könnte weitere Inkarnationen erzeugen

```
procedure SWAP is new EXCHANGE(INTEGER);
procedure SWAP is new EXCHANGE(DATE);
```

Wir schaffen auf diese Art und Weise weitere Überladungen von SWAP, die durch ihre Parametertypen unterschieden werden können. Genauso wie generische Unterprogramme können wir auch generische Pakete vereinbaren. Ein einfaches Beispiel ist das schon erwähnte Paket STACK.

Dies funktionierte nur mit Variablen vom Typ INTEGER. Außerdem war die Größe des Stacks durch MAX begrenzt.

```
generic
  MAX: NATURAL;
  type ITEM is private;
package STACK is
  procedure PUSH(X: ITEM);
  function POP return ITEM;
end STACK;

package body STACK is
S: array (1.. MAX) of ITEM;
TOP: INTEGER range 00. MAX;
-- etc as before but with INTEGER
-- replaced by ITEM
end STACK;
```

Wir können jetzt einen Kellerspeicher einer beliebigen Größe und eines beliebigen Typs erzeugen und verwenden.

```
declare
  package MY_STACK is new STACK(100,REAL);
  use MY_STACK;
begin
  ...
  PUSH(X);
  ...
  Y:=POP();
  ...
end;
```

Das Paket MY_STACK, das durch eine Inkarnation entsteht, verhält sich genauso wie ein normales, ausprogrammiertes Paket. Der use-Satz erlaubt das direkte Zugreifen auf PUSH und POP.

Es können alle Parameter, die für Unterprogramme anwendbar sind, und sogar Typen und Unterprogramme, generische Formalparameter sein. Die benannten Parameter können von der Übergabeart **in** oder **in out**, aber nicht **out** sein. Wie bei Unterprogrammen wird **in** implizit angenommen (obiges Beipiel die Verwendung von MAX). Ein generischer in-Parameter verhält sich wie eine Konstante, deren Wert durch den entsprechenden aktuellen Parameter bereitgestellt wird. Ein in-out-Parameter verhält sich jedoch wie eine Variable und benennt den entsprechenden aktuellen Parameter um. Weiter darf der aktuelle Parameter keine Komponente eines nichteingeschränkten, diskriminierten Verbundes sein, da die Existenz dieser Komponente vom Wert der Diskriminante abhängt.

Innerhalb eines generischen Rumpfes können die formalen generischen Parameter relativ frei verwendet werden. Allerdings gibt es verschiedene Stellen, an denen ein Ausdruck statisch sein muß, wie z.B. in den Alternativen einer case-Anweisung oder Variante, im Wertebereich einer Typdefinition usw. In all diesen Situationen kann kein generischer Ausdruck verwendet werden, da der Ausdruck nicht statisch ist.

2.15.2 Typen als Parameter

Im vorangegangenen Abschnitt traten Typen als generische Parameter auf. Sie waren von der Form

```
type T is private;
```

Innerhalb eines generischen Unterprogramms oder Pakets kann man in diesem Fall annehmen, daß die Zuweisung und die Gleichheit definiert sind. T verhält sich in der generischen Einheit fast wie ein privater Typ außerhalb des Paketes, das ihn definiert. Diese Analogie erklärt die Schreibweise für den Formalparameter. Der entsprechende aktuelle Parameter muß natürlich die Zuweisung und die Gleichheit bereitstellen und kann daher jeder beliebige Typ, außer einem begrenzt privaten, sein.

Ein generischer Formalparameter kann auch andere Formen annehmen.

```
type T is limited private;
```

In diesem Fall sind die Zuweisung und die Gleichheit nicht automatisch vorhanden. Der entsprechende aktuelle Parameter kann jeder beliebige Typ sein.

Jede der obigen Formen könnte Diskriminanten haben

```
type T(...) is private;
```

und der aktuelle Typ muß dann Diskriminanten mit denselben Namen, Untertypen und Vorbesetzungswerten aufweisen.

Der Formalparameter könnte auch einer der folgenden sein:

```
type T is (<>);
type T is range <>;
type T is digits <>;
type T is delta <>;
```

Im ersten Fall muß der aktuelle Parameter ein diskreter Typ sein (Aufzählungstyp, ganzzahliger Typ). In den anderen Fällen muß der aktuelle Parameter ein ganzzahliger Typ, bzw. ein Gleitpunkttyp oder ein Festpunkttyp sein. Innerhalb der generischen Einheit sind die passenden, vordefinierten Operationen und Attribute vorhanden.

```
generic
  type T is (<>);
function NEXT(X: T) return T;

function NEXT(X: T) return T is
begin
  if X=T'LAST then
    return T'FIRST;
  else
    return T'SUCC(X);
  end if;
end NEXT;
```

In diesem Beispiel verlangt der Formalparameter T, daß der aktuelle Parameter ein diskreter Typ ist. Da alle diskreten Typen die Attribute FIRST, LAST und SUCC besitzen, können sie im Rumpf der Funktion verwendet werden.

Anwendungsbeispiele:

```
function TOMORROW is new NEXT(DAY);
```

Dann würde

```
TOMORROW(SUN)=MON
```

liefern.

```
function NEXT_WORKDAY is NEXT(WEEK_DAY);
```

```
function NEXT_DIGIT is new NEXT(INTEGER range 0..9);
```

Ein formaler generischer Parameter kann auch ein Feldtyp sein. Der aktuelle Parameter muß dann auch ein Feldtyp mit demselben Komponententyp und denselben Einschränkungen (soweit es welche gibt), derselben Anzahl von Dimensionen und denselben Indexuntertypen sein. Entweder müssen beide Felder eingeschränkt oder uneingeschränkt sein. Sind sie eingeschränkt, dann müssen die Indexwertebereiche für die Indizes gleich sein. Ein generischer Formalparameter kann von einem vorangegangenen Formalparameter abhängen. Das wird bei Feldern oft der Fall sein. Als Beispiel soll eine Funktion SUM generisch vereinbart werden, die die Elemente vom Typ REAL eines Feldes mit ganzzahligem Index addiert.

```
generic
  type INDEX is (<>);
  type FLOATING is digits <>;
  type VEC is array (INDEX range <>) of FLOATING;
function SUM(A: VEC) return FLOATING

function SUM(A:VEC) return FLOATING is
  RESULT: FLOATING:=0.0;
begin
  for I in A'RANGE loop
    RESULT:=RESULT+A(I);
  end loop;
  return RESULT;
end SUM;
```

Man beachte, daß INDEX nicht ausdrücklich im generischen Rumpf erscheint, obwohl er ein Formalparameter ist. Trotzdem wird er implizit verwendet, da der Schleifenparameter I vom Typ INDEX ist.

Eine Inkarnation könnte erzeugt werden durch

```
function SUM_VECTOR is new SUM(INTEGER,REAL,VECTOR);
```

Die letzte Möglichkeit für einen Formalparameter ist die eines Zugriffstyps (Zeigers). Die allgemeine Form ist

```
type A is access T;
```

wobei T ein vorangegangener Formalparameter sein kann aber nicht sein muß. Der aktuelle Parameter, dem A zugeordnet wird, muß auf T zeigen. Einschränkungen bezüglich des Zugriffstyp müssen auch gleich sein. Abschließend beachte man, daß es kein Konzept für einen formalen Verbundtyp gibt und zwar deshalb, weil die innere Struktur von Verbundtypen wilkürlich ist und die Möglichkeiten für Anpassungen daher gering sind.

2.15.3 Unterprogramme als Parameter

In ADA können Unterprogramme nur Parameter von generischen Einheiten sein. Man könnte sich eine generische Funktion erzeugen

```
generic
  with function F(X: REAL) return REAL;
function INTEGRATE(A,B: REAL) return REAL;
```

die die Funktion

$$\int_a^b f(x)dx$$

berechnet.

Um eine bestimmte Funktion zu integrieren, müssen wir eine Inkarnation von INTEGRATE mit unserer Funktion als aktuellen generischen Parameter erzeugen. Angenommen, wir wollen

$$\int_0^p e^t \sin t \, dt$$

berechnen, dann schreiben wir

```
function G(T: REAL) return REAL is
begin
  return EXP(T)*SIN(T);
end;

function INTEGRATE_G is new INTEGRATE(G);
```

Der Ausdruck `INTEGRATE_G(0.0,p)` liefert dann das Ergebnis.

Man beachte, daß ein Unterprogramm als Formalparameter durch **with** eingeleitet wird. (Das einleitende **with** ist notwendig, um syntaktische Mehrdeutigkeiten zu vermeiden). Das entsprechende Unterprogramm, das als aktueller Parameter bereitgestellt wird, muß Parameter und ein Resultat (wenn es eines gibt) desselben Typs, derselben Übergabeart und mit denselben Einschränkungen aufweisen. Auf die Parameternamen und die Vorbesetzungswerte kommt es nicht an. Die Spezifikation des formalen Unterprogramms kann von vorangehenden formalen Typen abhängen. Alle Ausdrücke, die im generischen Teil vorkommen (in einer Einschränkung oder als Vorbesetzungswert) werden soweit als möglich zum Zeitpunkt der Vereinbarung der generischen Einheit berechnet. Ein Ausdruck könnte jedoch Bezugnahmen auf einen Typ enthalten, der selbst ein generischer Parameter ist. In diesem Fall muß ein Teil der Berechnungen bis zum Zeitpunkt der Inkarnation verschoben werden. Ausdrücke in der Spezifikation der generischen Einheit werden immer bei der Inkarnation berechnet, als ob das Paket oder Unterprogramm direkt vereinbart worden wäre.

2.16 Prozesse

2.16.1 Parallelität

Bis jetzt wurden nur sequentielle Programme betrachtet. Bei vielen Anwendungen ist es jedoch sinnvoller, ein Programm aus mehreren parallel ablaufenden Aktivitäten aufzubauen, die je nach Bedarf zusammenarbeiten. Das gilt besonders für Programme, die zeitabhängige physikalische Prozesse in der realen Welt beeinflussen. In ADA werden parallele Aktivitäten mit Hilfe von Prozessen beschrieben. In einfachen Fällen ist die syntaktische Form eines Prozesses der eines Paketes sehr ähnlich. Sie besteht aus einer Spezifikation, die die Schnittstelle zu anderen Prozessen beschreibt, und einem Rumpf, der das dynamische Verhalten des Prozesses bestimmt.

```
task T is               -- specification
  ...
end T;

task body T is          -- body of the task
  ...
end T;
```

In einigen Fällen gibt es keine Schnittstelle zu anderen Prozessen, wodurch sich die Spezifikation auf

```
task T;
```

reduziert.

Als einfaches Beispiel für Parallelität soll uns hier eine Famile, die zum Einkaufen geht dienen. Bisher konnte das Einkaufen nur durch eine Prozedur (SHOPPING) realisiert werden.

```
procedure SHOPPING is
begin
  BUY_MEAT;
  BUY_SALAD;
  BUY_WINE;
end;
```

Der Nachteil ist die sequentielle Ausführung der einzelnen Einkaufsschritte BUY_MEAT, BUY_SALAD und BUY_WINE. Viel effizenter wäre es wenn sich die Familie aufteilt und jeder einen bestimmten Gegenstand kauft, so daß z.B.

die Mutter das Fleisch kauft, die Kinder den Salat und der Vater den Wein. Diese parallele Lösung kann erreicht werden duch

```
procedure SHOPPING is
task GET_SALAD;
task body GET_SALAD is
begin
  BUY_SALAD;
end GET_SALAD;

task GET_WINE;
task body GET_WINE is
begin
  BUY_WINE;
end GET_WINE;

begin
  BUY_MEAT;
end SHOPPING;
```

Bei dieser Lösung wird die Mutter als Hauptprozeß dargestellt. Sie ruft BUY_MEAT von der Prozedur SHOPPING auf. Die Kinder und der Vater werden als untergeordnete Prozesse betrachtet und führen die lokal vereinbarten Prozesse GET_SALAD und GET_WINE aus, die wiederum BUY_SALAD und BUY_WINE aufrufen.

Das Beispiel veranschaulicht die Vereinbarung, Aktivierung und Beendigung von Prozessen. Ein Prozeß ist ein Programmteil wie ein Paket und wird auf ähnliche Art und Weise in einem Unterprogramm, einem Block, einem Paket oder einem anderen Prozeßrumpf vereinbart. Eine Prozeßspezifikation kann auch in einer Paketspezifikation vereinbart werden, wobei dann der Prozeßrumpf im entsprechenden Paketrumpf vereinbart werden muß. Eine Prozeßspezifikation kann jedoch nicht in der Spezifikation eines anderen Prozesses vereinbart werden, sondern nur in dessen Rumpf. Die Aktivierung eines Prozesses geschieht automatisch. Im obigen Beispiel werden die lokalen Prozesse aktiv, wenn der Kontrollfluß in der umfassenden Programmeinheit (Stammeinheit) das **begin** nach der Prozeßvereinbarung erreicht. Ein Prozeß wird beendet, sobald er sein abschließendes **end** erreicht. Der Prozeß GET_SALAD ruft daher nur die Prozedur BUY_SALAD auf und wird dann sofort beendet. Ein Prozeß, der im Vereinbarungsteil eines Unterprogramms, eines Blocks oder eines Prozeßrumpfs vereinbart wird, ist von dieser Einheit abhängig. Es gilt die Regel, daß eine Einheit nicht verlassen werden kann, bis alle abhängigen Prozesse beendet sind. Diese Bedingungsregel garantiert, daß Objekte, die in der Einheit vereinbart wurden und daher für lokale Prozesse möglicherweise sichtbar sind, nicht verschwinden können, solange noch ein Prozeß existiert, der auf sie zugreifen könnte. (man beachte, daß ein Prozeß nicht von einem Paket abhängen kann; später mehr dazu.)

Man nimmt an, daß das Hauptprogramm (Hauptprozedur) selbst von einem hypothetischen Hauptprozeß aufgerufen wird. Wir können nun die Aktivitäten der Reihe nach verfolgen, wenn dieser Hauptprozeß die Prozedur SHOPPING aufruft. Zuerst werden die Prozesse GET_SALAD und GET_WINE vereinbart. Wenn der Hauptprozeß das **begin** erreicht, werden diese abhängigen Prozesse parallel zum Hauptprozeß aktiviert. Die abhängigen Prozesse rufen ihre Prozeduren auf und werden beendet. In der Zwischenzeit ruft der Hauptprozeß BUY_MEAT auf und erreicht das **end** von SHOPPING. Der Hauptprozeß wartet nun bis die abhängigen Prozesse beendet sind, wenn sie es bis dahin nicht schon sind. Das entspricht der Mutter, die darauf wartet, daß der Vater und die Kinder mit ihren eingekauften Waren zurückkommen.

2.16.2 Das Rendezvous

Im Beispiel SHOPPING beeinflußten sich die verschiedenen Prozesse nach ihrer Aktivierung nicht. Im allgemeinen ist es aber so, daß sich Prozesse gegenseitig beeinflussen, solange sie aktiv sind. Bei ADA geschieht das mit Hilfe des Rendezvous-Konzepts. Das ist ähnlich wie bei den Menschen, wenn sich zwei Leute treffen, etwas gemeinsam tun und dann unabhängig voneinander weitergehen. Ein Rendezvous zwischen zwei Prozessen läuft ab, wenn ein Prozeß einen Eingang aufruft, der in einem anderen Prozeß vereinbart ist. Ein Eingang wird in der Prozeßspezifikation vereinbart, ähnlich einer Prozedur in einer Paketspezifikation.

```
task T is
  entry E(...);
end;
```

Ein Eingang kann in-,out- oder in out-Parameter wie eine Prozedur besitzen. Im Gegensatz zur Funktion kann er keinen Ergebniswert zurückliefern. Ein Eingang wird ähnlich wie eine Prozedur aufgerufen.

```
T.E(...);
```

Ein Prozeßname kann nicht in einem use-Satz vorkommen, und deshalb ist die punktierte Schreibweise notwendig, um den Eingang außerhalb des Prozesses aufzurufen. Ein lokaler Prozeß kann einen Eingang seiner Stammeinheit direkt aufrufen. Die Anweisungen, die während eines Rendezvous ausgeführt werden sollen, werden durch eine entsprechende accept-Anweisung im Rumpf des Prozesses beschrieben, der die Vereinbarung des Eingangs enthält. Eine accept-Anweisung hat gewöhnlich die Form

```
accept E(...) do
  -- sequence of statements
end E;
```

Die Formalparameter des Eingangs E werden in der accept-Anweisung wiederholt. Nach dem **end** kann der Name des Eingangs stehen oder nicht. Ein signifikanter Unterschied zum bisher üblichen liegt darin, daß der Rumpf der accept-Anweisung nur eine Folge von Anweisungen ist. Für lokale Vereinbarungen oder die Behandlung von Ausnahmebedingungen muß ein lokaler Block geschrieben werden.

Ein Prozeß ruft einen Eingang in einem anderen Prozeß auf. Die entsprechende accept-Anweisung wird aber von dem Prozeß ausgeführt, der den Eingang besitzt. Außerdem kann die accept-Anweisung so lange nicht ausgeführt werden, bis ein Prozeß den Eingang aufruft und der Prozeß, der den Eingang besitzt, die accept-Anweisung erreicht. Natürlich wird eines von beiden zuerst stattfinden, und der betreffende Prozeß wird dann einstweilen aufgeschoben, bis der andere seine entsprechende Anweisung erreicht. Sobald dies eintritt, wird die Anweisungsfolge der accept-Anweisung vom aufgerufenen Prozeß ausgeführt, während der aufrufende Prozeß aufgeschoben bleibt. Diese Wechselwirkung wird Rendezvous genannt.

Wenn das Ende der accept-Anweisung erreicht wird, ist das Rendezvous abgeschlossen, und beide Prozesse laufen unabhängig voneinander weiter. Der Parameterübergabemechanismus entspricht genau dem eines Unterprogrammaufrufs. Man beachte, daß Ausdrücke in der aktuellen Parameterliste berechnet werden, bevor der Aufruf durchgeführt wird.

Man könnte das Einkaufsbeispiel nun wie folgt weiterentwickeln. Der Prozeß GET_SALAD erhält zwei Eingänge. Einen, damit die Mutter den Kindern das Geld für den Salat geben kann, und einen, um ihnen nachher den Salat wieder abzunehmen. Das Analoge geschieht für GET_WINE. Die Prozeduren BUY_SALAD, BUY_WINE und BUY_MEAT erhalten den Parameter Geld.

```
procedure SHOPPING is

task GET_SALAD is
  entry PAY(M: in MONEY);
  entry COLLECT(S: out SALAD);
end GET_SALAD;

task body GET_SALAD is
  CASH: MONEY;
  FOOD: SALAD;
begin
  accept PAY(M: in MONEY) do
    CASH:=M;
  end PAY;

  FOOD:=BUY_SALAD(CASH);

  accept COLLECT(S: out SALAD) do
    s:=FOOD;
  end COLLECT;
```

```
end GET_SALAD;

-- GET_WINE similarly

begin
  GET_SALAD.PAY(50);
  GET_WINE.PAY(100);
  MM:=BUY_MEAT(200);
  GET_SALAD.COLLECT(SS);
  GET_WINE.COLLECT(WW);
end;
```

Sobald die Prozesse GET_SALAD und GET_WINE aktiv werden, treffen sie auf die accept-Anweisungen und warten, bis der Hauptprozeß ihre Eingänge PAY aufruft. Nach dem Aufruf der Funktion BUY_MEAT, ruft der Hauptprozeß die COLLECT-Eingänge auf.

Als weiteres Beispiel betrachten wir das Produzenten - Konsumenten Problem. Dabei ist ein Prozeß bereitzustellen, der als einziger Puffer fungiert zwischen einem oder mehreren Prozessen, die Elemente produzieren und einem oder mehreren Prozesse, die sie konsumieren.

```
task BUFFERING is
  entry PUT(X: in ITEM);
  entry GET(X: out ITEM);
end;

task body BUFFERING is
  V: ITEM;
begin
  loop
    accept PUT(X: in ITEM) do
      V:=X;
    end PUT;
    accept GET(X: out ITEM) do
      X:=V;
    end GET;
  end loop;
end BUFFERING;
```

Andere Prozesse können nun Elemente produzieren bzw. konsumieren, indem sie `BUFFERING.PUT(...);` bzw. `BUFFERING.GET(...);` aufrufen. Der Zwischenspeicher für das Element ist die Variable V. Der Prozeßrumpf ist eine Endlosschleife, die eine accept-Anweisung für PUT und eine GET enthält. Der Prozeß nimmt daher abwechselnd Aufrufe von PUT und GET an, die die Variable V füllen und entleeren. Jedem Eingang ist eine Warteschlange von Prozessen zugeordnet, die darauf warten, vom Eingang bedient zu werden; diese Warteschlange wird nach der first in first out Regel behandelt. Die Anzahl der Prozesse in der Warteschlange des Eingangs E wird durch das Attribut E'COUNT bereitgestellt. Es gilt die Einschränkung, daß dieses Attribut nur innerhalb des Prozeßrumpfes des Prozesses, der den Eingang besitzt, verwendet werden kann. Ein Eingang kann mehrere entsprechende accept-Anweisungen haben (gewöhnlich aber nur eine).Jede Ausführung einer accept-Anweisung entfernt einen Prozeß aus der Warteschlange. Ein Eingang muß keine Paramter haben, wie bespielsweise

```
entry SIGNAL
```

Er kann dann durch

```
T.SIGNAL;
```

aufgerufen werden. Eine accept-Anweisung braucht keinen Rumpf zu haben, wie beispielsweise

```
accept SIGNAL;
```

In diesem Fall ist der Zweck des Aufrufs nur der, eine Synchronisation zu erreichen und nicht Informationen weiterzugeben.

Es gibt keine besonderen Einschränkungen für die Anweisungen in einer accept-Anweisung. Sie können Eingangsaufrufe, Unterprogrammaufrufe, Blöcke und weitere accept-Anweisungen enthalten (möglicherweise sogar für denselben Eingang). Andererseits kann eine accept-Anweisung nicht in einem Unterprogrammaufruf vorkommen, sondern muß in der Anweisungsfolge des Prozesses enthalten sein. Die Ausführung einer return-Anweisung in einer accept-Anweisung entspricht dem Erreichen des abschließenden end und beendet daher das Rendezvous.

Ein Prozeß kann einen seiner eigenen Eingänge aufrufen, bleibt aber natürlich stecken (Deadlock). Vor einem Deadlock sollte der Compiler warnen.

2.16.3 Zeitüberwachung und Prozeßsteuerung

Ein ADA- Programm kann mehrere Prozesse enthalten. Zum besseren Verständnis sollte man sich die Prozesse so vorstellen, daß jeder seinen eigenen Prozessor hat, so daß ein Prozeß nicht auf andere Prozesse in der Ausführung warten muß. In der Praxis werden die meisten Implementierungen nicht jedem Prozeß einen eigenen Prozessor zuweisen können. Meist hat der Rechner nur einen Prozessor. Es wird dann notwendig sein, dem Prozessor(en) mit Hilfe eines Algorithmus zur Prozeßsteuerung jene Prozesse zuzuweisen, die gerade ausgeführt werden können. Dies kann auf unterschiedliche Art realisiert werden. Ein einfacher Mechanismus ist der Zeitscheiben-Mechanismus. In diesem Fall wird jedem Prozeß eine feste Rechenzeit gewährt. Auch denen, die zur Zeit nicht rechnen können, weil sie z.B. auf ein Rendezvous warten. Es wird Rechenzeit verschwendet. In der Realität wird man intelligentere Lösungen finden, die z.B. die Rechenzeit eines wartenden Prozesses an den nächsten zuteilen. Weiter ist es oft der Fall, daß Prozesse unterschiedliche Prioritäten besitzen müssen, da sie dringender sind als andere.

In ADA ist es erlaubt, die verschiedensten Prozeßsteuerungsstrategien zu verwenden. Wenn eine Implemetierung das Prinzip der Prioritäten verwendet, dann kann die Priorität durch ein Pragma angezeigt werden, das irgendwo in der Prozeßspezifikation steht.

```
task BUFFERING is
  pragma PRIORITY(7);
  entry PUT...
  ...
end;
```

Für das Hauptprogramm befindet sich das Pragma im äußersten Vereinbarungsteil. Die Priorität muß ein statischer Ausdruck vom Untertyp PRIORITY des Typs INTEGER sein. Der tatsächliche Wertebereich des Untertyps PRIORITY hängt jedoch von der Implementierung ab. Der Wertebereich läßt sich mit den Attributen PRIORITY'FIRST und PRIORITY'LAST feststellen.

Man beachte, daß die Priorität eines Prozesses statisch ist und daher während der Ausführung eines Programmes nicht geändert werden kann. Eine größere Priorität zeigt einen höheren Grad an Dringlichkeit an. Mehrere Prozesse können dieselbe Priorität aufweisen. Ein Prozeß muß aber die Priorität nicht explizit angeben. Die Wirkung von Prioritäten auf die Prozeßsteuerung in ADA wird durch folgende Regel aus dem LRM beschrieben.

Wenn zwei Prozesse mit verschiedenen Prioritäten für eine Ausführung in Frage kommen und dieselben Verarbeitungsresourcen verwenden, dann darf es nicht passieren, daß der Prozeß mit der niedrigeren Prioität ausgeführt wird und der Prozeß mit der höheren Priorität warten muß.

Bei einem Rendezvous kommt es bei der Bestimmung der Prioritäten zu einem Konflikt, da zwei Prozesse beteiligt sind. Weisen beide Prozesse eine explizite Definition der Priorität auf, so wird das Rendezvous mit der höheren Priorität durchgeführt. Hat nur ein Prozeß eine expizite Definition der Priorität, dann wird das Rendezvous mit dieser Priorität ausgeführt. Hat kein Prozeß eine definierte Priorität, so ist die Priorität des Rendevous undefiniert.

Die Rendezvous-Regeln garantieren, daß ein Prozeß mit einer hohen Priorität nicht durch einen Prozeß mit einer niedrigeren Priorität durch ein Rendezvous aufgehalten wird. Die Warteschlange ist von den Prioritäten unbeeinflußt und arbeitet wie bisher nach der "first in first out"-Methode.

Bei der Verwendung von Prioritäten sollte sorgfältig vorgegangen werden. Sie sind als Mittel zur Regulierung der Dringlichkeit einer Aufgabe gedacht und sollten nicht zur Synchronisation verwendet werden.

Ein Prozeß kann aus verschiedenen Gründen aufgehalten werden. Dies erreicht man mit der delay-Anweisung z.B.

```
delay 3.0;
```

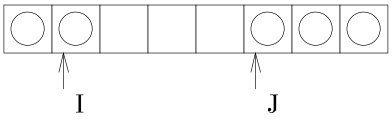
Diese Anweisung verzögert den Prozeß für zumindest drei Sekunden. Der Ausdruck nach dem reservierten Wort delay gehört zum vordefinierten Festpunkttyp DURATION und gibt die Zeit in Sekunden an. Eine delay-Anweisung mit Null oder einer negativen Zahl als Argument hat keine Wirkung. Obwohl der Typ DURATION von der Implementierung abhängt, wird garantiert, daß er uns Zeitintervalle von mindestens einem Tag (86400 s) zur Verfügung stellt.

2.16.4 Select-Anweisung

Die Select-Anweisung soll an Hand eines Beispiels erklärt werden. Es soll ein Ringpuffer mit n Elementen erstellt werden.

```
task BUFFERING is
  entry PUT(X: in ITEM);
  entry GET(X: out ITEM);
end;

task body BUFFERING is
  N: constant:=8;
  A: array (1..N) of ITEM;
  I,J: INTEGER range 1..N:=1;
  COUNT: INTEGER range 1..N:=0;
begin
  loop
    select
      when COUNT < N =>
      accept PUT(X: in ITEM) do
        A(I):=X;
      end;
      I:=I mod N+1; COUNT:=COUNT+1;
    or
      when COUNT > 0 =>
      accept GET(X: out ITEM) do
        X:=A(J);
      end;
      J:=J mod N+1; COUNT:=COUNT-1;
    end select;
  end loop;
end BUFFERING;
```

Der Puffer wird durch das Feld der Länge N realisiert. Die Variablen I und J zeigen auf das nächste freie, bzw. auf das zuletzt verwendete Pufferelement, COUNT ist die Anzahl der vollen Pufferelemente.



Ziel dieses Prozesses ist es, Elemente nach der "first in first out" - Regel dem Puffer hinzuzufügen oder vom Puffer zu entfernen. Es soll verhindert werden, daß der Puffer überläuft oder vollständig geleert wird. Das wird mit einer allgemeineren Form der Select-Anweisung erreicht, die die Verwendung von Schutzbedingungen (guarding conditions) einschließt.

Eine select-Anweisung beginnt mit dem reservierten Wort **select** und endet mit **end select**. Sie enthält zwei oder mehr Alternativen, die durch **or** getrennt werden. In diesem Beispiel besteht jede Alternative aus einer accept-Anweisung und zwei Zuweisungen. Bei einer select-Anweisung gibt es verschiedene Möglichkeiten der Ausführung.

- Wenn keine der select-Anweisungen ausgeführt werden kann, wird der Prozeß angehalten, bis eine select-Anweisung ausgeführt werden kann.
- Liegt nur ein Aufruf oder mehrere Aufrufe für eine der select-Anweisungen vor, dann wird nur diese select-Anweisung ausgeführt.
- Liegen für verschiedene select-Anweisungen Anforderungen vor, so wird willkürlich, nicht deterministisch, um das Gleichgewicht zwischen den select-Zweigen aufrecht zuerhalten, eine select-Anweisung ausgewählt.

In unserem Beispiel werden noch Schutzbedingungen verwendet. Damit beginnt jede select-Anwesung mit

```
when condition =>
```

Darauf folgen accept-Anweisung und weitere Anweisungen. Das Verhalten ist wie bei der oben beschriebenen select-Anweisung ohne Schutzvorkehrungen. Es werden nur die Zweige berücksichtigt, für die die Bedingungen wahr sind. Ein solcher Zweig wird ausgewählt und das entsprechende Rendezvous durchgeführt. Nach der Anweisung kann ein Zweig weitere Anweisungen enthalten. Diese werden als Teil der select-Anweisung, aber außerhalb des Rendezvous ausgeführt. Die Schutzbedingungen sind also Bedingungen, die wahr sein müssen, bevor eine Dienstleistung angeboten werden kann.

In unserem Beispiel kann PUT nur akzeptiert werden, wenn COUNT $<$ N ist. Ebenso kann GET nur akzeptiert werden, wenn COUNT $>$ 0 ist. Nachdem das Rendezvous abgeschlossen ist, werden I bzw. J und COUNT auf den neuesten Stand gebracht. Eine nicht vorhandene Sicherheitsbedingung wird als wahr angenommen. Wenn sich alle Sicherheitsbedingungen als falsch herausstellen, wird die Ausnahmebedingung SELECT_ERROR aktiviert. Eine Sicherheitsbedingung muß nach der Durchführung eines Rendezvous nicht mehr richtig sein, weil globale Variablen verändert worden sein könnten. In unserem Beispiel geht alles gut und SELECT_ERROR kann nie aktiviert werden.

Eine select-Anweisung kann verschiedene Formen annehmen. Ein oder mehrere Zweige können mit einer delay-Anweisung beginnen und nicht mit einer accept-Anweisung.

```
select
  accept A(...) do
  ...
  end;
or
  accept B(...) do
  ...
  end;
or
  delay 10.0 * MINUTES;
  -- time out statements
end select;
```

Wenn innerhalb von 10 Minuten weder ein Aufruf von A noch von B empfangen wird, dann wird der dritte Zweig gewählt und die Anweisungen, nach der delay-Anweisung werden durchgeführt. Gibt es mehrere delay-Anweisungen wird die kürzeste gewählt. Man sollte beachten, daß jede delay-Anweisung abgebrochen wird, sobald eine der accept-Anweisungen befolgt wird. Die delay-Anweisung wird in diesem Zusammenhang häufig zur Zeitüberwachung eingesetzt.

Eine weitere Form der select-Anweisung ist mit einem else-Zweig.

```
select
  accept A(...) do
  ...
  end;
or
  accept B(...) do
  ...
  end;
else
  -- alternative statements
end select;
```

In diesem Fall steht vor dem letzten Zweig else statt or, und dieser besteht nur aus Anweisungen. Der else-Zweig wird sofort gewählt, wenn keiner der anderen Zweige unmittelbar akzeptiert werden kann. Eine select-Anweisung mit einem else-Zweig ist ähnlich einer Anweisung mit einem Zweig, der mit delay 0.0; beginnt. Er bricht sofort ab, wenn keine Kunden zum Bedienen da sind. Eine select-Anweisung kann nicht einen else- Zweig und delay-Alternativen besitzen. Wenn eine select-Anweisung einen else-Zweig besitzt, kann nie SELECT_ERROR aktiviert werden. Der else-Zweig hat keine Sicherheitsbedingung und kann deshalb immer ausgeführt werden.

Es gibt zwei weitere Formen der select-Anweisung, die sich von den bisher besprochenen ziemlich unterscheiden. Sie betreffen einen einzelnen Eingangsaufruf und nicht eine oder mehrere accept-Anweisungen.

Der Eingangsaufruf bei Zeitüberschreitung bewirkt,daß eine Anweisungsfolge als Alternative zu einem Eingangsaufruf ausgeführt wird, wenn dieser nicht innerhlab der festgelegten Zeitdauer akzeptiert wird.

```
select
  A.PUT(...);
or
  delay 1*MINUTE;
  -- sequence of statements
end select;
```

Im nächsten Fall wird sofort die Alternative gewählt, wenn der Aufruf nicht sofort akzeptiert wird (bedingter Eingangsaufruf).

```
select
  A.PUT(...);
else
  -- sequence of statements
end select;
```

Bedingte Aufrufe und Aufrufe bei Zeitüberschreitung sind nützlich, wenn ein Prozeß nicht übermäßig verzögert werden soll. Abschließend sei noch erwähnt, daß Zeitüberschreitungen besonders sorgfältig behandelt werden müssen, wenn das Attribut COUNT verwendet wird. Eine Entscheidung, die auf dem Attribut COUNT beruht, kann ungültig sein, weil ein Aufruf bei Zeitüberschreitung unvermutet einen Prozeß aus einer Eingangswarteschlange entfernen könnte.

2.16.5 Prozeßtypen

Manchmal ist es nützlich, Prozesse mit ähnlichen Eigenschaften zu haben. Es kann nicht vorhergesagt werden, wieviele dieser Prozesse benötigt werden. Diese Prozesse müssen auf dynamische Art und Weise geschaffen und eingesetzt werden. Eine Schablone für Prozesse mit ähnlichen Eigenschaften wird durch eine Prozeßtypvereinbarung bereitgestellt. Diese sieht aus wie eine Prozeßvereinbarung, nur daß in der Spezifikation nach dem reservierten Wort **task** das Wort **type** steht.

```
task type T is
  entry E(...):
end T;

task body T is
  ...
end T;
```

Für den Prozeßrumpf gelten die bisherigen Regeln. Um einen aktuellen Prozeß zu erzeugen, verwendet man die normale Form einer Objektvereinbarung.

```
X: T;
```

Dadurch wird ein Prozeß X des Typs T vereinbart.

Prozeßobjekte können in allen Strukturen wie gewöhnlich verwendet werden. Man könnte z.B. Felder, bestehend aus Prozessen, vereinbaren.

```
AT: array (1..10) of T;
```

Ebenso Verbunde, die Prozesse enthalten

```
type REC is
  record
    CT: T;
    ...
  end record;

R: REC;
```

Die Eingänge solcher Prozesse werden aufgerufen, indem man den Namen des Prozeßobjektes angibt.

```
X.E(...);
AT(I).E(...);
R.CT.E(...);
```

Prozeßobjekte sind keine Variablen, sondern verhalten sich wie Konstanten. Eine Prozeßvereinbarung erzeugt einen Prozeß, der permanent an ein Objekt gebunden ist. Daher ist für Prozeßtypen weder eine Zuweisung noch ein Vergleich erlaubt. Ein Prozeßtyp ist daher ein begrenzt privater Typ und kann auch wie dieser eingesetzt werden.

Unterprogrammparameter können Prozeßtypen sein. Die Übergabetechnik ist immer "by reference"; d.h. die Parameter werden nie kopiert, sondern die formalen und aktuellen Parameter beziehen sich immer auf denselben Prozeß. Ein Prozeß, der als Prozeßobjekt vereinbart wird, hängt von dem umschließenden Block, Unterprogramm oder Prozeßrumpf ab, in dem er vereinbart ist (siehe auch Abhängigkeit von Prozessen weiter vorn). Die Aktvierung eines Prozesses kann man sich als zweiphasigen Vorgang vorstellen. Zuerst werden die Vereinbarungen des Prozeßrumpfes abgearbeitet, währenddessen darf die Stammeinheit nicht weiter ablaufen (Behandlung von Ausnahmebedingungen sonst nicht geklärt). Erst wenn die Abarbeitung der Vereinbarungen vollendet ist, fängt der neue Prozeß parallel zum vereinbarenden Prozeß zu laufen an.

Prozeßobjekte können in einem Paket vereinbart werden, und obwohl sie nicht vom Paket abhängig sind, werden sie dennoch am Anfang (begin) des Paketrumpfes aktiviert. Besitzt der Paketrumpf keine Initialisierungsanweisung (kein begin) wird eine leere Initialisierungsanweisung angenommen. Hat ein Paket keinen Rumpf, wird ein Rumpf mit einer leeren Initialisierungsanweisung angenommen.

Prozesse können auch durch Zugriffstypen (Zeiger) erzeugt werden.

```
type REF_T is access T;
```

Der Prozeß wird erzeugt indem man einen Objektkonstruktor (Allokator) auf die übliche Art verwendet.

```
RX: REF_T:=new T;
```

Der REF_T ist ein normaler Zugriffstyp. Daher sind Zuweisungen und Vergleiche zwischen den Objekten des Typs erlaubt. Der Eingang E des Prozesses kann wie üblich aufgerufen werden.

```
RX.E(...);
```

Prozesse, die durch Zugriffstypen erzeugt werden, folgen hinsichtlich ihrer Aktivierung und Abhängigkeiten etwas anderen Regeln. Sie werden gleichzeit mit der Abarbeitung des Allokators aktiviert. Weiter sind solche Prozesse nicht von der Einheit abhängig, in der sie erzeugt werden, sondern von dem Block, Unterprogrammrumpf oder Prozeßrumpf, der die Vereinbarung des Zugriffstyps selbst enthält. Wenn ein Prozeß eine Komponente eines Objekts ist, dann sind die Aktivierungs- und Abhängigkeitsregeln so, als ob der Prozeß das ganze Objekt wäre. Das gilt sowohl für direkt vereinbarte Objekte als auch für Objekte, die durch Zugriffstypen erzeugt wurden.

2.16.6 Beendigung von Prozessen und Ausnahmebedingungen

Ein Prozeß kann auf verschiedene Arten enden, beispielsweise, indem er in das abschließende **end** läuft. Häufig bestehen Prozeßrümpfe aber aus Endlosschleifen; d.h. der Prozeß endet nie auf natürliche Art. Auch kann die Einheit, von der der Prozeß abhängt, nie verlassen werden. Es ist möglich, daß sich ein Prozeß automatisch mit Hilfe einer besonderen Form der select-Anweisung selbst beendet.

```
select
  accept A(...) do
  ...
  end;
or
  accept B(...) do
  ...
  end;
or
  terminate;
end select;
```

Die terminate-Alternative wird dann durchgeführt, wenn die Einheit, von der der Prozeß abhängt, ihr Ende erreicht hat und alle abhängigen Prozesse beendet sind oder eine terminate-Alternative enthalten. In der Praxis bedeutet das, daß alle Prozesse, die Dienstleistungen erbringen, eine terminate-Alternative enthalten sollen, um sich selbst beenden zu können. Eine terminate-Alternative kann eine Sicherheitsbedingung enthalten. Sie kann aber nicht in einer select-Anweisung mit einer delay-Alternative oder einem else-Zweig auftreten. Die Durchführung einer terminate-Alternative kann als eine normale Beendigung klassifiziert werden; der Prozeß ist unter Kontrolle und endet freiwillig. Eine extreme Lösung ist es, einen Prozeß sofort und bedingungslos mit der abort-Anweisung zu beenden.

```
abort X,AT(3),RX.all;
```

Wenn ein Prozeß abgebrochen wird, dann werden alle Prozesse, die von ihm abhängen, oder Unterprogramme und Blöcke, die von ihm gerade aufgerufen werden, ebenfalls sofort beendet. Wenn sich der Prozeß in einer Warteschlange befindet, dann wird er aus dieser entfernt. Ist der Prozeß gerade mit einem Rendezvous beschäftigt, dann müssen die Folgen für den Partnerprozeß einkalkuliert werden. Wird der Aufrufer abgebrochen, dann ist der aufgerufene Prozeß nicht berührt und das Rendezvous wird vollständig durchgeführt. Wenn dagegen der aufgerufene Prozeß sofort beendet wird, dann wird dem aufrufenden Prozeß die Ausnahmebedingung TASKING_ERROR zugestellt. Die abort-Anweisung hat eine sehr einschneidende Wirkung und sollte deshalb nur in außergewöhnlichen Situationen verwendet werden.

Eine etwas weniger extreme Möglichkeit ist die Aktivierung der speziellen Ausnahmebedingung FAILURE, die Attribut jedes Prozesses ist.

```
raise T'FAILURE;
```

Diese Ausnahmebedingung verhält sich wie andere, hat aber Vorrang gegenüber diesen. Die Ausnahmebedingung FAILURE kann auf übliche Art und Weise durch **others** behandelt werden oder durch

```
when T'FAILURE =>    -- nur im Prozessrumpf
```

Wenn die Ausnahmebedingung FAILURE aktiviert wird und der Prozeß verzögert oder aufgeschoben ist, wird die Verzögerung, bzw. der Aufschub aufgehoben, so daß die Ausnahmebedingung sofort behandelt werden kann. Befindet sich der Prozeß in einer Eingangswarteschlange, wird er aus dieser entfernt. Befindet sich der Prozeß als aufrufender Partner gerade in einem Rendezvous, dann wird das Rendezvous zuendegeführt, bevor er eine Behandlung der Ausnahmebedingung sucht. Ist der Prozeß der aufgerufene Partner, dann wird dem aufrufenden Prozeß die Ausnahmebedingung TASKING_ERROR übergeben.

Ein wichtiger Unterschied zwischen sofortiger Beendigung und Verwendung von Ausnahmebedingungen ist, daß im letzteren Fall abhängige Prozesse nicht betroffen sind. Eine Ausnahmebedingung wird aus einer Einheit nicht verbreitet, solange nicht alle abhängigen Prozesse beendet sind. Wird eine Ausnahmebedingung überhaupt nicht durch einen Prozeß behandelt, dann wird der Prozeß sofort beendet (vgl. Hauptprogramm) und die Ausnahmebedingung geht verloren. Sie wird nicht zur Stammeinheit weitergegeben, weil sie dort zerstörend wirken würde. Beim Versuch, den Eingang eines beendeten Prozesses aufzurufen, wird TASKING_ERROR beim Aufrufer aktiviert. Weiter wird allen Prozessen, die sich in einer Warteschlange eines Prozesses befinden, TASKING_ERROR übergeben, wenn der betreffende Prozeß beendet wird. Abschließend sei noch das Attribut TERMINATED erwähnt, mit dem abgefragt werden kann, ob ein Prozeß beendet ist.

2.16.7 Vergleich von Prozessen und Paketen

Prozesse und Pakete haben lexikalische Änlichkeiten - beide besitzen Spezifikationen und Rümpfe. Es gibt aber auch viele Unterschiede:

- Ein Prozeß ist eine aktive Konstruktion; ein Paket eine passive.
- Ein Prozeß enthält in seiner Spezifikation nur Eingänge. Eine Paketspezifikation kann alles andere, aber keine Eingänge besitzen. Ein Prozeß kann keinen privaten Teil besitzten.
- Ein Paket kann generisch sein, ein Prozeß nicht. Die Wirkung eines parameterlosen generischen Prozesses kann durch einen Prozeßtyp erreicht werden. Der Prozeß wird häufig durch ein generisches Paket eingekapselt.
- Ein Paket kann einen use-Satz haben, ein Prozeß nicht.
- Ein Paket kann eine Bibliothekseinheit sein, ein Prozeß nicht. Ein Prozeßrumpf jedoch kann eine Untereinheit sein.

Der Hauptunterschied ist der, daß man Pakete als das wichtigste Hilfsmittel für Stukturierungsprozesse betrachten sollte, während Prozesse für die Synchronisation gedacht sind. Typische Untersysteme bestehen daher aus einem (generischen) Paket, das einen oder mehrere Prozesse enthält. Dies hat den Vorteil, daß man die vollständige Kontrolle über die zur Verfügung stehenden Hilfsmittel ausübt.

3. Die Programmierumgebung Ada World

3.1 Die Programmentwicklung mit dem Alsys Ada Compiler

Die Programmentwicklung mit dem ADA-Compiler von Alsys kann in folgende Schritte eingeteilt werden, die dem normalen Design und Spezifikationsprozeß entsprechen:

1. Anlegen einer Programm-Bibliothek mit dem Library Manager und dem Befehl NEW
2. Schreiben des Programmcodes für die verschiedenen Programmeinheiten (units) mit einem Texteditor (vi, elvis,jed oder joe)
3. Kompilieren jeder einzelnen Programmeinheit (unit) mit dem Ada Compiler, um einen entsprechenden Objektcode zu erzeugen
4. Binden (binding) der einzelnen Objekte (objekt units) mit dem Ada Binder, um ein entsprechendes Objektfile zu erzeugen
5. Linken des Objektfiles mit den "Alsys Runtime Executive Programs" und anderen benötigten externen Einheiten, um ein ausführbares Programm zu erzeugen
6. Testen des erzeugten Programms

Enthält das Sourcefile, das dem Compiler in Schritt 3 übergeben wird, eine oder mehrere zu kompilierende Einheiten, dann wird die Kompilierung sequentiell in der angegebenen Reihenfolge durchgeführt. Die zu kompilierenden Einheiten müssen in passender Reihenfolge angegeben werden. Eine Einheit muß kompiliert sein, bevor sich weitere Einheiten auf sie beziehen. Entdeckt der Compillier, daß sich Einheiten auf noch nicht kompilierte Einheiten beziehen, gibt er eine Fehlermeldung aus. Im Normalfall können die Schritte 4 und 5 zusammen ausgeführt werden.

3.1.1 Programmbibliotheken (Program Libraries)

Der Ada Compiler behandelt jede Komplilierung als ein Stück eines größeren Programms. Das bedeutet, daß

- das gesamte Programm, und nicht nur die jeweilige Kompilierung, auf Konsistenz der Deklarationen und den Gebrauch von Unterprogrammen und Daten geprüft wird.
- die mächtigen Sichtbarkeitsregeln von Ada mit Hilfe des use und with-Satzes, die Verwendung von include-Files überflüssig machen.
- größere Module in kleinere Untereinheiten (subunits) aufgeteilt werden können, ohne sich um die Sichtbarkeit kümmern zu müssen.

Um diese mächtigen Möglichkeiten zu implementieren, ist es notwendig, daß der Ada Compiler Informationen in den Programmbibliotheken für zukünftige Kompilierungen und Programmbindungen ablegt. Die Ada Programmbibliothek (Program Library) ist die Basis aller Informationen, die bei der Programmentwicklung benötigt werden. Für jede bekannte Unit enthält die Programmbibliothek folgende Informationen.

- den vollen Namen der Unit
- den original Sourcefile-Namen
- Datum und Zeit der Kompilierung
- Art der Unit (Spezifiktion, Body oder Subunit; Funktion, Prozedur oder Paket; generische Einheit usw.)
- Status der Einheit (kompiliert oder nicht)
- erzeugter Code der Unit
- interne Repräsentation der Deklarationen von Typen, Objekten und Unterprogrammen, die außerhalb der Einheit sichtbar sind
- Abhängigkeiten gegenüber anderen Units (with, generic usw.)

Der Binder benutzt die Abhängigkeitsinformationen in der Programmbibliothek, um die Konsistenz der verschiedenen Einheiten untereinander zu überprüfen und um ein einziges Objektfile später zu generieren. Das bedeutet, daß der gesamte Objektcode, der vom Compiler erzeugt wird, erst in den Programmbibliotheken abgelegt wird, bis dann der Binder ein ausführbares Programm erzeugt.

3.1.2 Multi-Library Environment

Um die Konstruktion größerer Programme und die Zusammenarbeit mehrerer Programmierer zu erleichtern, unterstützt die Ada-Programmierumgebung die Einrichtung verschiedener Programmbibliotheken.

- Der Familiy Manager erzeugt und löscht Application Families
- Der Library Manager erzeugt und löscht Application Libraries in einer Appllication Family
- Der Unit Manager erlaubt es dem Benutzer, Units innerhalb einer Library zu untersuchen und Links zwischen verschiedenen Application Libraries zu erzeugen.

Kopieren und Linken von Units in verschiedenen Bibliotheken

Wie bereits oben erwähnt, kann eine Unit in einer Programmbibliothek erzeugt werden, indem man den Quellcode in die Unit kompiliert. In einer Multi-Library-Environment kann eine Unit auch von einer Bibliothek in eine andere importiert werden ohne die Einheit neu zu übersetzten. Dies kann durch Setzen eines Links oder durch Kopieren geschehen. Das Kopieren und Linken erfolgt mit dem Befehl acquire des Unit Managers. Unter gewissen Umständen erzeugt die Programmierumgebung automatisch Links bei der Kompilierung.

Die Multi-Library-Umgebung ermöglicht es weiter, daß ganze Bibliotheken in derselben Programmfamilie, oder von einer zu einer andern Familie kopiert werden.

Das Kopieren, Linken, und Duplizieren von ganzen Bibliotheken ist ein sehr mächtiges Mittel bei der Programmentwicklung.

3.1.3 Bibliotheksfamilien

Eine Gruppe zusammengehörender Bibliotheken (libraries) wird als Familie (family) bezeichnet. Es gibt zwei verschiedene Bibliothekstypen:

Application Libraries und Installation Libraries.

Dementsprechend gibt es auch zwei verschiedene Familientypen:

Application Families und Installation Families.

Es dürfen bis zu 15 Bibliotheken sich in einer Familie befinden. Application Families werden vom Benutzer für die Entwicklung seiner Programme erzeugt, wogegen Installation Family eine Bibliothek ist, die sich alle Application Families teilen. Die Installation Family wird vom Compiler zur Verfügung gestellt. Sie enthält eine Library, die "Predefind Installation Library", die folgende in Ada vordefinierten Pakete und Unterprogramme, so wie sie im Language Reference Manual beschrieben werden, bereitstellt.

- STANDARD, die Standard Ada Objekte und Typen sowie deren Operationen.
- SYSTEM, die Definition der implementierungsabhängigen Konstanten, Typen und Funktionen
- TEXT_IO, SEQUENTIAL_IO, DIRECT_IO und IO_EXCEPTIONS, die die Ein- und Ausgabemöglichkeiten der Sprache definieren.
- CALENDAR, Funktionen für Zeit und Datum
- UNCHECKED_CONVERSION , erlaubt ungeprüfte Konvertierung von Typen
- UNCHECKED_DEALLOCATION, erlaubt die Freigabe von Speicher
- SYSTEM_ENVIRONMENT, Zugriff auf die UNIX-Ebene
- GENERIC_ELEMENTARY_FUNCTIONS, elementare mathemat. Funktionen
- INTERRUPT_MANAGER, Behandlung von Interrupts

3.1.4 Aktualisierung einer Programmbibliothek durch Kompilierung

Wird eine Unit erfolgreich kompiliert, dann werden die Informationen in der entsprechenden Programmbibliothek aktualisiert. Ist die Kompilierung nicht erfolgreich, dann werden die alten Informationen beibehalten und die Bibliothek nicht aktualisiert.

3.1.5 Abhängigkeiten zwischen Units

In Ada können Abhängigkeiten zwischen Units vereinbart sein. Der Binder benutzt die Abhängigkeitsinformationen in der Programmbibliothek, um festzustellen, welche Units in ein Objektfile eingebunden werden müssen. Sind eingebunde Units wieder von anderen Units abhängig, dann müssen auch diese mit eingebunden werden. Jedesmal wenn eine Unit neu kompiliert wird, müssen alle anderen Units, die von dieser Unit abhängen, ebenfalls neu kompiliert werden. Dies gilt nicht nur für die direkten Abhängigkeiten, sondern auch für die indirekten, d.h. transitiven Abhängigkeiten.

3.1.6 File- und Library-Namen

Die Filenamen in Ada World folgen der UNIX-Konvention.

Folgende Namen können Ada-Sourcecode bezeichnen:

bingo	– ohne Extension
bingo.ada	– ADA Source Code
bingo.ads	– ADA Spezifikation
bingo.adb	– ADA body

Die ADA Kompilierumgebung erzeugt verschiedene Files

name.lis	Ein Listing File des Compilers
name.adl	File, das ein Teil der ADA Programmbibliothek ist. Man sollte diese Datei nicht löschen, kopieren oder umbenennen
name.inf	File, das ein Teil der ADA Programmbibliothek ist. Man sollte diese Datei nicht löschen, kopieren oder umbenennen
name.add	Ein File der ADA-Familiy. Sollte nicht gelöscht, umbenannt oder kopiert werden.
name.o	Objektfile, erzeugt durch den Binder oder UNIX Assembler
name.lnk	Command file für den UNIX Linker
name	Ausführbares Programm
name.map	bind and link maps
name.cui	Tabellen für den AdaProbe-Debugger und -Viewer

Ada Programmbibliotheken werden durch Verzeichnisse angegeben. Das Verzeichnis enthält die kompilierten Einheiten der Programmbibliothek.

3.2 Ada World Command Language

Die Ada World wird mit dem Befehl ada aufgerufen und meldet sich mit dem Prompt Ada.. Die folgenden Kommandos sind von diesem Prompt aus möglich:

compile	ruft den Ada Compiler auf
bind	ruft den Ada Binder auf
family_manager	ruft den Ada Family Manager auf
lib_manager	ruft den Ada Library Manager auf
unit_manager	ruft den Ada Unit Manager auf
Workbench	ruft die Workbench auf, von der aus verschiedene Tools aufgerufen werden können

Familiy Manager, Library Manager und der Unit Manager haben jeweils ihre eigene Umgebung, die durch die Eingabe der Befehle family_manager, lib_manager und unit_manager aufgerufen wird.

Folgende Befehle sind in allen Befehlsebenen verfügbar:

default	setzt, löscht oder zeigt die eingestellten Parameter für die verschiedenen Befehle an
system	führt Befehle auf der Shell-Ebene aus, ohne die aktuelle Befehlsebene zu verlassen
help	Hilfe zu der jeweiligen Befehlsebene.
global	setzt globale Einstellungen für die verschiedenen Befehlsebenen
quit	kehrt zur vorhergehenden Befehlsebene zurück. Auf der untersten Ebene der Ada World kehrt man zum Betriebssystem zurück.

3.2.1 Befehlssyntax

Befehle in Ada World haben dieselbe Synatx wie Prozeduraufrufe in Ada:

in benannter Form:

```
compile (source => bingo.ada, library => /usr/smith/games);
```

oder durch Position:

```
compile (bingo.ada,/usr/smith/games);
```

oder sowohl in benannnter als auch in positioneller Form

```
compile (bingo.ada, library => /usr/smith/games)
```

Wenn positionelle und benannte Parameter gemeinsam verwendet werden, dann müssen wie in Ada alle positionellen Parameter am Anfang der Parameterliste auftreten.

Es können beliebig viele Leerzeichen zwischen den Befehlen eingefügt werden. Ein Befehl kann sich über mehrere Zeilen erstrecken, wenn das letzte Zeichen in der Zeile ein Komma, eine öffnene Klammer oder ein Pfeil => ist. Jede Zeile kann kommentiert werden.

```
compile (source => bingo.ada,          -- source file name
         library => /usr/smith/games   -- program library name
         optiobs =>
           (errors => 25,              -- 25 maximum errors
            checks => stack));         -- only stack overflow
```

Ein Programm kann direkt von der Betriebssystemebene aus kompiliert werden. Zum Beispiel kann der einzelne Befehl

```
ada compile bingo.ada, /usr/smith/games
```

gleichgesetzt werden mit

```
ada
compile (bingo.ada, /usr/smith/games);
quit;
```

Auf der Betriebssystemebene dürfen keine Zeichen benutzt werden, die als meta Zeichen der Shell behandelt werden könnten. Normalerweise wird das > Zeichen von der Shell zur Ausgabeumlegung verwendet. Will man z.B. "=>" auf Shell-Ebene verwenden, müßen die Default-Einstellungen entsprechend geändert werden.

3.2.2 Tastaturabkürzungen

In der Ada World ist es möglich, die eingebenen Befehle auf verschiedene Art und Weise abzukürzen. Als Beispiel sei hier nur die Namensabkürzung vorgestellt. In den meisten Fällen reicht es, die Anfangsbuchstaben eines Befehls oder Parameters einzugeben. File- und Verzeichnisnamen können ohne Anführungszeichen geschrieben werden. Die Angabe in positioneller Form erspart die Angabe des Parameter, dem Werte zugewiesen werden. Sämtliche Kommas können durch Blanks ersetzt werden.

Beispiel:

```
compile (source => "bingo.ada",
         library => "/usr/smith/games",
         options => (errors => 25, level => code),
         display => (text => yes));
```

kann abgekürzt werden durch

```
c (s => "bingo.ada",
   l => "/usr/smith/games",
   o => (e => 25, l => c),
   d => (t => y));
```

Weitere Abkürzungsmethoden findet man im Alsys User's Guide auf Seite 16ff.

3.2.3 Der Default-Befehl

Wie schon erwähnt, kann der default-Befehl in jeder Befehlsebene aufgerufen werden. Es werden die default-Parameter ausgegeben, gesetzt oder gelöscht.

Setzen von Werten

Es können jederzeit die default-Werte während einer Sitzung neu gesetzt werden.

Beispiel:

```
default.compile(level => parse);  -- Set default for level
                                     parameter of compile
                                     command to parse
```

```
default(library => my_lib);       -- set my_lib to be the
                                     default library for the
                                     current level
```

Löschen von Werten

Default-Werte können jederzeit gelöscht werden. Werden sie beim nächsten Aufruf nicht explizit angegeben, fragt sie das System nach.

Beispiel:

```
default.compile (level =>):  -- clear default for level parameter
                                of compile command
default(library =>)          -- clear default for all library
                                defaults
```

Anzeige von Werten

Es können alle default-Parameter mit dem Befehl default ausgegeben werden:

Beispiel:

```
default;           -- display parameter defaults for all
                      commands at current level

default.compile    -- display parameter defaults for
                      compile command
```

3.2.4 Der Quit-Befehl

Mit dem quit-Befehl kehrt man zur vorhergehenden Ebene zurück. Die Default-Einstellungen der jeweiligen Ebene können dabei gespeichert werden. Dazu muß

```
QUIT (DEFAULTS => KEEP,
      HISTORY  => KEEP);
```

gesetzt sein.

3.2.5 Der System-Befehl

Mit dem system-Befehl können UNIX-Befehle ohne die jeweilige Systemebene zu verlassen, ausgeführt werden.

Beispiel:

```
system.ls;

system.less bingo.ada;
```

3.2.6 Der Help-Befehl

Mit dem help-Befehl kann man Hilfe zur aktuellen Befehlsebene erhalten.

Beispiel:

```
help;
```

3.3 Der Compile-Befehl

Mit dem compile-Befehl können Ada-Sourcefiles kompiliert werden. Die Syntax lautet:

```
compile ( source  => source_name,
          library => library_name,
          options => ...
          ...
         );
```

Es würde zu weit führen, sämtliche Optionen und Paramter auszuführen. Die wichtigsten Parameter sind:

source: Der zu kompilierende Quellcode

library: Bibliothek, in die kompiliert wird

Beispiel:

```
compile ( source  => bingo.ada,
          library => /usr/smith/games);
```

Ein Übersetzungsprotokoll wird dabei in die Datei bingo.lis geschrieben. Wird kein Library-Name angegeben, so wird die default-Einstellung verwendet.

3.4 Der Bind-Befehl

Die Eingabe des bind-Befehls besteht aus

- der Ada Programmbibliothek, die die zu bindenden Einheiten enthält
- der Name des Hauptprogramms, bzw. der Hauptprozedur innerhalb der Bibliothek

Der Binder findet aufgrund des Namens der Hauptprozedur und der Informationen in der Programmbibliothek die zu bindenden Einheiten. Die Ausgaben können folgende sein:

- ein Objektfile, das den Code und die Daten aller Units der Library enthält

- ein UNIX-Linker-Command-File
- ein ausführbares Programm, das vom UNIX-Linker erzeugt wurde
- eine Informationsdatei, die zum Debuggen benutzt werden kann

Die Syntax für den Aufruf des Binders lautet:

```
bind ( program  => unit_name,
       library  => library_name,
       options  => (level => link), ...
       ...
      );
```

Es würde zu weit führen, sämtliche Optionen und Paramter auszuführen. Die wichtigsten Parameter sind:

program: Die Hauptprozedur (main unit) des übersetzten Programms

library: Der Name der Bibliothek, in der der Binder die Hauptprozedur finden kann.

Beispiel:

```
bind ( program  => bingo,
       library  => /usr/smith/games);
```

Soll gleich ein ausführbares Programm erzeugt werden, dann muß in den Options level = link gesetzt werden (default-Einstellung).

3.5 Ausführung eines Programms

Um ein Programm auszuführen, muß nur der Name der Hauptprozedur (Hauptprogramm) auf der Systemebene eingegeben werden oder man ruft das Programm direkt aus der Ada-Umgebung auf, indem man den Befehl system.name eingibt.

Beispiel:

```
bingo          -- in der Shell-Umgebung
```

oder

```
system.bingo   -- in der Ada-Umgebung
```

3.6 Der Family Manager

Der Family Manager dient zum Verwalten von kompletten Programmfamilien. Man erreicht die Befehlsebene des Family Managers durch Eingabe des Befehls family_manager auf der Ebene von Ada World.

Der Family Manager verfügt neben den bereits oben erwähnten, in allen Ebenen verfügbaren Befehlen, über folgende wichtige Befehle.

3.6.1 Der New-Befehl

Mit dem new-Befehl innerhalb des Family Managers ist es möglich, eine neue Familie zu erzeugen. Die Syntax lautet:

```
new ( family    => family_name,
      overwrite => yes|No|confirm);
```

Dabei bedeuten

family_name: Der neue Programmbibliotheksfamilienname. Es muß sich dabei um einen UNIX-Filenamen handeln. Es kann ein Familienname mit dem default-Befehl gesetzt werden.

overwrite:

yes: Sofort einen existierenden Filenamen überschreiben.

no: Falls eine Familie mit dem angegebenen Namen bereits existiert, dann überschreibe nicht.

confirm: Falls eine Familie mit dem angegebenenen Namen existiert, dann wird eine Bestätigung zum Überschreiben verlangt (Voreinstellung)

Beispiel:

```
new ( family => /usr/smith/adafam);
```

3.6.2 Der Erase-Befehl

Der erase-Befehl kann benutzt werden, um eine existierende Familie zu löschen.

Die Syntax lautet:

```
erase ( family  => family_name,
        confirm => yes|no);
```

dabei bedeuten:

family_name: wie beim new-Befehl

confirm:

yes: immer bestätigen (Voreinstellung)

no: keine Bestätigung verlangen

Beispiel:

```
erase ( family => /usr/smith/adafam);
```

3.6.3 Der Rename-Befehl

Mit dem rename-Befehl können bereits existierende Familien umbenannt werden.

Syntax:

```
rename ( family    => old_family_name,
         as        => new_family_name,
         overwrite => yes|no|confirm);
```

dabei bedeuten

old_family_name: der alte Familienname

new_family_name: der neue Familienname

overwrite: wie beim new-Befehl

Beispiel:

```
rename ( family => /usr/smith/adafam,
         as     => /usr/smith/newfam);
```

Neben diesen Befehlen gibt es noch weitere, wie mend und unlock, die verwendet werden, wenn die Bibliotheken durch Systemabstürze beschädigt wurden.

3.7 Der Library Manager

Mit dem Library Manager können die verschiedenen Bibliotheken verwaltet werden. Man ruft den Library Manager durch die Eingabe des Befehls library_manager in Ada World auf.

Neben den bereits oben, in allen Befehlsebenen verfügbaren Kommandos, gibt es weitere wichtige Befehle:

3.7.1 Der New-Befehl

Der New-Befehl erzeugt eine neue Programmbibliothek. Die neue Bibliothek enthält Links zu allen Paketen und Unterprogrammen der "predefind Library".

Syntax:

```
new ( library   => library_name,
      overwrite => yes|no|confirm,
      parent    => library_name | predefined,
      annotate  => character_string);
```

dabei bedeuten

library_name: Der neue Bibliotheksname

overwrite: Wie beim new-Befehl des Family Managers

parent: Normalerweise die vordefinierten Bibliotheken oder die "Eltern-Bibliotheken"

annotate: Beschreibung zur Bibliothek

Beispiel:

```
new ( library => /usr/smith/adalib);
```

3.7.2 Der Erase-Befehl

Mir dem erase-Befehl kann eine bereits existierende Bibliothek gelöscht werden.

Syntax:

```
erase ( library => library_name,
        confirm => yes|no);
```

dabei bedeuten

library_name: Der zu löschende Libray-Eintrag

confirm: Wie erase-Kommando beim Family Manager

Beispiel:

```
erase ( library => /usr/smith/adalib);
```

3.7.3 Der Other-Befehl

Mit dem other-Befehl kann die aktuelle Familie gewechselt werden. Die nachfolgenden Befehle des Library Managers beziehen sich auf die neu gewählte Familie.

Synatx:

```
other (family => installation | public | family_name);
```

dabei bedeutet

installation: Die installation-family wird aktuelle Familie

public: Die public-family wird aktuelle Familie

family_name: Der Name der neuen Familie

Beispiel:

```
other (family => /usr/smith/adafam);
```

3.7.4 Der Rename-Befehl

Mit dem rename-Befehl kann der Name einer existierenden Programm Bibliothek geändert werden.

Synatx:

```
rename ( library   => old_library_name,
         as        => new_library_name,
         overwrite => yes|no|confirm);
```

dabei bedeuten

old_library_name: Der alte Library-Name

new_library_name: Der neue Library-Name

overwrite: Wie beim new-Befehl

Beispiel:

```
rename ( library => /usr/smith/adalib,
         as      => /usr/smith/newlib);
```

3.7.5 Der List-Befehl

Der list-Befehl zeigt Informationen über die Libraries in der aktuellen Familie an.

Syntax:

```
list ( libraries => pattern,
       display   =>
         (output => SCREEN | file_name,
          detail => NAMES | BRIEF | FULL));
```

dabei bedeutet

pattern: Ein Pattern, der bestimmt, welche Libraries aus der aktuellen Famile angezeigt werden sollen (voreingestellt: *, d.h. alle anzeigen).

Beispiel:

```
list;
```

3.8 Der Unit Manager

Mit dem Unit Manager ist es möglich, die verschiedenen Units in einer Programmbibliothek zu verwalten. Neben den in allen Ebenen verfügbaren Befehlen, existieren auch hier noch eine Reihe weiterer Befehle. Im Bereich des Prozeßrechner-Praktikums werden keine dieser Befehle unbedingt benötigt, so daß ich hier auf den Projekt Development Guide S. 123ff verweisen möchte. Der Unit Manager wird mit dem Befehl unit_manager betreten. Einige nützliche Befehle sind:

3.8.1 Der List-Befehl

Der List-Befehl zeigt Informationen über die Units in der aktuellen Library an.

Synatx:

```
list ( units   => pattern,
       display => ...
       ...
     );
```

dabei bedeutet

pattern: Ein Pattern, der bestimmt, welche Units aus der aktuellen Library angezeigt werden sollen (voreingestellt: *, d.h. alle anzeigen).

3.8.2 Der View-Befehl

Mit dem view-Befehl können Informationen über den gesamten Inhalt der aktuellen Programmbibliothek angezeigt werden.

Syntax:

```
view;
```

Folgende Informationen werden angezeigt:

- der Name der Bibliothek
- der Name der Familie zu der die Bibliothek gehört
- die Anzahl der Links auf Units in anderen Bibliotheken
- die Anzahl der kompilierten Units in der Bibliothek
- den aktuellen Zugriffsstatus

3.8.3 Der Acquire-Befehl

Mit dem acquire-Befehl ist es möglich, Units aus anderen Bibliotheken zu linken. Links können nur auf Units erzeugt werden, die

- sich innerhalb der Libraries derselben Familie befinden, oder
- sich in den Libraries der INSTALLATION-Familie befinden.

Syntax:

```
acquire (units     => pattern,
         kind      => specification | body | all,
         from      => library_name,
         by        => link | copy,
         overwrite => yes | no | confirm,
           (output => screen | file_name,
            detail => brief | full));
```

dabei bedeuten

pattern: Pattern für die Unit

kind: Wohin der Link gehen soll

from: Name der Library, die die Unit enthält und auf die der Link zeigen soll.

by: Linken oder kopieren.

overwrite: Überschreiben ja oder nein, oder bestätigen.

output: Bildschirm oder Datei

detail: Kurz oder ausführlich

Beispiel:

```
acquire (units => "test",            -- creates a link unit
         from  => /usr/smith/games); -- to test in the
                                     -- library /usr/smith/games
```

3.8.4 Der Erase-Befehl

Mit dem erase-Befehl ist es möglich, mehrere Units zusammen mit ihren Abhängigkeiten aus einer Library zu löschen. Dazu muß sich der Unit Manager aber im Update-Mode befinden.

Syntax:

```
erase (units   => pattern,
       confirm => yes | no,
       kind    => specification | body | alternate);
```

Beispiel:

```
erase (units => "test",
       kind  => alternate);
```

3.8.5 Der Other-Befehl

Mit dem other-Befehl kann die aktuelle Bibliothek gewechselt werden. Die nächsten Befehle beziehen sich dann auf die neue Bibliothek.

Syntax:

```
other (library => library_name,
       mode    => read | update);
```

Beispiel:

```
other (library => /usr/smith/games, -- change to library
                                       /usr/smith/games
       mode    => update);          -- open for reading
                                       and writing
```

3.8.6 Der Transfer-Befehl

Mit dem Transfer-Befehl kann man sich Informationen (Programmlisting) aus der Programmbibliothek anzeigen lassen.

Syntax:

```
transfer (unit   => unit_name,
          kind   => specification | body,
          component => source | options,
          to        => screen | file_name);
```

dabei bedeuten

unit: Name der Einheit

kind: Informationen aus der Spezifikation oder aus dem body

component: Quellcode (default) oder Optionen anzeigen

to: Bildschirm oder Datei

Beispiel:

```
transfer (unit => "test");
```

3.9 Die Workbench

Die Workbench stellt eine Reihe von verschiedenen Programmiertools zur Verfügung. Dazu gehören:

- AdaProbe: Debugger für Ada-Programme
  AdaProbe ist ein Debugger für Ada-Programme. Der Debugger wird in der Workbench mit dem Befehl adaprobe aufgerufen. Der Debugger ist menügesteuert und benutzt den vierten Bildschirm der LynxOS Rechner.

  AdaProbe ermöglicht

  - das Setzen von Breakpoints in einem Ada Programm
  - die kontrollierte, schrittweise Ausführung eines Programms (step und trace-Befehle)
  - die Manipulation von Ada Objekten, das heißt das Beobachten und Setzen von Varibalen ist jederzeit möglich.
  - die direkte Manipulation von Objekten auf Speicherebene, das heißt es können direkt Werte aus Speicheradressen ausgelesen werden, bzw. geschrieben werden. Das direkte Ändern von Programmcode ist ebenfalls möglich.
  - Zum Erlernen von AdaProbe gibt es ein AdaProbe-Tutorial, das mit allen wichtigen Elementen von AdaProbe bekanntmacht.
- AdaMake ermöglicht das Kompilieren mehrerer Ada-Source Files ähnlich zum Make-File in C. Im wesentlichen ist AdaMake ein Script der Ada-World, mit den entsprechenden Kompilierungsanweisungen.
- AdaXref ist ein Tool zum Erzeugen und Anzeigen von Querverweisen in Ada Programmen.
- AdaReformat ist ein recht nützlicher Pretty Printer für Ada-Programme.
- AdaVerify überprüft die Syntax eines Ada-Programms.

4. Literaturverzeichnis

- Kaucher, Klatte, Ulrich, Wolff v. Gutenberg: Programmiersprachen im Griff , Band 4: ADA, BI Hochschultaschenbücher
- Andreas Schwald, Ada: Eine Einführung, BI Hochschultaschenbücher
- Alsys Ada Software Development Environment for 80386 UNIX - Version 5.1.2, Toolset Manuals
- Alsys Ada Software Development Enviroment for 80386 UNIX, User Manuals
- The Operating System LynxOS, User Manual
- The Programming Language ADA Reference Manual,American National Standards Institue, ANSI/MIL-STD-1815A-1983, edited by G. Goos and J. Hartmanis, Springer Verlag