Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

Fakultät 5: Informatik, Elektrotechnik und Informationstechnik
Abteilung Formale Konzepte
Universität Stuttgart

25.10. + 08./15./22./29.11. + 06./13.12.2006 / Version 20. Dezember 2006

Inhaltsverzeichnis

1 Programmierung im Kleinen **2**

1.0 Was Sie bis hier her gelernt haben sollten 2

1.1 Algorithmen und Programme 2

1.1.1 Definition Algorithmus und Programm 2

1.1.2 Aspekte von Programmen 4

1.1.3 Rollenspiel-Metapher 5

1.2 Sprachen zur Beschreibung der Syntax von Sprachen 6

1.2.1 EBNF 8

1.2.2 Syntaxdiagramme 8

1.3 Beschreibung der Syntax von Ada 95 mit EBNF 9

1.4 Standarddatentypen `integer` und `float` – oder – Ada und das strenge Typkonzept 11

1.5 Arithmetische und logische Ausdrücke 15

1.5.1 Arithmetische Ausdrücke 15

1.5.2 Logische Ausdrücke 16

1.6 Auswertung von Ausdrücken – oder – der Baum: eine Struktur, viele Gesichter 18

1.6.1 Bäume: Begriffe und Definitionen 18

1.7 Bereiche und deren Verwendungen – oder – Arrays (und wie man dessen Elemente sortieren kann) 19

1.7.1 Ein einfacher Sortieralgorithmus: Sortieren durch Minimumsuche 22

1.7.2 Weitere Anmerkungen und Besonderheiten bei der Verwendung von Feldern in Ada 95 25

1.8 Das Blockkonzept in Ada 95 – oder – Parameterübergabemechanismen: Theorie und Praxis 26

1.8.1 Parameterübergabe-Mechanismen 27
1.8.2 Semantik von Prozeduren und Funktionen 28
1.8.3 Goldene Regeln beim Entwurf von Prozeduren und Funktionen und ein Beispiel, wie man es nicht machen sollte 30
1.9 Rekursion – oder – wie sich Probleme durch sich selbst lösen 32
1.9.1 Ein allgemeines Schema zur Rekursion 34
1.10 Zeichen und Zeichenketten – oder – Fallunterscheidungen ohne `if` 35
1.11 Übungsaufgaben 38

1 Programmierung im Kleinen

1.0 Was Sie bis hier her gelernt haben sollten

- Informatik ist mehr als das, was mit Computern zu tun hat
- eine ungefähre Vorstellung der Begriffe Algorithmus und Programm
- der Softwareentwicklungsprozess umfasst weit mehr als nur die Implementierung eines Algorithmus in einer Hochsprache auf dem Rechner
- eine ungefähre Vorstellung der grundlegenden Konzepte von Programmiersprachen
  - Variablen, Typen, Blockkonzepte (bisher in Form von Unterprogrammen)
  - Kontrollstrukturen (Fallunterscheidung, for- und while-Schleifen)
  - Rekursion

Mit diesem Wissen sollten Sie bereits in der Lage sein, kurze Programme zu verstehen und diese durch copy-&-modify-&-try-&-error zielgerichtet (d.h. kein blindes Raten) an ähnliche Aufgabenstellungen anzupassen.

Mit dieser ungefähren Vorstellung im Gepäck werden wir nun nochmal an den Anfang zurückgehen und die Begriffe etwas formaler und genauer fassen.

1.1 Algorithmen und Programme

Die Definition „Informatik“ aus dem Duden Informatik ist für den Moment hinreichend genau. Die Begriffe Algorithmus und Programm wurden bis jetzt jedoch nur umgangssprachlich verwendet.

1.1.1 Definition Algorithmus und Programm

Was zeichnet einen Algorithmus aus?

- Verarbeitungsvorschrift
- präzise formuliert

- kann von jedem ohne weitere Erläuterungen durchgeführt werden
  - je nach Kontext kann zur Durchführung noch weiteres Wissen notwendig sein – dieses ist dann aber eindeutig und nicht spezifisch für den Algorithmus, z.B. „bilde zu $f(x)$ die erste Ableitung $f'(x)$“
- enthält der Algorithmus umgangssprachliche Elemente, so müssen diese eindeutig interpretierbar sein
- Endlichkeit der Darstellung

Dies sind für den Moment die wichtigsten Eigenschaften (weitere Forderungen und Eigenschaften sind eher von theoretischem Interesse).

Zu Punkt 4: Kochrezepte haben i.A. nicht die Eigenschaft eines Algorithmus („eine Prise Salz“, „ein wenig Öl“, ...).

Zu Punkt 5: Ein Beispiel: Zur Ausgabe aller natürlichen Zahlen können wir folgenden Algorithmus angeben:

1. merke dir die Zahl 0
2. gebe die gemerkte Zahl aus
3. erhöhe die gemerkte Zahl um 1 und merke dir nun diese (und nur diese) Zahl
4. gehe zu Schritt 2

Dieser hat eine endliche (sogar sehr kurze) Darstellung. Gibt man hingegen alle Zahlen explizit aus wie hier

1. gebe die Zahl 0 aus
2. gebe die Zahl 1 aus
3. gebe die Zahl 2 aus
4. gebe die Zahl 3 aus
5. $\vdots$

– so ist dies kein Algorithmus, da die Darstellung nicht endlich ist. In der Praxis wären solche Berechnungsvorschriften mangels Möglichkeit zur Speicherung auf externen Datenträgern sowieso ungeeignet.

Auf die Frage, ob ein Algorithmus unendlich lange läuft, kommen wir in einigen Wochen zurück. In der Praxis ist man in der Regel an Algorithmen interessiert, die für jede Eingabe nach endlich vielen Schritten die Berechnung abgeschlossen haben (Ausnahmen sind z.B. Betriebssysteme oder Steuerungssysteme).

Umgangssprachlich können wir Algorithmus mit dem Begriff „eindeutiges Kochrezept“ übersetzen.

Im Allgemeinen gibt es für ein gegebenes Problem mehr als einen Algorithmus (sogar unendlich viele).

Was unterscheidet nun ein Programm von einem Algorithmus?

- eindeutiger Formalismus einer Programmiersprache
- Bezug auf bestimmte Darstellung der verwendeten Daten
- Schnittstellen zu anderen Programmen (z.B. Betriebssystem) und Hardware
- Ausführbarkeit auf einem Computer

Mit der Umsetzung eines Algorithmus in ein auf einem Computer ausführbaren Programm kommen in der Regel noch die beiden folgenden Eigenschaften bzw. Einschränkungen dazu

- Näherungen (z.B. bei reellen Zahlen)
- endlicher Speicher

Ein Algorithmus kann in verschiedene Programme umgesetzt werden (verschiedene Programmiersprachen, verschiedene Betriebssysteme, verschiedene Computerhardware). Ein Algorithmus ist somit die abstrakte Formulierung aller Programme, die ihn beschreiben.

1.1.2 Aspekte von Programmen

Lexikalische Einheiten (Bezeichner, Literale, reservierte Wörter, Begrenzer, Trennzeichen, Kommentare) – dies sind die Bausteine eines Programms.

- Lexikalische Einheiten entsprechen dem, was in natürlicher Sprache z.B. ein Wort oder Satzzeichen ist - jede Einheit hat eine Bedeutung in sich.
- Bezeichner sind die Namen von Variablen und Prozeduren.
- Literale sind direkt hingeschriebene Werte (z.B. das Ganzzahl-Literal `42` oder das Text-Literal `"Das ist ein Text"`).
- Reservierte Wörter (auch Schlüsselwörter genannt) sind die direkt in der Sprache Ada 95 verankerten Wörter wie z.B. „`procedure`“, „`begin`“, „`while`“, „`if`“ oder „`end`“.
- Kommentare beginnen mit „`--`“ und enden am Zeilenende.
- Begrenzer sind Zeichen wie Klammern, (Vergleichs-)Operatoren (`+`,`-`,`*`,`/`,<,>) oder der Zuweisungsoperator „`:=`“.
- Trennzeichen sind das Zeilenende und in der Regel Leerzeichen und Tabulatoren (außer in Text-Literalen und Kommentaren).
- Der Programmtext wird durch Begrenzer und Trennzeichen in die anderen lexikalischen Einheiten aufgeteilt. (Dies ist auch der erste Schritt, den ein Übersetzer der Sprache Ada 95 durchführt.)

Neben diesen Bausteinen gibt es einige grundlegenden Konzepte:

- Variablen-, Typ-, Block-Konzepte

- Datenstrukturen (`integer`, `boolean`, `character`, ...)
- Operatoren (`+`, `-`, `*`, `/`, `mod`, `and`, `or`, `&`, ...)
- Kontrollstrukturen (`if`, `for`, `while`, ...)

Den Bereich über den Aufbau und Bedeutung von Programmen fassen wir in der Semiotik (Lehre von Zeichen, Zeichensystemen und Zeichenprozessen) zusammen:

- Syntax (formaler Aufbau)
- Semantik (Bedeutung)
- Pragmatik (Wechelwirkung mit der „Außenwelt“ – dies meint, dass es die Konzepte und Bausteine von Programmiersprachen genau so gibt, weil es entsprechendes in der Realität gibt, das im Rechner modelliert werden soll)

Zur Einführung der Datenstrukturen, Operatoren und Kontrollstrukturen werden wir die Syntax und Semantik definieren. Dazu benötigen wir insbesondere formale Beschreibungsmöglichkeiten für Syntax (hier: Erweiterte Backus-Naur-Form (EBNF), Syntaxdiagramme).

1.1.3 Rollenspiel-Metapher

Unter anderem zur Motivation, warum überhaupt eine formale Beschreibung sinnvoll ist, betrachten wir, welche Rollenverteilung es beim Programmieren im Allgemeinen gibt.

- Programmierer: schreibt Programme (in einer Hochsprache); er alleine ist dafür verantwortlich, was das von ihm geschriebene Programm leistet (Semantik des Programms) – hier: Sie!
- Ausführer: – hier: AdaLogo bzw. Übersetzer von Ada 95 (+ Windows/Linux)
  - prüft die Programme auf syntaktische Korrektheit (d.h., er prüft, ob das Programm ein für den Rechner formuliertes „detailliertes, eindeutiges Kochrezept“ ist)
  - führt das Programm (falls es syntaktisch korrekt ist) mechanisch Schritt für Schritt nach dem „Kochrezept“ aus; der Ausführer hat keine Vorstellung davon, was der Programmierer sich gedacht haben könnte – er tut genau das und nur das, was im Programm steht
- Benutzer: – hier: Sie! (und bei den Übungsaufgaben der Korrektor)
  - lässt den Ausführer Programme ausführen
  - ist dabei für Eingaben und insbesondere Interpretation(!) der Ausgaben zuständig

Notwendige Kenntnisse für den Programmierer

- Syntax und Semantik der Programmiersprache
- Problemstellung, welche Eingaben sollen zu welchen Ausgaben verarbeitet werden

- ggf. Kreativität und Genialität (oder Expertenwissen von außen), wenn effiziente Lösungen oder ähnliches gefragt sind

Notwendige Kenntnisse für den Ausführer

- Zur Überprüfung, ob es sich um ein Programm handelt: Syntax der Programmiersprache
- Voraussetzung: Eindeutige Beschreibungen ⇝ Formale Darstellung (speziell, wenn das Ausführen mechanisch durchgeführt wird) ⇝ Erweiterte Backus-Naur-Form (EBNF), Syntaxdiagramme
- Zur Ausführung des Programms: Semantik der Programmiersprache

Notwendige Kenntnisse für den Benutzer

- Keine, wenn das Programm entsprechend geschrieben ist

Das Programm aus Sicht des Ausführers

- Eingabe für den Ausführer: hier: eine Textdatei mit dem Programm in AdaLogo bzw. Ada 95
- 1. Aufgabe: Analyse des Programms, Zerlegung des Textes in Lexikalische Einheiten
- 2. Aufgabe: Überprüfung auf formal korrekten Aufbau
- ggf. Rückmeldung an den Programmierer ⇝ dieser muss sich über die lexikalischen Einheiten bewusst sein, um die Fehlermeldungen des Ausführers verstehen zu können
- 3. Aufgabe: Ausführen des Programms

Formale Syntax ist auch für den Programmierer sinnvoll, da diese den strukturellen Aufbau der Sprache verdeutlicht.

1.2 Sprachen zur Beschreibung der Syntax von Sprachen

Der wesentliche Aufbau von Programmen in Ada 95 (und der meisten anderen Hochsprachen) lässt sich relativ leicht beschreiben. Wir stellen hier zwei Möglichkeiten vor

- (E)BNF - (Erweiterte) Backus-Naur-Form
  - wird z.B. auch im Ada Reference Manual verwendet
- Syntaxdiagramme
  - sind etwas anschaulicher, aber in der Praxis auch etwas umständlicher zu handhaben

Hier sollen zunächst nur die wenigen Bausteine der EBNF und der Syntaxdiagramme vorgestellt werden, so dass wir damit die Syntax von Ada 95 formal beschreiben können.

Welche weitergehenden Eigenschaften die EBNF und Syntaxdiagramme haben, was genau damit beschrieben werden kann und wo die Grenzen der Möglichkeiten liegen, werden wir später im Semester aufgreifen. Für den Moment reicht erst einmal das Wissen, dass alles, was mit EBNF dargestellt werden kann, auch mit Syntaxdiagrammen möglich ist und umgekehrt.

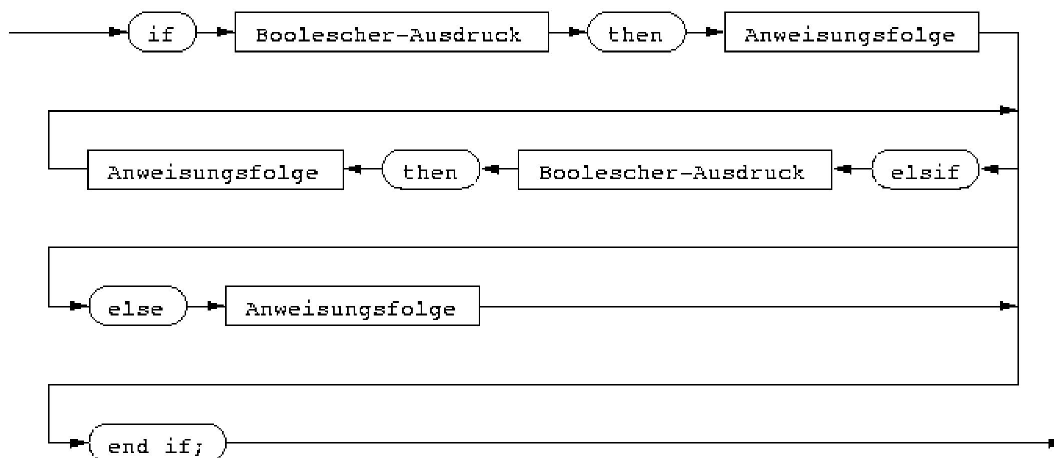
Wir führen hier nun an dem Beispiel des `if-then-elsif-else`-Konstrukts vor, wie sich dessen Syntax mittels Syntaxdiagramm und EBNF darstellen lässt.

Darzustellen ist folgender Aufbau: Eine `if-then-elsif-else`-Anweisung beginnt stets mit `if`, gefolgt von einem Boole'schen Ausdruck. Dann kommt das Schlüsselwort `then`, gefolgt von einer Anweisungsfolge. Optional kommen dann beliebig viele (oder auch garkeine) `elsif`-Zweige (jeder Zweig beginnt mit dem Schlüsselwort `elsif`, gefolgt von einer Anweisungsfolge. Optional kann dann noch ein `else`-Zweig folgen (Schlüsselwort `else` plus Anweisungsfolge). Abgeschlossen wird die `if-then-elsif-else`-Anweisung in jedem Fall durch `end if;`.

In EBNF formulieren wir das wie folgt – wir geben die EBNF-Regeln gleich so an, dass auch sinnvolle Einrückungen mit dargestellt werden:

```
<if-then-elsif-else>  ::=    "if" <Boolescher-Ausdruck> "then"
                                  <Anweisungsfolge>
                            { "elsif" <Boolescher-Ausdruck> "then"
                                  <Anweisungsfolge> }
                            [ "else"
                                  <Anweisungsfolge> ]
                              "end if;"
```

Hier werden also Iterationen und optionale Elemente mit geschweiften bzw. eckigen Klammern dargestellt. In Syntaxdiagrammen werden diese durch Verbindungen mit Pfeilen modelliert.



Es sei hier nochmals betont, dass es dabei um die Syntax (also den formalen Aufbau) des `if-then-elsif-else`-Konstrukts geht. Die Semantik, also die Bedeutung für den Ablauf in einem Programm, z.B., dass die Anweisungen im `then`-Zweig nicht ausgeführt werden, wenn die Bedingung sich zu `false` auswertet, ist etwas anderes. Die Syntax besagt nur, dass nach dem `then` stets eine Folge von Anweisungen stehen muss, nicht dass diese auch in jedem Fall ausgeführt wird.

1.2.1 EBNF

Hier noch einmal das Wesentliche über die EBNF kurz zusammengefasst:

- Formale Beschreibungssprache für Zeichenketten
- Terminale (direkt aufgeschriebene Zeichenketten): z.B. `"Text"`, `"42"`, ... – in Anführungszeichen (zum Teil findet man auch Varianten ohne Anführungsstriche, z.B. das Ada Reference Manual)
- Nichtterminale (Variablen): z.B. <if-Anweisung>, <Ausdruck>, <Ziffer>, ... – in spitze Klammern
- „wird definiert durch“: „::=“
- Alternativen: z.B. <Ziffer> ::= ”0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9” – Trennung durch senkrechten Strich
- Optionale Elemente: z.B. [”else” <Anweisungs-Folge>], ... – in eckige Klammern, 0 oder 1 Mal
- Iterationen: z.B. { ”elsif” <boolescher-Ausdruck> ”then” <Anweisungs-Folge> } – geschweifte Klammern, 0, 1, 2, ... Mal

Es gibt in der Literatur diverse Variationen davon (z.B. Markierung des Endes einer Regel durch einen Punkt, (..)* statt {..}, ...).

1.2.2 Syntaxdiagramme

Hier noch einmal das Wesentliche über die Syntaxdiagramme kurz zusammengefasst:

- Formale Beschreibungssprache für Zeichenketten
- Terminale (direkt aufgeschriebene Zeichenketten): z.B. Text, 42, ... – in Kreise (oder Ellipsen)
- Nichtterminale (Variablen): z.B. if-Anweisung, Ausdruck, Ziffer, ... – in Rechtecke
- „wird definiert durch“: Name steht über dem Diagramm
- Kreise und Rechtecke werden durch Pfeile verbunden
- Alternativen, optionale Elemente, Iterationen: durch Aufsplittung der Pfeile/Verbindungen – die durch ein Syntaxdiagramm definierten Zeichenfolgen ergeben sich durch alle möglichen Reihenfolgen vom Anfang zum Ende des Syntaxdiagramms zu gelangen.

1.3 Beschreibung der Syntax von Ada 95 mit EBNF

In der Literatur ist EBNF im Vergleich zu Syntaxdiagrammen als Syntax-Beschreibungssprache deutlich häufiger anzutreffen. Wir geben hier noch einige weitere Beispiele aus dem Umfeld von Ada 95:

Programmaufbau: Dieser ist bei Ada 95 stets gleich – nach dem „`with`-und-`use`-Bereich“ kommt das Schlüsselwort `procedure`, gefolgt von einem Namen des Programms, danach das Schlüsselwort `is`, dann der Deklarationsteil (dort werden Variablen und Prozeduren deklariert). Nun sind alle für die Ausführung des Programms notwendigen Bausteine bekannt – es folgt das Schlüsselwort `begin`, gefolgt von einem Anweisungsteil, dann dem Schlüsselwort `end` – optional kann hier noch der Name des Programms wiederholt werden – und ein abschließendes „`;`“. Danach dürfen keine weiteren Anweisungen mehr folgen. In EBNF formuliert liest sich dies so (die Einrückungen sind wie oben Empfehlungen zur Strukturierung des Codes):

```
<Ada-95-Programm>  ::=  <with-und-use-Anweisungen>
                        "procedure" <Bezeichner> is
                           <Deklarationen>
                        "begin"
                           <Anweisungsfolge>
                        "end" [<Bezeichner>] ";"
```

In EBNF lassen sich manche Dinge nicht beschreiben, z.B., dass der Bezeichner am Ende identisch mit dem am Anfang sein muss. Solche zusätzlichen Eigenschaften werden wir hier umgangssprachlich hinzufügen.

Kommentare sind überall erlaubt. Sie beginnen mit „`--`“ und gehen bis zum Zeilenende. Diese ließen sich zwar auch in EBNF formulieren, würden aber die Lesbarkeit der Definitionen beeinträchtigen. Beachten Sie: Kommentare dürfen (fast) überall beginnen, z.B. auch nach dem Wort `procedure` oder vor dem Wort `is`, wir werden die Möglichkeiten von Ada in dieser Hinsicht jedoch nicht überstrapazieren – setzen Sie Kommentare stets so, dass das Programm übersichtlich bleibt (in den hier angegebenen EBNF-Regeln z.B. stets nur am Ende der Zeilen).

Der Einfachheit halber werden hier alle **reservierten Wörter** klein geschrieben – Ada 95 unterscheidet bei reservierten Wörtern und Bezeichnern keine Groß- und Kleinschreibung. Dies ließe sich zwar ebenfalls in EBNF ausdrücken, aber `"procedure"` ist schlicht kürzer und besser lesbar als `("p"|"P")("r"|"R")("o"|"O")...("e"|"E")`.

Einige Bestandteile müssen wir noch genauer definieren: Deklarationen sind z.B. eine beliebig häufige Wiederholung von Variablen- und Prozedurdeklarationen, wobei keine feste Reihenfolge vorgegeben ist.

Wir schauen uns hier als Beispiel die **Variablen-Deklaration** genauer an: Nach einem Namen für die Variable kommt ein „`:`“, gefolgt von dem Datentyp. In Ada kann man noch optional vor dem Datentyp das Schlüsselwort `constant` angeben, um eine Variable als unveränderlich zu markieren. Ebenso kann einer Variablen gleich bei der Deklaration ein Anfangswert zugewiesen werden. Wir beschreiben dies mit den EBNF-Regeln wie folgt:

```
<Variablen-Deklaration>  ::=
        <Bezeichner> ":" [ "constant" ] <Datentyp> [ ":=" <Ausdruck> ] ";"
```

Welchen Sinn hat die Definition als Konstante? Auch hier ist wieder die Les- und Wartbarkeit das Argument: Ohne ein entsprechendes Ada-Konstrukt wäre es nur ein Versprechen des Programmierers, den Inhalt dieser Variablen nie zu verändern – durch das Schlüsselwort `constant` wird es aber durch den Ausführer garantiert, dass dieser Wert im Programm gleich bleibt (Ausnahmen gibt es auch hier, wir werden im Rahmen vom Konzept der Blöcke darauf zurückkommen – Stichwort: lokale/globale Variablen).

In Ada kann man auch gleichzeitig mehrere Variablen vom selben Typ in einer Anweisung deklarieren, ggf. erhalten dann alle Variablen denselben Anfangswert. Z.B.

```
Zaehler, Nenner, Ergebnis : integer := 1;
```

Versuchen Sie selbst, in obiger EBNF-Regel die Syntax so zu erweitern, dass mehrere durch Komma getrennte Variablen-Bezeichner erlaubt sind.

Bezeichner: Zum Benennen der Dinge, mit denen wir arbeiten (Variablen, Prozeduren und noch auch einiges andere), dürfen wir uns in Ada Namen (Bezeichner, englisch: identifier) ausdenken, die nur wenigen Einschränkungen unterliegen:

- Jeder Name muss mit einem Buchstaben anfangen
- Als weitere Zeichen sind erlaubt: Buchstaben, Ziffern und der Unterstrich '_', aber keine sonstigen Zeichen, auch keine Leerzeichen
- Es dürfen nicht mehrere Unterstriche hintereinander vorkommen, das letzte Zeichen des Namens darf kein Unterstrich sein.

Bezeichner dürfen beliebig lang sein; wie bei den Schlüsselwörtern werden Groß- und Kleinbuchstaben nicht unterschieden („`Muh`“ und „`muH`“ sind in Ada ein und derselbe Name).

Einige erlaubte Bezeichner:

„`c`“, „`c1`“, „`ein_langer_bezeichner`“

und ein paar Wörter, die keine zulässigen Bezeichner sind:

„`1c`“, „`keine leerzeichen`“, „`unerlaubtes_zeichen!`“

In EBNF lässt sich dies sehr kompakt formulieren (überlegen Sie selbst, dass somit alle Bezeichner gemäß obigen Eigenschaften beschrieben werden):

```
<Bezeichner>  ::=  <Buchstabe> { [ _ ] <Ziffer> | [ _ ] <Buchstabe> }
```

Einige Bezeichner sind für Ada-Sprachelemente reserviert, etwa „`procedure`“. Andere, wie etwa „`Get`“ stehen für Dinge, die in der Ada-Sprachbibliothek bereits definiert sind. Die Regeln, in wieweit man diese Namen für eigene Sachen nutzen darf, sind recht kompliziert; normalerweise wird man das auch nicht machen wollen (Ausnahmen folgen ggf. im Laufe des Semesters).

Dass diese Schlüsselwörter als Variablen-Bezeichner nicht erlaubt sind, lässt sich zwar in EBNF formulieren, macht die Regeln aber unnötig kompliziert – wir fügen diese Eigenschaft hier nur umgangssprachlich hinzu (Übung: versuchen Sie EBNF-Regeln anzugeben, die alle `<Bezeichner>` wie oben erzeugen mit Ausnahme des Schlüsselwortes `begin`).

Die Auswahl geeigneter Bezeichner ist schon ein Schritt in Richtung eines guten Programmierstils: Aussagekräftige, aber nicht zu lange Namen, erhöhen die Lesbarkeit von Programmen!

Obwohl Ada Groß-/Kleinschreibung ignoriert, wollen wir uns hier im Sinne der Lesbarkeit an folgende übliche Konvention halten:

- Schlüsselwörter werden stets klein geschrieben
- Bezeichner für Variablen, Konstanten, Prozeduren und Funktionen sollten immer mit einem Großbuchstaben beginnen, die anderen Buchstaben sollten klein geschrieben sein, außer eventuell hinter einem Unterstrich (wie z.B. bei „Ergebnis_in_Prozent“)

1.4 Standarddatentypen `integer` und `float` – oder – Ada und das strenge Typkonzept

Berechnungen beruhen meist auf Zahlen, wir werden im Rechner also insbesondere die Menge der ganzen Zahlen $\mathbb{Z} := \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ abbilden wollen:

Ada 95 stellt hierfür die Datentypen `integer` (entspricht $\mathbb{Z}$), `natural` (entspricht $\mathbb{N} \cup \{0\}$) und `positive` (entspricht $\mathbb{N}$) zur Verfügung.

Desweiteren bietet Ada 95 die Möglichkeit über Attribute Eigenschaften und Grenzen von Datentypen während der Laufzeit abzufragen und somit ohne Änderung des Ada-Codes das Programm an Eigenheiten des Zielrechners anzupassen. Wir betrachten hier zunächst nur die Attribute `integer'first` und `integer'last`, die die kleinste und größte mit dem Datentyp `integer` darstellbare Zahl angeben.

Wir nutzen die Attribute im folgenden Beispiel der Berechnung der Fakultät, um Laufzeitfehler durch Überschreiten des erlaubten Zahlenbereichs abzufangen. Würde man diese Grenzen als feste Zahlen im Programm kodieren, so müsste man auf anderen Rechnern unter Umständen das Programm ändern, um es auf die Möglichkeiten des neuen Rechners anzupassen. Durch Verwendung der Attribute nutzt das Programm auf unterschiedlichen Rechner-Architekturen immer den dort verfügbaren Zahlenbereich optimal aus.

Als Beispiel dient uns folgendes Programm zur Berechnung der Fakultät einer Zahl n, d.h. $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$:

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use Ada.Text_Io, Ada.Integer_Text_Io;

procedure Fakultaet is
   N : natural;
   Fak : natural := 1;
begin
   Put("Geben Sie eine natuerliche Zahl ein: ");
   Get(N);
   Put("Ich berechne jetzt die Fakultaet von "); Put(N); New_Line;

--   Put("Die groesste darstellbare Zahl ist "); Put(natural'last); New_Line;

   for Faktor in 2..N loop
      if Fak<=natural'last/Faktor then -- mit Abfrage Fak*Faktor<=natural'last
         Fak := Fak*Faktor;      -- würde der Überlauf bei der Abfrage erzeugt
      else
         Fak := 0;
      end if;
   end loop;
```

```
    if Fak=0 then
       Put("Es ist ein Fehler aufgetreten"); New_Line;
    else
       Put("Das Ergebnis ist "); Put(Fak); New_Line;
    end if;
end;
```

Das Programm tut zwar, was es soll, trotzdem ist folgendes unschön:

- bisher keine Zuweisung der Art `X:=Fak(N)` möglich
- Ausgabe der Zahlen unschön
- Standardeinstellungen sind für mehrzeilige Ausgaben gut geeignet (evtl. etwas breit), für einzelne Zahlen aber unnötig viele Leerzeichen
- recht eingeschränkter Zahlenbereich und unpraktische Fehlerbehandlung

Auf den letzten Punkt werden wir am Ende der Vorlesung im Rahmen des Exception-Handlings zurückkommen. Um die anderen drei Punkte kümmern wir uns jetzt.

In der Mathematik ist eine Funktion (von einer Menge A in eine Menge B) eine Abbildung, die jedem Element eines Definitionsbereichs genau ein Element eines Wertebereichs zuordnet. Wir werden dies in Ada sinngemäß übernehmen – dazu müssen wir lediglich das Konzept der Prozeduren etwas erweitern, indem wir neben den Parametern auch einen Rückgabewert fordern. Im Beispiel der Fakultät können wir folgende Funktion einführen:

```
function Fakult(N: Natural) return Natural is
   Ergebnis : Natural := 1;
begin
   for Faktor in 2..N loop
      if Ergebnis<=natural'last/Faktor then
         Ergebnis := Ergebnis*Faktor;
      else
         Ergebnis := 0;
      end if;
   end loop;
   return Ergebnis;
end;
```

Diese wird im Deklarationsteil eingefügt und kann nun nach dem `begin` des eigentlichen Programms z.B. in Zuweisungen `Fak := Fakult(7);` oder auch in einer Ausgabe `Put(Fakult(N));` verwendet werden.

Die Syntax einer Funktion in Ada lautet formuliert als EBNF-Regeln:

```
<Funktion>  ::=  "function" <Bezeichner> "(" <Parameter-Liste> ")"
                                          "return" <Datentyp> "is"
                    <Deklarationen>
                 "begin"
                    <Anweisungsfolge>
                 "end" [<Bezeichner>] ";"
```

Auch hier muss der Bezeichner, wenn er beim `end` wiederholt wird, identisch gewählt werden. Außerdem fordern wir wenigstens eine `return`-Anweisung. Mit dieser wird die Funktion beendet und der Wert der Funktion an den aufrufenden Programmblock zurückgegeben.

Bei der Ausgabe der `integer`-Zahlen erlaubt Ada die Angabe der Breite, die für die Zahl reserviert werden soll (hat die Zahl mehr Stellen, so werden trotzdem alle Stellen ausgegeben. Hat die Zahl weniger Stellen, so werden links entsprechend Leerzeichen eingefügt). Z.B. `Put(zahl,width=>5);`, `Put(a+b,width=>7);`, `Put(42,0);` – das Wort `width` kann also weggelassen werden. Auch hier geben wir die EBNF-Schreibweise an:

```
<Ganzzahlausgabe>  ::=
     "Put" "(" <int.-Ausdruck> ["," [ "width=>" ] <int.-Ausdruck>] ")" ";"
```

Das ist nicht die ganze Wahrheit, reicht aber für den Moment :-) Mit `default_width := 4;` bzw. `Ada.Integer_Text_IO.default_width := 4;` kann der Default-Wert für die Breite bei der Ausgabe von ganzen Zahlen z.B. auf 4 geändert werden. Mit dem Wert 0 benötigt die Ausgabe jeweils nur die minimal nötige Breite. Explizite Angaben in Put haben grundsätzlich Vorrang vor den Default-Werten.

In der nun folgenden Umsetzung des Programms zur Studienstiftung in Ada werden wir uns mit dem Datentyp `float` für Gleitkommazahlen und dessen formatierte Ausgabe beschäftigen.

```
with Ada.Text_Io, Ada.Integer_Text_Io, Ada.Float_Text_Io;
use Ada.Text_Io, Ada.Integer_Text_Io, Ada.Float_Text_Io;

procedure Stiftung is

    Betrag : float := 100000.0;
    Ausgaben : float := 30000.0;
    Zinsen : float := 4.0;
    Inflation : float := 2.0;
    Jahr : natural := 1;
begin
--   Put(float'digits); New_Line; -- Anzahl der gültigen Dezimalstellen

    while Betrag>=Ausgaben and Betrag<Float(2000000) loop

      Put("Ausgaben im "); Put(Jahr, width=>0); Put(".ten Jahr betragen ");
      Put(Ausgaben, exp=>0, aft=>2, fore=>10); Put("Euro."); New_Line;

      Betrag:=Betrag-Ausgaben;
      Put("Guthaben vor Zinsen im "); Put(Jahr, width=>0);
      Put(".ten Jahr betragen ");
      Put(Betrag, exp=>0, aft=>2, fore=>10); Put("Euro."); New_Line;

      Betrag:=Betrag+Betrag*Zinsen/100.0;
      Put("Guthaben nach Zinsen nach dem "); Put(Jahr, width=>0);
      Put(".ten Jahr betragen ");
      Put(Betrag, exp=>0, aft=>2, fore=>10); Put("Euro."); New_Line;
      Put(Betrag, 10,2,0); Put("Euro."); New_Line;
```

```
    Jahr:=Jahr+1;

    Ausgaben:=Ausgaben+Ausgaben*Inflation/100.0;
  end loop;

  if Betrag<= Ausgaben then
    Put("pleite im Jahr "); Put(Jahr, width=>0); New_Line;
  else
    Put("juchhu, 2. Stiftung"); New_Line;
  end if;

end;
```

Für die formatierte Ausgabe von Gleitkommazahlen können wir Angaben über die Anzahl der Vorkomma- und Nachkommastellen machen, sowie die für den Exponenten. Wir geben hier zunächst die EBNF-Darstellung an:

```
<Gleitkommazahlausgabe>  ::=
     "Put" "(" <float-Ausdruck> [ "," [ "fore=>" ] <int.-A.> ]
         [ "," [ "aft=>" ] <int.-A.>] [ "," [ "exp=>" ] <int.-A.>] ")" ";"
```

Dies ist eine etwas vereinfachte Darstellung; bei Angabe von fore, aft und exp dürfen diese drei in beliebiger Reihenfolge auftreten. Beispiele: Put(Zahl,exp=>0);, Put(A*B+C,aft=>2);, Put(X,3,4,0);. Lässt man die Parameter-Bezeichner fore, aft und exp weg, so wird der erste Ausdruck für fore verwendet, der zweite für aft und der dritte für exp – der Lesbarkeit halber sollten man diese Bezeichner aber immer mit hinschreiben. Auch für float-Zahlen kann man die Default-Werte für die Ausgabe ändern: [Ada.Float_Text_IO.]default_fore := 4 setzt den Default-Wert für fore (analog für aft und exp). Explizite Angaben in Put haben grundsätzlich Vorrang vor den Default-Werten.

Wir lernen hier auch einiges über das strenge Typkonzept in Ada 95:

- Ada unterscheidet zwischen integer-Zahlen und float-Zahlen, letztere haben stets einen Dezimalpunkt „.“ und mindestens eine Nachkommastelle (ggf. eine 0).
- Es findet keine automatische Konvertierung zwischen Datentypen statt. Die Umwandlung kann über float(<Ausdruck>) in eine Gleitkommazahl und über integer(<Ausdruck>) in eine Ganzzahl erfolgen. Bei der Umwandlung in eine Ganzzahl wird mathematisch gerundet (und nicht nur wie bei manchen anderen Programmiersprachen die Nachkommastellen abgeschnitten).
- Die scheinbaren Ausnahmen wie natural/integer sind in Ada aber Unterbereiche und keine eigenständigen Typen – wir kommen bald darauf zurück
- Wo möglich, können Typen explizit konvertiert werden: <Datentyp>(<Ausdruck>), z.B. float(3*4), integer(float(42)/5.7) ... Achtung! Es können nicht beliebige Typumwandlungen durchgeführt werden (dies wird man in der Regel auch nicht wollen) und es treten ggf. Rundungsfehler auf.

Wozu dient ein solch strenges Typkonzept? Es bietet einige Vorteile:

- Durch das (strenge) Typkonzept werden bestimmte Flüchtigkeitsfehler vermieden.
- Keine Mehrdeutigkeiten bei Ausdrücken wie z.B. `x:float; ...x*5;`
- Tippfehler, die in anderen Sprachen nicht erkannt werden, werden hier ggf. zu formalen Fehlern und können so vom Ausführer erkannt werden.
- Die Lesbarkeit und Wartbarkeit von Programmen wird erhöht.

Die meisten Programmiersprachen unterstützen ein Typkonzept. Diese unterscheiden sich zum Teil aber erheblich. Ada setzt das Typkonzept sehr konsequent um. Andere Sprachen (z.B. C++) sind hier deutlich laxer (betrachten Sie z.B. den „Typ“ boolean in der Sprache C++). Ohne Typkonzept lassen sich Flüchtigkeitsfehler oft schwer finden.

1.5 Arithmetische und logische Ausdrücke

Unter dem Begriff *Ausdruck* verstehen wir hier eine Verarbeitungsvorschrift, deren Ausführung einen Wert liefert. Wir haben Ausdrücke bereits kennengelernt: Z.B. als Verknüpfung von Zahlen mit Operatoren (wie den Grundrechenarten) oder als Parameter bei der Ausgabe. Ausdrücke können überall dort stehen, wo ein Wert eines bestimmten Typs erwartet wird. Wir betrachten als einen solchen Fall die *Zuweisung*: Auf der linken Seite einer Zuweisung „`:=`“ steht stets eine Variable, auf der rechten Seite stets ein Ausdruck, in EBNF formuliert lautet dies:

```
<Zuweisungs-Anweisung>  ::=  <Bezeichner> ":=" <Ausdruck> ";"
```

Wichtig ist dabei, dass der Ausdruck sich zu einen Wert vom selben Typ auswertet ($\rightsquigarrow$ starkes Typkonzept in Ada). Zwischenschritte bei der Auswertung des Ausdrucks können dabei durchaus von anderem Typ sein: Betrachten wir den Ausdruck `Integer(Float(123/4)*5.67+0.8)`, so würde zunächst `123/4` ganzzahlig dividiert werden (Ergebnis ist die `integer`-Zahl 30), dann die 30 in `float` gewandelt werden, diese mit 5.67 multipliziert (Ergebnis ist die `float`-Zahl 170.1), 0.8 addiert (`float`-Zahl 170.9) und schließlich bei der Typ-Umwandlung auf die `integer`-Zahl 171 gerundet. Dieser Wert würde dann ggf. einer `integer`-Variablen zugewiesen werden können, aber nicht einer `float`-Variablen.

Achtung: Ada 95 rundet korrekt (also Aufrunden ab „.5“) – dies ist bei vielen anderen Programmiersprachen anders!

Wichtig! Es wird immer der Wert eines Ausdrucks zugewiesen, nicht der Ausdruck selbst – Variablen sind Wertebehälter, keine Ausdrucksbehälter.

Ein Ausdruck ist eine Anweisung einen Wert zu berechnen. Ausdruck und Wert haben zwar miteinander zu tun, sind aber wesentlich verschiedene Dinge.

1.5.1 Arithmetische Ausdrücke

Etwas vereinfacht stellt sich ein Arithmetischer Ausdruck so dar:

```
<Arith.-Ausdruck>  ::=  [ "+" | "-" ] <Term> { <Arith.-Operator> <Term> }
           <Term>  ::=  "(" <Arith.-Ausdruck> ")" | <Literal> | <Bezeichner> |
                        "abs" "(" <Arith.-Ausdruck> ")" | <Funktionsaufruf>
<Arith.-Operator>  ::=  "+" | "-" | "*" | "/" | "mod" | "rem" | "**"
```

(die ganze Wahrheit steht im Ada-Reference-Manual, 4.4).

Dabei steht `abs(<Arith.-Ausdruck>)` für die Betragsfunktion und `x**y` für die Exponentation (x^y, $y \geq 0$) (für `y` ist hier nur `integer` bzw. `natural/positive` erlaubt). Ebenfalls nur für `integer` sind `mod` und `rem` definiert (diese Operatoren berechnen jeweils den Rest einer ganzzahligen Division und unterscheiden sich nur durch die verschiedene Handhabung bei negativen Zahlen – probieren Sie die möglichen Kombinationen positiver und negativer Zahlen jeweils mit `mod` und `rem` selbst aus).

Die Klammern bei `abs` sind in Ada 95 optional, wir wollen sie aber hier der Lesbarkeit halber immer verwenden.

In dem Paket `Ada.Numerics.Elementary_Functions` stehen Funktionen für `float`-Zahlen wie Wurzel (`Sqrt`), Logarithmus (`Log`) und Trigonometrische Funktionen zur Verfügung (Details im Ada-Reference-Manual, A.5.1).

Prioritäten der gängigen Operatoren in Ada 95 Durch Klammerungen kann man stets Eindeutigkeit erreichen. Ist dies nicht gegeben, so hat Ada folgende (von der Mathematik und anderen Programmiersprachen bekannte) Prioritätsregeln:

1. `abs, **`
2. `*, mod, /, rem`
3. `+, -` (als Vorzeichen)
4. `+, -` (als Addition/Subtraktion)

Gewisse Kombinationen, inbesondere solche, die für Normalsterbliche „mathematisch nicht eindeutig sind“, sind verboten und müssen mit Klammerungen eindeutig gemacht werden, z.B. `x/-y`, `x**y**z`, `x**abs(y)`. (Warum ist `x**abs(y)` nicht eindeutig? Selbst probieren und Fehlermeldung deuten ...)

Auch hier: Klammerungen erhöhen die Lesbarkeit!

1.5.2 Logische Ausdrücke

Die Vergleichsoperatoren „=“, „/=“, „<“, „<=“, „>=“, „>“ haben den Ergebnis-Typ `Boolean` (entweder `true` oder `false`), wobei bei `float`-Zahlen die Vergleiche „=“, „/=“ aufgrund der unvermeidlichen Rundungsfehler mit Vorsicht zu genießen sind. Man bezeichnet aufgrund des Ergebnis-Typs Boolean die logischen Ausdrücke auch als Boolesche Ausdrücke. Auch hier folgt eine etwas vereinfachte Darstellung in EBNF:

```
<Boole.-Ausdruck>  ::=  <Relation> { <Boole.-Operator> <Relation> }
       <Relation>  ::=  "(" <Boole.-Ausdruck> ")" |
                        "true" | "false" | "not" <Relation> |
                        <Arith.-Ausdruck> <Vergleichsop.> <Arith.-Ausdruck>
<Boole.-Operator>  ::=  "and" | "or" | "xor" | "and then" | "or else"
```

(die ganze Wahrheit steht im Ada-Reference-Manual, 4.4).

Bei `and`, `or` und `xor` werden in Ada 95 grundsätzlich beide Operanden ausgewertet. Die Semantik ist

- `x and y` wertet sich zu `true` aus genau dann, wenn beide Operanden den Wert `true` haben
- `x or y` wertet sich zu `true` aus genau dann, wenn mindestens einer der Operanden den Wert `true` hat
- `x xor y` wertet sich zu `true` aus genau dann, wenn die beide Operanden verschieden sind, d.h. genau einer den Wert `true` hat

Die Operatoren „`and then`“ sowie „`or else`“ erlauben eine verkürzte Auswertung der Booleschen Ausdrücke. Z.B.

```
if x/=0 and then z/x > 10 then <Anweisung> end if;
```

oder

```
while x=0 or else z/x > 5  loop <Anweisung> end loop;
```

und vermeiden somit unter Umständen geschachtelte `if`-Anweisungen. Die Mächtigkeit der Programmiersprache wird dadurch nicht erhöht. Überlegen Sie selbst, wie Sie im obigen Beispiel der `while`-Schleife ohne den Operator `or else` auskommen könnten.

Als Besonderheit in Ada 95 gibt es keine Prioritätsregeln zwischen den Booleschen Operatoren, insbesondere verlangt Ada 95 grundsätzlich eine Klammerung, wenn verschiedene Boolesche Operatoren in einem Ausdruck vorkommen, so etwas wie `x and y or z` ist also verboten und muss geklammert werden zu `(x and y) or z` oder `x and (y or z)`.

Das `not` – `not x` wertet sich zu `true` aus genau dann, wenn `x` den Wert `false` hat und umgekehrt – nimmt eine Sonderstellung ein und ist in der Priorität auf der Ebene von `**` und `abs` eingeordnet. Auch hier hat der Benutzer durch Klammerungen für Eindeutigkeit zu sorgen, d.h. `not abs(y) > 4` ist nicht erlaubt (probieren Sie es aus).

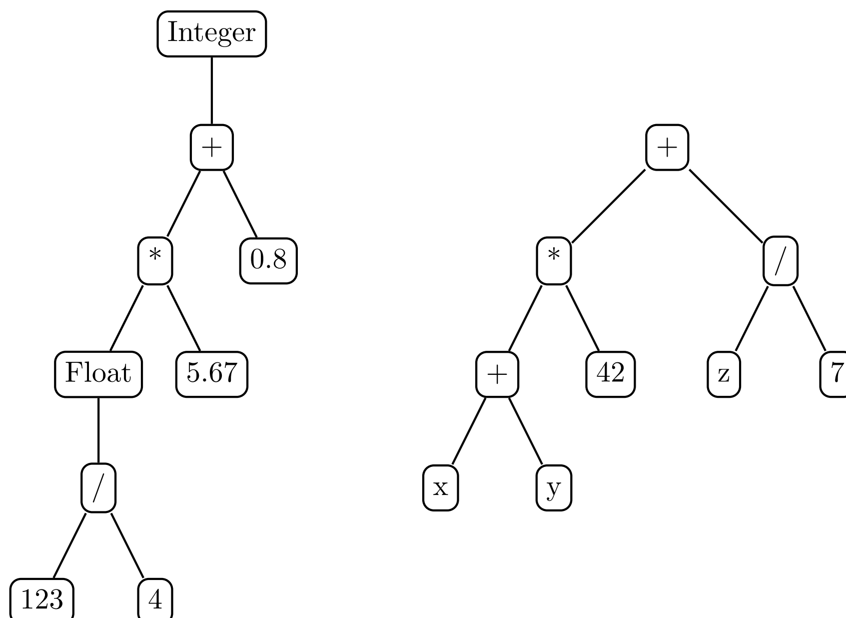
Vergleichsoperatoren für Boolean: Außer mit den logischen Operatoren können Boolesche Variablen auch mit den Vergleichsoperatoren „=“, „/=“, „<“, „<=“, „>=“, „>“ verknüpft werden, wobei `false < true` gilt. Diese Möglichkeit des Vergleichs Boolescher Variablen wäre nicht nötig und lässt sich leicht mit den logischen Operatoren nachbilden: `x /= y` entspricht `x xor y` (damit also `x = y` auch `not (x xor y)`), `x < y` ist identisch mit `(not x) and y` (finden Sie Ausdrücke mit je maximal zwei logischen Operatoren, um die verbleibenden Vergleichsoperatoren nachzubilden).

Über die Vergleichsoperatoren hinaus lässt sich auch das `xor` noch relativ leicht auf einen Ausdruck zurückführen, der nur `not`, `and` und `or` verwendet (hier sind schon fünf Operatoren nötig – selber probieren). Sogar auf das `or` könnte man noch verzichten, da `x or y` identisch mit `not (not x and not y)` ist. Wir nehmen all dieses hier aber nur als theoretische Möglichkeit zur Kenntnis und nutzen bei der Formulierung der Booleschen Ausdrücke die Möglichkeiten von Ada 95 im Sinne einer guten Lesbarkeit.

1.6 Auswertung von Ausdrücken – oder – der Baum: eine Struktur, viele Gesichter

Wir wollen hier nicht in die Tiefen des Compiler-Baus einsteigen, können aber die Gelegenheit nutzen eine grundlegende und sehr wichtige Struktur in der Informatik kennenzulernen: den Baum. Bei der Auswertung von Ausdrücken kommt dieser als Rechenbaum vor.

Hierzu zwei Beispiele: Betrachten wir nochmals den Term `Integer(Float(123/4)*5.67+0.8)`, sowie den Term `(x+y)*42+z/7` – die zugehörigen Rechenbäume sind diese:



Erstellen Sie selbst Beispiele und untersuchen Sie, in welcher Reihenfolge Ada 95 diese auswerten könnte. Ein schönes Web-Applet findet sich hierzu unter `http://fom.berlios.de/` (der Link „start applet“ dort startet das Applet).

Die Auswertung solcher Rechenbäume ist stets eindeutig und kommt insbesondere ohne Klammerungen aus. Überlegen Sie selbst, dass zu jedem Rechenbaum ein (ggf. geklammerter) arithmetischer Ausdruck existiert und umgekehrt.

Wir stellen außerdem fest, dass die Auswertungsreihenfolge und der Durchlauf durch einen Baum der rekursiven Aufrufe (wir erinnern uns an das Beispiel der Türme von Hanoi) große Ähnlichkeiten aufweisen. Wir werden im weiteren Verlauf der Einführung in die Informatik noch viele Gebiete kennenlernen, in denen Bäume vorkommen und verwendet werden. Auf die Gemeinsamkeiten dieser verschiedenen Gebiete bzgl. der Struktur „Baum“ werden wir zu gegebener Zeit aufmerksam machen.

1.6.1 Bäume: Begriffe und Definitionen

Im Allgemeinen kann in den Knoten der Bäume beliebiges stehen, in manchen Anwendungen interessiert auch nur die Struktur, so dass dann die Knoten unbeschriftet sind. Prinzipiell können auch die Kanten beschriftet sein, wir werden auf solche Anwendungen aber erst viel später zurückkommen,

Baumartige Strukturen spielen in der Informatik eine so wichtige Rolle, dass wir gleich die wichtigsten Begriffe kennenlernen sollten.

Definition: Zur Definition eines Baums beginnt man interessanterweise besser mit dem Plural: Ein *Wald* ist eine (möglicherweise leere) Menge von Bäumen.

Dann lässt sich leicht sagen, was ein Baum ist: ein *Baum* besteht aus einem Knoten (*Wurzel* genannt) und einem Wald der Unterbäume.

Insgesamt haben wir also eine Menge von Knoten vor uns, zwischen denen gewisse Beziehungen bestehen. Zur Beschreibung dieser Beziehungen greift man auf eine Analogie zu familiären Beziehungen zurück: ein Knoten ist der *Vater* der Wurzelknoten seiner Unterbäume, die entsprechend *Söhne* heißen (es ist eine rein männlich Familie...).

Knoten, deren Wald der Unterbäume leer sind, heißen *Blätter* (hier ist der Sprachgebrauch nicht einheitlich: manchmal wird die Wurzel als Blatt ausgeschlossen, bei uns aber nicht). In obigen Rechenbäumen können in den Blättern nur Konstanten und Variablennamen vorkommen, jedoch niemals Operatoren (selbst überlegen, warum).

Da Elemente einer Menge keine feste Anordnung haben, haben wir auch keine Anordnung der Söhne eines Knotens; daher ist ein Rechenbaum kein Baum im Sinne dieser Definition (die Reihenfolge der Operanden z.B. in `x/y` ist wesentlich!). Das hindert uns aber nicht daran, das Vokabular sinngemäß auch für diese Struktur zu verwenden — spielt die Reihenfolge der Unterbäume eine Rolle, so spricht man von *geordneten Bäumen*.

1.7 Bereiche und deren Verwendungen – oder – Arrays (und wie man dessen Elemente sortieren kann)

Wir haben bisher Ada als eine Sprache mit einem strengen Typkonzept kennengelernt, stellen aber fest, dass sich `integer` mit den Datentypen `natural` und `positive` verträgt. Dies liegt daran, dass die Datentypen `natural` und `positive` in Ada 95 Subtypen sind. Ein Subtyp ist eine Beschränkung eines Typs auf einen Teilbereich.

Ein Beispiel: bei der Bearbeitung des Datums auf den letzten Übungsblättern, wäre es unter Umständen hilfreich gewesen, den Wertebereich für den Monat auf 1..12 einzuschränken. Dieses hätte man mit folgender Anweisung tun können:

```
subtype Monatstyp is integer range 1..12;
```

– allgemein:

```
subtype <Typname> is <Datentyp> <Constraint>;
```

Auch hier gilt wieder das Prinzip der Les- und Wartbarkeit: Es ist gelegentlich praktisch, wenn Bereichsüberschreitungen automatisch als Fehler erkannt werden. In Ada sind unter Anderem `natural` und `positive` vordefiniert:

```
subtype natural  is integer range 0..integer'last;
subtype positive is integer range 1..integer'last;
```

Für den Subtypen stehen alle Operationen und Ausgabemöglichkeiten zur Verfügung, die auch der ursprüngliche Typ bereit stellt. Insbesondere können in Ausdrücken beliebige Subtypen eines Typs miteinander verarbeitet werden. Die Berechnungen finden dabei innerhalb des Typs statt, nicht innerhalb des Subtyps. Mit obigem `subtype Monatstyp`, wäre z.B. folgende Deklaration denkbar:

```
monat : Monatstyp := 11+5-7;
```

Der Ausdruck würde als Integer-Ausdruck ausgewertet (dies trifft auch zu, wenn in dem Ausdruck nur Variablen eines Subtyps auftreten). Zwischenergebnisse können den Subtypbereich also überschreiten – erst bei der Zuweisung wird überprüft, ob der Wert den Constraint erfüllt. Falls nicht, erzeugt das Programm einen Laufzeitfehler.

Die Constraints werden in der Regel durch Einschränkung auf einen Bereich (Schlüsselwort „`range`") angegeben. Bereiche stellen Intervalle dar: hier gilt zunächst, dass `a..b` der Menge $\{x \mid a \leq x \leq b\}$ entspricht, insbesondere ist `a..b` leer, wenn `a` größer als `b` ist. Wir werden später sehen, dass sich solche Bereiche nicht nur für Ganzzahl-Typen definieren lassen, konzentrieren uns im Moment jedoch nur auf die Ganzzahlbereiche.

Bereiche haben wir schon bei `for`-Schleifen kennengelernt. Dort wird der Schleifenvariablen bei jedem Durchlauf beginnend mit dem kleinsten Element jeweils das nächstgrößere Element des Intervalls zugewiesen. Ist der Bereich leer, so wird der Schleifenrumpf garnicht ausgeführt. Will man die Elemente in umgekehrter Reihenfolge, also vom größten zum kleinsten, bearbeiten, so ist dies mit dem Schlüsselwort `reverse` möglich:

```
for i in reverse 1..5 loop
   put (i);
end loop;
```

gibt die Zahlen 5, 4, 3, 2 und 1 aus.

```
for i in 5..1 loop
   put (i);
end loop;
```

tut hingegen garnichts, da der Bereich `5..1` leer ist – es gibt keine Zahlen, die sowohl größer gleich 5 als auch kleiner gleich 1 sind.

Bereiche lassen sich auch in Booleschen Bedingungen verwenden: in der Anweisung

```
if Zaehler in 3..9 then <Anweisung> end if;
```

würde der `then`-Zweig genau dann ausgeführt, wenn `Zaehler>=3 and Zaehler<=9` gilt.

Manchmal möchte man nicht die Möglichkeiten und Eigenschaften von z.B. `integer` übernehmen, sondern vielleicht eigene Funktionen und Operatoren definieren (wir werden später darauf zurückkommen, wie wir z.B. den Operator „+" oder andere für eigene Typen umdefinieren können). In diesem Fall definieren wir einen neuen Ganzzahltyp mit

```
type <Bezeichner> is range <range>
```

– obiger Monatstyp sähe als eigenständiger Typ dann so aus:

```
type Monatstyp is range 1..12;
```

Als Besonderheit nehmen wir zur Kenntnis, dass auch hier die Berechnungen nicht innerhalb des beschränkten Bereichs stattfinden, sondern der gesamte zur Verfügung stehende Ganzzahlbereich genutzt wird. Es wird erst bei der Zuweisung überprüft, ob der Wert in dem definierten Bereich liegt.

Die Ganzzahloperationen werden für so definierte Typen übernommen, können aber umdefiniert werden, ohne die Operationen für `integer` und dessen Subtypen zu verändern. D.h. auch, dass ein so definierter Ganzzahltyp in Ausdrücken nicht mit z.B. `integer` kombiniert werden kann – es ist ein anderer Typ, die Operatoren „`+`“, “`-`“, etc. sind nur für Operanden vom jeweils gleichen Typ definiert. Es muss hier also eine explizite Typ-Konvertierung durchgeführt werden (`Integer(...)+...` oder `Monatstyp(...)+...` – je nach dem, ob der Ergebnistyp der Operation vom Typ `integer` oder `Monatstyp` sein soll.

Auch bei der Ausgabe solcher Ganzzahltypen schlägt das strenge Typkonzept von Ada wieder zu. Hierbei gibt es prinzipiell zwei Möglichkeiten:

- Bei Ganzzahltypen, deren Wertebereich innerhalb dem von `integer` liegt, kann man sich mit der Umwandlung `Integer(...)` behelfen.

- Es ist aber auch möglich Zahlenbereiche zu definieren, die über die Grenzen von `integer` hinausgehen. Ada 95 erlaubt bei Zahl-Literalen dabei die Verwendung von Unterstrichen, um die Lesbarkeit zu erhöhen – semantisch haben diese hier keine Bedeutung. Beispiel:
  `type Grosszahlen is range -1_000_000_000_000..+1_000_000_000_000;`
  (Anmerkung: die Möglichkeit der Verwendung von Unterstrichen besteht auch bei Fließkommazahlen – `euler : float := 2.71828_18284_59045_23536;` ist z.B. möglich – die Unterstriche können dabei beliebig eingesetzt werden, jedoch nie zwei Unterstriche hintereinander).

  Um solche Zahlen ein- und ausgeben zu können müssen wir uns neue Ein- und Ausgabe-Routinen definieren. Ada 95 liefert uns dazu eine „Schablone“, die wir wie folgt einbinden können:
  `package Grosszahlen_IO is new ada.text_io.integer_io(num=>Grosszahlen);`
  Wir fügen diese (als eigene Zeile) in dem Bereich der `with`- und `use`-Anweisungen ein. Es stehen dann wie bei `integer`-Zahlen Prozeduren `Put` und `Get` zur Verfügung, die wir mit z.B. `Grosszahlen_IO.Put(...)` aufrufen können. Fügen wir noch ein `use Grosszahlen_IO;` dort mit ein, so reicht wieder die Angabe von `Put(...)` – Ada sucht sich die passende Put-Prozedur dann selbstständig heraus. Die Ausgabe kann wie bei `integer`-Zahlen auch durch Angabe von `width => ...` beeinflusst werden. Der Wert `default_width` ist ebenfalls bekannt und standardmäßig immer ein Zeichen breiter als die größte Zahl. Zum Ändern dieses Wertes muss in jedem Fall dann `Grosszahlen_IO.default_width:=...` verwendet werden, weil sonst nicht klar wäre, ob vielleicht die Breite von `integer`-Zahlen gemeint war. Ggf. macht uns der Compiler hierauf aufmerksam.

  In obiger `package`-Deklaration kann der Name zwar prinzipiell beliebig gewählt werden, wir wollen aber zur besseren Lesbarkeit uns stets an die Form `<Datentyp>_IO` (oder `<Datentyp>_Text_IO`) halten (in Anlehnung an die uns schon bekannten Ada-Bibliotheken `Ada.Integer_Text_IO` und `Ada.Float_Text_IO`).

1.7.1 Ein einfacher Sortieralgorithmus: Sortieren durch Minimumsuche

Wir wollen nun einen einfachen Sortieralgorithmus entwerfen und sehen, wie wir diesen in Ada 95 umsetzen können. Wollen wir z.B. 1000 Adressen sortieren, so wäre die Verwendung von 1000 Variablen dafür mehr als umständlich. Ada stellt dafür Arrays (auch Feld, Vektor oder Reihung genannt) zur Verfügung. Die eben eingeführten Subtypen und Bereiche werden in Ada für die Index-Mengen der Arrays verwendet.

Arrays kann man sich als Tabelle vorstellen, bei der in der einen Spalte die Indizes und in der anderen Spalte die zugehörigen Werte aufgelistet sind. Eine etwas mathematischere Sprechweise: Wir haben eine Abbildung von einer Indexmenge I in die Wertemenge des Zieltyps T und weisen jedem Indexwert ein Objekt des Typs T zu.

Die Deklaration einer Feld-Variable hat in Ada die Form

```
<Var.-Bezeichner> : array ( <range> ) of <Datentyp> [ := <Ausdruck> ] ;
```

Wie solche Ausdrücke zur Initialisierung von Feldern aussehen, entnehmen wir dem unten folgenden Beispiel des Sortieralgorithmus. Im Gegensatz zu anderen Programmiersprachen kann die Indexmenge in Ada ein beliebiger Ganzzahlbereich sein (dieser muss also nicht wie z.B. in C++ mit 0 beginnen).

Eine Möglichkeit zur Sortierung einer Menge von Zahlen ist, sich zunächst das kleinste Element herauszusuchen. Von den restlichen Zahlen sucht man sich dann wiederum das kleinste heraus, usw. Wir haben hier die `N` zu sortierenden Zahlen in einem Feld mit Indizes 1..N abgespeichert. Auf einer noch recht abstrakten Sichtweise sind die ersten Schritte des Algorithmus folgende:

1. Suche das Minimum im Feld mit den Indizes `1..N`
2. Tausche dieses Minimum mit dem Element im Feld an der Position `1`
3. Suche das Minimum im Feld mit den Indizes `2..N`
4. Tausche dieses Minimum mit dem Element im Feld an der Position `2`
5. ...

Wir erkennen, dass wir im `i`-ten Durchlauf jeweils das kleinste Element im Teilfeld mit den Indizes `i..N` suchen und dann das Minimum mit dem Element an der Position `i` vertauschen. Dies lässt sich mit einer `for`-Schleife beschreiben. Eine solche Iteration lässt sich in die zwei Teilschritte „Minimum finden“ und „Vertauschen“ aufteilen – solch ein Vorgehen der schrittweisen Verfeinerung nennt man Top-Down-Entwurf. Führt man diesen zunächst auf Papier auf diese Art und Weise durch, so lassen sich anschließend leicht diese kleinen und übersichtlichen Bausteine in Ada umsetzen und jeder für sich auf Korrektheit überprüfen, danach können diese zu den jeweils nächstgrößeren Komponenten zusammengesetzt werden und diese wiederum auf Korrektheit überprüft werden. In diesem Sortierbeispiel haben wir nur zwei solcher Ebenen – bei umfangreicheren Problemen können dies leicht auch vier, fünf oder mehr werden. Ohne diese strukturierte Herangehensweise lassen sich schwierigere Problemstellungen kaum in übersichtliche und lesbare Programme umsetzen. Wir wollen diese Methode auch schon bei unseren kleinen Übungsaufgaben anwenden, auch wenn wir hier manchmal in Versuchung geraten, uns gleich an den Rechner zu setzen und loszuprogrammieren.

Nach obiger Vorgehensweise entsteht folgendes Programm zur Sortierung von Elementen (wir verzichten hier auf die Eingabe durch den Benutzer und geben das Feld der zu sortierenden Zahlen vor – auch, um die Möglichkeiten zur Initialisierung von Feldern in Ada zu demonstrieren):

```
--------------------------
-- Autor: sl
-- Datum: 29.11.2006
-- Idee: Sortiere Zahlen (suche jeweils Minimum im entsprechenden Teilfeld
--       und tausche dieses an den Anfang des jeweiligen Teilfeldes)
--------------------------

with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Sortiere is

   N : constant natural := 9; -- Deklaration als Konstante,
                              -- um unbedachte Änderungen zu vermeiden
   feld : array (1..N) of integer := (34,27,26,29,85,74,94,23,16);
          -- (5 => 41, 3 =>23, others =>7); -- Teilinitialisierung ist möglich
          -- in beliebiger Reihenfolge Paare der Form <Index> => Wert angeben
          -- others => <Wert> initialisiert alle anderen Feldelemente
          -- obiges liefert die Initialisierung (7,7,23,7,41,7,7,7,7)

   procedure Ausgabe is
   begin
      Put("Ausgabe:"); New_Line;
      for I in 1..N loop
         Put_Line(I'Img &        -- Put_Line entspricht Put(..); New_Line;
                  "-te Zahl: " & -- <Zahl>'img liefert die Zeichenkette zur Zahl
                  Feld(I)'img);  -- & verknüpft Zeichenketten und erlaubt so
                                 -- lange Zeichenketten im Programm auf mehrere
                                 -- Zeilen verteilt übersichtlich anzugeben
      end loop;
   end Ausgabe;

   function Bestimme_Minindex(Ind_Links,Ind_Rechts:Integer) return Integer is
      Bisheriges_Minimum : Integer := Ind_Links;  -- gibt jeweils den Index an,
                      -- an dessen Position das kleinste Element des Teilfeldes
                      -- mit Indexbereich Ind_links..Ind_rechts steht
   begin
      for I in Ind_Links+1..Ind_Rechts loop
         if Feld(I)<Feld(Bisheriges_Minimum) then
            Bisheriges_Minimum := I;
         end if;
      end loop;
      return Bisheriges_Minimum;
   end Bestimme_Minindex;
```

```
   procedure Vertausche(Ind1,Ind2:Integer) is
      Hilfs_Int : integer;
   begin
      Hilfs_Int := Feld(Ind1);
      Feld(Ind1):= Feld(Ind2);
      Feld(Ind2):= Hilfs_Int;
   end Vertausche;

   Zeichen : character;

begin

   Put("Dieses Programm gibt Zahlen sortiert aus"); New_Line;

   Ausgabe;

   for I in 1..N-1 loop -- N-1, weil das größte Element dann automatisch
                        -- an der letzten Position steht
      -- hier folgt ein Sortierschritt
      Vertausche(I, Bestimme_Minindex(I,N));
      Ausgabe; -- nur zur Kontrolle
      Get_Immediate(Zeichen); -- liest ein einzelnes Zeichen
                              -- (ohne Return drücken zu müssen)
   end loop;

end Sortiere;
```

Fassen wir die hier neu eingeführten Elemente nochmal (in der Reihenfolge des Auftretens) zusammen:

- Deklarationen von Feldern haben die Syntax
  `<Var.-Bezeichner> : array ( <range> ) of <Datentyp> [ := <Ausdruck> ] ;`
- Der Ausdruck liefert die Werte zur Initialisierung. Die Werte sind durch Kommata getrennt und mit runden Klammern eingefasst. Achtung: Die Anzahl der Elemente muss mit der Kardinalität des Indexbereichs übereinstimmen.
- Will man nur bestimmte Feldelemente initialisieren, so kann man Paare der Form `<Index> => <Wert>` angeben und mit `others => <Wert>` den verbleibenden Elementen einen Wert zuordnen. Statt Werten können hier grundsätzlich auch beliebige Ausdrücke mit entsprechendem Ergebnis-Typ stehen.
- Der Zugriff auf das Feldelement mit Index `Index` erfolgt über `(Index)`, angehängt an den Variablennamen des Feldes.
- Für Zahlvariablen existiert das Attribut `'Img`, dass die zur Zahl zugehörige Zeichenkette liefert (dies ist leider in ObjectAda nicht implementiert).
- Der Operator „`&`“ verknüpft Zeichenketten und erlaubt so, sehr lange Zeichenketten (z.B. zur Ausgabe) übersichtlich auf mehrere Zeilen zu verteilen.

- `Put_Line` gibt Zeichenketten aus und führt danach einen Zeilenvorschub aus. Kleiner Nachteil: Dies ist nur für Zeichenketten definiert, jedoch nicht z.B. für Zahlen. Wer es einheitlich möchte, bleibt daher bei der bisher verwendeten Methode `Put(...); New_Line;`.
- Der Datentyp für einzelne Zeichen lautet `character`.
- Speziell zur Programmierung von Menüs oder, um wie hier einfach auf einen beliebige Tastendruck zu warten, eignet sich die Funktion `Get_Immediate(...)`, die ein Zeichen von der Tastatur einliest, ohne dass die Eingabe durch das drücken von <Return> bestätigt werden muss.

1.7.2 Weitere Anmerkungen und Besonderheiten bei der Verwendung von Feldern in Ada 95

Das Typkonzept von Ada ist bzgl. Arrays sehr streng. Insbesondere können keine Zuweisungen zwischen Feldern durchgeführt werden, wenn für die Felder nicht zuvor ein eigener Typ mittels `type` eingeführt wurde, z.B.:

```
type Dreierfeld is array (1..3) of integer;
feld1, feld2: Dreierfeld;
feld3, feld4: array (1..3) of integer;
begin
   feld1 := (3,4,7); -- ist möglich
   -- auch Zuweisungen der Form (1 => 3, others => 7),
   -- dies entspricht (3,7,7)
   feld3 := feld1; -- ist nicht möglich, obwohl die Dimensionen übereinstimmen
   feld2 := feld1; -- ist möglich, da eigener spezieller Typ definiert wurde
   feld4 := feld3; -- Zuweisungen zwischen anonymen Arrays sind nicht möglich
   feld2(2..3) := feld1(1..2); -- Zuweisungen von Teilfeldern sind ebenfalls
   -- nur bei gleichem, benannten Typ möglich - es reicht dabei,
   -- wenn die Größe des Indexbereichs übereinstimmt
```

Als Nebeneffekt tritt dies auch auf, wenn Arrays als Parameter bei Prozeduren oder Rückgabewerten von Funktionen verwendet werden sollen – auch hier muss man mittels `type` zunächst einen eigenen Typ einführen. Ada unterscheidet also zwischen den mittels `type` deklarierten *benannten* Arrays und den direkt (ohne den Zwischenschritt mit `type`) deklarierten *anonymen* Arrays.

Die Idee dahinter ist, dass z.B. ein `array (1..2) of integer` vielfältige Interpretationsmöglichkeiten hat und Zuweisungen somit nicht immer sinnvoll sind. Ein Tupel (5,7) könnte so sowohl als Bruch, als auch als Punkt in der Ebene oder als komplexe Zahl interpretiert werden. Definiert der Programmierer jedoch vorher einen Typ für Brüche, so kann der Übersetzer Fehlzuweisungen als formale Fehler erkennen und so zu lesbareren Programmen beitragen.

In Ada 95 lassen sich auch mehrdimensionale Arrays definieren. Dazu sind bei der Definition die verschiedenen Bereiche in den einzelnen Dimensionen durch Kommata zu trennen. Die Typ-Definition für ein Gitter mit 3 mal 3 Feldern, in dem jeweils ein Ganzzahl-Wert stehen soll, lautet in Ada z.B.

```
type Gitter is array (1..3,1..3) of integer;
```

Die Deklaration einer Variablen dieses Typs und Initialisierung mit einem Startwert könnte so aussehen:

```
Beispiel : Gitter := ((1,2,3),(3,4,5),(4,5,6));
```

Im Hauptprogramm kann dann z.B. mit `Beispiel(3,1)` auf das dritte Tupel, Index 1 (hier der Wert 4) zugegriffen werden.

Ada unterscheidet dies übrigens von einem Feld aus Feldern, wobei wir auch hier zur Deklaration den Zwischenschritt über einen eigenen Typ gehen müssen:

```
type Gitter is array (1..3) of array (1..3) of integer; -- geht so nicht
type Gitter_Reihe is array (1..3) of intger;
type Tolles_Gitter is array (1..3) of Gitter_Reihe; -- so geht es
Mein_Gitter : Tolles_Gitter := ((1,2,3),(3,4,5),(4,5,6));
-- die Initialisierung sieht hier wie beim 2-dimensionalen Array aus
```

Der Zugriff erfolgt dann über `Tolles_Gitter(3)(1)` auf das erste Element in der dritten Reihe. Mit `Tolles_Gitter(3)` greift man auf die komplette Reihe zu.

1.8 Das Blockkonzept in Ada 95 – oder – Parameterübergabemechanismen: Theorie und Praxis

Blöcke haben wir bisher in Form von Prozeduren und Funktionen kennengelernt. Diese stellen in sich abgeschlossene Programmteile dar, die jeweils eine bestimmte Aufgabe erledigen – wir sprechen hier von einem Kontrakt, der festlegt, was der Aufruf genau bewirkt und welche Vor- und Nachbedingungen bei dem Aufruf gelten sollen bzw. durch diese sichergestellt werden. Diese Vor- und Nachbedingungen sollten in Form von Kommentarzeilen im Programm und ggf. auch (bei größeren Softwareprojekten) in der Dokumentation festgehalten werden.

Beispiel Wurzelfunktion: Die Wurzel einer Zahl ist nur für nichtnegative Zahlen definiert. Es ist nun die Frage, ob das aufrufende Programm oder die Prozedur dafür verantwortlich ist, dass die Eingabe den Bedingungen entspricht. Dieses ist Teil des Kontrakts. Prüfen sowohl das aufrufende Programm als auch die Wurzelfunktion selbst, ob die Eingabe größer gleich null ist, so werden die Abfragen unnötig doppelt ausgeführt. Ist die Wurzelfunktion für die Prüfung zuständig, so ist z.B. auch eine Definition nötig, was passiert, wenn die Eingabe negativ ist.

Ada 95 stellt das Konzept des Blocks auch außerhalb von Prozeduren und Funktionen zur Verfügung. Will man z.B. ein Feld aufgrund einer Benutzereingabe dimensionieren, so wäre dies wie folgt möglich:

```
...
Get(x);
...
declare
  feld : array (0..x) of integer;
begin
  ...
end;
```

Blöcke können überall stehen, wo auch Anweisungen stehen können. Der Aufbau ist identisch dem von Prozeduren und Funktionen, außer dass Blöcke keine Parameterlisten und keine Rückgabewerte haben. Sie beginnen mit dem Schlüsselwort `declare`, es folgen wie bei Prozeduren und Funktionen Deklarationen, gefolgt von `begin`, Anweisungen und abgeschlossen durch `end;`.

Randbemerkung: Eine weitere Form von Blöcken haben wir mit den `for`-Schleifen kennengelernt – diese werden intern als eine Art `declare`-Block mit einer implizit lokal definierten Schleifen-Variable betrachtet. Dies ist auch der Grund, warum wir die Schleifen-Variablen nicht zu deklarieren brauchen und die Schleifen-Variable nach Verlassen der Schleife nicht mehr sichtbar ist. Als Besonderheit ist zu beachten, dass die Schleifen-Variable im Schleifen-Rumpf wie eine Konstante behandelt wird – man kann nur lesend darauf zugreifen. (Hinweis: Dies gilt mangels impliziter Schleifen-Variablen nicht für `while`-Schleifen!)

1.8.1 Parameterübergabe-Mechanismen

Wir haben bisher verschiedene Arten von Parametern kennengelernt:

- Parameter, deren Wert beim Aufruf übergeben wird (z.B. `Put(...)`)
- Parameter, die nach Aufruf einen Wert bekommen (z.B. `Get(...)`)

Wir haben noch nicht kennengelernt:

- Parameter, die einen Wert an eine Prozedur übergeben und von der Prozedur verändert werden.

Wir unterscheiden folgende Begriffe:

- Formalparameter – dies sind die Elemente der Parameterliste in der Prozedur- oder Funktionsdeklaration. Die Parameter in der Parameterliste in der Form <`Bezeichner`>`:`<`Datentyp`> durch Semikolons getrennt angegeben, Parameter gleichen Typs können durch Kommata getrennt zusammengefasst werden.
- Aktualparameter – dies sind die Werte der Ausdrücke oder Variablennamen, mit denen die Prozeduren und Funktionen aufgerufen werden.

Auf optionale Parameter, wie wir Sie bei den Ausgabe-Routinen `Put(...)` kennengelernt haben, gehen wir hier nicht näher ein (bei Bedarf kann man dies im Ada Reference Manual, 6.1, nachlesen).

Aus klassischer Sicht gibt es folgende drei Parameterübergabe-Mechanismen:

- call-by-value (es wird nur der Wert der Variablen übergeben)
- call-by-reference (es wird eine Referenz zu einer Variablen übergeben, d.h., die übergebene Variable wird automatisch mitverändert, wenn der formale Parameter in der Prozedur verändert wird)
- call-by-name (in der Prozedur wird der formale Parameter an allen Stellen durch den Namen der übergebenen Variablen ersetzt)

In den meisten Programmiersprachen sind die Konzepte call-by-value und call-by-reference umgesetzt. In Ada 95 ist die Sprechweise und Semantik der Übergabe-Mechanismen etwas anders.

Optional lässt sich für jeden formalen Parameter eine der folgenden Richtungsangaben machen:

- `in`, Eingangsparameter – dies ist auch die Defaulteinstellung, wenn man nichts angibt. Der übergebene Parameterwert wird im Rumpf der Prozedur als Konstante behandelt und kann nicht, wie in manchen anderen Programmiersprachen, wie eine lokale Hilfsvariable verwendet und verändert werden.
- `out`, Ausgangsparameter – diese haben zu Beginn noch keinen Wert und dürfen in der Prozedur nach einer Zuweisung beliebig verwendet werden. Wird die Prozedur verlassen, so wird der Inhalt dem Aktualparameter zugewiesen.
- `in out`, Durchgangsparameter – hier wird dem formalen Parameter bei Aufruf der Wert des Aktualparameters zugewiesen. Der formale Parameter kann in der Prozedur wie eine normale Variable benutzt und verändert werden. Wird die Prozedur verlassen, so wird der Inhalt dem Aktualparameter zugewiesen.

Bei den Richtungsangaben `out` und `in out` muss der Aktualparameter ein Variablenname sein (sonst ist keine Zuweisung des Wertes des Aktualparameters möglich). Bei `in`-Variablen kann der Aktualparameter ein beliebiger Ausdruck sein, der sich zu einem Wert des dem formalen Parameters entsprechenden Typs auswertet.

Die verschiedenen Richtungsangaben können in beliebiger Reihenfolge stehen.

Funktionen haben in Ada 95 grundsätzlich ausschließlich Eingangsparameter. Die Angabe von `in` ist optional – `out` oder `in out` können bei Funktionen nicht verwendet werden.

Übung: Beschreiben Sie die Syntax der Prozedur- und Funktions-Deklaration als EBNF oder Syntaxdiagramm.

1.8.2 Semantik von Prozeduren und Funktionen

Innerhalb von Prozeduren und Funktionen können wir – neben den formalen Parametern – ebenfalls Variablen und weitere Prozeduren deklarieren. Hier stellt sich die Frage, was erlaubt ist (Antwort: fast alles), welche Semantik dies dann hat (Antwort: ist alles exakt definiert – aber in Details leider nicht von allen Compilern einheitlich umgesetzt; die Abweichungen sind aber nicht tragisch), insbesondere, wenn Variablen in Prozeduren diegleichen Bezeichner haben, muss definiert werden, was in Ausdrücken und bei Zuweisungen passiert.

Dies führt auf die Begriffe

- Gültigkeit bzw. Lebensdauer und
- Sichtbarkeit

von Variablen, Prozeduren und Funktionen. Ist im folgenden von einem Block die Rede, so umfasst dies alle drei uns bekannten Möglichkeiten: Prozeduren, Funktionen und `declare`-Blöcke.

Eine Variable oder Prozedur bzw. Funktion ist **gültig** bzw. **lebendig** vom Zeitpunkt der Deklaration an bis zum Ende des umschließenden Blocks (wir fassen das Hauptprogramm auch als Block auf).

Innerhalb eines Blocks kann ein Variablenname nicht zweimal bei einer Deklaration verwendet werden.

Die in einem Block erklärten Variablen heißen **lokal**, sie sind außerhalb der Block-Deklaration nicht **sichtbar**. Wir können also für unsere Teilaufgaben eigene Variablen erklären, die sich mit gleichen Namens, die in anderen Prozeduren erklärt werden, nicht beißen. Diese müssen dabei nicht einmal vom selben Typ sein. Eine Variable `zahl` kann so in einer Prozedur vom Typ `integer` sein und in einer anderen Prozedur vom Typ `float` – Deklarationen in verschiedenen Blöcken referenzieren verschiedene Speicherstellen.

Bei jedem Aufruf einer Prozedur oder Funktion wird ein neuer Satz lokaler Variablen angelegt und beim Verlassen automatisch wieder entfernt. Wir kommen bald im Zusammenhang mit Rekursion wieder darauf zurück.

Dies ist nicht der einzige Fall, bei dem Variablen mit gleichem Namen auftreten können. Prinzipiell können Deklarationen geschachtelt sein, z.B. eine Deklaration im Deklarationsteil einer anderen Prozedur. Variablen, die in einem umfassenden Block deklariert wurden, sind weiterhin sichtbar – diese heißen **globale** Variablen (der Block muss umfassend sein, Variablen in parallel liegenden Blöcken sind nicht sichtbar).

Wenn lokale und globale Variablen denselben Namen haben, passiert erstmal nichts. Wir können aber, solange die lokale Variable gültig ist, nicht auf die globale Variable mit gleichem Namen zugreifen, die globale Variable nennt man dann **verschattet**.

Betrachten wir nun die Semantik des Aufrufs einer Prozedur oder Funktion genauer:

- Die Abarbeitung des derzeitigen Programmteils wird unterbrochen.
- Es wird überprüft, ob die Werte der Aktualparameter zu den Typen der zugehörigen Formalparameter kompatibel sind.
- Für die Formalparameter und lokalen Variablen der aufgerufenen Prozedur wird ein neuer Speicherbereich zur Verfügung gestellt und die Werte der Aktualparameter den Formalparametern zugewiesen.
- Die Anweisungen der aufgerufenen Prozedur werden ausgeführt, bis wir beim Ende ankommen oder die Prozedur durch eine `return`-Anweisung vorzeitig verlassen wird.
- Gab es formale `out`- oder `in-out`-Parameter, so wird den entsprechenden Aktualparametern der Wert der formalen Parameter zugewiesen.

  Bei Funktionen merken wir uns den über die `return`-Anweisung übermittelten Wert.
- Der Speicherbereich für die lokalen Variablen und formalen Parameter wird gelöscht.
- Die Abarbeitung des vorhin abgebrochenen Programmteils wird an der Stelle nach Aufruf der Prozedur fortgesetzt bzw. der von der Funktion zurückgegebene Wert weiterverarbeitet.

Bei Prozeduraufrufen entspricht dies dem Ersetzen des Prozeduraufrufs durch einen `declare`-Block mit entsprechenden lokalen Variablen, die mit den Werten der Aktualparameter ersetzt

werden – es wird quasi der Programmtext der aufgerufenen Prozedur an die entsprechende Stelle kopiert. Man spricht daher auch von der *Kopierregel*. In der Praxis werden solche textuellen Ersetzungen (und das somit notwendige erneute Übersetzen des Programms) natürlich nicht durchgeführt. Stattdessen werden in einem bestimmten Speicherbereich des Programms (dem sogenannten Kellerspeicher) u.a. die neuen Variablen angelegt und diese dort nach Beenden der zugehörigen Prozedur automatisch wieder gelöscht. Details hierzu können z.B. im Duden Informatik nachgelesen werden.

1.8.3 Goldene Regeln beim Entwurf von Prozeduren und Funktionen und ein Beispiel, wie man es nicht machen sollte

Als Richtlinie für verständliche und strukturierte Programme dienen folgende Regeln für Prozeduren und Funktionen:

- Umfang: überschaubar – in der Regel nicht mehr als eine Bildschirmseite
- Vermeiden Sie Zugriffe auf globale Variablen und spezielle Daten außerhalb der Prozedur.

Welche Vorteile hat dies?

- Wie die Aufgabe einer Prozedur oder Funktion erledigt wird, sollte nach außen nicht sichtbar sein. Dies ermöglicht ein Top-Down-Vorgehen: das Gesamtprodukt wird in Teile zerlegt, von denen zunächst nur gesagt wird, was sie tun werden; die Implementierung kommt später (Beispiel: Prozedur `Vertausche` im Sortierbeispiel).
- Ein weiterer Grund für dieses „Information-Hiding“ ist, dass sich das „wie“ der Umsetzung noch ändern kann. Wenn wir später ein schnelleres Verfahren für eine Teilaufgabe finden, so ist es hilfreich, wenn dies nur Änderungen in der entsprechenden Prozedur nach sich zieht und wir nicht im gesamten Programm suchen müssen, wo die Änderung überall nachvollzogen werden muss.
- Wir vermeiden Seiteneffekte, die durch Veränderung globaler Variablen auftreten. Diese sind oft eine Quelle schwer zu findener Programmierfehler.

Hier ein Beispiel, um etwas zu üben: Achtung! Dies ist nur ein Übungsbeispiel, es zeigt auch, wie man nicht programmieren sollte!

```
with ada.text_io;
use ada.text_io;

procedure glokal is
   a,b:integer := 2;
   c:integer := 7;

  procedure confu(d,b: in out integer) is
     c: integer := 3;
  begin
     b:=d+a;
```

```
      a:=b+d;
   end confu;

   procedure sion(c,a: in integer) is
      d:integer := a+b;
   begin
      b:=c+d;
   end sion;

   procedure ausgabe(a,b,c: integer) is
   begin
      put_line(a'img & b'img & c'img);
   end ausgabe;

begin
   ausgabe(c,a,b);
   confu(b,a);
   ausgabe(b,c,a);
   sion(a,c);
   ausgabe(a,b,c);
end glokal;
```

Die Sichtbarkeit einer Variablen kann man sich als Stapel vorstellen: Wird ein Variablenname noch nicht verwendet, so legt man bei der entsprechenden Deklaration einen Wertebehälter neu an. Wird eine Variable lokal deklariert, deren Name bereits in einem umfassenden Block verwendet wurde, so fügen wir einen neuen Wertebehälter hinzu (der Typ muss nicht identisch sein) – dieser wird auf den bereits vorhandenen Stapel der Wertebehälter mit diesem Namen gestellt. Die darunter liegenden Wertebehälter sind verschattet und nicht zugreifbar. Bei Beenden eines Blockes werden die in diesem Block deklarierten Wertebehälter von den zugehörigen Stapeln entfernt (dieses ist immer der oberste Wertebehälter, selbst überlegen!).

Randbemerkung: In Sonderfällen (folgendes Beispiel) muss man jedoch aufpassen:

```
declare
   x : integer := 7;
   procedure Falle is
   begin
      Put(x);
   end;
begin
   declare
      x : integer := 3;
   begin
      Falle;
   end;
end;
```

Die Ausgabe hier ist 7! In der aufgerufenen Prozedur `Falle` wird die Variable `x` ausgegeben – dies ist jedoch das im äußeren Block deklarierte `x`. Der innere Block ist von der Prozedur `Falle` aus nicht sichtbar, somit wird von `Falle` aus auf das `x` mit dem Wert `7` zugegriffen.

1.9 Rekursion – oder – wie sich Probleme durch sich selbst lösen

Als Beispiel für die Rekursion betrachten wir folgende Funktion `fibo` zur Berechnung der Fibonaccizahlen. Diese sind definiert durch `fibo(0)=fibo(1)=1` und `fibo(n)=fibo(n-1)+fibo(n-2)` für $n \geq 2$. In Ada 95 formuliert lautet dies:

```
function fibo(n:natural) return natural is
begin
  if n<=1 then
    return 1;
  else
    return fibo(n-1)+fibo(n-2);
  end if;
end;
```

Versuchen Sie mit den in Abschnitt 1.8 eingeführten Regeln nachzuvollziehen, was der Aufruf von `Put(fibo(4));` bewirkt.

Mit den Richtungsangaben lassen sich manche Funktionen und Prozeduren elegant formulieren. Wir betrachten hier nun einige weitere rekursive Beispiele, die ganz nebenbei auch noch ein paar neue Ada-Feinheiten in Aktion zeigen:

```
type vektor is array (integer range <>) of integer;
```

Dies definiert einen Array-Typ, dessen Bereichsgrenzen erst bei Deklaration der Variablen festgelegt werden müssen (das „`<>`“ spricht man „Box“). Will man ein Array mit Indexbereich 1 bis 7, so lautet die Deklaration für eine Variable mit Namen `glofeld` (hier gleich mit Initialisierung der Werte):

```
glofeld : vektor(1..7) := (5,7,2,8,3,9,1);
```

In diesem Beispiel hätte man auch das `(1..7)` weglassen können, denn durch die Initialisierung ist die Anzahl der Elemente ersichtlich (die Verwendung von `others=>...` ist bei der Initialisierung ohne Angabe des Indexbereichs nicht möglich – selbst überlegen, warum dies so ist).

Gibt man keinen Indexbereich an, so beginnt dieser beim kleinsten möglichen Index – hier also bei -2147483648 (dies ist `integer'first`). Man hätte oben allerdings auch `positive range <>` schreiben können, dann hätte der Indexbereich bei 1 begonnen (außer man gibt ihn explizit an).

Eine wichtige Anwendungsmöglichkeit solcher Arrays mit variablen Bereichsgrenzen ist als Typ in Parameterlisten. Wir betrachten hier als Beispiel die rekursive Berechnung des Maximums eines Feldes – dieses ist entweder das erste Element oder das Maximum des restlichen Arrays:

```
function Maximum (Feld:Vektor) return integer is
begin
  if Feld'length=1 then
    return Feld(Feld'first);
  else
    return integer'max(Feld(Feld'first),Maximum(Feld(Feld'first+1..Feld'last)));
  end if;
end;
```

Dabei sind als Aktualparameter nun alle Varianten vom Typ `Vektor` erlaubt – wir müssen also unsere Funktion nicht für jede Array-Größe neu definieren. Der formale Parameter `Feld` erbt die Bereichsgrenzen vom Aktualparameter. Obwohl dies keine vorab bekannten Werte sind, können wir in Ada über Attribute sinnvoll mit solchen Parametern arbeiten:

- `<Bezeichner>'first` gibt den kleinsten Index des Arrays an
- `<Bezeichner>'last` gibt den größten Index des Arrays an
- `<Bezeichner>'range` gibt den Bereich des Arrays an – z.B. zur Ausgabe:
  `for i in Feld'range loop Put(Feld(i)); end loop;`
- `<Bezeichner>'length` gibt die Anzahl der Elemente des Arrays an

Neben diesen haben wir noch das schöne Ada-Konstrukt `<Typname>'max` benutzt. Dieses ist eine für jeden skalaren Typen implizit definierte Funktion, die zwei Parameter des Typs erwartet und den größeren der beiden als Ergebniswert zurückgibt (analog ist auch die Funktion `<Typname>'min` definiert). Skalare Typen sind die uns bereits bekannten Zahltypen sowie Aufzählungstypen (werden wir noch kennenlernen).

Obige Funktion zur Berechnung des größten Werts eines Arrays geht nun folgendermaßen vor: Wenn das Array nur aus einem Element besteht, dann ist dies der größte Wert – sonst ist es das Maximum vom ersten Element und dem größten Element des restlichen Arrays. Wir haben hierzu die Funktion `integer'max` verwendet und rekursiv die Funktion `Maximum` mit dem Rest des Arrays aufgerufen.

Übungsaufgabe für Fortgeschrittene: Schauen Sie sich das Ada-Paket `ada.calendar` und/oder `ada.real_time` an und benutzen Sie es, um experimentell die Laufzeit obiger `Maximum`-Funktion in Abhängigkeit der Array-Größe zu ermitteln (da die Laufzeiten sehr kurz sind, wiederholen Sie das Programm mit Hilfe einer for-Schleife ausreichend oft, um verwertbare Ergebnisse zu bekommen).

Hinweis an alle: Obige Funktion zur Maximum-Bestimmung ist eine sehr elegante und kurze Umsetzung des Problems in ein Programm – bzgl. der Laufzeit hat sie jedoch einen Haken, da in jeder Rekursionsebene das restliche Array als Parameter übergeben wird – wir kommen auf dieses Beispiel nochmal zurück, wenn wir nach Weihnachten Zeiger kennengelernt haben und damit dieses Problem beheben können.

Auch unser Sortierprogramm lässt sich analog mit wenigen Zeilen beschreiben: Suche das Minimum, tausche es an die erste Position und sortiere dann das restliche Array – in Ada lautet das dann so (die Funktion `Bestimme_Minindex(Feld)` bestimme den Index des kleinsten Elementes des Arrays `Feld` – diese ist nicht durch Ada gegeben, man muss Sie selbst schreiben): Nehmen Sie dazu die Funktion `Bestimme_Minindex` aus dem Sortierbeispiel in Abschnitt 1.7 (mit zu ändernden formalen Parametern) oder wandeln Sie obige Funktion `Maximum` entsprechend um, dass der Index des kleinsten Elementes ermittelt wird anstelle des Werts des größten Elements:

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

```
procedure Sortiere (Zusort: in out Vektor) is
begin
  if Zusort'first<Zusort'last then -- sonst ist nur ein Element im Array
    Vertausche(Zusort(Zusort'first),Zusort(Bestimme_Minindex(Zusort)));
    Sortiere(Zusort(Zusort'first+1..Zusort'last));
  end if;
end;
```

Auch hier lässt sich der Overhead durch die Übergabe des restlichen Arrays mit Hilfe von Zeigern ($\rightsquigarrow$ nach Weihnachten) vermeiden.

1.9.1 Ein allgemeines Schema zur Rekursion

Obige Beispiele zur Rekursion haben viele strukturelle Gemeinsamkeiten, die wir noch zu einem allgemeinen Schema zusammenfassen wollen:

- Ist die Problemgröße sehr klein (in der Regel wird hier die Größe 1 oder 0 verwendet), so wird das Problem direkt gelöst – dies ist die *Abbruchbedingung* für die Rekursion.
- Ansonsten lösen wir das Problem, indem wir es auf eines oder mehrere kleinere Probleme derselben Art zurückführen und diese Teillösungen dann mit (in der Regel) wenigen Schritten zur Lösung des gesamten Problems zusammenfügen – dies ist das *Bildungsgesetz* für die Rekursion.

Im Beispiel der Fibonaccizahlen ist die Abbruchbedingung `n<=1` und sonst lösen wir das Problem für die kleineren Problemgrößen `n-1` und `n-2`. Bei der Maximum-Bestimmung und dem Sortieralgorithmus ist die Abbruchbedingung das Erreichen eines 1-elementigen Feldes und sonst die Reduzierung auf die Problemlösung für ein Feld mit einem Element weniger und einem zusätzlichen Vergleich bzw. Vertauschungsschritt.

Wir geben hier noch das Schema in einer halbformalen Darstellung an:

```
procedure/function <Etwas_Rekursives>(<Parameter>) [return <Datentyp>] is
   [<Deklarationen>]
begin
   if <Abbruchbedingung> then
      <Löse das Problem direkt>
   else
      <Löse ein oder mehrere kleinere Teilprobleme>
      -- dies beinhaltet rekursive Aufrufe von <Etwas_Rekursives>(...)
      <Füge die Teillösungen zur Gesamtlösung zusammen>
   end if;
end;
```

Versuchen Sie, sich stets bei rekursiven Lösungsansätzen an dieses Schema zu halten. Wenn – wie bei obigem Sortierbeispiel – beim Abbruch garnichts zu tun ist, so kann man natürlich (wie oben geschehen) die Abbruchbedingung entsprechend umformulieren und die rekursiven Schritte zur Lösung in den `then`-Zweig einfügen.

1.10 Zeichen und Zeichenketten – oder – Fallunterscheidungen ohne `if`

Bevor wir zu Zeichenketten kommen, lernen wir zunächst noch kurz den Datentyp für einzelne Zeichen kennen : `character`. Literale von diesem Typ werden immer in einfachen(!) Hochkommata eingeschlossen, z.B. `'A'`. Wir können diese mit den Ein-/Ausgabe-Routinen `Get` und `Put` bearbeiten.

Auch zum Datentyp `character` gibt es in Ada 95 Attribute. Die beiden wichtigsten sind:

- `Character'Pos('A')` gibt die Position an, an der der Buchstabe `A` in der Aufzählung des Datentyps `Character` steht – Ergebnis: 65.
- `Character'Val(65)` gibt das Zeichen an, das an der 65-ten Position in der Aufzählung des Datentyps `Character` steht – Ergebnis: A.
- Es gilt `zahl = Character'Pos(Character'Val(zahl))` und `zeichen = Character'Val(Character'Pos(zeichen))`.

Die Attribute `Pos` und `Val` sind für alle diskreten Typen (z.B. auch alle Ganzzahl-Typen) definiert, werden in der Regel aber nur bei `Character` und ggf. bei Aufzählungstypen (kommen später) verwendet.

Speziell, wenn man interaktive Menüs auf Textbasis gestalten will, ist es hilfreich, ein Zeichen einlesen zu können, ohne danach die Return-Taste drücken zu müssen. Dazu dient die Prozedur `Get_Immediate(<Bezeichner>)`. Sowie eine Taste gedrückt wird, bekommt die Variable `<Bezeichner>` den entsprechenden Wert.

Die Weiterverarbeitung des Zeichens bei einem Menü geschieht in der Regel in Form einer Fallunterscheidung. Eine Möglichkeit solcher Fallunterscheidungen haben wir mit dem `if-then-elsif-else`-Konstrukt kennengelernt, eine zweite Möglichkeit, die `case`-Anweisung folgt nun.

Als Beispiel wollen wir das eingelesene Zeichen untersuchen.

```
Zeichen : character;
...
Get_Immediate(Zeichen);
...
case Zeichen is
   when 'a'             => Put("Ein a");
   when 'b' | 'e' | 'h' => Put("Ein b, e oder h");
   when 'k'..'r' | 'z'  => Put("Irgendwas zwischen k und r oder ein z");
   when others          => Put("Das hab ich noch nie gesehen");
end case;
```

Nach dem Schlüsselwort `case` folgt der Bezeichner der Variablen, deren Inhalt untersucht werden soll, und das Schlüsselwort `is`. Sodann folgt eine Auflistung von Fällen, jeder eingeleitet durch `when`, gefolgt von einzelnen Literalen und/oder Bereichen des entsprechenden Datentyps, die jeweils durch einen senkrechten Strich voneinander getrennt sind. Jeder Fall wird durch das `=>` und einer Anweisungsfolge (zumindest eine `null`-Anweisung, wenn in dem betreffenden Fall nichts auszuführen ist) abgeschlossen. Optional kann noch für noch nicht aufgeführte Möglichkeiten mit `when others` ebenfalls eine Anweisungsfolge angegeben werden.

Bei Verwendung der `case`-Anweisung sind zwei Dinge zu beachten:

- Jeder mögliche Wert des Datentyps muss in der `case`-Anweisung aufgeführt sein, d.h., der Fall `when others` darf nur dann entfallen, wenn alle möglichen Werte zuvor aufgeführt wurden.
- Die Werte aus je zwei Fällen müssen disjunkt sein, d.h., jede mögliche Eingabe muss eindeutig zu einen der angegeben Fälle zuzuordnen sein.

In manchen Fällen ist eine `case`-Anweisung eleganter und einfacher zu formulieren (insbesondere auch lesbarer und besser zu verstehen), in anderen ist die Möglichkeit über `elsif` einfacher. Hier hilft zur Auswahl nur die Erfahrung. Lösen Sie zur Übung folgende Aufgabe: Lassen Sie ein Zeichen eingeben und unterscheiden Sie sodann folgende Fälle:

- klein geschriebener Vokal,
- groß geschriebener Vokal,
- klein geschriebener Konsonant,
- groß geschriebener Konsonant,
- klein geschriebener Umlaut,
- groß geschriebener Umlaut,
- eine der Ziffern 0 bis 9,
- etwas anderes.

Sie werden dabei die Vor- und Nachteile der `case`-Anweisung kennenlernen.

Anmerkung zum Abschluss der `case`-Anweisung: Diese lässt sich auf jeden diskreten Datentyp anwenden, z.B. `character`, `integer` und seine Subtypen, selbstdefinierte Ganzzahlbereiche, Aufzählungstypen (kommen später). Statt einem Variablennamen kann man auch einen Ausdruck angeben, so ist z.B. folgendes möglich (für das Wandeln eines Zeichens in den zugehörigen Großbuchstaben mittels `To_Upper(Zeichen)` wird das Paket `Ada.Character.Handling` benötigt):

```
case To_Upper(Zeichen) is
   when 'A'    => Put("a oder A");
   when others => Put("was anderes");
end case;
```

Anmerkung zum Abschluss des Datentyps `character`: Die oben aufgeführten Zeichen-Bereiche lassen sich auch in Boole'schen Ausdrücken (z.B. `if Zeichen in 'a'..'z' then ...`) und als Laufvariablen von `for`-Schleifen verwenden (z.B. `for z in character range 'a'..'z' loop ...`). Es ist sogar möglich solche Bereiche als Indexmenge für Arrays zu verwenden (z.B. `Koordinaten : array (character range 'x'..'z') of float;`, Zuweisung dann z.B. mittels `Koordinaten('x'):=4.2;`).

Zeichenketten in Ada 95: Zeichenketten sind vom Datentyp `string`. Dieser ist genau genommen ein Array mit variablen Bereichsgrenzen, wie wir ihn ähnlich weiter oben schon kennengelernt haben:

```
type String is array (Positive range <>) of Character;
```

Damit ist auch klar, dass für Ada das Zeichenliteral `'A'` (vom Typ `character`) und das Zeichenkettenliteral `"A"` (vom Typ `array (1..1) of character`) zwei grundsätzlich verschiedene Dinge sind – wir sehen hier wieder das strenge Typkonzept in Ada.

Da Strings so wichtig in einer Programmiersprache sind, gibt es für diese – obwohl es eigentlich Arrays sind – die Möglichkeit Literale in doppelten Hochkommata einzuschließen. Sonst müsste man statt `"Hallo"` das unschön aussehende `('H','a','l','l','o')` verwenden.

Wir haben Strings bisher nur als Literale kennengelernt, als wir Text über die Prozedur `Put` ausgegegeben haben, z.B. `Put("Das doppelte Hochkomma '""' muss man doppelt angeben.");` erzeugt die Ausgabe `Das doppelte Hochkomma '"' muss man doppelt angeben.`. Außerdem haben wir schon gelernt, wie wir mehrere Strings konkatenieren (d.h. zusammenfügen) können – der Operator „`&`" erledigt dies.

Wir werden hier am Beispiel der Strings noch einige Möglichkeiten demonstrieren, die auch bei allen anderen Arten von Arrays möglich sind.

Eines vorweg: Die maximale Größe eines Strings wird in Ada bei der Deklaration der Variablen festgelegt. Eine Variable `text1 : string(1..20);` hat also immer 20 Zeichen. Wird die Länge durch eine Initialisierung implizit festgelegt, z.B. `text2 : string := "Das ist ein Text.";` so hat die Variable `text2` von nun an den Indexbereich `1..17` und fasst somit 17 Zeichen – ein nachträgliches ändern des Indexbereichs ist (wie auch bei anderen Array-Typen) nicht möglich. (Fortgeschrittene mögen sich das Paket `Ada.Strings.Unbounded` anschauen und mit diesem arbeiten – wir werden die Verarbeitung von Zeichenketten hier nicht weiter vertiefen.)

Möglichkeiten in Beispielen:

- Sei `Text : string := "Mathematik";` deklariert. Die Zuweisung `Text(1..5) := "Infor";` ersetzt den Inhalt an den Indizes 1 bis 5, `Put(Text);` gibt dann das Wort `Informatik` aus.
  - Dies ist auch bei Zahlen möglich, z.B. `feld(4..6) := (1,2,3);`
  - Man beachte auch hier die Unterscheidung zwischen `character` und `string` – welche der vier Zuweisungen sind erlaubt?
    - `text(3) := 'A';`
    - `text(3) := "A";`
    - `text(3..3) := 'A';`
    - `text(3..3) := "A";`
- Konkatenation mit dem Operator „`&`": Außer in Ausdrücken (z.B. bei der Ausgabe) ist der Operator auch bei Zuweisungen möglich: `text := "Lewan" & "dowski";` – die Länge des Strings muss dabei stimmen, ansonsten müsste man hier `text(1..11) := "Lewan" & "dowski";` schreiben (vorausgesetzt, die Variable `text` umfasst mindestens 11 Zeichen). Dies erlaubt z.B. die Zuweisung sehr langer Strings, die nicht auf einer Zeile Platz finden.

– Auch dies ist bei Zahlen möglich. Mit der Deklaration `feld1 : vektor(1..7); feld2 : vektor(3..6);` wäre eine Zuweisung `feld1 := feld2 & (7,12,23);` erlaubt. (Randbemerkung: Außer der Anzahl der zugewiesenen Elemente muss hier auch der Typ übereinstimmen – dies kann also nur bei Arrays mit variablen Bereichsgrenzen (z.B. `(integer range <>)`) funktionieren.)

Zwei Attribute in Zusammenhang mit Strings wollen wir hier nicht verheimlichen. Diese dienen in gewissem Sinne zur Typ-Umwandlung von und nach `integer`

- `integer'image(...)` liefert den String zum Integer-Ausdruck – die Kurzform `'img`, die auch direkt auf Objekte vom Typ `integer` angewendet werden konnte, haben wir schon kennengelernt. Während letztere in Object Ada nicht zur Verfügung steht, ist die Variante `integer'image(...)` in allen Ada 95 Übersetzern vorhanden.
- `integer'value(...)` wandelt einen String in den zugehörigen Integer-Wert.
- Analog gibt es diese Attribute auch für `float`-Zahlen: `float'image` und `float'value`.

Im Rahmen dieser Vorlesung (und als Vorbereitung auf die Einführung in die Informatik II) reicht dieses Wissen zu Strings in Ada 95 aus. Interessierte mögen sich über das Internet oder Bücher weitere Möglichkeiten zur Zeichenketten-Verarbeitung mit Ada aneignen.

1.11 Übungsaufgaben

Zum Teil wird in diesen Übungsaufgaben auch noch zusätzlicher Stoff vermittelt. Die Inhalte der Übungen gehören zu den klausurrelevanten Themen dazu.

1. **Summe der natürlichen Zahlen:** In der Vorlesung wurde eine Funktion zur Berechnung der Fakultät geschrieben. Dabei wurden über das Attribut `integer'last` Fehler durch Überschreiten des erlaubten Zahlenbereichs abgefangen. Wir wollen dieses Prinzip nun auf eine andere Problemstellung übertragen.

   (a) Schreiben Sie ein Programm, das die Summe der natürlichen Zahlen 1+2+3+...+n für eine einzugebene Zahl n berechnet. Dabei soll ein Überschreiten des Zahlenbereichs im Programm abgefangen werden.

   (b) Eine Anekdote über den Mathematiker Gauß erzählt, dass er in der Grundschule in Mathematik die Aufgaben immer so schnell gerechnet hat, dass der Lehrer ihm die Aufgabe stellte, die Zahlen von 1 bis 50 zu addieren, in der Hoffnung, dass so der junge Gauß einige Zeit beschäftigt sei. Dieser lieferte die korrekte Antwort 1275 jedoch nach ungewöhnlich kurzer Zeit: Er hatte sich die Zahlen 1 bis 50 einmal aufsteigend und einmal absteigend aufgeschrieben und gesehen, dass die übereinanderstehenden Zahlen jeweils 51 ergaben.

   ```
    1  2  3  ...  48 49 50
   50 49 48  ...   3  2  1
   ```

   Die Summe der Zahlen 1 bis 50 war also die Hälfte von 50 mal 51, also 1275.
   Geben Sie eine allgemeine Formel für die Summe der natürlichen Zahlen

   $$\sum_{i=1}^{n} i$$

   an und beweisen Sie deren Korrektheit mit vollständiger Induktion.

(c) Verwenden Sie die Formel aus Teil (b) und schreiben Sie damit Ihr Programm aus Teil (a) neu.

(d) Geben Sie an, mit welchen Zahlen Sie Ihr Programm getestet haben, bis Sie sich sicher waren, dass es korrekt arbeitet. Begründen Sie Ihre Antwort.

2. **Freitag der 13. – Teil 2:** Wenn Programme umfangreicher werden, bieten Prozeduren und Funktionen eine Möglichkeit Programme strukturiert aufzubauen und den Ablauf in überschaubare Teile zu gliedern.

   Wir wollen hier nun ein Programm schreiben, das ein Datum einliest, den Wochentag an diesem Datum berechnet und diesen dann ausgibt. Der heute gültige Gregorianische Kalender beginnt am Freitag, 15. Oktober 1582. Informationen zum Gregorianischen Kalender finden Sie z.B. unter `http://de.wikipedia.org/wiki/Gregorianischer_Kalender`

   (a) Überlegen Sie sich zunächst, welche Bausteine Sie benötigen. Das Hauptprogramm sollte nur folgende Befehle enthalten:

```
begin
   Eingabe_Datum; -- liest Werte für die Variablen Tag, Monat und Jahr ein
   Wochentag := Berechne_Wochentag(Tag,Monat,Jahr);
   Ausgabe_Wochentag(Wochentag);
end;
```

   Zumindest die Funktion `Berechne_Wochentag` wird sich sicherlich in kleinere Bausteine aufteilen. Solch ein Baustein sollte nie so groß werden, dass er nicht mehr auf eine (Bildschirm-)Seite passt.

   (b) Implementieren Sie Ihre Bausteine aus Teil (a). Beachten Sie dabei, dass das eingegebene Datum gültig sein muss, insbesondere soll es nach dem 15. Oktober 1582 liegen.

   (c) Wir werden später im Semester noch lernen, wie man eigene Datentypen, z.B. für ein Datum, definiert. Überlegen Sie sich, welche Konstrukte Ihnen bei der Programmierung dieser Aufgabe gefehlt haben, die aus Ihrer Sicht das Programm übersichtlicher, besser lesbar oder einfacher zu schreiben gemacht hätten.

   (d) **Zusatzaufgabe**: Vergleicht man die Wochentage vom 1.1.2000 und 1.1.2400, so sieht man (unter Berücksichtigung der Schaltjahrregeln), dass sich der Kalender alle 400 Jahre bzgl. der Wochentage exakt wiederholt.

   Untersuchen Sie, wie oft der 13. eines Monats auf einen Montag, Dienstag, . . . oder Sonntag fällt.

3. **Typ-Umwandlungen:** Die Typ-Umwandlung von `float` nach `integer` rundet mathematisch, d.h. ab „.5“ wird aufgerundet. In manchen Anwendungen möchte man jedoch einfach die Nachkommastellen ignorieren und mit einer `integer`-Zahl weiterrechnen. Schreiben Sie solch eine Funktion zur Typ-Umwandlung von `float` nach `integer`. Überlegen Sie sich ausreichend Testfälle und fügen Sie die entsprechenden Ausgaben im Hauptprogramm in der Form „`Die Zahl 4.567 lautet ohne Nachkommastellen 4`“ mit ein.

4. **Aufwandsabschätzungen:** Geben Sie für folgendes Programmfragment die Anzahl der Schritte abhängig von der Eingabe `N` an (jeder Vergleich, jede Zuweisung, jeder Operator, jeder Prozedur- und Funktionsaufruf ... soll jeweils als ein Schritt gezählt werden).

```
...
  N : natural;

  function SoWhat (Maxi : natural) return natural is
    Sum : natural := 0;
  begin
    for I in 1..Maxi loop
      Sum := Sum + I;
    end loop;
    return Sum;
  end SoWhat;

begin
  Get(N);
  while N>0 loop
    Put(SoWhat(N));
    N := N-1;
  end loop;
end;
```

5. **Rolle Rückwärts**

   (a) Schreiben Sie eine rekursive Prozedur, die solange Zahlen einliest, bis die Zahl 0 eingeben wurde, und die eingegebenen Zahlen danach in umgekehrter Reihenfolge wieder ausgibt. Bei Eingabe der Zahlen 5 <Return>, 12 <Return>, 7 <Return>, 0 <Return> soll die Ausgabe 7 12 5 lauten. Zur Lösung dieser Aufgabe sind die bisher gelernten Sprachelemente von Ada 95 ausreichend!

   Fügen Sie Ihre Idee in den Programmkopf als Kommentarzeilen mit ein und kommentieren Sie Ihr Programm. Achten Sie auch auf sinnvolle Einrückungen.

   (b) Bei der einfachsten Lösung der Teilaufgabe (a) sind die eingebenen Zahlen nach der Ausgabe verloren. Wie müssten Datenstrukturen aussehen, um die Zahlen danach noch weiterverarbeiten zu können? Muss die Anzahl der einzugeben Zahlen dazu vorher bekannt sein? Skizzieren Sie eine mögliche Lösung. (Eine Variante werden wir bald in der Vorlesung kennenlernen, eine weitere später im Semester)

6. **Phythagoräische Zahlen-Tripel:** Der Satz von Phythagoras besagt, dass in einem rechtwinkligen Dreieck die Summe der Quadrate der Katheten gleich dem Quadrat der Hypothenuse ist. Im Allgemeinen sind in einem rechtwinkligen Dreieck nicht alle drei Kantenlängen ganzzahlig. Es gibt jedoch Beispiele, bei denen dies der Fall ist, z.B. $3^2 + 4^2 = 5^2$.

   (a) Schreiben Sie ein Programm, das eine Zahl n einliest und alle Zahlen-Tripel (a, b, c) mit $1 \leq a \leq b \leq c \leq n$ ausgibt. Für $n = 20$ soll die Ausgabe z.B. so aussehen:

```
Berechnung aller Phythagoraeischen Zahlen-Tripel (a,b,c) mit 1<=a<=b<=c<=20:
   1. Zahlen-Tripel: (3,4,5)
   2. Zahlen-Tripel: (5,12,13)
   3. Zahlen-Tripel: (6,8,10)
   4. Zahlen-Tripel: (8,15,17)
   5. Zahlen-Tripel: (9,12,15)
   6. Zahlen-Tripel: (12,16,20)
```

(b) **Zusatzaufgabe (sehr schwer, nur für Tüftler)**: Der Aufwand bei Teil (a) ist bei der einfachen Lösung proportional zu n^3. Wir wollen dies hier nun wesentlich beschleunigen.

Für alle ungeraden n ist $(n, (n^2 - 1)/2, (n^2 + 1)/2)$ ein Phythagoräisches Zahlen-Tripel (d.h. auch, es gibt unendlich viele solcher Zahlen-Tripel, die nicht nur Vielfache voneinander sind), dabei beträgt die Differenz zwischen der zweiten und dritten Zahl stets genau 1.

Entwickeln Sie einen Ausdruck, der die Phythagoräischen Zahlen-Tripel mit Differenz 2 zwischen der zweiten und dritten Zahl beschreibt, verallgemeinern Sie dies (Hinweis: Die so gefundenen Zahlen sind nicht immer ganzzahlig, bei der Ausgabe müssen solche Tripel dann ausgelassen werden).

Schreiben Sie mit dem so entwickelten Ausdruck ein Programm zur Berechnung der gesuchten Zahlen-Tripel. Vergleichen Sie experimentell die Laufzeiten von der einfachen und Ihrer neuen Lösung, fügen Sie den Vergleich als Kommentarzeilen mit ein. Können Sie auch mathematisch abschätzen, wie groß der Aufwand zur Berechnung nun ist?

7. **Zahlen raten:** Ein einfaches Spiel ist, sich eine ganze Zahl (z.B. zwischen 0 und 100) auszudenken und den Gegenspieler diese Zahl erraten zu lassen. Als Hinweis verrät man dem Gegenspieler bei jedem Raten nur, ob die geratene Zahl mit der ausgedachten Zahl übereinstimmt, kleiner oder größer ist.

(a) Spielen Sie dieses Spiel zunächst untereinander und versuchen Sie Ihr Vorgehen dabei umgangssprachlich als Algorithmus zu formulieren (kein Ada-95-Programm). Begründen Sie, warum Ihr Vorgehen beim Raten effizient ist.

(b) Schreiben Sie ein Programm, das einen Spieler zunächst eine Zahl einlesen lässt. Der andere Spieler soll nun versuchen diese zu erraten. Das Programm gibt Hinweise, ob die geratene Zahl stimmt, kleiner oder größer ist. Ist die geratene Zahl außerhalb des Bereichs, der aufgrund der bisherigen Tipps schon bekannt ist, so soll darauf hingewiesen werden.

Statt die Zahl von dem einen Spieler eingeben zu lassen, können Sie sich von folgendem Programm inspirieren lassen, wie man mit Ada 95 Zufallszahlen erzeugen kann.

```
with Ada.Text_IO, Ada.Numerics.Float_Random;
use Ada.Text_IO, Ada.Numerics.Float_Random;

procedure RandDemo is
  G : generator;
  Zufall : float;
begin
  Reset (G);
  for i in 1..20 loop
    Zufall := Random (G);
    if Zufall < 0.5 then Put_Line("Kopf"); else Put_Line("Zahl"); end if;
  end loop;
end RandDemo;
```

Details finden sich im Abschnitt A.5.2 des Reference Manuals, grob gesagt ist folgendes zu tun: Ein Objekt vom Typ `Generator` deklarieren, dieses *einmal* als Pa-

rameter von `Reset` verwenden, anschließend beliebig oft als Parameter von `Random`, das Ergebnis letzterer Funktion ist dann ein „zufälliger“ Wert im Bereich zwischen 0 und 1. (Da die direkte Möglichkeit ganzzahlige Zufallszahlen in Ada 95 zu erzeugen etwas komplizierter ist, begnügen wir uns hier mit dieser etwas einfacheren Variante – schreiben Sie sich ggf. eine `function Random_Int(n:natural) return natural` selbst, die eine Zufallszahl zwischen 0 und `n-1` erzeugt.)

(c) Erweitern Sie Ihr Programm aus Teil (b) nun so, dass der Computer die Zahl errät, die Sie sich ausdenken (diese soll dann von Ihnen im Programm nicht mehr eingegeben werden). Der Computer soll dabei erkennen, wenn Sie ihn durch widersprüchliche Angaben in die Irre führen wollen.

Zu Beginn soll das Programm fragen, ob der Computer oder wie in Teil (b) der Benutzer raten soll.

8. **Primzahlen:** Ein sehr einfaches Primzahltestprogramm überprüft, ob die eingegebene Zahl $n \geq 2$ sich ganzzahlig durch eine Zahl aus dem Intervall von 2 bis $n-1$ teilen lässt. Ist dies für keine Zahl i mit $2 \leq i \leq n-1$ der Fall, d.h. n hat außer der 1 und sich selbst keine Teiler, so ist n eine Primzahl. (Anmerkung: Primzahlen haben immer genau 2 Teiler: die 1 und sich selbst; d.h., die 0 und 1 sind keine Primzahlen)

(a) Hat eine Zahl n einen Teiler k, dann hat sie auch einen Teiler n/k, d.h., man muss nicht alle Zahlen bis $n-1$ auf die Teilereigenschaft überprüfen. Nutzen Sie dies und schreiben Sie eine effiziente „`function istPrimzahl(n:natural) return Boolean`“. Fügen Sie die Begründung für Ihr Vorgehen als Kommentar mit ein.

(b) Alle ganzen Zahlen $n \geq 2$ lassen sich eindeutig als Produkt von Primzahlen schreiben, die sogenannte Primfaktorzerlegung. Schreiben Sie eine Prozedur mit Parameter n, die die Primfaktorzerlegung von n ausgibt. Benutzen Sie Ihre Prozedur, um im Hauptprogramm die Primfaktorzerlegungen von 15, 24, 108, 153 und 1001 auszugeben.

(c) Will man alle Primzahlen bis zu einer Zahl n bestimmen, so kann man diese mit dem Sieb des Eratosthenes berechnen. Zunächst schreibt man sich dazu alle Zahlen von 2 bis n auf. Die 2 ist Primzahl und man markiert alle echten Vielfachen der 2 (diese lassen sich ganzzahlig durch 2 teilen und sind somit keine Primzahlen). Die nächste unmarkierte Zahl, die 3, ist Primzahl und man markiert wieder alle echten Vielfachen dieser Zahl. Dieses wird jeweils mit der nächsten unmarkierten Zahl wiederholt.

Die unmarkierten Zahlen sind nicht Vielfache einer kleineren Zahl, sie sind also Primzahlen. Wir erhalten so der Reihe nach 2, 3, 5, 7, 11, ...

Schreiben Sie eine Prozedur `Primzahlsieb` mit Parameter n, die das Sieb das Eratosthenes durchführt und dann alle Primzahlen bis zur Zahl n ausgibt. (Hinweis: Sie können Parameter einer Prozedur als Feldgrenzen verwenden)

(d) **Zusatzaufgabe (schwer):** Überlegen Sie, ob man das Sieb des Eratosthenes benutzen kann, um die Berechnung der Primfaktorzerlegung zu beschleunigen. Welchen Aufwand hat Ihr Algorithmus aus Teil (b), mit welchem Aufwand müsste man bei Verwendung des Siebs des Eratosthenes rechnen? Skizzieren Sie Ihre Überlegungen auch an einigen Beispielen.

9. **Das NIM-Spiel:** Gegeben ist ein Haufen von n Hölzchen. Die zwei Spieler A und B ziehen abwechselnd und nehmen dabei jeweils 1, 2 oder 3 Hölzchen vom Haufen weg. Wer das letzte Hölzchen wegnimmt, hat verloren.

(a) Entwickeln Sie eine Strategie, mit der man dieses Spiel gewinnen kann.

Wir spielen das Spiel hier mit 17 Hölzchen. Der Computer soll beginnen. Programmieren Sie so, dass der Computer dabei versucht zu gewinnen. Achten Sie auf sinnvolle Kommentare, Einrückungen und „sprechende“ Variablennamen. Fügen Sie Ihre Strategie, mit der der Computer spielt, als Kommentarzeilen mit ein.

(b) **Zusatzaufgabe:** Spielt man das Spiel zu dritt, so verändern sich die Strategien. Zeigen Sie, dass man (unabhängig davon, wer beginnt) verliert, wenn die anderen beiden Spieler sich verbünden. Gibt es sinnvolle Strategien, wenn solch ein Bündnis zu Beginn nicht existiert?

10. **Hotel Devil's Door:** In diesem merkwürdigen Hotel sind die Zimmer mit den Zahlen 1 bis 100 nummeriert. Nachts, wenn alle schlafen, rennen kleine Teufel durch die Gänge und schlagen die Türen auf und zu.

- Der erste Teufel öffnet alle Türen.
- Der zweite Teufel schließt jede zweite Tür. Das Zimmer 1 ist also noch offen, das Zimmer 2 geschlossen, Zimmer 3 offen, usw.
- Der dritte Teufel verändert den Zustand jeder dritten Tür: Ist die Tür offen, so schließt er sie – ist sie geschlossen, so öffnet er sie. Er schließt also Tür 3, öffnet Tür 6, schließt Tür 9, usw.
- Der i-te Teufel verändert den Zustand jeder i-ten Tür – analog zum dritten Teufel.

Wir wollen nun das nächtliche Treiben in diesem seltsamen Hotel simulieren.

(a) Schreiben Sie ein Programm, das das nächtliche Türen-auf-und-zu-schlagen im Hotel Devil's Door simuliert. Geben Sie jeweils den Zustand der Türen aus, nachdem ein weiterer Teufel durch die Gänge gelaufen ist.

(b) In welchen Zimmern kann man schlafen, so dass am nächsten Morgen die Zimmertür geschlossen ist? In welchen Zimmern, deren Tür morgens geschlossen ist, wurde die Tür am seltensten auf und zu gemacht? Begründen Sie Ihre Antwort.

(c) **Zusatzaufgabe:** In welchen Zimmern hört man vom Türenklappern am wenigsten, in welchen am meisten (solange die eigene Tür offen ist, hört man stets, wenn eine anderen Tür geöffnet oder geschlossen wird)? Ermitteln Sie die Lösung durch ein Programm.

11. **Trick 17:** In einem Programm eines Kommilitonen sehen Sie folgende Anweisungen für die beiden `integer`-Variablen `a` und `b`: `b:=b-a; a:=a+b; b:=a-b;`

Was bewirken diese drei Zuweisungen? Bewerten Sie dieses Vorgehen.

12. **Binärzahlen:** Diese Aufgabe soll uns ein tieferes Verständnis der Datentypen im Inneren unseres Rechners vermitteln.

Im Rechner sind alle Daten durch Vektoren von Bits (mit Wert 0 oder 1) dargestellt. Wir wollen hier nun die Addition von positiven ganzen Zahlen mit 8 Bits simulieren. Mit 8 Bits lassen sich $2^8 = 256$ verschiedene Werte darstellen. Bei der gewöhnlichen Darstellung von Zahlen zur Basis 2 ist der Wert einer Zahl in Binärdarstellung $b_7b_6b_5b_4b_3b_2b_1b_0 = \sum_{i=0}^{7} 2^i \cdot b_i$, so hat 00010110 den Wert $0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 4 + 2 = 22$.

Bei der Addition zweier Zahlen geht man wie beim schriftlichen Addieren zweier Dezimalzahlen stellenweise vor (beginnend bei der niedrigstwertigen Stelle). Addieren wir

zu 00010110 eine 1, so erhalten wir 00010111. Addieren wir zu dieser Zahl nochmals 1, so ergibt die Addition auf die letzte Stelle einen Überlauf, der auf die Stelle davor übertragen wird. Dort steht jedoch auch eine 1, sodass erneut ein Überlauf passiert usw. Wir erhalten schließlich als Ergebnis 00011000, die Binärdarstellung der Dezimalzahl 24.

Zur Simulation verwenden wir einen `type SimulNat is array (0..7) of Boolean;` – der Wert an Index `i` sei dabei genau dann `true`, wenn das i-te Bit der simulierten Zahl den Wert 1 hat.

- Schreiben Sie eine Funktion, die als Eingabe eine 8-Bit-Zahl erhält und den zugehörigen Dezimalwert zurückgibt.
- Schreiben Sie eine Funktion, die als Eingabe eine Dezimalzahl erhält und die zugehörige 8-Bit-Zahl zurückgibt.

  Hinweis: Ist die Zahl ungerade, so wird das Bit mit der Wertigkeit 2^0 eine 1 sein. Die höherwertigen Stellen erhält man, indem man die Dezimalzahl halbiert und erneut testet, ob diese nun gerade oder ungerade ist. Beispiel: Die Zahl 13 ist ungerade, das Bit 0 muss 1 sein. Ganzzahldivision der 13 durch 2 ergibt 6 – 6 ist gerade, das Bit 1 ist also 0. Ganzzahldivision der 6 durch 2 ergibt 3 – 3 ist ungerade, das Bit 2 ist demnach 1. Ganzzahldivision der 3 durch 2 ergibt 1 – 1 ist ungerade, das Bit 3 ist ebenfalls 1. Ganzzahldivision der 1 durch 2 ergibt 0 – alle höherwertigen Bits sind 0. Als 8-Bit-Zahl erhält man somit die 00001101 – beachten Sie, dass das niedrigstwertige Bit ganz rechts steht (so wie auch bei den Dezimalzahlen die Einer ganz rechts stehen).
- Schreiben Sie eine Prozedur `AddEins`, die auf eine gegebene Zahl in Binärdarstellung 1 addiert. Von welcher Art muss der Parameter sein? Wie erkennt man, falls die resultierende Zahl sich nicht mehr mit 8 Bit darstellen lässt?
- Schreiben Sie eine Funktion `Addiere`, die zwei Binärzahlen mit je 8 Bit als Eingabe erhält und als Ergebnis ebenfalls eine 8-Bit-Zahl zurückgibt. Erkennen Sie auch hier, falls ein Überlauf passiert.
- Will man negative Zahlen darstellen, so deutet man in der Regel das höchstwertige Bit (hier Bit 7) als -2^7. Dies ist die sogenannte Zweierkomplement-Darstellung. Der mit 8 Bit darstellbare Bereich ist nun -128 bis 127.

  Zu einer positiven 8-Bit-Zahl erhält man die zugehörige negative 8-Bit-Zahl, indem man alle Bits negiert und auf die so erhaltene Zahl 1 mit obigem Algorithmus addiert. Beispiel: Negiert man die 8-Bit-Darstellung von 22 (dies ist 00010110), so erhält man zunächst 11101001, Addition von 1 führt zu 11101010. Diese Zahl hat den Wert $-128 + 64 + 32 + 8 + 2 = -22$.

  Schreiben Sie eine Prozedur `Negiere`, die ein 8-Bit-Zahl mit Dezimalwert `x` in eine 8-Bit-Zahl mit Dezimalwert `-x` verwandelt.

  Fügen Sie als Kommentarzeilen mit ein, warum dies funktioniert. Bei welcher Zahl funktioniert das Verfahren nicht?
- **Zusatzaufgabe (sehr schwer, nur für Tüftler):** Bei der Zweierkomplement-Darstellung gibt jeweils das höchstwertige Bit an, ob eine Zahl negativ ist.

  Eine auf den ersten Blick sehr merkwürdig anmutende Zahldarstellung ist die zur Basis „-2“. Die Binärzahl 1101 hat dann z.B. den Wert $1 \cdot (-2)^3 + 1 \cdot (-2)^2 + 0 \cdot (-2)^1 + 1 \cdot (-2)^0 = -8 + 4 + 1 = -3$. Die Dezimalzahl +9 hätte die Binärdarstellung 11001 (16-8+1).

Finden Sie Algorithmen zur Negierung und Addition solcher Zahlen, die zur Basis -2 dargestellt sind.

Hinweis: Diese Zahldarstellung demonstriert lediglich, dass es Alternativen zur Zweierkomplement-Darstellung gibt. In der Praxis findet sie meines Wissens nach keinerlei Anwendung.

Demonstrieren Sie die von Ihnen implementierten Prozeduren und Funktionen durch geeignete Testfälle im Hauptprogramm.

13. **Permutationen:** Als Permutation bezeichnet man die Umordnung einer vorgegebenen Zahlenfolge. Beispiel: Die Permutationen zur Menge $\{1, 2, 3\}$ lauten (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1). Zu einer Menge mit n Elementen, gibt es stets $n!$ Permutationen (die erste Stelle ist eine von n möglichen Zahlen, abhängig von der ersten Stelle können $n - 1$ Zahlen für die zweite Stelle ausgewählt werden, usw. – es ergeben sich $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!$ Möglichkeiten).

    - Schreiben Sie eine Prozedur, die zum Parameter n alle Permutation der Menge $\{1, 2, \ldots, n\}$ ausgibt. Schreiben Sie dazu eine rekursive Prozedur, die ein Array nach und nach füllt unter Beachtung, dass der bisherige Inhalt des Array stets zu einer Permutation ergänzt werden kann.
    - **Alternativ-Aufgabe (schwerer):** Finden Sie einen Algorithmus, der zu einer gegebenen Permutation, die lexikographisch nächstgrößere Permutation findet (in obigem Beispiel sind die Permutation lexikographisch geordnet). Benutzen Sie diesen, um nacheinander alle Permutationen der Menge $\{1, 2, \ldots, n\}$ auszugeben.

14. `for`**-Schleifen**: Wir haben gelernt, dass die Laufvariblen von `for`-Schleifen implizit deklariert werden. So gesehen stellt eine `for`-Schleife ebenfalls einen Block dar.

    Simulieren Sie eine `for`-Schleife durch einen Block und eine `while`-Schleife. Testen Sie Ihr Vorgehen an dem Beispiel:

```
for i in 1..10 loop
  for i in 1..10 loop
    put(i,0);
  end loop;
end loop;
```

Fügen Sie als Kommentarzeilen neben Ihrer Idee hinzu, ob und (wenn ja) welche Eigenschaften der Laufvariablen von der Simulation nicht abgebildet werden.

15. **Zusatzaufgabe Programmanalyse (schwer):** Der Student Exit hat bei unserer Tutorin Goto das folgende unkommentierte Programm abgegeben.

```
with text_io, ada.Integer_Text_Io;
use text_io, ada.Integer_Text_Io;

procedure Exit_Considered_Harmful is
   Zahl1, Zahl2, Zahl3 : Integer := 0;
begin
   put("Erste Zahl: ");
   get(Zahl1);
   Put("Zweite Zahl: ");
   Get(Zahl2);
   for I in 1..Zahl1 loop
      Zahl2 := Zahl2 + 3;
      exit when Zahl2 > 100;
      for J in Zahl1..Zahl1+10 loop
         exit when Zahl1 > Zahl2;
         Zahl2 := Zahl2 - Zahl1;
         if I+15 > J then
            Zahl3 := Zahl3 + 1;
            exit when Zahl3 > Zahl1;
         end if;
         Zahl2 := Zahl2 + Zahl1 + 1;
      end loop;
   end loop;
   Put("Ergebnis: ");
   Put(Zahl1-Zahl3);
   Put(Zahl2);
end Exit_Considered_Harmful;
```

Beide stehen vor einem Rätsel, was das Programm tatsächlich tut. Da sich unter Ihnen Spezialisten für die Transformation von bestehenden Programmen und die Reformulierung von Schleifen befinden, transformieren Sie das Programm in eine Version `ohneexit.adb`, die keine `exit`-Anweisungen mehr enthält und sich identisch zum Ursprungsprogramm verhält. Vereinfachen Sie das Programm und versehen Sie es mit sinnvollen Kommentaren, sodass die Verständlichkeit erhöht wird. Sie können davon ausgehen, dass die Eingaben für das Programm immer zwischen 0 und 200 liegen. Das obige Programm ist auf der Webseite der Vorlesung erhältlich.

Hinweis1: Die Überführung in ein Programm ohne `exit` ist relativ einfach. Um das Programm jedoch in eine leicht lesbar, klar verständliche Form zu bringen, ist vermutlich ein wesentlich größerer Aufwand notwendig. Wägen Sie ab, wieviel Zeit Sie investieren möchten.

Hinweis2: Falls Sie die Anweisung `exit when` nicht kennen: Es handelt sich dabei um eine Sprunganweisung, welche die aktuelle (innerste) Schleife sofort verlässt, falls die Bedingung hinter `when` erfüllt ist. Die Programmausführung wird direkt hinter der Schleife fortgesetzt.

16. **Die Familie Recursi:** In der Familie Recursi, bekommt jedes männliche Familienmitglied im Laufe seines Lebens zwei Söhne, und zwar stets den ersten Sohn im Alter von 23 und den zweiten im Alter von 29 Jahren. Schreiben Sie ein Ada-Programm zur Berechnung der Anzahl der männlichen Familienmitglieder. Lesen Sie zu Programmbeginn das Alter des Stammvaters Adamo Recursi ein und geben dann das Ergebnis aus. Verwenden Sie für die eigentliche Berechnung eine rekursive Hilfsfunktion.

17. **Wald von Paganovo:** Im Zauberwald von Paganovo gibt es einen höchsten Baum mit Höhe $h_{\max}$. Alle weiteren Bäume gehorchen dem folgenden magischen Gesetz: Einen Baum der Höhe h gibt es im Wald nur dann, wenn es auch einen mit der Höhe $2(h+13)$ oder einen mit Höhe $h+37$ gibt. Natürlich hat jeder Baum eine positive Höhe. Schreiben Sie eine Ada-Funktion `Kleinster(h: Float) return Float`, die aus der Höhe des größten Baumes die des kleinsten Baumes berechnet. Bauen Sie diese Funktion in ein Hauptprogramm ein, das die Höhe des größten Baumes einliest und zum Schluss das Ergebnis ausgibt.

18. **Die gerechte Erbschaft:** Zwei Personen erben n Goldklumpen. Diese besitzen die Werte $g_1, g_2, \ldots, g_n$ Euro. Ihr Gesamtwert beträgt $G = g_1 + g_2 + \cdots + g_n$. Kann man diese Goldklumpen (ohne sie zu zerkleinern) so in zwei Hälften teilen, dass jede Person genau den Wert $G/2$ erbt? (Sie können davon ausgehen, dass die Werte g_i ganzzahlig sind. Testen Sie Ihre Programme unter Anderem mit den Werten $g_1 = 9$, $g_2 = 7$, $g_3 = 5$, $g_4 = 4$, $g_5 = 3$, $g_6 = 2$)

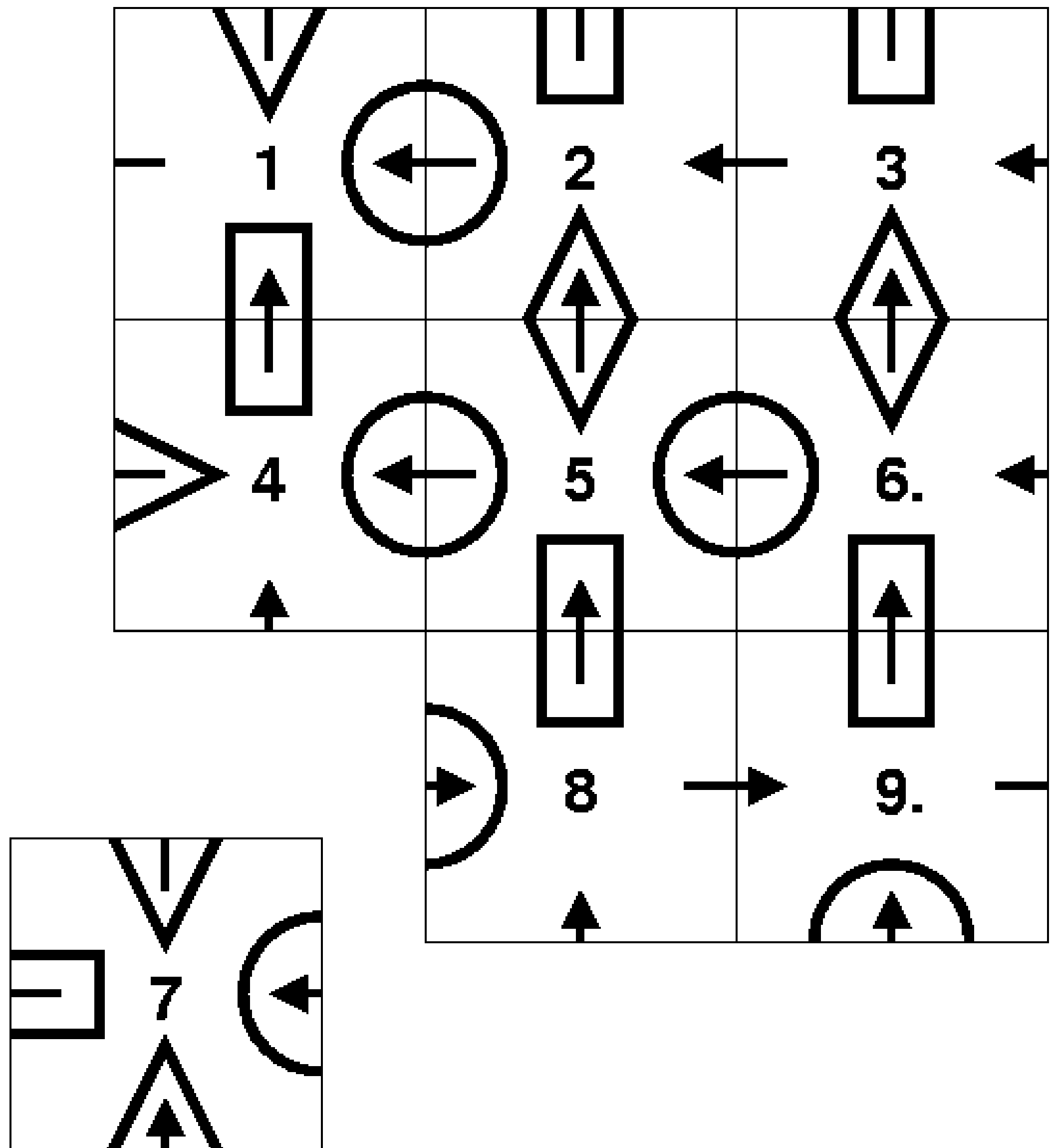
    (a) Schreiben Sie einen Algorithmus in Ada 95, der alle möglichen Aufteilungen systematisch durchprobiert (d.h., entweder ist g_i in der ersten Menge oder in der zweiten Menge, den Rest teilt man rekursiv auf) und am Ende angibt, ob es eine Aufteilung gibt, so dass die entstehenden Teilmengen jeweils den Wert $G/2$ haben.

    (b) Schreiben Sie einen Algorithmus in Ada 95, der systematisch in einem `array (0..G) of Boolean` die Indizes k mit `true` markiert, für die es eine Teilmenge der g_i gibt, deren Summe gerade k ist. Eine gerechte Erbschaft ist dann möglich, wenn der Index $G/2$ dabei mit `true` markiert wurde.

    (c) (2 Punkte, mittel) Schätzen Sie die Laufzeiten Ihrer beiden Algorithmen ab und vergleichen Sie diese. (Wenn Sie nur eine Teilaufgabe gelöst haben, schätzen Sie nur die Laufzeit des entsprechenden Algorithmus ab.)

19. **Verflixte Puzzelei:** Beim Aufräumen haben Sie im Keller ein Puzzle gefunden. Die 9 Teile sollen so zu einem Quadrat zusammengefügt werden, dass an jeder Kante ein durchgehender Pfeil und ggf. eine komplette geometrische Figur entsteht. Nach etwas Probieren haben Sie immerhin 8 Teile passend zusammengefügt, nur das Teil mit der Nummer 7 passt nicht in die verbleibende Ecke.

    Schreiben Sie ein Ada 95 Programm, das alle möglichen Lösungen des Puzzles ausgibt (geben Sie für die 3 Reihen jeweils an, welche Teile dort verwendet und ob diese um 90, 180 oder 270 Grad gedreht wurden).



Zusatzaufgabe: Erweitern Sie Ihr Programm, so dass Lösungen, die durch Drehung aus anderen Lösungen hervorgehen, nicht ausgegeben werden.

20. **Der Weihnachtsbaum**: Weihnachten naht, das Jahr 2006 ist fast zu Ende und überall sieht man Weihnachtsbäume, sogar auf dem Bildschirm!?!

    Schreiben Sie ein Ada-95-Programm, das in Abhängigkeit von einem Parameter `n` Weihnachtsbäume auf dem Bildschirm ausgibt. Die Weihnachtsbäume sollen nach folgendem Muster gebaut sein:

```
                                                          *
                                                         /-\
                                                         / \
                                 *                      /---\
                                /-\                      / \
          *                     / \                     /   \
         /-\                   /---\                   /-----\
          |                      |                        |
Schwabenversion (n=1), Studentenversion (n=2), Professorenversion (n=3) usw.
```

    (a) Schreiben Sie ein Ada-95-Programm, das zunächst eine Zahl `n` einliest und dann den entsprechenden Baum (mit Hilfe von Schleifen) zentriert auf dem Bildschirm ausgibt. Die Bildschirme, die Ihnen zur Verfügung stehen, können 25 Zeilen mit je 79 Zeichen darstellen (definieren Sie entsprechende Konstanten in Ihrem Programm). Es gibt eine maximale Grenze für `n`, so dass der Baum noch auf dem Bildschirm angezeigt werden kann. Wird ein größeres `n` eingegeben, so soll eine Erklärung auf dem Bildschirm ausgegeben werden, die den Benutzer zu mehr Bescheidenheit aufruft.

    (b) **Zusatzaufgabe:** Lesen Sie das `n` nun als Übergabeparameter aus der Kommandozeile ein. Der Aufruf `weihnachstbaum 3` sollte also dann die Professorenversion ausgeben, der Aufruf `weihnachstbaum 42` würde sich nicht mehr auf 79 Zeichen Breite darstellen lassen und würde zu dem entsprechenden Hinweis auf dem Bildschirm führen. Programmaufrufe, die nicht die Form `weihnachtsbaum <Zahl>` haben, sollen nur einen Bedienungshinweis ausgeben. Hinweis: Um Argumente von der Kommandozeile zu lesen benötigen Sie das Paket `Command_Line`.

21. **4 gewinnt:** Programmieren Sie das Spiel „4 gewinnt“. Die Regeln finden Sie unter `http://de.wikipedia.org/wiki/4_gewinnt`. (Ein Computerspieler braucht nicht programmiert zu werden.)

22. **Memory:** Programmieren Sie ein Memory-Spiel. Das Spielprinzip ist z.B. unter `http://de.wikipedia.org/wiki/Memory` erklärt. (Ein Computerspieler braucht nicht programmiert zu werden.)

23. **Die gesprochene Zahl:** Wir wollen zu den Zahlen das ausgeschriebene Wort von einem Programm berechnen lassen. Z.B.:

    - 42 → zweiundvierzig,
    - 425 → vierhunderfünfundzwanzig,
    - 1300 → eintausenddreihundert,
    - 24201 → vierundzwanzigtausendzweihunderteins,
    - 101000 → einhunderteintausend,
    - 400016 → vierhunderttausendsechzehn.

Schreiben Sie ein Ada 95 Programm, das eine natürliche Zahl zwischen 1 und 999999 einliest und korrekt in Worten ausgibt. Erläutern Sie zunächst das Konstruktionsprinzip (fügen Sie dies als Kommentarzeilen in Ihrem Programm ein).

24. **Türme von Hanoi:** Dieses klassische Beispiel für rekursive Programmierung haben wir bereits in der Einführung mit AdaLogo betrachtet. Es gibt jedoch auch einen einfachen iterativen Lösungsalgorithmus, dieser lautet umgangssprachlich: *Lege zu Beginn die kleinste Scheibe einen Stapel weiter nach rechts. Wiederhole (bis man fertig ist): Lege die zweitkleinste bewegbare Scheibe auf den Stapel, auf dem nicht die kleinste Scheibe liegt, und lege dann die kleinste Scheibe einen Stapel weiter nach rechts (lag die Scheibe bereits auf dem Stapel ganz rechts, so lege sie auf den Stapel ganz links).*

    Implementieren Sie diesen Algorithmus in Ada 95.

25. **Funktionale Programmierung:** Ada 95 ist eine imperative (befehls- oder anweisungsorientierte) Programmiersprache. Im dritten Semester werden Sie auch funktionale Programmiersprachen kennenlernen. Bei diesen kann man auf Zuweisungen und Schleifen verzichten – die einzigen Sprachelemente sind die Fallunterscheidung und die Rekursion. Bei den Beispielen zur Berechnung der Fakultät und der Fibonacci-Zahlen haben wir bereits ein wenig funktional programmiert. Prinzipiell lässt sich jede berechenbare Funktion auch funktional programmieren, d.h. nur mit Fallunterscheidung und Rekursion.

    - (mittel) Versuchen Sie zunächst `for`-Schleifen und `while`-Schleifen durch Rekursion zu simulieren (die Fallunterscheidung wird dabei nur zum Rekursionsabbruch benutzt).
    - (knifflig) Programmieren Sie eine `function istPrim(n:natural) return boolean`, die testet, ob der formale Parameter `n` eine Primzahl ist. Verwenden Sie auch hier ausschließlich Fallunterscheidung und Rekursion.
    - (noch etwas schwerer) Programmieren Sie eine `function ntePrimzahl(n:natural) return natural`, die zum formalen Parameter `n` die `n`-te Primzahl berechnet. Verwenden Sie auch hier ausschließlich Fallunterscheidung und Rekursion.

26. **Durcheinandergewürfelt:** (Anmerkung für Studierende, die noch Sprachschwierigkeiten haben: diese Aufgabe ist nur bedingt für Sie geeignet.) Sind in einem Text manche Buchstaben vertauscht, so erkennen wir dies und verstehen den Inhalt trotzdem. Dieses Vertauschen kann sehr weit gehen: Solange in jedem Wort der erste und letzte Buchstabe an seiner Position bleibt, so ist der Mensch in der Lage, die Buchstaben instinktiv in die richtige Reihenfolge zu bringen. Versuchen Sie es selbst:

    Snid in eeinm Txet mnhcae Buhsecabtn vresauhtct, so erneknen wir dies und vserhteen den Ihnlat todtrzem. Deiess Vetrahuscen kann sher weit geehn: Slgaone in jdeem Wrot der estre und lttzee Buhabstce an senier Poiotsin bieblt, so ist der Mecsnh in der Lage, die Bcsshtuabn ititnnskiv in die rcitgihe Reeinlfhoge zu bienrgn.

    Programmieren Sie daraus ein Spiel, bei dem ein Spieler einen Satz eingibt, das Programm die Wörter des Satzes nach obigen Regeln durcheinanderwürfelt und den anderen Spieler nun raten lässt, wie der Satz im Original hieß.

27. **n-Damen-Problem:** Auf einem Schachbrett kann eine Dame waagerecht, senkrecht und diagonal auf jedes Feld, das in diesen Richtungen liegt, wechseln. Eine Dame A bedroht eine andere Dame B, wenn die Dame B auf einem der Felder steht, auf das die Dame A wechseln kann.

Schreiben Sie ein Programm, das alle Stellungen von 8 Damen auf einem 8×8-Schachbrett berechnet, so dass diese 8 Damen sich paarweise nicht bedrohen. Geben Sie auch die Gesamtzahl der gefundenen Lösungen an. Welche Anzahlen ergeben sich für n von 1 bis 10, wenn man n Damen auf einem $n \times n$-Schachbrett platzieren soll? Überlegen Sie sich, welche Laufzeit ihr Programm hat.

Modifizieren Sie das Programm so, dass Lösungen, die aus anderen Lösungen durch Drehung hervorgehen, nicht ausgegeben werden.

Zusatzaufgabe: Modifizieren Sie das Programm weiter, so dass auch Lösungen, die aus anderen Lösungen durch Spiegelung hervorgehen, nicht ausgegeben werden.

28. **Programmanalyse:** Gegeben ist folgender Block:

```
declare
  x : integer := 5;
  procedure increase is
  begin
    x := x+1;
  end;
begin
  increase; put(x,width=>0);
  declare
    x : integer := 3;
  begin
    increase; put(x,width=>0);
  end;
  increase; put(x,width=>0);
end;
```

Welche Ausgabe wird hier erzeugt?

29. Ein Klausuraufgabenklassiker ... Die verschiedenen Parameterübergabemechanismen in Ada sorgen – wenn sie wie hier etwas durcheinander verwendet werden – immer wieder für Verwirrung. Analysieren Sie das folgende Ada-Programm *Scope*. Welche Ausgabe erzeugt es?

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;

procedure Scope is
   A:Integer:=3;
   B:Integer:=2;
   C:Integer:=1;

   procedure Write2int(A,B:in Integer) is
   begin
      Put(A,3);
      Put(B,3);
      New_Line;
   end;
```

```
   procedure P1(E,F: in Integer) is
   begin
      A:=E+F;
      B:=E-F;
      Write2Int(A,B);
   end;

   procedure P2(A,D: out Integer) is
   begin
      A:=B*2;
      D:=C*3;
      Write2Int(B,C);
   end;

   procedure P3(C,A:in out Integer) is
   begin
      C:=A+B;
      A:=B-C;
      Write2int(A,C);
   end;

begin
   P1(A,B);
   P2(B,C);
   P3(A,B);
   P2(B,C);
end;
```