Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

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

26.10. + 04./09./16./23./30.11.2005 / Version 7. Dezember 2005

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 $\rightsquigarrow$ Formale Darstellung (speziell, wenn das Ausführen mechanisch durchgeführt wird) $\rightsquigarrow$ 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 $\rightsquigarrow$ 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.

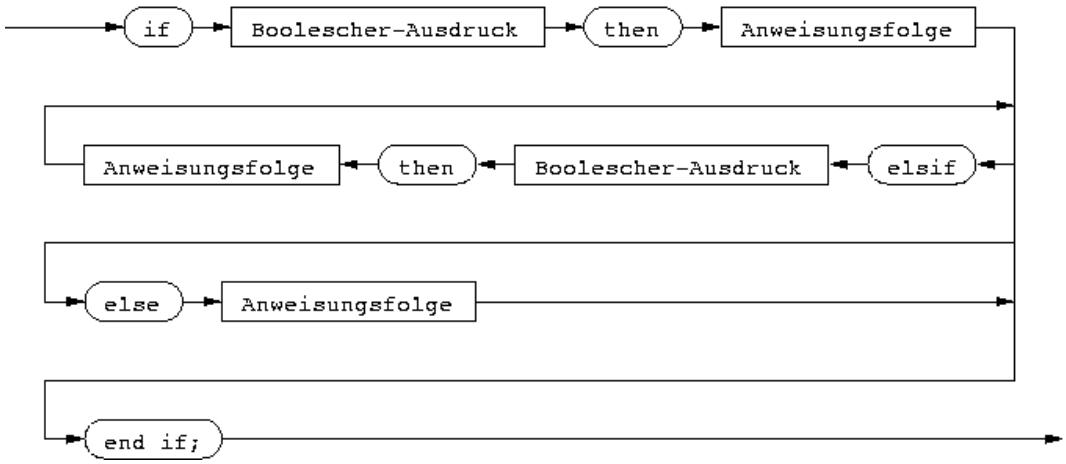
Wir führen hier nun an dem Beispiel des `if-then-elsif-else`-Konstrukts vor, wie sich dessen Syntax mittels Syntaxdiagramm und EBNF darstellen lässt.

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 Vorlesung 9.11.+16.11.

- Programme haben in Ada 95 immer denselben Aufbau. Zunächst kommen die with- und use-Anweisungen, dann das reservierte Wort `procedure`, gefolgt von einem Bezeichner für den Programmnamen, dann das Wort `is`, Variablen- und Prozedur-Deklarationen, das Wort `begin`, eine Anweisungsfolge, das Wort `end`, optional nochmal der oben gewählte Bezeichner für den Namen des Programms und abschließend ein Semikolon. In EBNF liest sich dies so: ... (kommt noch)
- 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.
- Ebenso sind Kommentare 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.
- Ebenso werden hier der Einfachheit halber alle reservierten Wörter klein geschrieben – Ada 95 unterscheidet bei reservierten Wörtern und Bezeichner keine Groß- und Kleinschreibung. Dies ließe sich 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: (kommt noch)
- 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).

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; Groß- und Kleinbuchstaben werden nicht unterschieden („Muh“ und „muH“ sind 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!“

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

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:

- Datentyp integer, natural, positive
- Ada 95 bietet 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 angibt.

Wir nutzen die Attribute in folgenden Beispiel der Berechnung der Fakultät, um Laufzeitfehler durch Überschreitung 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.

```
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;
```

- Programm zur Berechnung der Fakultät
- 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
- Beispiel aus der Übung: Studienstiftung

```
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;
```

- unschön:
  - Rundungsfehler auf ganze Euro $\rightsquigarrow$ Datentyp float
- wir sehen dabei gleich:
  - Strenges Typkonzept in Ada 95
  - Konvertierung zwischen Datentypen
  - Formatierte Ausgabe für Datentyp float

Zusammenfassung:

- ```
  function <Bezeichner> ( <Parameter-Liste> ) return <Datentyp> is
    <Deklarationen>
  begin
    <Anweisungs-Folge>
  end [<Bezeichner>];
  ```
  ```

- wenigstens eine `return`-Anweisung

- Datentyp `integer`:
  `Put(<integer-Ausdruck> [, [width=>] <integer-Ausdruck>] );`
  z.B. `Put(zahl,width=>5);`, `Put(a+b,width=>7);`, `Put(42,0);`

  – das ist nicht ganze Wahrheit, reicht aber für den Moment :-)

- Datentyp `float`:
  `Put(<float-A.> [, [fore=>] <integer-A.>] [, [aft=>] <integer-A.>]`
  `[, [exp=>] <integer-A.>] );`
  z.B. `Put(zahl,exp=>0);`, `Put(a*b+c,aft=>2);`, `Put(x,3,4,0);`

- lässt man die Parameter-Bezeichner `fore`, `aft`, `exp` weg, so wird der erste Ausdruck für `fore` verwendet, der zweite für `aft` und der dritte für `exp`

- der Lesbarkeit halber Bezeichner mit hinschreiben

- Ada verfolgt ein strenges Typkonzept, d.h., es finden keine expliziten Typumwandlungen statt (Ausnahmen wie `natural`/`integer` – dies sind in Ada aber Unterbereiche und keine eigenständigen Typen – ggf. später)

- 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).

  – Es treten ggf. Rundungsfehler auf.

- Vorteile:

  – Durch das (strenge) Typkonzept werden bestimmte Flüchtigkeitsfehler vermieden.

  – Keine Mehrdeutigkeiten bei Ausdrücken wie z.B. `x:float; ...x*5;`

  – Tippfehler, die in andere 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.

- Nachtrag: 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.

  Analog für `float`-Zahlen `[Ada.Float_Text_IO.]default_fore/aft/exp := 4`.

  Explizite Angaben in Put haben grundsätzlich Vorrang vor den Default-Werten

Ein wenig Theorie - Zuweisungen und Ausdrücke:

- Die Zuweisung: auf der linken Seite einer Zuweisung „`:=`“ steht stets eine Variable, auf der rechten Seite stets ein Ausdruck, in EBNF ausgedrückt:
  `<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 z.B. `Integer(Float(123/4)*5.67+0.8)` so würde zunächst `123/4` ganzzahlig dividiert werden (Ergebnis `integer`-Zahl 30), die 30 in `float` gewandelt werden, diese dann mit 5.67 multipliziert (Ergebnis `float`-Zahl 170.1), 0.8 addiert (`float`-Zahl 170.9) und dann bei der Typ-Umwandlung auf 171 gerundet. Dieser Wert würde dann ggf. einer `integer`-Variablen zugewiesen werden können, aber nicht einer `float`-Variablen.
- 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 also zwar miteinander zu tun, sind aber wesentlich verschiedene Dinge.

Arithmetische Ausdrücke

Etwas vereinfacht stellt sich ein Arithmetischer Ausdruck so dar:

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

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

Dabei steht `abs(<AAus>)` für die Betragsfunktion und `x**y` für die Exponentation (x^y, $y \geq 0$) (für `y` ist nur `integer` bzw. `natural/positive` erlaubt). Ebenfalls nur für `integer` sind `mod` und `rem` definiert.

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

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

Prioritäten der gängigen Operatoren in Ada 95

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!

Die Vergleichsoperationen „=“, „/=“, „<“, „<=“, „>=“, „>“ haben den Ergebnis-Typ `Boolean` (entweder `true` oder `false`), wobei bei `float`-Zahlen die Vergleiche „=“, „/=“ aufgrund der unvermeidlichen Rundungsfehler mit Vorsicht zu genießen sind.

Boole'sche Ausdrücke:

```
<BAus> ::= <Rel> {<BoOp> <Rel>}
<Rel> ::= "true" | "false" | "not" <Rel> |
           "("<BAus>")" | <AAus><VglOp><AAus>


<BoOp> ::= "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 Boole'schen 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 Boole'schen Operatoren, insbesondere verlangt Ada 95 grundsätzlich eine Klammerung, wenn verschiedene Boole'sche 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).

Außer mit den logischen Operatoren können Boole'sche Variablen auch mit den Vergleichsoperatoren „=“, „/=“, „<“, „<=“, „>=“, „>“ verknüpft werden, wobei `false < true` gilt. Diese Möglichkeit des Vergleichs Boole'scher 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 Boole'schen Ausdrücke die Möglichkeiten von Ada 95 im Sinne einer guten Lesbarkeit.

Wie wertet Ada 95 Ausdrücke aus?

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. Betrachten wir hierzu einige Beispiele: (an der Tafel)

- `Integer(Float(123/4)*5.67+0.8)`
- `(x+y)*42+z/7`

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 :-)).

Definition von Bäumen:

- 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.
- Baumartige Strukturen spielen in der Informatik eine so wichtige Rolle, dass wir gleich die wichtigsten Begriffe kennen lernen 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).
- 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*.

1.4 Vorlesung 23.11.

- Ada hat ein strenges Typkonzept, aber `integer` verträgt sich mit den Datentypen `natural` und `positive`?

  Die Datentypen `natural` und `positive` sind in Ada 95 Subtypen.

- Was ist ein Subtyp? 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 vordefiniert sind unter Anderem:
  ```
  subtype natural is integer range 0..integer'last;
  subtype positive is integer range 1..integer'last;
  ```

- Die Constraints werden in der Regel durch Einschränkung auf einen Bereich `range a..b` angegeben.

- 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 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). Erst bei der Zuweisung wird überprüft, ob der Wert den Constraint erfüllt. Falls nicht, erzeugt das Programm einen Laufzeitfehler.

- Bereiche (Schlüsselwort „`range`“) 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.

- 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 Boole'schen Bedingungen verwenden: in der Anweisung `if Zaehler in 3..9 then ...` würde der `then`-Zweig genau dann ausgeführt, wenn `Zaehler>=3 and Zaehler<=9` gilt.

- Manchmal möchte man nicht die Möglichkeiten von z.B. `integer` übernehmen, sondern vielleicht eigene Funktionen und Operanden 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 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.

- Bei der Ausgabe solcher Ganzzahltypen schlägt das strenge Typkonzept von Ada wieder zu.

  – Bei Ganzzahltypen, deren Wertebereich innerhalb dem von `integer` liegt, kann man sich noch mit der Umwandlung `Integer(monat)` 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);`
  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;` mit ein, so reicht wieder die Angabe von `Put(...)`. Die Ausgabe kann auch durch Angaben `width =>` beeinflusst werden. Der Wert `default_width` ist 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 hier vielleicht die Breite von `integer`-Zahlen gemeint war.

Wir wollen uns 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. Wir werden die Deklaration und Zugriff auf Arrays gleich im Beispiel sehen. Nur so viel vorweg: die eben eingeführten Subtypen bzw. 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.

Wir betrachten vorab noch als kleines einführendes Beispiel das Einlesen von 5 Zahlen und deren Ausgabe in umgekehrter Reihenfolge.

```
-- sl - Info 1 (autip), Vorlesungsbeispiel Einlesen von Zahlen
-- und Ausgabe in umgekehrter Reihenfolge
-- 23.11.2005
-- Idee: lese Zahlen in ein Array ein und gib dieses Array rückwärts wieder aus

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

procedure Gib5ZahlenAus is

   N : Natural :=5;
   type Vektor is array (1..N) of integer; -- sonst Zuweisungen der Art feld1:=feld2
   Feld : Vektor; --      nicht möglich wären (gilt nicht für alle Ada Compiler ...)

   begin
      for I in 1..N loop
         Put(I'Img & ". Zahl: "); -- i'img liefert die Zeichenkette der Zahl i
         --                       -- der Operator & verknüpft Zeichenketten
         Get(feld(I));
      end loop;

      for I in reverse 1..N loop
         Put_Line(I'Img & ". Zahl ist " & Feld(I)'Img);
         -- Put_line gibt eine Zeile mit Zeilenvorschub aus
         -- (gibt es nur für Zeichenketten!)
      end loop;
```

```
end Gib5ZahlenAus;
```

Wir haben im Beispiel noch ein paar neue Konstrukte eingeführt: `x'Img` gibt die Zeichenkette des Werts des Ausdrucks `x` an (dies ist leider in ObjectAda nicht implementiert). Der Operator „&“ verknüpft Zeichenketten. Dieses beide zusammen erlaubt, die Ausgabe von Text mit Variablen deutlich übersichtlicher zu gestalten, als eine Aneinanderreihung von mehreren `Put`-Aufrufen.

Die Initialisierung und der Zugriff auf Elemente eines Feldes demonstriert dieses kleine Sortierprogramm (einige Details sind dabei noch unschön, wir werden bei Gelegenheit sehen, wie man es besser machen kann).

```
-- sl - Info 1 (autip), Vorlesungsbeispiel Sortieren von Zahlen
-- 23.11.2005
-- Idee: Suche die kleinste Zahl, tausche diese nach vorn
--       Suche die kleinste Zahl im Feld von 2 bis Ende, tausche diese an Pos. 2
--       Suche die kleinste Zahl im Feld von i bis Ende, tausche diese an Pos. i
--       Fertig, wenn i = Ende

with Ada.Text_Io, Ada.Integer_Text_Io;
use Ada.Text_io, ada.integer_text_io;

procedure Sortiere9Zahlen is

   N : Natural :=9;
   type Vektor is array (1..N) of Integer;
   Feld : Vektor := (27,13,8,17,37,12,21,5,19);
   -- Initialisierung ist auch partiell möglich: (5 => 42, 3 => 17, others => 0)
   -- initialisert das Feld zu (0,0,17,0,42,0,0,0)

   function Minimumsuche(Anfang,Ende:Integer) return Integer is
      -- liefert die Position des kleinsten Elements im Feld(anfang,ende)
      Bisher_Min:integer:=anfang; -- zu Beginn ist feld(anfang) das kleinste Element
   begin
      for I in Anfang+1..Ende loop
         if Feld(I)<Feld(Bisher_Min) then
            Bisher_Min:=I;
         end if;
      end loop;
      return Bisher_Min;
   end Minimumsuche;

   procedure Vertausche(I,J:Integer) is
      -- Vertauscht im Feld die Werte an den Positionen i und j
      Zwsp:Integer;
   begin
      zwsp:=feld(i);
      Feld(I):=Feld(J);
      Feld(J):=Zwsp;
```

```
   end Vertausche;

   begin
      put("Wir sortieren die Zahlen");
      for I in 1..N-1 loop
         Put(feld(I)'Img & ","); -- x'img liefert die Zeichenkette der Zahl x
      end loop;
      put_line(feld(n)'img);

      for I in 1..N-1 loop
         -- minimumsuche(i,n) sucht den Index des zu vertauschenden Elements
         Vertausche(I,Minimumsuche(I,N));
         -- vertauscht zunächst die 1. Position mit dem kleinsten Elemente
         -- dann (i=2) die 2. Pos. mit dem kleinsten El. des verbleibenden Feldes usw.
      end loop;

      put("Sortiert:");
      for I in 1..N-1 loop
         Put(feld(I)'Img & ","); -- x'img liefert die Zeichenkette der Zahl x
      end loop;
      put_line(feld(n)'img);

end Sortiere9Zahlen;
```

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.

1.5 Vorlesung 30.11.

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;`.

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 Semikolon getrennt angegeben, Parameter gleichen Typs können durch Komma 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 Prozedur- und Funktions-Deklaration als EBNF oder Syntaxdiagramm.

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).

Was passiert nun, wenn lokale und globale Variablen denselben Namen haben?

Erstmal nichts. Wir können, solange die lokale Variable gültig ist, nicht auf die globale Variable mit gleichem Namen zugreifen, die globale Variable nennt man dann **verschattet**.

Was passiert beim Aufruf einer Prozedur bzw. Funktion?

- Die Abarbeitung des derzeitigen Programmteils wird unterbrochen.
- Für die formalen Parameter und lokalen Variablen der aufgerufenen Prozedur wird ein neuer Speicherbereich zur Verfügung gestellt.
- 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.

Goldene Regeln beim Entwurf von 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;
```

1.6 Vorlesung 7.12.

Als Beispiel für 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 dem oben 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 -2147483648. 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, z.B.:

```
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(1+feld'first..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 Typ 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).

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 `min_index(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 `Minimumsuche` aus dem alten Sortierbeispiel (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(min_index(zusort)));
    Sortiere(zusort(1+zusort'first..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.

Zeichen und Fallunterscheidungen

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, vielleicht auch 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, dabei muss nur die Anzahl der zugewiesenen Elemente übereinstimmen. Mit der Deklaration `feld1 : vektor(1..7); feld2 : vektor(3..6);` wäre eine Zuweisung `feld1 := feld2 & (7,12,23);` erlaubt.

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(string)` 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.

Programmierung – Pause

Wir unterbrechen hier zunächst die Einführung in die Programmiersprache Ada 95 und wenden uns drei formalen/theoretischen Themengebieten zu, bevor wir die Programmierung wieder aufgreifen. Dies ermöglicht Ihnen über die Weihnachtsferien ggf. den Stoff zu wiederholen und Lücken zu schließen.

Ausblick:

- 14.12.: Grammatiken und Formale Sprachen (+ 2. Testklausur)
- 21.12.: Grenzen der Programmierkunst, Berechenbarkeit (+ Vorlesungsumfrage)
- 11.01.: Aufwandsabschätzungen formal gefasst: Die O-Notation
- 18.01.: Hierarchie der Datentypen – Aufzählungstypen, Verbunde, Beginn Pointer
- 25.01.: Fortsetzung Pointer
- 01.02.: Pakete, Abstrakte Datentypen (+ 3. Testklausur (eine Woche später als ursprünglich geplant!))
- 08.02.: Exceptions
- 15.02.: Objektorientierte Programmierung, Polymorphie, dynamische Bindung
- 08.03.: Nur für höhere Semester: Klausur zur Info I+II – die Aufgaben stellt ein letztes Mal Herr Dr. Zimmer. – wer (wie alle jetzigen Erstsemester) die Klausur nach dem Sommersemester schreibt (Termin 14.8.), muss mit Aufgaben basierend auf den diesjährigen Inhalten rechnen; es wird dann keine Aufgaben mehr basierend auf der Vorlesung von Herrn Dr. Zimmer geben.