Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

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

19.10.(V+Ü) + 26.10.2005(Ü) / Version 16. November 2005

Inhaltsverzeichnis

0 Einführung

0.1 Organisatorisches

0.1.1 Vorlesung und Übung

- Vorlesung: Mittwochs 14:00–15:30, Raum V38.03
- Übung: Mittwochs 11:30–13:00, Raum V9.02
  - Wöchentliche Übungsblätter ab 26.10.

  - Abgabe und Bearbeitung über das System eClaus
    `http://eclaus.informatik.uni-stuttgart.de`
    Ablauf der Übungen: Ausgabe jeweils in der Übung, Abgabe jeweils 9:45 Uhr am Tag der folgenden Übung (also erstmals am 4.11., danach jeweils Mittwochs, also 9.11., 16.11., ..., jeweils 9:45 Uhr). Abgabe erfolgt ausschließlich elektronisch über `eClaus.informatik.uni-stuttgart.de` – versuchen Sie nach Möglichkeit die Abgabe nicht in der letzten Minute zu machen!
  - Eintragen in die Übungsgruppe:
    username: `online-Info1autip` ; password: `lewand05`
  - eClaus-Account wird eingerichtet, Passwort wird per E-Mail zugesendet
  - ab dann mit Ihrer Matrikelnummer und dem zugesandten Passwort einloggen

- Scheinbedingungen:
  - 50% der erreichbaren Punkte, 14 Übungsblätter a 20 Punkte
  - Bestehen von 2 der 3 Testklausuren, voraussichtliche Termine: 16.11., 14.12., 25.1.
  - aktive Teilnahme in den Übungen (mind. 2 Mal Aufgaben vorrechnen)

0.1.2 Informatik im Studiengang Autip

- 1. Semester: Einführung in die Informatik 1 : 2V + 2Ü
- 2. Semester: Einführung in die Informatik 2 : 4V + 2Ü
- 14.8.2006: Klausur Einführung in die Informatik 1+2
- Zulassungsvoraussetzung: Übungsschein in Informatik 1 oder 2
- 3. Semester: Einführung in die Informatik 3 : 3V + 2Ü
- ca. 10.3.2007: Klausur Einführung in die Informatik 3
- weitere Vorlesungen im Hauptstudium

0.2 Ein kurzer Überblick

0.2.1 Definition Informatik

Definiert man Mathematik, so würde (hoffentlich) keiner auf die Idee kommen zu sagen: „Mathematik ist alles, was mit Taschenrechnern zu tun hat“. Als Informatiker wird man hingegen öfters mit folgender „Definition“ konfrontiert:

> **Definition:** Informatik ist alles, was mit Computern zu tun hat

Das ist **falsch!** – Definition aus dem Duden Informatik:

> Informatik ist die Wissenschaft von der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von Informationen, besonders der automatischen Verarbeitung mithilfe von Digitalrechnern (Computer).

Lernziele sind also insbesondere **nicht** der Beherrschung von Windows oder Linux und den darauf laufenden Programmen.

0.2.2 Lernziele

- Grundkonzepte der Informatik:
  - Modellbildung und Abstraktion
  - Herangehensweisen zur Problemlösung
  - Algorithmen: Entwicklung und Bewertung
  - Standardalgorithmen und -datenstrukturen
- Kenntnis einer höheren Programmiersprache (aktiv, nicht nur passiv!), hier : Ada 95
- Am Rande gestreift:
  - ungefähre Vorstellung der Computerhardware
  - Aspekte des Software-Engineerings
- auch(!): soziale Kompetenzen:
  - Teamarbeit, ...

0.2.3 Der Softwareentwicklungsprozess – ein Beispielproblem

Als Beispiel für die Aufgabenfelder der Informatik (auch wenn es für uns in dieser Veranstaltung weit außerhalb unserer Reichweite ist) betrachten wir ein System zur Wettervorhersage. Eine Herausforderung für die „systematische und automatisierte Informationsverarbeitung“: wir haben

> Eingabedaten: eine Flut von Messwerten, die uns die aktuelle Wettersituation und deren bisherige Entwicklung liefern

und sollen daraus

> Ausgabedaten: die Wettervorhersage für morgen

gewinnen. Ich habe extra ein kompliziertes Beispiel gewählt, weil hier wohl kaum jemand der Versuchung erliegt, sofort zum Computer zu rennen und ein Programm einzutippen – hier ist hoffentlich klar, dass einiges an Vorarbeit geleistet werden muss, bevor wir den Computer einschalten.

Modellbildung: Um unsere Aufgabe mit dem Rechner angehen zu können, müssen wir ein Modell des relevanten Teils der Realität entwickeln – Stichworte sind hier Abstraktion und Formalisierung.

- Für viele Problemstellungen liefert die Mathematik einen mächtigen Satz Werkzeuge zur präzisen Beschreibung der vorkommenden Größen und deren Beziehungen. Im Fall der Wettervorhersage lassen sich etwa Strömungsvorgänge mittels (recht ekliger) Differentialgleichungen beschreiben.

- Außer Mathematik brauchen wir bei diesem Schritt in der Regel Hilfe aus der Anwendungsdisziplin, hier aus der Meteorologie, aus der Physik und möglicherweise aus weiteren Disziplinen.
- Am Ende der Modellbildung ist aus dem unpräzisen Szenario („das Wetter“) eine präzise Beschreibung (in Größen wie „Temperaturverteilung im Gebiet XY“) geworden, die als Grundlage für die weiteren Arbeiten fassbar ist.

Meist wird in diesem Prozess erst klar, was der Kunde eigentlich will: vielleicht wollte er gar kein Programm, das die Temperaturverteilung vorherberechnet, sondern nur eines, das Bauernregeln auswertet?

Formalisierung und Abstraktion sind wichtige Säulen des Problemlösungsprozesses (auch wenn der Sinn bei den kleinen Beispielen, wie wir sie in den Übungen behandeln können, vielleicht nicht so offensichtlich ist)!

Computergerechte Repräsentation: Bei der Formalisierung hatten wir zwar schon im Hinterkopf, dass mal ein Programm entstehen soll, aber im Mittelpunkt der Überlegungen stand die Beschreibung des Problems, nicht seine Lösung: eine Differentialgleichung ist kein Programm!

Der nächste Schritt bewegt sich in Richtung Rechner. Das formalisierte Problem wird in eine für den Rechner geeignete Form aufbereitet:

- Was für Daten sollen gespeichert werden $\rightsquigarrow$ Datenstrukturen
- Was soll mit diesen Daten passieren $\rightsquigarrow$ Algorithmen (Berechnungsvorschriften)

Auswahl/Entwicklung von Datenstrukturen: Wie sieht eine geeignete Repräsentation der Problemgrößen auf einem Rechner aus?

- Unser Rechner hat z.B. einen zwar großen, aber endlichen Speicher: schon bei der Darstellung (reeller) Zahlen sind Näherungen notwendig.
- Wesentlich problematischer: Feldgrößen (etwa der Temperaturverteilung über einem Gebiet); prinzipiell unendlich viele Funktionswerte. Hier müssen noch mal Mathematiker und Informatiker (und der Kunde) gemeinsam ran, um eine geeignete diskretisierte (endliche) Näherungsdarstellung der verwendeten Größen zu finden.

So mathematiklastig geht es nicht immer zu, die Auswahl geeigneter Datenstrukturen für die verwendeten Größen ist aber fast immer ein wesentlicher Schritt der Programmentwicklung. Das Rad nicht neu erfinden (zumal ein Vorrat von Rädern zur Verfügung steht, auf die wir selber vermutlich nicht kommen würden): Kenntnis von Standard-Datenstrukturen ist hier für den Erfolg sehr wichtig!

Auswahl/Entwicklung von Algorithmen: Mit diesen Daten soll natürlich auch etwas passieren: aus der Aufgabenstellung ist ein Berechnungsverfahren zu gewinnen, ein Algorithmus.

Hierbei sollte man zwar schon an die folgende Implementierungsarbeit denken, aber sich noch nicht mit zu vielen Details (etwa der verwendeten Programmiersprache) befassen.

Implementierung: Ist man sich über Algorithmen und Datenstrukturen im Klaren, muss das ganze in ein Programm(-system) umgesetzt werden, das unser Rechner verarbeiten kann.

Wichtige Frage dabei: was für Werkzeuge gibt es bereits, die uns die Arbeit erleichtern?

- Unumgänglich für praktisch jede Programmentwicklung: Übersetzer einer Hochsprache.

  Dies ermöglicht uns, ein Programm hinzuschreiben, ohne uns um die hässlichen Details der Hardware zu kümmern.

  Die Programmiersprache, die wir in dieser Vorlesung benutzen werden, heißt Ada 95.

- Gibt es fertige Programmmodule, die uns unterstützen, z.B.: sollen wir zum Abspeichern unserer Daten ein Datenbanksystem kaufen oder programmieren wir alles selber? Auch hier gilt mal wieder: nicht das Rad neu erfinden! Im Netz sind umfangreiche Programmsammlungen verfügbar, zum Teil (nicht immer!) in sehr guter Qualität.

- Im Beispiel Wettervorhersage sollten wir uns die vorhandenen Programmsysteme zum Lösen von Differentialgleichungen ansehen, auf jeden Fall die vorhandenen Numerikbibliotheken sichten; ziemlich sicher werden wir ein Programm zur Visualisierung dazukaufen.

- Bei einem komplexen Vorhaben wird auch Software nützlich sein, die die Programmentwicklung unterstützt (für unsere Übungsaufgaben wird das keine Rolle spielen): Dokumentation der Programmentwicklung, Unterstützung von Teamarbeit etc. sind Aufgaben, die man nicht unterschätzen sollte.

Was sonst noch zum Entwicklungsprozess dazugehört:

- Analyse des Produktes: Arbeitet es korrekt? Arbeitet es effizient? Kritisches Hinterfragen während aller Entwicklungsschritte!

  Die Formalisierung ermöglicht es, präzise Aussagen dazu zu machen. (Ob die Formalisierung selber korrekt war, ist ein anderes Problem)

- Darstellung und Interpretation der Ergebnisse

- Wartung und Weiterentwicklung

- ...

0.3 Materialien und Literatur

Selbstständiges Arbeiten ist notwendig!

- Erlernen von Ada, Programmierpraxis gewinnen

- Nachlesen der in der Vorlesung angesprochenen theoretischen Konzepte

- Skript zur Vorlesung

  `http://www.fmi.uni-stuttgart.de/fk/lehre/ws05-06/autip1/`

  dort auch Folien der letztjährigen Vorlesung (Dr. Zimmer)

- Unterlagen zur Hauptfach-Vorlesung Informatik (Prof. Claus)
  `http://www.fmi.uni-stuttgart.de/fk/lehre/ws05-06/info1/default.htm`
- Skript der letztjährigen Hauptfach-Vorlesung (Prof. Lagally)
  `http://pcbs13.informatik.uni-stuttgart.de/ifi/bs/lehre/ei1/2004/htm/ei1ws04.htm`
  Diese beiden werden im Weiteren kurz als „Claus bzw. Lagally, Kap. so und so“ zitiert.

Die Literatur zu Ada 95 ist fast ausschließlich Englisch und richtet sich in der Regel an Menschen mit einigen Programmierkenntnissen in anderen Hochsprachen. Zwei Standardwerke seien hier genannt:

- J.G.P. Barnes, Programming in Ada 95, 2. Auflage, Addison-Wesley, 1998
- M. Nagl, Softwaretechnik mit Ada 95, 2. Auflage, vieweg, 2003

Ein sehr gutes Buch für Anfänger in englischer Sprache ist von

- Jan Skansholm: Ada 95 - From the Beginning, Third Edition, Addison-Wesley, 1997

(die erste und zweite Auflage behandeln die Vorversion Ada 83). Dieses Buch liegt in 3 Exemplaren in der Unibibliothek vor (derzeit aber leider alle 3 verliehen, Stand 20.10.).

Eine Sammlung von Links zu Ada 95 findet man auf der Ada-Seite der Fachschaft Informatik und Softwaretechnik

- `http://fachschaft.informatik.uni-stuttgart.de/quietschies-online/ada.htm`

Weitere Links auf der Seite zu dieser Vorlesung :-)

- `http://www.fmi.uni-stuttgart.de/fk/lehre/ws05-06/autip1/`

Diskussionen über Vorlesungsinhalte und Übungsaufgaben (bitte keine fertigen Lösungen präsentieren!) sind möglich und erwünscht. Die Fachschaft Informatik und Softwaretechnik stellt eigens dafür ein Forum

- `http://swt.uni-stuttgart.de/forum/`

zur Verfügung. Auch Fragen und Probleme zur Installation der Ada-Software können hier diskutiert werden.

0.4 Der Rechner und wir

- Zum Bearbeiten der Aufgaben Zugang zu Rechner unbedingt notwendig!
- Zugang über RUS oder Account im Poolraum der Informatik (Universitätsstr. 38) beantragen.
- Bequemer: eigener Rechner
  - Wesentliche Voraussetzung: Ada-Übersetzter, ist für Windows/Linux (kostenlos) verfügbar ($\rightsquigarrow$ Ada-CD-Rom, siehe Link auf der Vorlesungsseite).
  - Finden Sie sich so in Arbeitsgruppen zusammen, dass pro Gruppe ein Rechner (Windows/Linux) vorhanden ist.

0.5 Eine spielerische Einführung in Ada 95 mit dem System AdaLogo

Das System AdaLogo (`http://www.adalogo.de.vu/`) wurde im Sommersemester 2005 im Rahmen eines Softwarepraktikums erstellt. Es stellt einen Teil der Sprache Ada 95 in einer graphischen Benutzerumgebung zur Verfügung und hat einige Befehle, um einfache Graphiken zu erstellen.

Wir wollen hier einige Grundlegende Konzepte der Programmierung, einfache Datentypen, Ausdrücke, Kontrollstrukturen (Fallunterscheidungen, Schleifen) sowie Prozeduren anhand einfacher Beispiele illustrieren.

- Zeichne ein Dreieck

Eine Anweisungsfolge in Umgangssprache könnte so lauten: Male irgendwie nach rechts, dann nach links oben und dann zurück zum Ausgangspunkt.

Mal davon abgesehen, dass vermutlich auch hier nicht unbedingt ein Dreieck herauskommen würde („irgendwie nach rechts“ muss nicht einmal eine Gerade sein), wären die Anweisungen für einen Algorithmus nicht exakt genug. Ein Algorithmus muss

- exakt formuliert sein,
- muss von jedem ohne weitere Erläuterungen auf die gleiche Weise durchgeführt werden können.

In der Regel beschränkt man sich außerdem auf einen vorher festgelegten Satz von Anweisungsmöglichkeiten. Wir werden dies später noch genauer fassen.

Ein Algorithmus zum Zeichnen eines (gleichseitigen) Dreiecks könnte so lauten: „Zeichne eine Gerade von 100 Längeneinheiten, drehe dich 120 Grad nach links, zeichne eine Gerade von 100 Längeneinheiten, drehe dich 120 Grad nach links, zeichne eine Gerade von 100 Längeneinheiten.“

Um sich zum Schluss wieder in der Anfangssituation zu befinden, könnte man sich nochmals um 120 Grad nach links drehen. In AdaLogo ist dies nun unser erstes Programm:

```
forward(100);
turn(120);
forward(100);
turn(120);
forward(100);
turn(120);
```

(diese Befehle schreiben wir zwischen das `begin` und `end` im linken Textfeld, wo das Programm im AdaLogo-System steht).

Ein Klick auf den „run“-Knopf (links oben) startet das Programm und das Dreieck erscheint auf der Zeichenfläche.

Wollen wir nun ein zweites Dreieck zeichnen, könnten wir unten anfügen:

```
forward(200);
turn(120);
forward(100);
turn(120);
forward(100);
turn(120);
```

In der ersten angefügten Anweisung müssen wir 200 Schritte vorgehen, um das zweite Dreieck nicht direkt über das erste zu zeichnen.

Die Befehle für das Zeichnen eines Dreieckes würde man umgangssprachlich vielleicht so formulieren: „Wiederhole 3 Mal das Zeichnen einer Gerade von 100 Längeneinheiten jeweils gefolgt von einer Drehung um 120 Grad nach links.“ Genau genommen müsste man noch dazu sagen, dass man nach der Drehung von dort aus weitermacht und nicht etwa an den Anfangspunkt zurückspringt. Letzteres wird in AdaLogo automatisch so gemacht. Die Befehle für das 3-malige Wiederholen lauten:

```
for i in 1..3 loop
  forward(100);
  turn(120);
end loop;
```

Dabei ist 1..3 als Intervall zu verstehen. Das Zeichen „i“ bezeichnet eine Variable (einen Wertebehälter), die im ersten Durchlauf den Wert 1 annimmt, im zweiten Durchlauf den Wert 2 und im dritten Durchlauf den Wert 3. Da wir hier diese Werte 1, 2 bzw. 3 nicht verwenden, sondern uns nur die Häufigkeit interessiert, hätten wir hier auch statt dessen 5..7 verwenden können (selbst probieren).

Ersetzen wir das `forward(200);` weiter oben durch `forward(100); forward(100);`, so können wir unsere zwei Dreiecke mit folgenden Zeilen zeichnen lassen:

```
for i in 1..3 loop
  forward(100);
  turn(120);
end loop;
forward(100);
for i in 1..3 loop
  forward(100);
  turn(120);
end loop;
```

Wollen wir nun kleinere Dreiecke zeichnen, so müssten wir an drei Stellen die 100 durch den entsprechend kleineren Wert ersetzen. Wir können auch hier einen Platzhalter verwenden. Wollen wir als Platzhalter die Variable `x` verwenden, so fügen wir vor das `begin` die Zeile `x : integer;` ein – `x` ist dabei der Name der Variable, `integer` besagt, dass wir in der Variablen `x` (ganze) Zahlen speichern wollen. Wir müssen nun der Variablen noch einen Wert zuweisen. Insgesamt erhalten wir also:

```
procedure NONAME is
  x : integer;
begin
  x := 80;
  for i in 1..3 loop
    forward(x);
    turn(120);
  end loop;
  forward(x);
  for i in 1..3 loop
    forward(x);
    turn(120);
  end loop;
end;
```

Zur besseren Lesbarkeit (und falls wir noch weitere Dreiecke zeichnen wollen), können wir die Schleife zum Zeichnen der Dreiecke zu einem Befehl zusammenfassen. Dazu können wir eigene Befehle definieren und vor dem `begin` unseres eigentlichen Programmes einfügen:

```
procedure NONAME is
  x : integer;

  procedure dreieck is
  begin
    for i in 1..3 loop
      forward(x);
      turn(120);
    end loop;
  end;

begin
  x := 80;
  dreieck;
  forward(x);
  dreieck;
end;
```

Wollen wir verschieden große Dreiecke zeichnen, gibt es die Möglichkeit auch Parameter für unsere selbstdefinierten Befehle zu definieren. Die Prozedur für das Dreieck könnte dann so lauten:

```
  procedure dreieck(n:integer) is
  begin
    for i in 1..3 loop
      forward(n);
      turn(120);
    end loop;
  end;
```

Rufen wir dann z.B. `dreieck(60)` auf, so wird der Wert 60 der Parametervariablen `n` zugewiesen und somit ein Dreieck mit Kantenlänge 60 gezeichnet.

Ersetzen wir das Hauptprogramm zwischen `begin` und `end` durch

```
x := 80;
dreieck(x);
forward(x);
dreieck(x);
turn(-120);
dreieck(160);
```

so erhalten wir zwei kleine Dreiecke und ein weiteres mit fest gewählter Kantenlänge 160, hier also doppelt so großes, darunter.

Klickt man auf „debug“, so kann man das Programm schrittweise ausführen lassen (jeder Klick auf „single step“ führt einen weiteren Schritt aus). Wenn Sie sich soweit mit dem Programm vertraut gemacht haben, können Sie sich über den User-Guide auf der Webseite `http://www.adalogo.de.vu` die Bedeutung der anderen Buttons aneignen.

Die Bezeichner für die Variablen dürfen auch länger sein, testen Sie es selbst aus. Wir werden bald auch hierfür genau definieren, welche Namen erlaubt sind und welche nicht. Auch der Name des Hauptprogramms (in AdaLogo per default „NONAME“) darf geändert werden (in Ada 95 ist vorgeschrieben, dass dieser mit dem Dateinamen übereinstimmt).

Ein wichtiges Element zur Beschreibung von Algorithmen sind Fallunterscheidungen: Z.B. könnte man als Parameter für die Dreieckskantenlänge nur bestimmte Werte zulassen wollen (wir fragen hier nicht weiter, warum wir das hier vielleicht wollen könnten). Die Befehle für solche Fallunterscheidungen orientieren sich auch hier an der englischen Sprache. Wollen wir z.B. in obigem Programm bei Werten von `x<40` statt ein Dreieck zu zeichnen lieber eine Nachricht ausgeben, so könnte die Befehlsfolge dafür so lauten:

```
if x < 40 then
  put("Ein Dreieck mit so kleiner Kantenlänge lohnt sich nicht."); new_line;
else
  dreieck(x);
end if;
```

Nach dem `if` steht also eine Bedingung (die wahr oder falsch sein kann – so etwas wie `if x then ...` mit obigem `x` vom Typ `integer` ist nicht erlaubt), und nach dem `then` eine oder mehrere Anweisungen. Ist die Bedingung nicht erfüllt, dann (und nur dann) werden die Anweisungen zwischen dem `else` und dem `end if;` ausgeführt. Manchmal möchte man mehrere Fälle unterscheiden. Hierzu bietet Ada das Konstrukt `elsif` an (ja, ohne „`e`“). Ein Beispiel:

```
if x < 40 then
  put("Ein Dreieck mit so kleiner Kantenlänge lohnt sich nicht."); new_line;
elsif x < 80 then
  dreieck(80);
elsif x > 200 then
```

```
  dreieck(200);
else
  dreieck(x);
end if;
```

Was macht dieses Programmstück? Probieren Sie es aus ...

Anmerkung: Prinzipiell lässt sich die Verwendung von `elsif` durch Schachtelung von `if`-Anweisungen vermeiden. Zugunsten der übersichtlicheren Darstellung von Fallunterscheidungen, d.h. zur Erhöhung der Lesbarkeit, sollte man `elsif` verwenden, wenn sonst der gesamte `else`-Zweig nur aus `if` ... `end if;` bestehen würde. Obiges Beispiel zu `if-then-elsif-else` ohne Verwendung von `elsif` würde so aussehen:

```
if x < 40 then
  put("Ein Dreieck mit so kleiner Kantenlänge lohnt sich nicht."); new_line;
else
  if x < 80 then
    dreieck(80);
  else
    if x > 200 then
      dreieck(200);
    else
      dreieck(x);
    end if;
  end if;
end if;
```

Im Beispiel zum Zeichnen der Dreiecke wurde der Inhalt der Schleifen-Variablen `i` nicht verwendet. Schleifen-Variablen können aber in Ausdrücken verwendet werden. Als Beispiel geben wir hier die Zahlen 5^2, 6^2 und 7^2 aus.

```
for i in 5..7 loop
  put("Die "); put(i); put(". Quadratzahl ist ");
  put(i*i); new_line;
end loop;
```

Der Ablauf ist identisch mit dem des folgenden Programmstücks:

```
i := 5;
put("Die "); put(i); put(". Quadratzahl ist ");
put(i*i); new_line;
i := 6;
put("Die "); put(i); put(". Quadratzahl ist ");
put(i*i); new_line;
i := 7;
put("Die "); put(i); put(". Quadratzahl ist ");
put(i*i); new_line;
```

Das `i` im letzten Programmstück muss im Gegensatz zum `i` im Beispiel mit `for`-Schleife noch mit `i:integer;` vor dem `begin` bekannt gemacht werden. Wir werden diesen Umstand in einigen Wochen noch genauer untersuchen. Für den Moment genügt zu wissen, dass wir auf Schleifen-Variablen nach Ende der Schleife nicht mehr zugreifen können.

Bis hierher haben wir die Konzepte Variablen, Typen, Zuweisungen, Fallunterscheidungen, Schleifen und Prozeduren ein klein wenig kennengelernt.

Die etwas komplizierteren Konzepte ... Bei den oben eingeführten `for`-Schleifen muss die Anzahl der Wiederholungen zu Beginn der for-Schleife bekannt sein! Eine Wiederholung, bis ein bestimmtes Ereignis eintritt, lässt sich mit `for`-Schleifen jedoch nicht realisieren. Das von der `for`-Schleife zu bearbeitende Intervall wird zu Beginn der Schleife festgelegt – bei `x:=3; for i in 1..x loop x:=i+1; end loop;` würde die Schleife also genau 3 Mal ausgeführt, da zu Beginn `x=3` gilt – die obere Grenze wird also nicht nach jedem Schleifendurchlauf neu bestimmt.

Wir schauen uns als Beispiel für **bedingte Schleifen** folgendes Szenario an: Herr S. Pendabel hat eine Stiftung zur Förderung von begabten Studenten eingerichtet. Dazu wurde ein Betrag von 1000000 Euro mit festem Zinssatz von 4% angelegt. Von den Erlösen sollen jedes Jahr 6 Studenten mit zu Beginn je 5000 Euro gefördert werden. Um die Inflation auszugleichen, erhöht sich dieser Betrag jährlich um 2%. Wir wollen nun untersuchen, wie sich das Vermögen der Stiftung entwickelt, und entwerfen ein Programm, dass abhängig von den Zinssätzen berechnet, in welchem Jahr entweder das Vermögen aufgebraucht ist oder wann es sich verdoppelt hat (dann können wir das Vermögen aufteilen und eine zweite Stiftung ins Leben rufen).

Wir benötigen also folgende Variablen (und initialisieren Sie mit obigen Werten); wir verwenden hier Ganzzahlen, da wir noch keine anderen Zahl-Typen kennengelernt haben, und begnügen uns mit dieser Näherung:

```
betrag : integer := 1000000;
ausgaben : integer := 30000;
zinsen : integer := 4;
inflation : integer := 2;
jahr : integer := 1;
```

In jedem Jahr ziehen wir vom Guthaben zunächst die Ausgaben ab, auf den verbleibenden Betrag erhalten wir die Zinsen. Die Ausgaben für das Folgejahr ergeben sich durch Addition des entsprechenden Anteils gemäß der Inflation. Dies wollen wir durchführen solange noch genug Geld für die Ausgaben vorhanden ist (also solange `betrag>=ausgaben`) und solange das Vermögen noch nicht auf 2 Millionen angewachsen ist. In AdaLogo formuliert liest sich dies so:

```
while betrag>=ausgaben and betrag<2000000 loop
  put("Ausgaben im "); put(jahr); put(".ten Jahr betragen ");
  put(ausgaben); put("Euro."); new_line;

  betrag:=betrag-ausgaben;
  put("Guthaben vor Zinsen im "); put(jahr); put(".ten Jahr betraegt ");
  put(betrag); put("Euro."); new_line;
```

```
    betrag:=betrag+betrag*zinsen/100;
    put("Guthaben nach Zinsen nach dem "); put(jahr); put(".ten Jahr betraegt ");
    put(betrag); put("Euro."); new_line;

    jahr:=jahr+1;
    ausgaben:=ausgaben+ausgaben*inflation/100;
  end loop;
```

Zum Schluss wollen wir noch wissen, ob wir uns nun über eine 2. Stiftung freuen können oder die Studierenden wieder ohne Förderung auskommen müssen:

```
  if betrag<= ausgaben then
    put("pleite im Jahr "); put(jahr); new_line;
  else
    put("juchhu, 2. Stiftung"); new_line;
  end if;
```

Als nächstes Problem schauen wir uns die Türme von Hanoi an: Das Spiel besteht aus drei Feldern, auf die Scheiben verschiedener Größe gelegt werden können. Zu Beginn sind alle Scheiben auf einem Feld, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben.

Bei jedem Zug darf die oberste Scheibe eines beliebigen Feldes auf eines der beiden anderen Felder gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Ziel des Spiel ist es, den kompletten Scheiben-Stapel auf ein anderes Feld zu versetzen.

Der Legende nach handelt es sich um 64 goldene Scheiben, die von Mönchen in einem fernöstlichen Tempel versetzt werden sollen. Ist diese Aufgabe erledigt, so sei das Ende der Welt gekommen.

Wir betrachten zunächst das Spiel mit 3 Scheiben.

```
   |     |     |
   =     |     |
  ===    |     |
 =====   |     |
-------------------
 Start Mitte  Ziel
```

Mit etwas Ausprobieren kommt man schnell auf die Lösung: Scheibe 1 auf Z, 2 auf M, 1 auf M, 3 auf Z, 1 auf S, 2 auf Z, 1 auf Z. Wollen wir nun die Lösung für 4 Scheiben, so könnten wir wieder etwas probieren und würden dann früher oder später auch hier auf eine Lösung kommen, ob es die kürzeste ist, wäre aber nicht so sicher (es geht mit 15 Schritten).

So wie wir oben verschiedene Anweisungen zu einer Prozedur zum Zeichnen von Dreiecken zusammengefasst haben, könnten wir uns auch hier behelfen: Zum Versetzen eines Turms mit 4 Scheiben, versetze zunächst die oberen 3 Scheiben auf die Mitte (mit welchen Einzelschritten ein Turm mit 3 Scheiben zu versetzen ist, haben wir oben ja gezeigt), legen die größte Scheibe auf das Zielfeld und versetzen dann erneut den Turm mit 3 Scheiben, dieses Mal von der Mitte auf das Zielfeld. Will man sich nicht merken, wie 3 Scheiben zu versetzen sind, so kann man auch dieses auf das versetzen von Türmen mit 2 Scheiben reduzieren.

Wir reduzieren unser „Rezept“ hier sogar noch eine Stufe weiter: Besteht der Turm aus nur 1 Scheibe, dann versetze ihn direkt auf das Zielfeld, ansonsten bewege den Turm bis auf die größte Scheibe auf das Zwischenfeld, die größte Scheibe auf das Ziel und dann den restlichen Turm auf das Zielfeld.

In AdaLogo (und auch in Ada 95) sieht das dann so aus: Wir nummerieren die Felder auch mit 1 bis 3 durch; ist etwas von 1 nach 3 zu bewegen, so ist das Zwischenfeld die fehlende Ziffer, diese lässt sich immer als Differenz zu 6 berechnen (6-von-nach) – bei 1 nach 3 also 6-1-3=2, bei 2 nach 1 also 6-2-1=3 usw.

```
procedure hanoi(hoehe:integer; von:integer; nach: integer) is
begin
  if hoehe = 1 then
    put("Scheibe 1 von Position "); put(von);
    put(" auf Position "); put(nach); new_line;
  else
    hanoi(hoehe-1, von, 6-von-nach); -- den 1 kleineren Turm versetzen

    put("Scheibe "); put(hoehe); put(" von Position "); put(von);
    put(" auf Position "); put(nach); new_line;

    hanoi(hoehe-1, 6-von-nach, nach);
  end if;
end;
```

Mit dem Aufruf `hanoi(4,1,3);` erhält man z.B. eine Anweisungsfolge zum Versetzen des Turms mit 4 Scheiben.

Dieses Prinzip, also das Lösen eines Problems durch Lösen eines oder mehrerer kleinerer Probleme der selben Art, nennt man **Rekursion**. Führen Sie das Programm schrittweise durch und versuchen Sie nachzuvollziehen, wie sich die Prozedur `hanoi` immer wieder mit kleineren Turmgrößen selbst aufruft.

Zum Abschluss der spielerischen Einführung noch ein Beispiel, wie mit Rekursion aus sehr kurzen Beschreibungen sehr komplexe Bilder entstehen können.

```
procedure lindenbaum(groesse:integer) is
begin
  if groesse<5 then
    forward(5); forward(-5);
  else
    forward(groesse*3/10); -- Stamm
    turn(25);
    lindenbaum(groesse*3/4); -- Zweig nach links geneigt
    turn(-45);
    lindenbaum(groesse*2/3); -- Zweig nach rechts geneigt
    turn(20);
    forward(-groesse*3/10); -- Rückkehr zur letzten Verzweigung
  end if;
end;
```

Rufen Sie die Prozedur mit `turn(90); lindenbaum(120);` auf (das Drehen um 90 Grad dient nur dazu, dass der Baum senkrecht steht).

Versuchen Sie auch hier den Aufbau schrittweise nachzuvollziehen.

Der Baum wirkt noch realisitischer, wenn man den linken und rechten Zweig manchmal vertauscht (oder z.B. die Winkel zufällig variiert). Zufallszahlen lassen sich in AdaLogo durch die Funktion `random(a,b)` erzeugen; dabei erhält man Zahlen `x` mit `a`$\leq$`x`$<$`b` (Hinweis: zur obigen Vertauschung der Zweige sind Zufallszahlen hilfreich, die entweder -1 oder 1 sind, der Ausdruck `random(0,2)*2-1` liefert diese – selbst überlegen!).

Die Verwendung von Zufallszahlen in Ada 95 ist etwas komplizierter, wir kommen später im Semester darauf zurück.

0.6 Übungsaufgaben

Ein Tipp (speziell für Studierende ohne Vorkenntnisse): Schauen Sie sich die Beispielprogramme im AdaLogo an und vollziehen Sie nach, was im Programm durch welche Befehle bewirkt wird. Sie können vielleicht Teile daraus wiederverwenden und zur Lösung der Aufgaben benutzen.

Noch eine Anmerkung, bevor es los geht: Programmieren ist ein „Handwerk“ – man lernt es nur, indem man es selbst durchführt (ein Schreinerlehrling wird das Hobeln einer Tischplatte nie erlernen, wenn er seinem Meister beim Hobeln immer nur zuschaut, er muss es selbst tun, um ein Gefühl dafür zu bekommen). Die Anfangshürde ist dabei immer die schwerste. Jeder muss diese aber selbst überwinden. Wenn Sie nicht weiter kommen, diskutieren Sie mit Ihren Kommillitonen über Ideen zur Lösung. Programmieren Sie diese aber unbedingt selbst aus.

Kommentieren Sie Ihr Programm! Dies heißt insbesondere, dass Sie neben Autor und Datum in den Kommentarzeilen die Idee für Ihr Vorgehen beschreiben sollten und die wesentlichen Bestandteile des Programms beschreiben. Dies ist in der Regel mehr als der Text „Zeichne n-Eck“.

Einrückungen machen Programme deutlich lesbarer! Nutzen Sie diese Möglichkeit.

Und noch ein Wort zum Kopieren von Lösungen anderer: Versuchen Sie unbedingt, zu Beginn des Semesters die Programme selbst zu schreiben, es wird definitiv nicht leichter bis zum Ende des Semesters. Wenn Sie selbst nicht weiterkommen, setzen Sie sich mit Ihren Kommillitonen zusammen, lassen Sie sich die Programmierschritte nochmals erklären oder schauen Sie Ihren Kommillitonen beim Programmieren zu, aber programmieren Sie die Aufgaben unbedingt selbst aus!

1. **Algorithmus:** Ein Algorithmus ist ein exakt formuliertes Verfahren, das von jedem, der den Text des Algorithmus erhält, auf gleiche Weise durchgeführt werden kann, ohne dass weitere Erläuterungen notwendig sind.

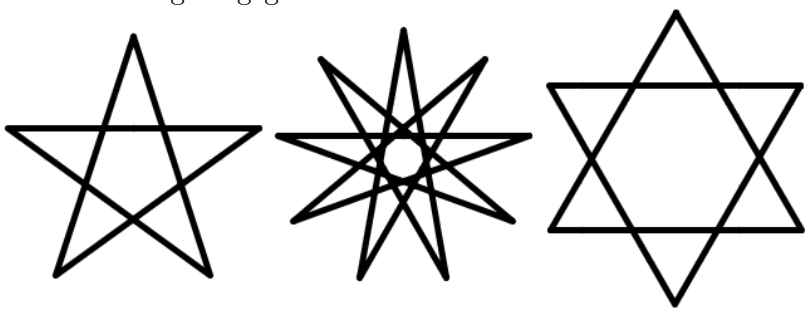
   Ein Computer führt in der Regel Algorithmen aus. Wichtig ist zunächst die Eindeutigkeit des Algorithmus. Hier ein umgangssprachliches Beispiel:

   Begib dich an die Nordostecke des Hauses in der Universitätsstraße 38. Warte bis Mitternacht. Peile den äußersten Deichselstern des Sternbilds des großen Wagen an und gehe dann genau 107 Schritte in diese Richtung. Grabe hier und du stößt in 2 Ellen Tiefe auf den Schatz.

   Ist diese Vorschrift eindeutig? Welche zusätzlichen Informationen fehlen?

2. **Zeichnen von regelmäßigen n-Ecken:** In Abschnitt 0.5 wurde eine `procedure dreieck(len:integer)` zum Zeichnen von Dreiecken mit Kantenlänge `len` erstellt. Erweitern Sie die Prozedur zu einer `procedure vieleck(anz:integer;len:integer)`, die als Parameter neben der Kantenlänge auch die Anzahl der Ecken übergeben bekommt. Fügen Sie die Idee als Kommentarzeilen im Programm hinzu.

3. **Zeichnen von Sternen mit n Zacken:** Schreiben Sie eine Prozedur zum Zeichnen von Sternen, die Anzahl der Zacken und die Kantenlänge sollen dabei als Parameter übergeben werden. Zunächst soll die Prozedur Sterne mit 4 Zacken (ein Quadrat) oder einer ungeraden Anzahl von Zacken zeichnen können. Wird als Parameter eine Zackenanzahl übergeben, die von der Prozedur nicht fehlerfrei gezeichnet werden kann, so soll eine entsprechende Meldung ausgegeben werden.



Zusatzaufgabe (schwer): Erweitern Sie Ihr Programm, so dass auch gerade Zackenanzahlen gezeichnet werden können. (Hinweis: Da AdaLogo derzeit nur ganze Zahlen verarbeitet, müssen Sie etwas tüfteln ... `len/cos(`α`)` kann für kleine Winkel α durch `len+len*`α^2`/2` angenähert werden, wobei α dann im Bogenmaß angeben werden muss, $\pi \approx$`22/7`, beachten Sie ggf. noch, dass der Operator „/“ ganzzahlig dividiert)

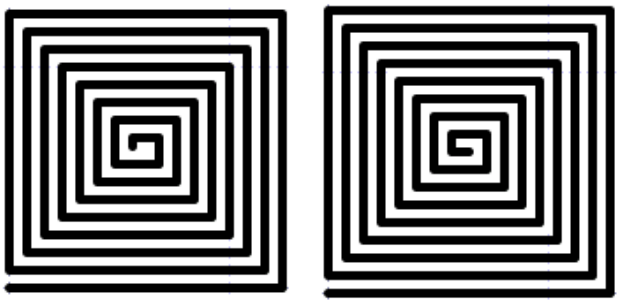
4. **Zeichnen von Kreisen:** Es gibt in AdaLogo keinen Befehl, um einen Kreis zu zeichnen. Die einfache Variante `for i in 1..360 loop forward(1); turn(1); end loop;` ist unbefriedigend, da wir dabei keinen Radius und keinen Mittelpunkt angeben können.

   (a) Kreise lassen sich durch Vielecke annähern. Erweitern Sie Ihre Prozedur aus Aufgabe 2 zu einer `procedure kreis(x:integer;y:integer;r:integer)`, die einen Kreis mit Mittelpunkt `(x,y)` und Radius `r` zeichnet. (Hinweis: Nutzen Sie die Beziehung zwischen Umfang und Radius eines Kreises!)

   (b) Die mathematische Definition eines Kreises besagt, dass genau die Punkte (p_x, p_y) zu einem Kreis mit Mittelpunkt (m_x, m_y) und Radius r gehören, für die $(p_x - m_x)^2 + (p_y - m_y)^2 = r^2$ gilt. Schreiben Sie mit Hilfe dieser Eigenschaft eine `procedure schoener_kreis(x:integer;y:integer;r:integer)`, die ebenfalls einen Kreis mit Mittelpunkt `(x,y)` und Radius `r` zeichnet.

   (c) Verwenden Sie Ihre Prozeduren aus den Teilaufgaben 4(a) oder 4(b), um die Olympischen Ringe zu zeichnen.

5. **Quadratische Spiralen:** Entwerfen Sie eine Prozedur für quadratische Spiralen. Dabei soll die Kantenlänge als Parameter übergeben werden und sich zwei Linien nur in den Eckpunkten berühren.

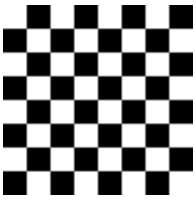


6. **(Runde) Spiralen:** Entwerfen Sie eine Prozedur für runde Spiralen. (Hinweis: der Abstand zwischen den einzelnen Linien wird hier vermutlich größer sein müssen)

7. **Geraden und Rechtecke:** Neben Kreisen sind Geraden und Rechtecke Grundelemente zum Zeichnen. Schreiben Sie folgende Prozeduren:

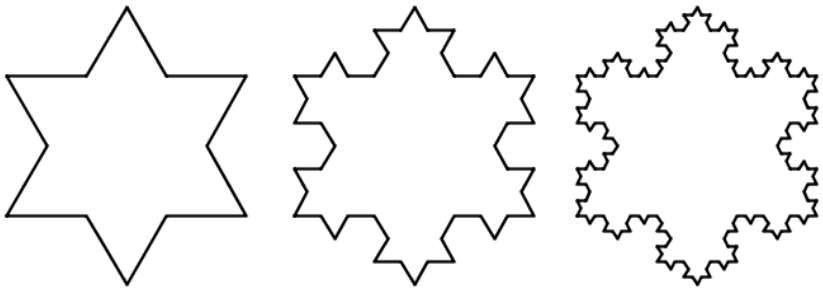
   - `procedure Linie(x1:integer;y1:integer;x2:integer;y2:integer)` -- zeichnet eine Gerade von den Koordinaten `(x1,y1)` nach `(x2,y2)`
   - `procedure Rechteck(x1:integer;y1:integer;x2:integer;y2:integer;filled:boolean)` -- zeichnet ein Rechteck; falls `filled=true` soll dieses Rechteck gefüllt sein

   Verwenden Sie diese Prozeduren, um ein Schachbrett zu zeichnen.



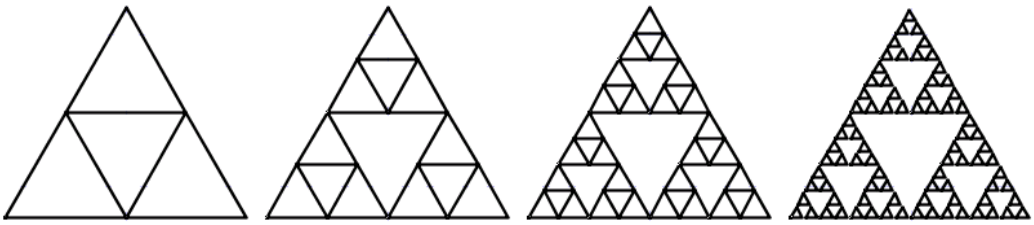
8. **Zeichnen selbstähnlicher Figuren:** Fraktale sind schöne Beispiele für Rekursion in der Programmierung.

   (a) Koch'sche Schneeflockenkurve (KSK): Man beginnt mit einem Dreieck (KSK der Stufe 0). Um aus dem Dreieck die KSK der Stufe 1 zu erhalten, ersetzt man bei jeder Kante das Mittelstück durch eine dreiecksähnliche Ausbuchtung, das Ergebnis ist der links abgebildete Stern. Ersetzt man bei diesem wiederum bei jeder Kante das Mittelstück durch zwei Kanten derselben Länge, so erhält man die KSK der Stufe 2 (in der Abbildung in der Mitte). Die KSK der Stufe 3 ist rechts abgebildet. Schreiben Sie eine Prozedur Schneeflocke, die als Parameter die Kantenlänge und die Stufe übergeben bekommt und die entsprechenden Koch'sche Schneeflockenkurve zeichnet. Dabei werden natürlich nicht schon gezeichnete Kanten gelöscht und neu gezeichnet, sondern Sie müssen sich überlegen, wie die KSK gleich richtig gezeichnet wird. (Beachten Sie, dass beim Zeichnen immer wenigstens 1 Schritt gezeichnet wird – testen Sie Ihr Programm mit den Eingaben (2,100), (3,100), (3,8) und anderen Ihrer Wahl)



(b) Sierpinski-Dreieck (SD): Auch hier ist der Ausgangspunkt ein Dreieck (Stufe 0). Das SD der Stufe 1 erhält man, indem man statt dem großen Dreieck, drei kleinere Dreiecke mit jeweils halber Kantenlänge zeichnet. Analog erhält man das SD der jeweils nächsten Stufe, indem statt einem Dreieck jeweils drei kleinere Dreiecke mit jeweils halber Kantenlänge gezeichnet werden. In der Abbildung sind von links nach rechts die Sierpinski-Dreiecke der Stufen 1, 2, 3 und 4 abgebildet.

Schreiben Sie eine Prozedur Sierpinski, die als Parameter die Kantenlänge und die Stufe übergeben bekommt und das entsprechende Sierpinski-Dreieck zeichnet. Es gilt der gleiche Hinweis wie bei der ersten Teilaufgabe.



9. **Programmanalyse:** Das AdaLogo-Programm

```
-----------------------------------------------
-- Autor: sl
-- Datum 04.11.2005
-- Frage: Welches Ergebnis wird in Abhängigkeit
--        des Wertes von zahl ausgegeben?
-- Idee : ?
-----------------------------------------------

with adalogo;
use  adalogo;

procedure Was_Tut_Das is
  zahl: integer := 42;
  erg : integer :=  0;
begin
  while erg*erg <= zahl loop
    erg := erg + 1;
  end loop;
  erg := erg - 1;
  Put("Das Ergebnis lautet: "); Put(erg); New_Line;
end;
```

berechnet eine Funktion $f : \mathbb{Z} \to \mathbb{Z}$ (Inhalt der Variable `erg` am Ende des Programms in Abhängigkeit des Inhalts der Variable `zahl` zu Beginn).

(a) Führen Sie das Programm von Hand schrittweise für verschiedene Anfangswerte der Variablen `zahl` durch. Welche Funktion f wird durch das Programm `Was_Tut_Das` realisiert? Begründen Sie Ihre Antwort.

(b) Wieviel Schritte benötigt das Programm zur Berechnung des Ergebnisses (abhängig von der Anfangsbelegung der Variablen `zahl`)? Begründen Sie Ihre Antwort.

(c) **Zusatzaufgabe (schwer):** Entwerfen Sie einen Algorithmus, der das Ergebnis mit deutlich weniger Schritten berechnet. Programmieren Sie diesen Algorithmus in AdaLogo aus. Fügen Sie dabei die Idee als Kommentarzeilen mit in das Programm ein und schätzen Sie ab, wieviel Schritte Sie nun zur Berechnung benötigen.

10. **Freitag der 13.:** Ist der Wochentag des 1. Januar bekannt, so lässt sich eindeutig bestimmen, auf welchen Wochentag ein Datum dieses Jahres fällt (wir gehen von Jahren aus, die kein Schaltjahr sind, also 28 Tage im Februar).

    Schreiben Sie eine Prozedur, die in Abhängigkeit des Wochentags am 1. Januar bestimmt, in welchen Monaten der 13. auf einen Freitag fällt.

11. **Umsetzung in Ada 95:** Diese Aufgabe soll Sie langsam zur Programmierung in Ada 95 hinführen.

    (a) Schreiben Sie das Programm aus Aufgabe 9 in Ada 95 um und kommentieren Sie es so, dass es gut verständlich ist (Sie dürfen dazu auch Variablennamen ändern).

    (b) Fügen Sie einige (mindestens 6) Syntaxfehler in Ihr Programm ein. Beschreiben Sie, welche 6 Fehler Sie eingefügt haben, geben Sie die Fehlermeldung des Compilers an und beurteilen Sie, ob diese Fehlermeldung hilfreich ist und den Fehler aus Ihrer Sicht zutreffend beschreibt. Die 6 Fehler sollten folgende Fälle beinhalten: zusätzliches Semikolon, fehlendes Semikolon, fehlendes Trennzeichen, Tippfehler bei Variablennamen, „=“ statt „:=“ bei einer Zuweisung.