Programmieren

in

FirstStep

W. Kowalk
Universität Oldenburg

Vorwort

Dieses Buch führt in eine moderne Programmiersprache ein, die wir ***FirstStep*** genannt haben. Es soll anhand von Beispielen, aber auch von grundlegenderen Betrachtungen gezeigt werden, wie heute komplexe Systeme programmiert und gewartet werden können. Dieses soll mittels einer einfachen, aber dennoch sehr mächtigen Sprache dargestellt werden. Die Umsetzung eines ***FirstStep***-Programms in gegenwärtige Sprachen, wie C++, kann entweder automatisch oder leicht manuell durchgeführt werden, so daß in ***FirstStep*** entwickelte Programme leicht auf jeden anderen Rechner übertragen werden können.

Oldenburg, im Oktober 1997 Wolfgang P. Kowalk

Inhaltsverzeichnis

1 Einleitung

Dieses Skript führt in die Programmierung in der Programmiersprache ***FirstStep*** ein. Die einzelnen Konzepte dieser Sprache werden systematisch in einzelnen Kapiteln zusammengestellt. Statt einer Syntax-Darstellung, die als Anhang aus der originalen Grammatik gegeben ist, werden aussagekräftige Beispiele verwendet, die eine direkte Umsetzung in ein Programm erlauben.

Da die Sprache ***FirstStep*** sich noch in der Entwicklung befindet, sollte die aktuellste Version jeweils aus dem Internet gezogen werden.

Neue Begriffe werden in **Fettschrift** (*bold*), die englische Übersetzung eines neuen Begriffs in **Kursivschrift** (*italic*) in Klammern hinter die Definition gesetzt; in der Regel wird hierfür die Nennform verwendet (Verben ohne *to*, Substantive und Adjektive im Nominativ Singular).

2 Was ist ein Programm?

Unter einem **Programm** (*program*) verstehen wir in diesem Skript eine Folge von Anweisungen, die Daten manipulieren. Unsere Vorstellung hiervon sollte es sein, daß ein Gerät, meist als Computer bezeichnet, Daten vorhält, die unter einem Namen referenziert werden können. Solche Namen sind einfache Wörter wie „`Wert`“, „`Anzahl`“, „`Name`“ oder „`Zeichen`“. Diese Namen sollten grundsätzlich so gewählt werden, daß sie den jeweiligen Wert, den sie repräsentieren, möglichst gut beschreiben.

2.1.1 Was sind Anweisungen?

Eine Anweisung besteht in der Regel aus einem Ausdruck, der einen Wert berechnet, sowie einem Namen, der diesen Wert annehmen soll.

2.1.2 Warum werden Folgen von Anweisungen verwendet?

Eine Programmiersprache stellt nur Anweisungen mit einer bestimmten Funktionalität zur Verfügung. Der Programmierer verwendet mehrere dieser Anweisungen in einer bestimmten Reihenfolge, um eine komplexere Funktionalität zu erhalten.

2.1.3 Was kann man mit Folgen von Anweisungen erreichen?

Die Frage stellt sich natürlich, ob man mit einer Folge von Anweisungen jeden Wert ermitteln kann. Dieses ist eine grundsätzliche Frage, die zunächst einmal klären muß, welche Werte man alles berechnen will. Es sei hier nur so viel gesagt, daß jeder berechenbare Wert auch von einer normalen Programmiersprache berechnet werden kann, soweit dem nicht Speicherplatz oder Laufzeitprobleme entgegenstehen.

2.1.4 Werden alle Programme durch Anweisungen geschrieben?

Man kann statt mit einfachen, elementaren Anweisungen auch mehrere Anweisungen zusammenfassen. Dieses wird häufig als „Unterprogramm“ bezeichnet. Wir werden später sehen, daß derartige Konzepte für die strukturierte Modellierung komplexer Programme notwendig sind.

Darüber hinaus untersucht die moderne Informatik weitere Konzepte, um möglichst einfach komplexe Programme zu erstellen. Beispielsweise wird versucht, Programme nicht durch die Befehlsreihenfolge, sondern durch die Festlegung des gewünschten Ergebnisses zu erstellen. Es bleibt dann einem Interpreter überlassen, selbst Programme zu entwickeln, die das gewünschte Zielproblem lösen. Die Techniken hierzu basieren jedoch auf Befehlen, die durch intelligente Methoden solche Probleme lösen wollen. Die Konzepte hierzu sollen im Rahmen dieses Skripts nicht behandelt werden.

2.2 Das erste Programm in ***FirstStep***

Wie kann nun ein einfaches Programm in einer Programmiersprache wie ***FirstStep*** geschrieben werden? Um ein Programm zu schreiben, welches zugleich eine Wirkung zeigt, kann man einen Text ausgeben. Dieses sieht in ***FirstStep*** etwa so aus:

```
DO
  write "Hallo Wolfgang, wie geht es dir?", newline;
DONE
```

Die Bedeutung ist also die folgende. Durch das Wort `DO` teilt das Programm dem Computer mit, etwas zu tun. Hat der Computer die Anweisungen ausgeführt, so teilt das Wort `DONE` dem Computer mit, daß alles getan ist, was getan werden sollte. Wörter mit einer bestimmten, von der Programmiersprache festgelegten Bedeutung, werden als **Schlüsselwörter** (*Keyword*) bezeichnet. Jede Programmiersprache kennt solche Schlüsselwörter. In ***FirstStep*** sind Schlüsselwörter leicht daran zu erkennen, daß sie entweder mit einem Kleinbuchstaben beginnen oder in Blockbuchstaben geschrieben sind. Beispielsweise ist auch „`write`“ ein solches Schlüsselwort.

Die eigentliche Anweisung in diesem Programm besagt, daß ein Text geschrieben werden soll. Woran erkennt das Programm, daß es sich um einen Text und nicht um eine Zahl handelt? An dem gleichen Indiz wie der Programmierer oder ein anderer Leser dieses Programms: Es folgt ein Text hinter dem „`write`“! Dieses scheint zunächst eine triviale Bemerkung zu sein, aber leider können viele Programmiersprachen nicht erkennen, um was für einen **Datentyp** (*data type*) es sich bei solchen Parametern handelt. Aus dem

Datentyp kann das Programm dann eine Anweisung generieren, die eine richtige Ausgabe erzeugt. Die obige Anweisung besteht somit aus einem „Befehlswort" (`write`) sowie einer Liste von Parametern. Hier sind es zwei, nämlich ein zu druckender Text und das Schlüsselwort „`newline`", welches die Ausgabe veranlaßt, eine Zeile weiter zu gehen. Somit gibt der obige Text dem Rechner den „Befehl", einen Text auszugeben und eine Zeile vorzugehen.

Solche Programme sind offenbar einfach erstellt. Statt eines Texts kann auch eine Zahl ausgegeben werden,

```
DO
  write 118, newline;
DONE
```

oder ein Zeichen,

```
DO
  write ‘X‘, newline;
DONE
```

oder mehrere Kombinationen von Texten, Zeichen und Zahlen:

```
DO
  write “Ich bin “,21,“ und mir geht es gut“, ‘!‘, newline;
DONE
```

Jedesmal scheint das Programm selbst zu erklären, was es eigentlich tut. Programme, die solche Eigenschaften haben, werden auch als „**selbsterklärend**" bezeichnet. Dieses ist wichtig, um ein Programm zu verstehen. In der Regel werden Programme häufiger gelesen als geschrieben, so daß eine Verständnishilfe beim Schreiben zum Lesen von Programmen sehr wichtig ist.

Das obige Programm wurde in drei Zeilen geschrieben. Man kann das durchaus einfacher machen, indem man alles in eine Zeile schreibt

```
DO write „Hallo Wolfgang, wie geht es dir?“, newline; DONE
```

oder auf noch mehr Zeilen verteilt:

```
DO

  write
    „Hallo Wolfgang, wie geht es dir?“
    ,
```

```
    newline
    ;


DONE
```

Ebenso lassen sich an verschiedenen Stellen Leerzeichen einfügen. Diese haben auf die Bedeutung eines Programms keinen Einfluß. Natürlich haben Leerzeichen in Texten (wie "`Hallo`" oder "`Ha ll o`") sehr wohl eine Bedeutung, und zwischen den Zeichen eines Schlüsselworts (`write` und nicht `wr it e`) oder eines Bezeichners für einen Speicherplatz darf ebenfalls kein Leerzeichen, Zeilenvorschub oder Tabulatorzeichen stehen. Aber ansonsten können diese Zeichen immer verwendet werden, um ein Programm übersichtlich zu strukturieren, wovon auch stets Gebrauch gemacht werden sollte, um das Lesen eines Programms zu vereinfachen.

Das letzte Beispiel zeigt zugleich die einzelnen Komponenten eines solchen Programms. Diese werden später genauer erläutert. Zunächst soll sich der Leser vor allem merken, daß Programme möglichst so geschrieben werden sollten, wie man es intuitiv tun würde, um die Lesbarkeit und somit die Verstehbarkeit zu erhöhen.

2.3 Programme mit Parametern

Natürlich ist ein Programm, welches immer die gleiche Ausgabe produziert, ziemlich langweilig. Es hat nur wenig nutzen, so daß das Programm von Werten abhängig sein sollte, die vom Benutzer des Programms eingegeben werden sollen. Ein einfaches Programm, welches einen Namen einliest und diesen wieder ausgibt, ist das folgende:

```
LINK
  text "Bitte geben Sie Ihren Namen ein ", NameDesBenutzers;
DO
  write "Hallo ", NameDesBenutzers, ", wie geht es dir?", newline;
DONE
```

Dieses Programm liest einen Namen ein und gibt ihn im Rahmen des Textes wieder aus. Damit haben wir die Möglichkeit, jeden Namen dieser Welt einzugeben und im Programm weiter zu verarbeiten.

Die Eingabe erfolgt hier übrigens abhängig von dem Programmaufruf. Wird hinter dem Programmname der Benutzername geschrieben, so wird dieses als Parameter genommen. Wird jedoch nichts geschrieben, so gibt das Programm den ersten Text aus und wartet auf eine Eingabe. Der Benutzer kann jetzt seinen Namen eingeben und dieser wird entsprechend weiter verarbeitet. Damit ist das Verhalten des Programms von der Art des Aufrufs abhängig, was einem Programmierer sehr viele Möglichkeiten gibt, solcher Programme flexibel und vielfältig zu benutzen.

Statt Texten werden von Computern häufig Zahlen verarbeitet. Soll beispielsweise eine Quadratzahl berechnet werden, so kann man schreiben

```
LINK
  real "Bitte geben Sie eine Zahl ein: ", Wert;
DO
  write "Das Quadrat von ", Wert, " ist ", Wert**2, newline;
DONE
```

Dieses Programm liest den Parameter auf dem Bezeichner “`Wert`“ ein und gibt das Ergebnis sofort wieder aus. Werden mehrere Parameter benötigt, so werden diese entsprechend nacheinander aufgeschrieben.

```
LINK
  real "Bitte geben Sie den ersten  Faktor ein: ", Wert1:
  real "Bitte geben Sie den zweiten Faktor ein: ", Wert2;
DO
  write
    "Das Produkt von ", Wert1, " und ", Wert2, " ist ", Wert1*Wert2,
    newline;
DONE
```

Wichtig ist auch hierbei einzusehen, daß Programme möglichst einfach und mit selbst erklärenden Benennungen geschrieben werden sollen. Andernfalls wird es praktisch unmöglich sein, auch die einfachsten Programm noch nach kurzer Zeit zu verstehen, zu verändern oder Fehler in ihnen zu finden. Aus diesem Grunde kann die Eingabe noch etwas verfeinert werden. Es ist möglich auch in dem „Link-Abschnitt“ beliebige Texte auszugeben, so daß auch der Benutzer eines Programms leicht verstehen kann, was er eingeben muß und was das Programm berechnet.

```
LINK
  write "Programm zur Multiplikation zweier Zahlen", newline;
  real "Bitte geben Sie den ersten  Faktor ein: ", Wert1;
  real "Bitte geben Sie den zweiten Faktor ein: ", Wert2;
DO
  write
    "Das Produkt von ", Wert1, " und ", Wert2,
     " ist ", Wert1*Wert2,
    newline;
DONE
```

Ein „gutes Programm“ ist somit nicht nur richtig und schnell, sondern auch und vor allem gut strukturiert und leicht verständlich. In dem letzten Beispiel wurde dieses noch einmal dadurch deutlich gemacht, daß die `write`-Anweisung, die sehr viele Parameter enthält, auf verschiedene Zeilen verteilt wurde und durch Einrücken der Zusammenhang zwischen den Anweisungen verdeutlicht wurde.

Aufgaben

1. Nennen Sie in dem letzten Beispiel sämtliche Schlüsselwörter!
2. In dem letzten Beispiel werden die Zeichen "`;`" und "`,`" in einer besonderen Bedeutung verwendet. Erläutern Sie diese. Wir werden im folgenden auch solche Zeichen zu den Schlüsselwörtern einer Programmiersprache rechnen.
3. Die Wörter „`Wert1`“ und „`Wert2`“ im letzten Beispiel werden auch als Variablen (*variable*) bezeichnet. Diese Benennung soll andeuten, daß sich ihr Wert ändern kann. Welche Programme in den bisherigen Beispielen verwenden keine Variablen, und welche anderen Variablen finden Sie in den Programmen?
4. Wie werden Variablen in ***FirstStep*** gebildet? Vergleichen Sie dieses mit der Bildungsregel für Schlüsselwörter.
5. Schreiben Sie ein Programm, welches einen Namen einliest und die erste Zeile eines Briefes „Sehr geehrter Herr ...,“ausgibt.

3 Bestandteile einer Programmiersprache

Offenbar besteht ein Programm aus verschiedenen Teilen, die in einer wohlgeordneten Weise zusammengestellt sind. Die vernünftige Anordnung dieser Teile stellt eine wichtige Möglichkeit dar, Programme übersichtlich zu gestalten. In diese Kapitel sehen wir uns die Struktur eines ***FirstStep***-Programms sowie dessen Komponenten etwas genauer an. Außerdem gehen wir tiefer auf den Sinn von Strukturierung ein.

3.1 Die Struktur eines ***FirstStep***-Programms

Die Programmiersprache ***FirstStep*** sieht verschiedene Bereiche vor, in die jeweils bestimmte Teile des Programms eingegeben werden können. Diese Bereiche umfassen beispielsweise die Verbindung zu anderen Programmen oder „Modulen“, die Definition von Datentypen oder Funktionen sowie die Deklaration von individuellen Bezeichnern für variable Werte. Außerdem gibt es einen Anweisungsteil, der wie bereits im letzten Kapitel gezeigt, mit „`DO`“ anfängt und mit „`DONE`“ aufhört. Die anderen Bestandteile eines ***FirstStep***-Programms kann man dem folgenden Schema entnehmen:

```
LINK
  -- Eingabeaufforderung
DISCLOSE
  -- Anbindung anderer Module an dieses Programms
DEFINE
  -- Definition von Datentypen und Funktionalitäten
DECLARE
  -- Deklaration von Datenbereichen und deren Bezeichnern
DO
  -- auszuführende Anweisungen
DONE
```

Zunächst wird die Verbindung zum Benutzer des Programms durch „`LINK`“ hergestellt, wie im ersten Kapitel in Beispielen gezeigt. Man kann hier verschiedene Datentypen eingeben wie Zeichen, Texte, ganze Zahlen, reelle Zahlen oder komplexe Zahlen sowie logische Abfragen durchführen.

Der `DISCLOSE`-Bereich erlaubt die Angabe von konstanten Texten, die als Dateinamen interpretiert werden; in diesen Dateien müssen entsprechende ***FirstStep***-Module gespeichert sein, die dann eingelesen und deren Funktionalitäten oder Datentypen in diesem Programm verwendet werden kann. Dieses stellt ein mächtiges Werkzeug zur Strukturierung komplexer Programme dar.

Der `DEFINE`-Bereich erlaubt die Festlegung von Eigenschaften von Daten bzw. Funktionen. Hier können beispielsweise Funktionen definiert werden, die an verschiedenen Stellen des Programms aufgerufen und ausgeführt werden können. Beispielsweise kann die Lösung eines Gleichungssystems, die Berechnung eines speziellen Polynoms oder die Eingabe eines Werts in einer solchen Funktion untergebracht werden. Zur deutlichen Unterscheidung werden sowohl Funktionen als auch Prozeduren definiert, wobei Prozeduren keinen Wert erzeugen, sondern einen anderen Effekt haben. Dafür werden wir noch Beispiele bringen.

In diesem Bereich können außerdem Datentypen definiert werden. Die üblichsten sind **Felder** (*array*), die eine bestimmte Anzahl von Werten mit gleichen Datentypen enthalten, sowie Datensätze (*record*), die Werten mit verschiedenen Datentypen enthalten können. ***FirstStep*** nennt einen **Datensatz** „`enclosure`“, was als Umhüllung oder auch Briefumschlag aufgefaßt werden kann. Dieses liegt daran, daß solche Enclosures mittels einem generellen Konstrukt, dem `disclose`, „geöffnet“ werden können.

Zusätzlich lassen sich in solche Datensätze noch Funktionen, die natürlich vor allem mit den Werten aus den jeweiligen Datensätzen arbeiten, definieren. Solche Datensätze werden aus historischen Gründen als **Klassen** (*class*) bezeichnet. Da solche Klassen auch die Definitionen und Deklarationen anderer Klassen übernehmen können, bilden sie ein mächtiges Werkzeug, um komplexe Programme zu erstellen.

Der `DECLARE`-Bereich erlaubt die Bereitstellung von Speicherplatz und die Zuordnung eines Namens zu diesem Speicherplatz. Hier lassen sich beispielsweise ganze Zahlen, Text oder andere Größen speichern, deren Werte sich abhängig von dem Programm verändern. Dieser Bereich stellt somit einen zentralen Bereich dar, um überhaupt die Funktionalität eines Programms, nämlich Daten zu berechnen, zu ermöglichen. Dazu werden den Variablen jeweils Werte zugewiesen, die in der Regel aus den Werten anderer Variablen schrittweise berechnet werden.

Der letzte Abschnitt ist der `DO`-Bereich. In diesem werden die Anweisungen aufgelistet, die den Variablen eines Programms Werte zuweisen. Da zugleich Ergebnisse ausgegeben werden können, kann das Programm die Resultate seiner Berechnung dem Benutzer einfach mitteilen. Natürlich können diese Ergebnisse auch längerfristig in Dateien gespeichert werden, oder auch andere Rechner geschickt werden. Dieses soll aber erst später genauer ausgeführt werden.

Die Struktur eines ***FirstStep***-Programms erlaubt also die Festlegung sowohl der Art der verwendeten Daten als auch der Ergebnisse durch schrittweise Berechnung. Dieses soll in einer möglichst übersichtlichen und für einen Leser leicht verständlichen Form geschehen. Dazu können zum einen Funktonen und Prozeduren, aber auch komplexere Datenstrukturen wie Felder, Datensätze oder Klassen verwendet werden.

3.2 Struktur und Strukturierung

Welchen Nutzen hat jetzt der Programmierer davon, wenn er seine Programme sehr übersichtlich schreibt? Diese Frage ist relativ zentral, obgleich sie in frühen Programmiersprache nahezu nicht beachtet wurde, und auch moderne Programmiersprache hierzu nicht immer optimale Unterstützung anbieten.

3.2.1 Kommentare

Ein kleiner Programm, welches nur aus zwei oder drei Anweisungen besteht, braucht sicherlich keine besondere Strukturierung. Man wird die Anweisung hintereinander schreiben und unter der Voraussetzung, daß der Leser eines Programm weiß, was das Programm tun soll, wird er diese Anweisungen schon verstehen. Dennoch sollte man die Möglichkeit, die Bedeutung dieser Anweisungen genauer zu beschreiben, nicht vernachlässigen. In ***FirstStep*** gibt es dazu die Möglichkeit, eine Bedeutung in den Programmtext hineinzuschreiben. Dieses sieht etwa so aus:

```
<<<< Programm zur Berechnung der Nullstelle eines Polynoms >>>>
DO
Ergebnis := (- Bpar + sqrt(Bpar* Bpar -4*Apar*Cpar) ) -- Zähler
               /(2*Apar)                              -- Nenner
DONE
```

Hier wird zunächst eine generelle Beschreibung des Programms gegeben und dann werden einzelne Teile erläutert. Solche zusätzlichen Anmerkungen werden auch als **Kommentare** (*comment*) bezeichnet. In ***FirstStep*** können beliebige Texte in Klammern „`<<<...>>>`“ gesetzt werden, wobei mindestens drei solcher „<“-Zeichen einen Kommentar einleiten, und mindestens drei „>“-Zeichen einen Kommentar beenden. Dazwischen kann jeder beliebige Text stehen, außer natürlich „>>>". Solche Kommentare werden verwendet, um größere Abschnitte eines Programmes zu erklären.

Danach folgt dann als Beispiel die Möglichkeit, Kommentare anzugeben, die nur für einen kurzen Abschnitt, bis zum Zeilenende gelten. Solche Kommentare helfen, einzelne Anweisungen zu erläutern, wenngleich in vielen Fällen einzelne Anweisung so sehr ihre Bedeutung selbst erklären, daß sich ein Kommentar erübrigt. In manchen Bereichen, z.B. bei komplexen mathematischen Auswertungen, sind solche **Kurzkommentare** aber dennoch sehr hilfreich und sollten entsprechend auch genutzt werden.

Dennoch sind Kommentare nicht immer hilfreich. Hier sollen einige Gründe angegeben werden, warum Kommentare auf jeden Fall nicht beliebig, sondern mit Verstand eingesetzt werden müssen.

1. Ein Programm, welches zu viele Kommentare enthält, kann leicht allein dadurch unübersichtlich werden. Aus diesem Grunde sollten längere Kommentare, so weit sie nötig sind, deutlich vom Programmtext abgesetzt werden. Auch Kommentare müssen, ähnlich wie Programme, strukturiert werden. Hier sollten ebenfalls aussagekräftige Überschriften, Einrückungen usw. gewählt werden. Allerdings sollten die Kommentare nicht das Handbuch oder die Dokumentation zu dem Programm ersetzen!
2. Kommentare, die einfache Vorgänge erläutern, sind meistens überflüssig. Die Anweisung

   ```
   Anzahl := Anzahl + 1;         -- inkrementiere Anzahl
   ```

   benötigt keinen Kommentar, da jeder, der etwas Verständnis von einer Programmiersprache hat, sofort die Bedeutung dieser Anweisung erkennt; hier wird ein Zähler inkrementiert. In solchen Situationen machen Kommentare solche Programme nur unübersichtlich und schwieriger zu verstehen. Besser ist es in jedem Fall, aussagekräftige Namen für solche Variable zu wählen, die einen Kommentar stets überflüssig machen!
3. Ein Kurzkommentar sollte eindeutig einer bestimmten Anweisung, die er kommentiert, zugeordnet werden. Es sollten also nicht mißverständlich eine andere Anweisung gemeint sein können. Deshalb sollten Kurzkommentare in der Regel hinter einer Anweisung stehen und möglichst nicht in der Zeile davor, niemals jedoch in der Zeile dahinter. Steht ein Kurzkommentar in der Zeile vor einer Anweisung, so sollte davor eine Leerzeile stehen.
4. Werden Anweisungen gelöscht, so müssen auch die Kommentare mitgelöscht werden. Nichts kann einen Leser mehr verwirren als Kommentare, die sich auf nicht mehr existierende Anweisungen beziehen! Aus diesem Grunde sollte in je-

der Zeile höchstens eine Anweisung und ein Kurzkommentar stets in der gleichen Zeile wie die Anweisung stehen; wird die Anweisung gelöscht, so hoffentlich gleich die ganze Zeile, also auch der Kommentar.

5. Werden Anweisungen kopiert und danach leicht modifiziert, so ist natürlich auch der Kommentar zu dieser Anweisung zu ändern. Das wird leider häufig vergessen und führt dann zu Anmerkungen, die nur noch verwirren können.
6. Kommentare sind keine Lyrik oder Stilübung. Sie müssen den jeweiligen Sachverhalt einfach und prägnant beschreiben, aber natürlich auch vollständig sein. Allzu weitschweifige Darbietungen sind nicht hilfreich und auch zu arbeitsintensiv. Hier kann u.U. auch ein Verweis auf die Dokumentation zu dem Programm sinnvoll sein.
7. Kommentare sollten erläutern und insbesondere den üblichen Fachjargon überwinden helfen. Daher sind irgendwelche sehr intimen Abkürzungen oder Anmerkungen kontraproduktiv.
8. Kommentare müssen sachlich sein. Gelegentlich findet man in Programmen sehr persönliche Meinungen des Programmierers zu Tagesereignissen, seiner Einschätzung der charakterlichen Eigenschaften seines Chefs bzw. anderer Personen oder andere saloppe Ausfälle, die in einem solch quasi-offiziellen Dokument nichts zu suchen haben!

Diese Liste zeigt, daß das Kommentieren sehr wohl gelernt werden muß, ehe es vernünftige Ergebnisse zeitigt. Soll ein Programm kommentiert werden, so geht man am besten in folgenden Schritten vor. Man überlege sich zunächst, wer das Programm verstehen soll. Dieser „virtuelle Leser“ wird ein gewisses Vorwissen haben, so daß triviale Fakten, wie die Auswertung von Ausdrücken oder ähnliches, als bekannt vorausgesetzt werden müssen. Nur darüber hinaus gehende Erläuterungen sind jeweils anzuführen.

Der Anfang eines Programms sollte einen Titel, eine sehr kurze Beschreibung der gesamten Aufgabe, den Namen des Autors, das Erstellungsdatum, sowie Revisionen, deren Daten und Autoren, enthalten. Dieses sollte deutlich hervorgehoben und von dem Programmtext abgesetzt werden.

Das strukturierte Programm sollte dann in logische Abschnitte untergliedert werden, ähnlich den Kapiteln eines Buches. Jeder Abschnitt wird wiederum mit einer entsprechenden Einführungskommentar versehen.

Innerhalb von Abschnitten werden umfangreichere Berechnungen, Definitionen usw. ebenfalls wiederum mit Einführungskommentaren versehen.

Einzelne Anweisungen sollten nur dann kommentiert werden, wenn ihre Bedeutung anders nur schwer verstanden werden kann. Bei einer guten Wahl der Bezeichner für Variablen ist dieses häufig überflüssig.

Um den Zusammenhang zwischen Kommentar und Anweisung deutlich zu machen, sollte der Kommentar möglichst in die Anweisung geschrieben werden. Dadurch wird beispielsweise verhindert, daß beim Löschen oder Umstellen von Anweisungen Kommentare „übrig“ bleiben und den Leser eines Programms verwirren. In ***FirstStep*** kann ein beliebiger, dann als Kommentar aufgefaßter Text vor den Zuweisungsoperator einer Anweisung und vor dem abschließenden Semikolon „`;`“ gesetzt werden.

```
<<<< Programm zur Berechnung der Nullstelle eines Polynoms >>>>
DO
Ergebnis "groessere Nullstelle einer Gl. 2. Grades" :=
    (- Bpar + sqrt(Bpar* Bpar -4*Apar*Cpar) )
    /(2*Apar) "allg. Formel zur Loesung einer Gl. 2. Grades";
DONE
```

Auch hier sollte jedoch eher sparsam kommentiert werden, da auch durch diese Konzepte nicht immer die Übersichtlichkeit vergrößert wird.

3.2.2 Strukturen in Ausdrücken

Größere Programme sollten durch geeignete informatische Methoden strukturiert werden. „Größer“ ist hier sicherlich als relativ anzusehen, so daß nur schwer eine Anzahl von Anweisungen oder Programmzeilen als „klein“, „mittel“ oder „groß“ angesehen werden kann. Umfangreiche Programme werden auf jeden Fall in mehrere Abschnitte gegliedert werden, welche dann zu strukturieren sind. Hierbei sollte der logische Gesichtspunkt im Vordergrund stehen, d.h. einer Gruppe von Anweisungen sollte eine logische Bedeutung zugeordnet werden können und diese wird dann im Kommentar erläutert.

Viele Programmiersprache bieten Konstrukte an, welche das Strukturieren von Programmen erleichtern. Ein wichtiges Konstrukt ist ein Ausdruck, der aus Sonderzeichen oder anderen Schlüsselwörtern zusammengesetzt ist. Der Ausdruck

```
Ergebnis := 2*(Wert1 + Wert2)/Anzahl + Basis;
```

erlaubt dem Programmierer, eine eigentlich recht komplexe Rechenoperation sehr viel einfacher zu schreiben. Tatsächlich besteht dieser Ausdruck aus verschiedenen Teiloperationen, die vom Compiler erkannt und in der richtigen Reihenfolge zusammengesetzt werden müssen. Ein Computer berechnet das Ergebnis beispielsweise durch die folgende Anweisungsfolge:

```
Hilf1 := Wert1 + Wert2;
```

```
Hilf1 := 2*Hilf1;
Hilf1 := Hilf1 / Anzahl;
Ergebnis := Hilf1 + Basis;
```

Es scheint klar zu sein, daß diese Art der Darstellung eines Programms sehr viel unübersichtlicher ist als ein strukturierender Ausdruck wie der obige Text. Dieses gilt auch für nicht mathematische Anweisungen wie

```
Text := "Anfang: " + TextWert[3,7] + " ----> ";
```

In diesem Beispiel wird ein neuer Textwert berechnet, indem drei Teiltexte aneinander gehängt werden, wobei von der Textvariablen nur ein Teilstück verwendet wird. Hat diese den Wert `TextWert == "Urvogel"`, so erhalten wir als Ergebnis

```
TextWert == "Anfang: vogel -----> "
```

Diese Notation ist zwar ungewohnt aber sicherlich verständlicher als eine explizite Zerlegung der einzelnen Zwischenschritte in elementare Anweisungen.

Diese Beispiele sollen zeigen, daß bereits ein aus Sonderzeichen, Konstanten und Variablen gebildeter Ausdruck eine komplexe Berechnung sehr viel übersichtlicher machen kann als die eigentlich dahinter steckende Anweisungsfolge. Allerdings erhält man in dem Falle, wenn man die gewohnte mathematische Notation verläßt, stets auch eine zunächst ungewohnte Notation, die hoffentlich so sinnfällig ist, daß auch der unbedarfte Leser leicht die Bedeutung der jeweiligen Ausdrücke versteht. Auch hier kann von der verwendeten Programmiersprache letzten Endes nicht zu viel Unterstützung gewährt werden.

3.2.3 Struturieren von Anweisungsfolgen

Es wurde bereits erwähnt, daß ein Programm die Reihenfolge, in der Anweisungen ausgeführt werden, festlegt. Im einfachsten Fall werden die einzelnen Anweisungen angegeben, jeweils getrennt durch ein besonderes Zeichen. Dieses ist in ***FirstStep*** ein Semikolon, in anderen Sprachen ein Punkt, das Zeilenende oder auch eine Klammerung. In den obigen Beispielen lassen sich verschiedene Anweisungsfolgen finden.

```
LINK
  write "Gebe die ersten vier Quadratzahlen aus.", newline;
DECLARE
  integer Zahl;
DO
  Zahl := 1;
```

```
    write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
    Zahl := 2;
    write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
    Zahl := 3;
    write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
    Zahl := 4;
    write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
  DONE
```

Man hat hier acht Anweisungen vorliegen, die nacheinander ausgeführt werden. Zunächst wird der Wert von `Zahl` auf eins gesetzt und dann ein Text mit dieser Zahl ausgegeben; die Ausgabe eines Textes gilt als <u>eine</u> Anweisung. Dann wird der Wert von `Zahl` auf zwei gesetzt und das Quadrat von zwei ausgegeben, usw. Das ist zwar ein mögliche Lösung, aber es ist offensichtlich, daß diese sehr umständlich und vor allem sehr inflexibel ist. Sollen beispielsweise die Quadratzahlen der Werte von 10 bis 20 ausgegeben werden, so kann dieses Programm nicht mehr benutzt werden.

Die Lösung dieser Problems liegt darin, sehr ähnliche Anweisungen mehrfach auszuführen, wozu sogenannte **Schleifen** (*loop*) verwendet werden. In ***FirstStep*** gibt es die sogenannte `for`-Schleife, die sämtliche vorkommenden Schleifenarten abdeckt.

```
  LINK
    write "Gebe die ersten vier Quadratzahlen aus.", newline;
  DO
    for Zahl := 1 to 4 do
      write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
    done for Zahl;
  DONE
```

Die Anweisung besteht also aus einer Bereichsangabe (`for Zahl:=1 to 4`) und einem Block mit einer oder mehreren Anweisungen (`do...done`). Hinter dem Schlüsselwort `done` steht dabei zur deutlichen Kennzeichnung, um welche Schleife es sich handelt, der Schleifenanfang (`for Zahl`). Dieses ist bei verschachtelten Schleifen, bei denen sich innerhalb des `do...done` –Blocks weitere Schleifen befinden, sehr hilfreich. Der Name der **Laufvariablen** (hier `Zahl`) wird somit zugleich zum Namen der Schleife. Dieses ist bei verschiedenen Anwendungen nützlich, wie wir noch sehen werden.

Man beachte eine weitere Vereinfachung. Die Laufvariable braucht hier nicht mehr als Variable deklariert zu werden. Sie ist **implizit** (*implicit*) deklariert. Ihr „Typ", d.h. die Art der Werte, die sie speichern kann, wird durch den Ausdruck rechts vom `:=`-Zeichen festgelegt. Hier ist es ein ganzzahliger Wert (`integer`).

Natürlich kann das obige Programm auch einfach parametrisiert werden, indem statt der festen Grenzen variable Grenzen eingegeben werden. Beispielsweise kann man schreiben:

```
LINK
  write "Gebe die Quadratzahlen aus.", newline;
  integer "Erste  Quadratzahl = ",  Erste;
  integer "Letzte Quadratzahl = ",  Letzte;
DO
  for Zahl := Erste to Letzte do
    write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
  done for Zahl;
DONE
```

Der Anwender dieses Programms erhält also wieder die Möglichkeit, durch einfache Eingabe der Wertebereiche die Berechnung genau zu steuern.

In manchen Anwendungen ist es interessant, wenn gewisse Werte nicht ausgegeben werden. Beispielsweise interessiert die Quadratzahl von 10 nicht besonders, da dieser Wert den meisten Leuten bekannt sein dürfte. Möchte man diese Zahl auslassen, so kann man schreiben:

```
LINK
  write "Gebe die Quadratzahlen aus.", newline;
  integer "Erste  Quadratzahl = ",  Erste;
  integer "Letzte Quadratzahl = ",  Letzte;
DO
  for Zahl := Erste to Letzte if Zahl<>10 do
    write Zahl, " zum Quadrat ergibt ", Zahl**2, newline;
  done for Zahl;
DONE
```

Dieses ist besonders dann wichtig, wenn bei einer Berechnung ein falscher Ausdruck entstehen würde, beispielsweise durch null dividiert wird. Die folgende Schleife berechnet eine Folge von Werten, die für bestimmte Parameter nicht definiert sind.

```
LINK
  write "Gebe eine Wertefolge f(Wert) aus.", newline;
  integer "Erster  Wert = ",  Erster;
  integer "Letzter Wert = ",  Letzter;
DO
  for Zahl := Erster to Letzter if Zahl-5<>0 do
    write
      "f(", Zahl, ") = ", (Zahl+Letzer-Erster) / (Zahl-5), newline;
  done for Zahl;
DONE
```

Man kann somit dem Programm sagen, für gewisse Werte die Berechnung nicht durchzuführen, d.h. den jeweiligen Schleifendurchlauf zu „überspringen“. Das ist gerade für selten auftretende Ausnahmefälle sehr wichtig. Da hinter dem `if` ein beliebiger logischer

Ausdruck stehen kann, kann diese Konstruktion für verschiedene Fälle eingesetzt werden.

In den letzten beiden Beispielen war stets die Bedingung „falsch“, d.h. nur wenn `Zahl` einen bestimmten Wert nicht hatte, wurde der Schleifenrumpf übersprungen. Das ist unter Umständen semantisch etwas verwirrend, da man auch mit mehrfachen Negationen rechnen muß. Aus diesem Grunde kann auch geschrieben werden

```
LINK
  write "Gebe eine Wertefolge f(Wert) aus.", newline;
  integer "Erster  Wert = ",  Erster;
  integer "Letzter Wert = ",  Letzter;
DO
  for Zahl := Erster to Letzter unless Zahl-5 == 0 do
    write
      "f(", Zahl, ") = ", (Zahl+Letzer-Erster) / (Zahl-5), newline;
  done for Zahl;
DONE
```

was exakt das gleiche bedeutet. Das Schlüsselwort `unless` (außer wenn …) sagt jetzt aber dem Programm direkt, daß es den Schleifenblock nur ausführen soll, wenn die nachfolgende Bedingung falsch ist. Man beachte, daß dieses jeweils zu exakt den gleichen Ergebnissen führt, so daß der Programmierer sich hier allein von seinem Sprachgefühl leiten lassen kann. Es sollte aber stets der einfachere logische Ausdruck gewählt werden, da logische „Umstellungen“ leicht mißverständlich sind.

An dieser Stelle soll noch ein wenig etwas zu logischen Ausdrücken gesagt werden. Diese sind in der Regel sogenannte **Aussagen**, die natürlicherweise wahr oder falsch sein können. Beispielsweise sind folgendes wahre Aussagen:

$2+2 = 4$; $10 > 8$; „Sonntag ist kein Werktag“; $(x-y)*(x+y) = x*x - y*y$;

Beispiele für falsche Aussagen sind

$2+3 = 6$; $10 > 12$; „Sonntag regnet es nie“; $(x+y)*(x+y) = x*x + y*y$;

Derartige Werte werden in der Informatik als die logischen Werte **wahr** (`true`) oder **falsch** (`false`) bezeichnet. Nach dem englischen Mathematik George Boole werden sie auch als **boolesche** (*boolean*) Werte bezeichnet.

In der Informatik werden häufig ähnliche Ausdrücke wie in der Mathematik verwendet, wobei die logische Gleichheit häufig durch “=“ ausgedrückt wird. In manchen Sprachen wird das Zeichen “=“ auch für die Wertzuweisung, d.h. unser Zeichen “`:=`“, verwendet, Um hier Verwechslungen vorzubeugen, verwenden wir für die logische Gleichheit das Zeichen “==“, wie es auch in C oder C++ üblich ist. Das Zeichen “=“, wird von uns gar nicht verwendet.

4 Werte und Ausdrücke

Unter einem „Wert“ wird häufig eine Zahl verstanden. In der Informatik wird dieser Begriff jedoch sehr viel weiter gefaßt. Hier kann auch ein Zeichen (‘A‘, ‘B‘, usw.) ein Text oder manchmal sogar ein komplexer Datensatz, der aus mehreren Zahlen, Zeichen usw. besteht, als Wert aufgefaßt werden. Auch könnte beispielsweise eine Berechnungsvorschrift, also eine Funktion, als Wert verstanden werden. Daher soll hier als Wert jedes vernünftige Ergebnis einer Berechnung verstanden werden. Mathematisch könnte man auch sagen, daß ein Wert ein Element einer wohldefinierten Menge ist.

Werte werden in der Regel berechnet, was häufig einfach durch einen Ausdruck durchgeführt werden kann. Für solche Ausdrücke werden häufig mathematische Ausdrücke verwendet, so daß ein Programmierer mit entsprechenden mathematischen Vorkenntnissen einfach solche Ausdrücke schreiben bzw. verstehen kann. Allerdings wird dieser Begriff in der Informatik auch auf nicht mathematische Ausdrücke angewendet.

In diesem Kapitel werden einfache Werte und Ausdrücke zur Berechnung solcher Werte behandelt. Dabei gehen wir zunächst auf Konstante ein und betrachten dann die verschiedenen Ausprägungen von Ausdrücken in ***FirstStep***.

4.1 Konstante

4.1.1 Einfache Konstante

Soll sich der Wert eines Merkmals nicht ändern, so nennt man diesen **konstant** (*constant*), das Merkmal eine **Konstante**. Beispiele sind Zahl- oder Textkonstanten wie

```
3.14159265, 42, "Dies ist ein Text"
```

oder Bezeichner, denen explizit ein Wert zugewiesen wird, der sich im Laufe des Programms nicht mehr ändert.

```
constant real Pi := 3.14159265
constant text Hallo := "Dies ist ein Text"
```

Im Programm kann jetzt beispielsweise statt der Zahl `3.1415...` jeweils `Pi` geschrieben werden, was einfacher und sicherer ist.

4.1.2 Aufzählungen

In verschiedenen Anwendungen ist es nötig, nicht nur einzelne Konstanten, sondern eine zusammengehörende Gruppe endlich vieler Merkmale zu bezeichnen. Beispielsweise sind in einem Bekleidungskatalog Farben wichtig (`Rot`, `Gelb`, `Grün` ...) oder bei Schreibwaren das Format von Papier (`DINA0`, `DINA1`, `DINA2`, `DINA3`, `DINA4`, `DINA5`, ...), oder die Wochentage in einem Kalender (`Montag`, `Dienstag`, ... `Sonntag`). Natürlich lassen sich solche Daten jeweils als Zahlen darstellen, was jedoch verschiedene Nachteile hat. Zum einen sind die Zahlen nicht selbsterklärend, so daß die Bedeutung der Werte jeweils in einem Kommentar angegeben werden müßte. Das ist unübersichtlich, umständlich und fehlerträchtig. Stattdessen wird man besser selbsterklärende Konstanten definieren, so daß jeder Wert jeweils einen eigenen Namen erhält. In ***FirstStep*** schreibt man für Konstante im allgemeinen

```
constant integer Montag := 0, Dienstag := 1, Mittwoch := 2;
```

Allerdings ist hier weder der Bereich der Werte vom Rechner überprüfbar, noch kann verhindert werden, daß mit diesen Zahlen gerechnet wird oder andere arithmetische Operationen durchgeführt werden, obgleich `Dienstag+Donnerstag` oder `Rot*Blau` sicherlich keine sinnvolle Bedeutung haben. Aus diesen Gründen ist es sinnvoll und üblich, Mengen von Werten einzuführen, die einem Datentypen entsprechen.

```
type Wochentag == values Montag, Dienstag, Mittwoch, Donnerstag,
                         Freitag, Samstag, Sonntag;
```

Die Deklaration ist zu lesen als: „Der Typ `Wochentag` umfaßt die Werte `Montag`, `Dienstag`, ..., `Sonntag`."

Ähnlich wie es einen Typ `integer` oder `real` gibt, so gibt es jetzt einen vom Programmierer definierten Typ `Wochentag`, dessen Werte jedoch dem Programmierer nicht bekannt sind, so daß auch das arithmetische Rechnen mit diesen Werten keinen Sinn macht und vom Compiler verhindert werden kann. Die Rechnerdarstellung dieser Größen wird in der Regel durch eine Abbildung auf eine aufsteigende Folge von Zahlen definiert, wenngleich dieses im Prinzip nicht nötig wäre. Mit der Deklaration

```
declare Wochentag Tag1, Tag2;
```

sind jetzt beispielsweise Wertzuweisungen möglich

```
Tag1 := Montag;
Tag2 := Dienstag;
Tag1 := Tag2;
```

4.1.3 Aufzählungen mit Ordnungen

Aber es ist nicht möglich, `Montag` und `Dienstag` zu addieren. In manchen Anwendungen ist es jedoch sinnvoll, die Werte einer Aufzählung mit einer Ordnung zu verbinden. Dazu ist in ***FirstStep*** zu definieren

```
type Wochentag == values Montag < Dienstag < Mittwoch <
                  Donnerstag < Freitag < Samstag < Sonntag;
```

Dann kann man schreiben

```
Montag < Dienstag;
if Tag1 • Freitag then ...
```

4.1.4 Aufzählungen mit Zyklen

Die Ordnung ergibt sich aus der Reihung der Werte bei ihrer Deklaration; die <-Zeichen deuten darauf hin, daß hier eine Ordnung mit definiert ist. Sie entspricht der Ordnung der natürlichen Zahlen. Wenn man eine totale Ordnung hat, so läßt sich auch ein Nachfolger bzw. Vorgänger zu einem Element definieren. In ***FirstStep*** läßt sich auch der übernächste, drittnächste usw. Nachfolger einfacher bestimmen.

```
Montag succ 1 == Dienstag;
Freitag pred 2 == Mittwoch:
```

Hierbei steht `succ` für *successor* (**Nachfolger**), während `pred` für *predecessor* (**Vorgänger**) steht. Allerdings hat der letzte Wert in dieser Folge keinen Nachfolger und der erste keinen Vorgänger. Schreibt man daher

```
type Folge == cycle values Montag < Dienstag < Mittwoch <
                Donnerstag < Freitag < Samstag < Sonntag;
```

so hat man einen Zyklus definiert. Dann gibt es jetzt einen Vorgänger zu `Montag` bzw. einen Nachfolger zu `Sonntag`:

```
Sonntag succ 1 == Montag;
Montag pred 2 == Samstag;
```

In einem solchen geschlossenen Zyklus ist ein Vergleich hinsichtlich der relativen Größe der Werte definiert.

4.1.5 Aufzählungen mit Schwellwerten

Manche Aufzählungen werden nicht als Zyklus, sondern als Zähler verwendet, die niemals über eine ober Grenze bzw. unter eine unter Schranke verändert werden. Dieses kann in ***FirstStep*** durch einen Schwellwert (gauge) realisiert werden.

```
type PegelTyp == gauge values Leer < Wenig < Mittel < Viel < Hoch;
```

Ein solcher Wert kann ähnlich wie ein Zyklus verwendet werden, außer daß ein durch den Befehl `succ` über die obere Grenze `Hoch` hinaus reichender Wert auf den oberen Wert abgebildet wird. Entsprechendes gilt für den unteren Wert.

```
declare PegelTyp Pegel;
do
  Pegel := Mittel;
  Pegel succ 1 == Viel;
  Pegel succ 2 == Hoch;
  Pegel succ 3 == Hoch;
  Pegel pred 1 == Wenig;
  Pegel pred 2 == Leer;
  Pegel pred 3 == Leer;
```

4.1.6 Aufzählungen mit expliziten Werten

FirstStep gestattet die explizite Angabe der Werte der einzelnen Elemente einer Aufzählung, einer Folge, von Zyklen oder von Schwellwerten.

```
type Folge ==  integer values Montag := dual 0001,
               Dienstag := dual 0010, Mittwoch := dual 0011,
               Donnerstag := dual 0100, Freitag := dual 0101,
               Samstag := dual 0110, Sonntag := dual 0111 ;
```

Dieses ist u.a. beim maschinennahen Programmieren sinnvoll; bei höheren Anwendungen kann es ebenfalls nützlich sein, einer Konstanten außer ihrer Ordnung auch einen Wert zuzuweisen. In ***FirstStep*** ist die obige Zuordnung daher erlaubt. Weiterhin können die Konstanten einer Folge verglichen werden, wobei allerdings nicht der Wert der Konstanten als Ordnungskriterium verwendet wird, sondern ihre Aufzählungsreihenfolge. Der Wert einer solchen Konstanten kann durch den Befehl

```
value(Montag) == dual 0001;
```

ermittelt werden. Statt ganzer Zahlen können auch `real`-Zahlen, Zeichen oder Texte als Werte von Konstanten verwendet werden.

4.1.7 Ein- und Ausgabe von Aufzählungsvariablen

Die Aufzählungstypen werden intern im Programm aber auch extern vom Anwender durch ihren Namen (`Montag`, `Dienstag`, usw.) repräsentiert. Wird somit eine Variable ausgegeben, so wird der Name ausgedruckt,

```
write "Heut ist ", Tag1, newline; -- Ausgabe: Heute ist Montag
```

Wird statt dessen eine solche Variable eingelesen, so ist ebenfalls der Name der jeweiligen Aufzählungskonstanten anzugeben. Wird ein falscher Name eingegeben, so druckt das Programm eine Fehlermeldung, listet die korrekten Namen auf und wartet erneut auf eine Eingabe.

```
read Tag1;
```

4.1.8 Zusammenfassung

Die folgende Tabelle stellt die möglichen Arten von Konstanten und die darauf sinnvollen Operationen zusammen; außerdem ist die Art der möglichen Werte angegeben.

Art	Syntax	Operationen	Ordnung	Werte
Konstante	constant <type>	==, <>, :=	wie <type>	notwendig
Aufzählung	values (,)	==, <>, :=	nein	optional
Folge	values (<)	==, <>, :=, <, ≤, >, ≥, succ, pred	wie Folge	optional
Zyklus	cycle values (<)	==, <>, :=, <, ≤, >, ≥, succ, pred	wie Folge	optional
Schwellwert	gauge values (<)	==, <>, :=, <, ≤, >, ≥, succ, pred	wie Folge	optional

4.1.9 Beispiele

Das folgende Beispielprogramm druckt alle Wochentage eines Jahres aus.

```
DEFINE
  type Weekdays == cycle values Sunday < Monday < Tuesday
               < Wednesday < Thursday < Friday < Saturday ;
  type Months == integer values
    Jan := 31 < Feb := 28 < Mar := 31 < Apr := 30 <
    May := 31 < Jun := 30 < Jul := 31 < Aug := 31 <
    Sep := 30 < Oct := 31 < Nov := 30 < Dec := 31;
DECLARE
  Weekdays Day;
DO
  write "First day of the year is "; read Day;
  for Month:=Jan to Dec do
    write newline, "    Sun  Mon  Tue  Wed  Thu  Fri  Sat";
    if Day<>Sunday then
      write newline, Month, "  ";
      for Weekday:=Sunday to Day pred 1 do
        write "     ";
      done for Weekday;
    end if;
    for Today:=1 to value(Month) do
      if Day == Sunday then
        write newline, Month,"  ";
      end if;
      write Today:2, " | ";
      Day := Day succ 1;
    done for Today;
    write newline;
  done for Month;
DONE
```

Erläuterungen zu dem Kalenderprogramm: Nachdem der erste Wochentag des Jahres bekannt ist, werden die Monate durchlaufen. Falls der erste Monatstag ein Sonntag ist, so muß nicht eingerückt werden; ansonsten muß so viele Wochentage, wie nicht in diesen Monat fallen, entsprechend weit eingerückt werden, um die Tagesdaten in die gleiche Spalte unter den jeweiligen Wochentagen zu schieben. Sodann geht man alle Tage des Monats durch und gibt jeweils das Datum aus. Immer, wenn Sonntag erreicht ist, geht man auf die nächste Zeile und gibt zunächst den Monatsnamen aus. Ansonsten wird stets das Monatsdatum ausgegeben, so daß auch in den Folgezeilen alle Daten genau untereinander und den entsprechenden Wochentagen stehen.

Das folgende Beispielprogramm gibt für eine Folge von Werten einen sogenannten Gray-Code, bei dem sich bei benachbarten Werten jeweils nur ein Bit in der Kodierung verändert. Dazu werden zunächst die entsprechenden Werte initialisiert. Hier wird besonders ausgenutzt, daß eine Aufzählungskonstante nicht nur eine Ordnung, sondern auch einen Wert haben kann, was hier der jeweilige ganzzahlige Graycode ist. Nachdem der Benutzer eine Anfangsposition eingegeben hat, werden von dieser ab alle Bezeichner und Werte des Graycodes auf dem Terminal ausgedruckt.

```
DEFINE
  type GrayCode == integer values
    Zero   := dual 0000, One      := dual 0001, Two    := dual 0011,
    Three  := dual 0111, Four     := dual 0101, Five   := dual 0100,
    Six    := dual 0110, Seven    := dual 1110, Eight  := dual 1100,
    Nine   := dual 1101, Ten      := dual 1111, Eleven := dual 1011,
    Twelve := dual 1001, Thirteen := dual 1000;
DECLARE
  GrayCode Position := Four;
DO
  write "Current Position is ", Position,
        " with value ", value(Position), newline;
  write "From Value "; read Position;
  for Lauf := Position to Thirteen do
    write "Position ", Lauf, " with value ", value(Lauf), newline;
  done for Lauf;
DONE
```

4.2 Elementare Werte

In ***FirstStep*** gibt es **singuläre Werte**, die in der Regel nur einen festen, beschränkten Speicherplatz belegen, **variable Werte**, die einen von den jeweiligen Werten abhängenden Speicherplatz benötigen, sowie **zusammengesetzte Werte**, deren Speicherplatzbedarf von der Deklaration abhängt, in der Regel aber fest ist. Im folgenden werden diese Werte näher beschrieben.

4.2.1 Logische Werte

Die logischen Größen können nur zwei Werte annehmen, nämlich **wahr** (*true*) und **falsch** (*false*). Logische Werte werden auch als **boolsche Variable** (nach dem britischen Mathematiker *G. Boole*) bezeichnet. Wahrheitswerte können Variablen zugewiesen werden, ähnlich wie es bei Zahlenwerten der Fall ist. In ***FirstStep*** werden sie auf die folgende Weise deklariert:

```
boolean  WahrheitsWert, IstGroesserAls;
do
  WahrheitsWert := true;
  IstGroesserAls := Wert > 2.5;
```

Wahrheitswerte können in Ausdrücken verwendet werden, wie den folgenden:

```
WahrheitsWert := IstGroesserAls and (Wert < 250);
IstGroesserAls := not Wahrheitswert;
```

In den meisten Fällen werden sie für logische Steuerungen in `if`- oder `for-while`-Schleifen verwendet. Es gibt im Prinzip 16 verschiedene Operatoren, von denen jedoch eine Reihe trivial sind. In ***FirstStep*** werden `and` (logisches UND), `or` (logisches ODER) und `not` (logisches NICHT) verwendet. Außerdem gibt es den Vergleich zweier logisches Ausdrücke (durch ==) bzw. die Ungleichheit (<>), so daß insgesamt sämtliche logischen Operatoren einfach dargestellt werden können, beispielsweise die Exklusiv-Oder-Operation durch <>, die Implikation durch `A or (not B)`, usw.

Zum Vergleich von Größen wie Zahlen, Zeichen oder Texten, werden die üblichen mathematischen Operatoren `"<"`, `"<="`, `">"`, `">="`, `"<>"`,`"=="`, wobei wahlweise auch die Operatoren `"=<"`, `"=>"`, `"><"` verwendet werden können.

4.2.2 Zeichen

Ein Zeichenwert ist einer von wenigen möglichen Werten, wozu insbesondere die Zeichen des Alphabets, aber auch Ziffern oder Sonderzeichen wie Satzzeichen oder mathematische Zeichen gehören. Das sogenannte ASCII-Alphabet fast 128 dieser Zeichen zusammen (siehe Anhang). Ein Zeichenwert wird in ***FirstStep*** als character bezeichnet. Eine entsprechende Variable wird folgendermaßen deklariert:

```
character Zeichen := "+", AZeichen :="A", XZeichen :="X";
do
  AZeichen := "a";
  if AZeichen == "X" then ...;
```

```
XZeichen := AZeichen;
```

Zeichen können in der Regel nur zugewiesen werden und verglichen werden. In ***First-Step*** können Zeichen auch in Texte umgewandelt werden sowie zu einem Zeichen ein Zahlenwert berechnet werden, der der Nummer dieses Zeichens im ASCII-Alphabet entspricht; entsprechend kann der jeweilige Vorgänger oder Nachfolger dieses Zeichens bestimmt werden. Dieses geschieht durch die Befehle

```
AZeichen := AZeichen succ 3;        -- Macht aus "A" ein "D".
AZeichen := AZeichen pred 3;        -- Macht aus "X" ein "U".
Nummer := | AZeichen |;
```

Dabei berechnet der Absolutstrich |...| den ASCCI-Zahlenwert, während der Nachfolger durch `succ`, der Vorgänger durch `pred` bestimmt wird.

4.2.3 Zahlen mit ganzen Werten

In ***FirstStep*** werden ganze Zahlen durch `integer`-Variablen dargestellt. ***FirstStep*** verwendet hierfür die Darstellung der Zwischensprache, hier C, so daß in der Regel 32-Bit-Zahlen verwendet werden. Man schreibt z.B.

```
integer Wert, GanzeZahl:= 2;
integer Zahl := GanzeZahl;
```

für die Deklaration einer Variablen mit ganzzahligem Typ. Diese Werte können in den üblichen Grundrechenarten verwendet werden, d.h. sie können addiert, subtrahiert, multipliziert und auch dividiert werden. Bei der Division ist jeweils darauf zu achten, daß das Ergebnis auch ganzzahlig sein kann, aber nicht muß. Außerdem können Zahlenwerte mit den Vergleichsoperatoren "<", "<=", ">", ">=", "<>","==" miteinander verglichen werden; das Ergebnis ist ein logischer Wert. Hier ist darauf zu achten, daß diese Operatoren jeweils „Werte" verwenden, d.h. es ist gleich, ob es sich um ganze oder reelle Zahlen (siehe nächster Abschnitt) handelt; ***FirstStep*** vergleicht die jeweiligen Werte miteinander, wie das in der Regel auch in der Mathematik unter dieser Operation verstanden wird.

4.2.4 Zahlen mit reellen Werten

In ***FirstStep*** werden reelle Zahlen durch `real`-Variablen dargestellt. ***FirstStep*** verwendet hierfür die Darstellung der Zwischensprache, hier C (double), so daß in der Regel 64-Bit-Zahlen verwendet werden. Man schreibt z.B.

```
real Wert, ReelleZahl:= 2.5;
integer Zahl := ReelleZahl;
```

für die Deklaration einer Variablen mit reelwertigem Typ. Diese Werte können in den üblichen Grundrechenarten verwendet werden, d.h. sie können addiert, subtrahiert, multipliziert und auch dividiert werden. Zusätzlich darf der Potenzoperator "`**`" verwendet werden; der Programmierer muß darauf achten, daß die Potenz einer negativen Zahl nicht definiert ist.

Reelle Zahlenwerte können mit den Vergleichsoperatoren "`<`", "`<=`", "`>`", "`>=`", "`<>`","`==`" miteinander verglichen werden. Hier ist darauf zu achten, daß diese Operatoren jeweils „Werte" verwenden, d.h. es ist gleich, ob es sich um ganze (siehe vorigen Abschnitt) oder reelle Zahlen handelt; ***FirstStep*** vergleicht die jeweiligen Werte miteinander, wie das in der Regel auch in der Mathematik unter dieser Operation verstanden wird.

4.2.5 Zahlen mit komplexen Werten

In ***FirstStep*** werden komplexe Zahlen durch `complex`-Variable dargestellt. Eine komplexe Zahl kann als paar von reellen Zahlen verstanden werden, so daß auch die Notation für komplexe Zahlen in ***FirstStep*** entsprechend ist. Man schreibt z.B.

```
complex CWert, ComplexZahl := ( 2.5 , -3.5 );
complex Zahl := ComplexZahl;
```

für die Deklaration einer Variablen mit komplexwertigem Typ. Diese Werte können in den üblichen Grundrechenarten verwendet werden, d.h. sie können addiert, subtrahiert, multipliziert und auch dividiert werden.

Reelle Zahlenwerte besitzen keine Ordnung, so daß sie auch nicht mit den Vergleichsoperatoren "`<`", "`<=`", "`>`", "`>=`", miteinander verglichen werden können. Natürlich können sie weiterhin auf Ungleichheit und Gleichheit ("`<>`","`==`") verglichen werden, soweit das bei reellen Zahlen überhaupt Sinn macht.

4.2.6 Texte

In ***FirstStep*** werden Texte durch `text`-Variable dargestellt. Ein Text ist eine Folge von Zeichen, deren Länge von dem aktuellen Wert des Texts abhängt. Deshalb müssen Text in ***FirstStep*** als variable Werte dargestellt werden. Man schreibt z.B.

```
text TWert, Name := "Wolfgang";
text Vorname := Name;
```

für die Deklaration einer Variablen mit textwertigem Typ. Texte können addiert und multipliziert werden, womit das Aneinanderhängen (Konkatenation) von Texten bzw. das Vervielfachen des gleichen Texts gemeint ist. Außerdem kann durch einen Feldoperator ein Teiltext ausgeblendet werden. Der folgende Textausdruck

```
Text1 := "Jo";                          -- "Jo"
Text2 := Text1 * 2;                     -- "JoJo"
Text3 := Text2 + "-Spiel";              -- "JoJo-Spiel"
Text4 := Text3[3:7];                    -- "Jo-Sp"
```

ist ein Beispiel für Texte mit verschiedenen Operatoren. Zu einem Text kann auch ein Zeichen „addiert“ werden, was die gleich Bedeutung hat wie die Konkatenation eines Texts, dessen Wert dieses Zeichen ist.

```
Text5 := Text4 + AZeichen + 's' + 's';  -- "Jo-Spass"
```

Die Textverarbeitung in ***FirstStep*** ist sehr komplex, aber dennoch effizient und sicher. Sie kann eingesetzt werden, ohne daß sich der Programmierer über „zu lange“ oder „zu kurze“ Texte usw. Gedanken machen muß oder über Speicherabfall von „Textresten“, dem sogenannten ***garbage***.

4.2.7 Felder (*array*)

In ***FirstStep*** werden mehrere Werte gleichen Typs durch **Felder** (*array*) zusammengefaßt. Ein Feld ist eine Folge von Werten, deren Länge bei der Deklaration angegeben werden muß, aber dennoch in verschiedenen Programmläufen unterschiedlich sein kann. Deshalb müssen Felder in ***FirstStep*** als variable Werte dargestellt werden. Man schreibt z.B.

```
real array Feld1[1:20] := 0.0 .. 0.0;
real array Feld2 [1:FeldLaenge, 2:12];
```

für die Deklaration eines Felds, dessen Variable einen reellwertigen Typ besitzen. Der Zugriff auf ein Feld geschieht über die Angabe einer entsprechenden Anzahl ganzzahliger Ausdrücke in der Index-Klammerung, also

```
Nummer := 22 + Feld1[Index+1];
Feld2 [Index, 8] := Nummer * Feld1[20];
```

Die Typen der Feldelemente können in ***FirstStep*** neben ganzen oder reellwertigen Zahlen auch Texte, Zeichen, komplexe Zahlen oder Datensätze sein.

4.2.8 Datensätze (*enclosure*)

In ***FirstStep*** werden mehrere Werte unterschiedlichen oder gleichen Typs durch **Datensätze** (*enclosure*) zusammengefaßt. Ein Datensatz ist eine Menge von Werten, die in allen Programmläufen gleich sein muß. Deshalb können Datensätze in ***FirstStep*** als singuläre Werte dargestellt werden. Man schreibt z.B.

```
define type Stammdaten == enclosure
                             text Name:="Name", Vorname:="Vorname";
                             text Strasse, Ort;
                             integer PLZ:=00000, Hausnummer:=0;
                          end enclosure
declare
  Stammdaten StammWert;
```

für die Deklaration eines Datensatzes mit Variablen textwertigen und ganzzahligen Typs. Der Zugriff auf die Elemente eines Datensatzes geschieht über die Punktnotation

```
Nummer := StammWert . PLZ;
StammWert . Strasse := "Ulmenweg";
StammWert . Hausnummer := 23;
```

Die Typen der Feldelemente können in ***FirstStep*** alle anderen Datentypen sein, einschließlich Felder, Datensätze oder Referenzen (siehe nächsten Abschnitt). Die Werte von Datensätzen können auch initialisiert werden, wie in dem Beispiel gezeigt wurde.

Da Datensätze Referenzen auf Datensätze des gleichen Typs enthalten können, kann man mit ihnen beliebige komplexe Strukturen erstellen, die auch als **Graphen** (*graph*) bezeichnet werden.

4.2.9 Referenzen (*reference*)

Eine Referenz ist ein Verweis auf ein Objekt. Referenzen können in ***FirstStep*** nur auf Datensätze sowie auf Klassen zeigen, da ohne Referenzen keine Graphen aufgebaut werden können. Referenzen auf andere Objekte sind in der Regel überflüssig und sollten, da sie relativ kompliziert zu behandeln sind, auch nicht für solche Objekte eingeführt werden. Man definiert

```
declare
  -> Stammdaten DatenVerweis;
```

für die Deklaration einer Referenz auf einen Datensatz. Der Zugriff auf die Elemente eines Datensatzes geschieht über die gleiche Punktnotation wie bei den Datensätzen, d.h. der Programmierer braucht nicht zwischen Datensatzobjekten und Datensatzreferenzen zu unterscheiden. Der wichtige Unterschiede besteht darin, daß eine Referenz noch kein Objekt bezeichnet. Dieses muß erst explizit instanziiert werden, beispielsweise durch

```
declare
  -> Stammdaten DatenVerweis :- Stammdaten objekt; -- erzeugt Datensatz
  Stammdaten Daten1, Daten2;                       -- Referenzen ohne Datensatz
do
  Daten1 :- DatenVerweis; -- Beiden Referenzen zeigen auf gleichen Datensatz
  Daten2 := Daten1;       -- Anlegen eines neuen Datensatz mit Werten von Daten1
```

Der Zuweisungsoperator für Referenzen ist "`:-`". Wird statt dessen das übliche Symbol "`:=`" verwendet, so ist das Kopieren der Werte eines Objekts auf ein anderes gemeint, welches natürlich den gleichen Typ haben muß. Sollte die Referenz noch den Wert "`null`" haben, also keine Referenz auf ein Objekt, so wird dieses in ***FirstStep*** automatisch angelegt.

Das größte Problem mit Referenz ist das Entstehen von Datensätzen, die im Speicher noch bereitgehalten werden, obgleich es keine Referenz mehr auf sie gibt, sie also niemals mehr erreicht werden können. Solche Daten werden als "***Garbage***" (engl. für *Abfall*) bezeichnet. Da ***FirstStep*** keine automatische Kontrolle dieses Abfalls durchführt, ist solch ein Garbage auf jeden Fall zu vermeiden. Nicht mehr benötigte Datensätze sollten entweder gelöscht oder noch besser wieder verwendet werden.

```
do
  discard(Daten1);        -- Löscht den Datensatz Daten1
```

4.2.10 Funktionsvariable

Funktionen führen immer wieder kehrende Rechenoperationen aus und sind vor allem zur Strukturierung komplexer Programme geeignet. Eine Funktion wird über den Namen der Funktion ausgeführt. Eine Funktionsvariable hat genau die gleiche Aufgabe wie eine normale Variable, d.h. es kann ihr als Wert ein Funktionsname zugewiesen werden und sie wird so aufgerufen wie eine Funktion aufgerufen wird. Deklaration und Aufruf sehen also folgendermaßen aus:

```
define
  type FunkType == function integer (real Wert);
declare
```

```
    FunkType FunkVariable := Funktion1;
  do
    Wert := FunkVariable(3.14);
    FunkVariable := Funktion2;
    Wert := FunkVariable(3.14);
```

Funktionsvariable können u.U. das Verhalten eines Programms recht undurchsichtig machen. Sie sollten daher nur verwendet werden, wenn es wirklich sinnvoll erscheint.

In ***FirstStep*** werden Funktionsvariable benutzt, um die sogenannten virtuellen Funktionen einzuführen, d.h. in Klassen können abgeleitete Klassen das Verhalten einer Basisklasse verändern. Siehe auch Abschnitt 11.5 unten.

5 Anweisungen

Eine Anweisung besteht in der Regel aus der Änderung eines Werts, da die Anweisung sonst keinen Sinn macht. Allerdings kann eine Anweisung auch in einem sogenannten Seiteneffekt bestehen, bspw. in der Ausgabe eines Werts in eine Datei, das Versenden einer Nachricht an einen anderen Rechner oder das Überprüfen eines Ausdrucks. Viele Anweisungen bestehen in der Regel aus der wiederholten Ausführung einzelner genau bestimmter Anweisungsfolgen, also den Schleifen, oder eine Anweisung besteht in der Ausführung einer von mehreren möglichen Anweisungsfolgen, den bedingten Anweisungen.

Die in ***FirstStep*** möglichen Anweisungen sollen in diesem und den beiden nachfolgenden Kapiteln systematisch zusammengefaßt werden. Dabei beginnen wir mit den einfachen Wertzuweisungen, gefolgt von den bedingten Anweisungen und den Schleifen. In diesem Kapitel werden die Wertzuweisungen behandelt.

5.1 Grundsätzliches zu Anweisungen

Anweisungen sind Folgen von Änderungen des Zustands eines Programms. Die Folge der Anweisung wird in der Regel durch ein Semikolon '`;`'. Allerdings kann dieses auch in vielen Fällen weggelassen werden, beispielsweise nach der letzten Anweisung in einem `do ... done` Block, oder gelegentlich sogar zwischen zwei normalen Anweisungen, wenn dieses nicht zu Mehrdeutigkeiten führt. Allerdings hilft ein solches Zeichen auch, die Programme zu strukturieren, so daß dieses möglichst vermieden werden sollte.

FirstStep erlaubt auch die Angabe einer „leeren" Anweisung, d.h. zwei direkt aufeinanderfolgende Semikolons. Dieses sollte jedoch in der Regel vermieden werden, da es eigentlich keinen Nutzen bringt und einen Leser evtl. verwirren könnte.

Anweisungen können über mehr als eine Zeile gehen, ohne daß besondere Zeichen am Ende der Zeile vorzusehen sind. Diese Möglichkeit sollte insbesondere benutzt werden, um komplexe Anweisungen, wie die `write`-Anweisung oder komplexe Funktionsaufrufe zu strukturieren.

5.2 Einfache Wertzuweisungen

Ein Wert, wie eine Zahl, Zeichen, Text usw., wird einer Variablen durch den Operator "`:=`" zugewiesen.

```
VarA := 25.5;
TextB := „"Text";
ComplexC := ( 2.4 , 3.6 );
```

Auf der linken Seite ist somit ein Ausdruck zu nennen, der ein Objekt kennzeichnet. Diese kann durchaus ein komplizierterer Ausdruck sein, wie eine Dereferenzierung oder ein Feldausdruck.

```
Datensatz.Hausnummer := 25;
Feld[27] := ( 2.3 , 4.7 );
```

Auf der rechten Seite muß ein Ausdruck stehen, der ein Wert ergibt. Wichtig ist, daß dieser Wert den gleichen Typ haben muß wie das Objekt auf der linken Seite, da ansonsten eine Wertzuweisung nicht sinnvoll durchgeführt werden kann. Der Compiler prüft dieses ab. Sind die Typen verschieden, so gibt es zwei Möglichkeiten:

- **Typanpassung**
  Kann der Wert der rechten Seite sinnvoll so umgewandelt werden, daß er dem Objekt der linken Seite zugewiesen werden, so nimmt ***FirstStep*** eine **Typanpassung** vor. Beispielsweise wird eine ganze Zahl in eine reelle Zahl umgewandelt, aber auch eine reelle Zahl gerundet und in eine ganze Zahl umgewandelt. Komplexe Zahlen, denen eine reelle Zahl zugewiesen wird, erhalten als Wert den jeweils reellen Wert. Außerdem werden Zeichen in Texte umgewandelt.

```
GanzZahl := 2.3;           -- Wert ist 2
RealZahl := 4 ;            -- Wert ist 4.0
CompZahl := 3.5;           -- Wert ist ( 3.5 , 0.0 )
TextVari := 'A';           -- Wert ist "A"
```

- **Fehlermeldung**
  Der Compiler weist während der Übersetzung einen Ausdruck mit fehlerhaftem Typ zurück.

Die zweite Möglichkeit ist eigentlich die rigorosere, erweist sich aber häufig, gerade bei Zahlen, als sehr unbeholfen. Daher werden in ganz wenigen, sinnvollen Fällen in ***FirstStep*** Typanpassungen auch automatisch durchgeführt.

5.3 Mehrfachzuweisungen

FirstStep erlaubt es, daß mehreren Variablen in einem Ausdruck ein Wert zugewiesen wird. Das vereinfacht einige Arten von Zuweisungen, wie das Vertauschen zweier Variablenwerte, ist darüber hinaus auch bei der Werterückgabe bei Funktionsaufrufen wichtig, da in ***FirstStep*** eine Funktion gleichzeitig mehrere Werte zurückgeben kann.

```
Wert1 , Wert2 := Wert2, Wert1;
VarA , VarB := 25 + VarA, Feld[24] + VarB;
Ergebnis1, Ergebnis2, Ergebnis3 := Funktion ( Parameter1, 25 + 4 );
```

Natürlich werden zunächst sämtliche rechten Seite ausgewertet, ehe die Variablen verändert werden, da ansonsten unbestimmte Ergebnisse vorliegen würden, die von der Reihenfolge der Auswertung abhängen. Außerdem sollten hier nur sehr einfache Ausdrücke verwendet werden, da sonst die Übersichtlichkeit leidet.

5.4 Wert- und Objektzuweisungen

Der Operator `:=` wird in ***FirstStep*** als „Wertzuweisung“ verstanden, d.h. es wird ein Wert berechnet und eine Kopie dieses Werts der Variablen auf der linken Seite zugewiesen. Auch bei komplexen Objekten wie Datensätzen und Feldern kann man somit „Werte“ zuweisen, indem Kopien der Objekte überwiesen werden. Bei Feldern ist lediglich darauf zu achten, daß die Feldelemente die gleichen Typen besitzen und daß die Anzahl der Indizes gleich ist. Die Indexbereiche dürfen unterschiedlich sein; das Feld auf der linken Seite erhält nach dieser Zuweisung die gleichen Indexbereich wie das Feld auf der rechten Seite; die alten Werte werden natürlich gelöscht.

Bei Datensätzen funktioniert dieses analog, mit einigen Einschränkungen. Da Datensätze wiederum Verweise auf Datensätze haben dürfen, werden diese Verweise nicht kopiert, sondern durch `null` ersetzt. Ist außerdem die linke Seite eine Referenz, die noch den Wert null hat, also auf kein Objekt verweist, so wird ein solches Objekt automatisch erzeugt. Die rechte Seite darf nicht den Wert `null` haben; dieses würde zu einem Laufzeitfehler führen.

Es ist auch möglich, das gleiche Objekt von verschiedenen Referenzvariablen aus zu referenzieren. Dazu wird der :−-Operator verwendet. Bei Verwendung dieses Operators hat nach der Zuweisung die linke Seite lediglich einen Verweis auf das gleiche Objekt wie die rechte Seite. Eine Änderung des Objekts auf der rechten Seite würde also auch eine Änderung des Objekts auf der linken Seite nach sich ziehen. Dieses kann zu sehr unverständlichen Programmen führen, so daß solche Objektzuweisungen möglichst nur verwendet werden sollen, wenn dieses unbedingt nötig ist, wenn beispielsweise Listen oder allgemeine Graphen verarbeitet werden sollen.

6 Bedingte Anweisungen

Häufig hängt die Fortsetzung einer Berechnung von den bisherigen Ergebnissen ab. Wird beispielsweise abhängig von einem Eingabewert eine bestimmte Berechnung durchgeführt, so muß die Fortsetzung des Programms von einer Bedingung abhängig gemacht werden. Dieses kann prinzipiell eine einfache `if-then-endif`-Anweisung sein. In vielen Fällen kann aber eine komplexere Anweisung die Darstellung sehr viel einfacher und übersichtlicher machen. Wir betrachten hier die verschiedenen Arten von bedingten Anweisungen, die ***FirstStep*** anbietet.

6.1 IF ... then ... else ... endif

Die einfachste Anweisung ist die sogenannte `if`-Anweisung, die folgendermaßen aussehen kann:

```
if     XVar > 0  then YVar := XVar**2
elseif XVar == 0 then YVar := 0.001
else write"Wrong input. Negative numbers not allowed!";
end if;
```

FirstStep erlaubt verschiedene Konstrukte, die dem Programmierer das Schreiben des Programms teilweise erleichtern. So kann letzte Schlüsselwort auf verschiedene Weisen geschrieben werden. Die ***FirstStep***-Grammatik erlaubt die folgenden Konstrukte:

```
ifAnweisung :: "if" ifAnwRest
               "then" anweisungen
               ["elseIf" bAusdruck "then" anweisungen]*
               ["else" anweisungen]
               ifEndAnw
ifEndAnw    :: "end if" | "end" | "endIf" | "done if" | "done"
```

wobei das Schlüsselwort auch leer sein kann. Dennoch sollte dieses nur angewendet werden, wenn keine Mißverständnisse auftreten können. Da insbesondere auch das Block-Ende als Ende-Anweisung genommen werden kann, kann ein vergessenes Ende auch leicht zu falschen Programmen führen. Aus diesem Grunde sollte als Ende-Anweisung möglichst immer `endIf` oder `end if` verwendet werden.

Die `else`- bzw. `elseIf`-Zweige können auch weggelassen werden. Ist die Bedingung falsch, so wird in diesem Fall überhaupt keine Anweisung ausgeführt. Gibt es eine oder mehrere `elseIf`-Anweisungen, so wird die Anweisung hinter der ersten wahren Bedingung ausgeführt; die folgenden Bedingungen werden nicht mehr ausgewertet. Ist keine Bedingung wahr, so wird die Anweisung hinter dem Schlüsselwort `else` ausgeführt. Fehlt dieses, so wird keine Anweisung dieser `if`-Anweisung ausgeführt.

6.2 IfCase-Anweisungen

Müssen mehrere Abfragen durchgeführt, so kann dieses natürlich übersichtlich mit der `ifCase`-Anweisung durchgeführt werden. Beispielsweise erhalten wir

```
ifCase
  ValX == 1 do anweisungen done
  ValX == 2 do anweisungen done
  ValX == 3 do anweisungen done
  ValX >  3 do anweisungen done
  else      do write „“Value out of range“,lf; done
end ifCase;
```

Semantisch bedeutet diese exakt das gleiche wie eine `if-elseIf-else-endIF`-Abfrage, läßt sich aber leichter strukturieren, so daß dieses in unübersichtlichen Fällen angewendet werden sollte.

6.3 Case-Anweisung

Müssen mehrere Abfragen auf einen Wert durchgeführt werden, so kann die `case`-Anweisung verwendet werden. Beispielsweise gilt

```
case Zeichen ==
  ‘A‘ do anweisungen done
  ‘B‘ do anweisungen done
  ‘C‘ do anweisungen done
  ‘D‘ do anweisungen done
```

```
    'E' do anweisungen done
    others do anweisungen done
  end case
```

Statt `others` darf auch `default` oder `else` verwendet werden. Nimmt `Zeichen` einen entsprechenden Wert an, so werden die Anweisungen des dahinter stehenden Blocks ausgeführt. Hat `Zeichen` irgend einen anderen Wert, so werden die Anweisungen des hinter `others` stehenden Blocks ausgeführt. Wenn keine Bedingung zutrifft, so werden keine Anweisungen ausgeführt. Diese Anweisung dient besonders der übersichtlichen Formulierung einfacher Abfragestrukturen. Statt Zeichen können auch ganze Zahlen oder Aufzählungen verwendet werden.

6.4 Range-Anweisungen

Werden Abfragen auf Wertebereiche durchgeführt, also insbesondere reelle Zahlen verwendet, so läßt sich die `range`-Anweisunge verwendet. Man schreibt beispielsweise

```
  range Zahlenwert <=
    1.0 do anweisungen done
    2.0 do anweisungen done
    3.0 do anweisungen done
    others do anweisungen done
  end range;
```

Hier wird die erste Anweisungsfolge ausgeführt, wenn `Zahlenwert` kleiner ist als 1.0 oder gleich diesem Wert ist; entsprechend wird die zweite Anweisungsfolge ausgeführt, wenn der Wert größer ist als 1.0, aber kleiner oder gleich 2.0, usw. Ist der Wert größer als 3.0, so wird die letzte Anweisungsfolge ausgeführt. Natürlich kann auch < verwendet werden, oder >, >=.

6.5 Guard-Anweisungen

Sollen Abfragen immer wieder überprüft und Anweisungen immer wieder bedingt ausgeführt werden, so kann die `guard`-Anweisung sinnvoll eingesetzt werden. Beispielsweise kann man schreiben

```
  guard
    XVar > 0.0001       do anweisungen done
    XVar < -0.0001      do anweisungen done
```

```
    XVar >= -0.0001 and XVar <= 0.0001
                        do anweisungen done
    XVar == 0.0 break;
  end guard;
```

um eine iterative Approximation zu formulieren (siehe vollständiges Beispiel unten). Sobald eine Bedingung wahr wird, wird die dahinter stehende Anweisungsfolge ausgeführt. Danach wird die nächste Bedingung geprüft usw. Nach der letzten Bedingung wird wieder die erste überprüft.

Durch das Schlüsselwort `repeat` kann wieder mit dem Testen der ersten Bedingung angefangen werden. Das Schlüsselwort `break` beendet die Wiederholung dieser Schleife. Nach einem Block kann man `then break` oder `then repeat` schreiben. Nach Ausführung des Blocks, wozu die davor stehende Bedingung wahr sein muß, wird entsprechend die Schleife verlassen oder es wird an den Anfang der Schleife zurückgesprungen. Dieses ließe sich im Prinzip auch durch die erneute Auswertung der Bedingung erreichen, wozu sich allerdings die Parameter der Bedingung nicht ändern dürfen.

Eine Anweisungsfolge, die immer ausgeführt werden soll, wird durch die Bedingung `true` eingeleitet. Entsprechend können durch `false` Anweisungen ständig unterdrückt werden. Dieses kann für das Testen solcher Schleifen interessant sein.

Das folgende Beispielprogramm soll eine gerade Zahl iterativ durch zwei teilen, bis ihr Wert 1 ist; wird die Zahl ungerade, so soll diese mit 3 multipliziert werden und 1 addiert werden.

```
  guard
    Zahl mod 2 == 0 do
                      Zahl := Zahl / 2
                    done then repeat
    Zahl == 1       do
                      write "Zahl hat den Wert 1",newline;
                    done then break
    true            do
                      Zahl := Zahl*3 + 1
                    done
  end guard;
```

Die theoretisch interessante Frage hierbei ist, ob diese Schleife stets teminiert. Bisher konnte das Gegenteil nicht bewiesen werden, aber auch nicht die Richtigkeit dieser Aussage.

Das nächste Beispiel soll die Nullstelle einer Funktion finden.

```
  LINK
    write "Nullstelle einer Funktion",newline;
```

```
  real "Parameter A = ",ParA;    -- Lese Parameter eines
  real "Parameter B = ",ParB;    -- Polynom 2. Grades ein
  real "Parameter C = ",ParC;
DEFINE
  function real Funk(real Wert) ==
  do
    return Wert*Wert*ParA + Wert*ParB + ParC;
  done function Funk;
DECLARE
  real XWert:=0, XLinks:=0.0, XRechts:=10.0;
  real Wert:=0, WertL := 0, WertR:= 0;
  boolean Start := true;
DO
  guard
    -- Suche Intervall mit verschiedenen Vorzeichen der Funktionswerte
    Start do
             WertL := Funk(XLinks);
             WertR := Funk(XRechts);
             if WertL*WertR> 0 then
               write "Funktionswerte ", WertL, " und ", WertR,
                     " haben das gleiche Vorzeichen.",newline;
               write "Neue Grenze links = "; read XLinks;
               write "Neue Grenze rechts = "; read XRechts;
             else
               Start := false;
             end if;
          done then repeat
    true do -- Berechne mittleren Funktionswert
           XWert := (XLinks + XRechts) / 2;
           Wert := Funk(XWert);
         done
    WertL * Wert == 0
         do -- Falls Funktionswert (oder an linker Stelle) null ist
           if WertL == 0 then XWert := XLinks; end if;
           write "Loesung == ", XWert, newline;
         done then break;
    Wert <= 1.0e-10 and Wert >= -1.0e-10
         do -- Falls Funktionswert genau genug approximiert
           write "Loesung == ", XWert, newline;
         done then break;
    WertL * Wert > 0
         do -- Falls Funktionswert gleiches Vorzeichen wie linker Wert
           XLinks := XWert;
           WertL := Wert;
         done then repeat;
    true do -- Falls Funktionswert gleiches Vorzeichen wie rechter Wert
           XRechts := XWert;
           WertR := Wert;
```

```
          done then repeat;
   end guard;
DONE
```

7 Wiederholte Anweisungen

Da Anweisungen in der Regel mehr als einmal auszuführen sind, ist eine Wiederholungsanweisung nötig. In ***FirstStep*** wird dieses durch eine sehr allgemeine `For`-Schleife realisiert.

7.1 Die Struktur einer `For`-Schleife

Eine For-Schleife besteht aus drei Abteilungen, dem Prolog, der lokale Variable definiert und initialisiert, einem Schleifenrumpf, der die Folge zu wiederholender Anweisungen angibt, und einem **Epilog**, der Anweisungen angibt, die nach der Beendigung der Schleife ausgeführt werden sollen. Eine Schleife bildet insbesondere einen lokalen Namensraum, der nach Beendigung der Schleife wieder verschwindet. Die Laufvariable ist nur innerhalb der Schleife bekannt. Das folgende Beispiel soll dieses verdeutlichen:

```
declare
       integer Wert := 0;
prolog
       if Grenze < 0  then exit ; end if;
for    Index:=1 to Zahl while Wert+Feld[Index]<=Grenze do
       Wert := Wert + Feld[Index];
epilog
       write "Nach ", Index,
             " Durchlaeufen betraegt die Summe ", Wert, newline;
done for Index;
```

Nach den Schlüsselwörtern `disclose`, `define`, `declare` können beliebige Variable definiert oder deklariert und initialisiert werden. Nach dem Schlüsselwort `prolog` können beliebige Anweisungen (außer `for`-Anweisungen) ausgeführt werden. Das Schlüsselwort `prolog` steht hier also statt des Schlüsselworts `do` in einem „normalen“ Block. Der Block beginnt es nach dem Schlüsselwort `do` hinter dem **Schleifenkopf**

Mit dem Schlüsselwort `for` wird dann die eigentliche Schleife mit dem **Schleifenkopf** eingeleitet. Sie hat als Namen den folgenden Bezeichner, der nur lokal in dieser Schleife bekannt ist und den Typ des Ausdrucks hinter dem Symbol `:=` hat; dieser Bezeichner wird als **Laufvariable** bezeichnet. Außer ganzen Zahlen können hier auch Aufzähltypen angegeben werde; es sind auch reelle Zahlen erlaubt, wenngleich hier das Problem der Rundungsfehler besteht. Nach dem Schlüsselwort `by` wird das Inkrement der Variable `Index` in jedem Durchlauf angegeben, nach dem Schlüsselwort `to` der letzte Wert, den die Laufvariable `Index` annehmen soll; danach wird nach Durchlauf der Anweisungen hinter dem Schlüsselwort `epilog` die Schleife beendet. Statt `to` kann auch `downto` stehen; dann sollte nach `by` ein negativer Wert stehen. Fehlen einzelne Schlüsselwörter mit den zugehörigen Werten, so wird dort der Wert eins bzw. nach `to` der Wert unendlich angenommen.

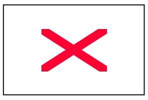
Nach diesen Ausdrücken können die Schlüsselwörter `while`, `until`, `if` oder `unless` in beliebiger Reihenfolge, gefolgt von einem logischen Ausdruck stehen. Die Schleife wird nicht mehr durchlaufen wenn nach `while` ein falscher, nach `until` ein wahrer Ausdruckswert folgt; sie wird statt dessen beendet, wenngleich vorher der Abschnitt nach dem Schlüsselwort `epilog` einmal durchlaufen wird. Steht nach `if` ein falscher, nach `unless` ein wahrer Ausdruck, so wird sofort der nächste Wert der Laufvariablen berechnet und der Test noch einmal durchgeführt; die Schleife wird also „übersprungen“.

Vor den Schlüsselwörtern `while` bzw. `until` kann auch das Schlüsselwort `repeat` stehen. Dann werden die jeweiligen Auswertungen am Ende der Schleife vor Berechnung des nächsten Werts der Laufvariablen durchgeführt.

In der Schleife kann durch die Anweisung `exit` bzw. `exit for Index` die Schleife verlassen werden; die Anweisungen nach dem Schlüsselwort `epilog` werden in diesem Falle **nicht** durchlaufen! Durch die Anweisung `repeat` bzw. `repeat for Index` wird die Schleife nach Berechnung des nächsten Werts der Laufvariablen wieder von vorne durchlaufen.

8 Ausdrücke mit wiederholter Berechnung

In der Mathematik werden Summen in einer standardisierten Notation beschrieben:



In der Informatik können solche Ausdrücke häufig durch elementare Schleifen nachgebildet werden, wie sie im letzten Kapitel definiert wurden. Für Standardausdrücke läßt sich jedoch einfach eine entsprechend Konstruktion in einer Programmiersprache einführen, damit entsprechende Aufgaben einfach gelöst werden können.

8.1 Verallgemeinerte Ausdrücke

Ein Ausdruck, der durch eine mehrfach wiederholte Operation berechnet wird, wird in ***FirstStep*** folgendermaßen dargestellt:

```
Ergebnis :=
  for Lauf := 1 to 100 compute (opr: <expression>);
```

Beispielsweise wird die Summe der ersten 100 Zahlen geschrieben als

```
Ergebnis :=
  for Lauf := 1 to 100 compute (+: Lauf);
```

Hierbei besteht der Rumpf der Schleife aus einer Klammerung, in der zunächst ein Operator, gefolgt von einem Doppelpunkt eingefügt wird, und dann ein Ausdruck, der einen Wert bestimmt. Als Operatoren sind in ***FirstStep*** zur Zeit vorgesehen

```
+, *, minimum, maximum, count, and, or
```

mit der jeweils bekannten Bedeutung; der `count`-Operator zählt die Anzahl wahrer Ausdrücke. Die jeweiligen Besonderheiten werden in den nächsten Abschnitten beschrieben.

Der Ausdruck, der die Wertefolge bestimmt, gehorcht den gleichen Bedingungen wie der entsprechende Ausdruck einer allgemeinen Schleife. Es können somit einzelne Werte ausgelassen werden (`if` bzw. `unless`), es können Bedingungen angegeben werden, unter denen die Ausführung abgebrochen wird (`while` bzw. `until`), und es kann zusätzlich noch festgelegt werden, daß die Ausführung beendet wird, sobald das Ergebnis sich nicht mehr ändern kann (`break`). Die letzte Bedingung bewirkt beispielsweise, daß bei einer logischen Bedingung die wahr (`or`) bzw. falsch (`and`) wird, der Schleifenablauf angebrochen wird, da eine logische Disjunktion, die einmal wahr wird, nicht mehr falsch werden kann, und ähnlich bei einer Konjunktion. Dieses läßt sich außerdem noch bei der Multiplikation einsetzen, da der Wert null niemals mehr durch Multiplizieren verändert werden kann.

Bei den arithmetischen Operatoren macht es u.U. Sinn, den Anfangswert einer Summation bzw. Multiplikation explizit festzulegen. Dieses kann durch einen Start-Wert festgelegt werden (siehe erstes Beispiel im nächsten Abschnitt). Zwar könnte dieser Wert auch auf den Ausdruck der `for`-Schleife addiert bzw. multipliziert werden, aber aufgrund von Rundungsfehlern führt dieses u.U. nicht zu den gleichen Ergebnissen. Die `start`-Anweisung kann ebenfalls benutzt werden, um den Typ der Summationsvariablen explizit auf `real` zu setzen, falls des arithmetische Ausdruck nicht eindeutig als `real` erkannt werden kann.

Eine weitere Besonderheit stellt der `index`-Wert dar. Es gibt eine Variable, die unmittelbar nach einer Mehrfachoperation angibt, wie wieviele Schritte die Laufvariable durchgeführt hat. Es wird also auch die Anzahl nicht ausgeführter Zwischenschritte (durch `if` oder `unless` abgefragt) gezählt. `index` ist somit eine ganzzahlige Variable, die direkt nach einer Mehrfachoperation benutzt werden muß. Bei der Minimum- bzw. Maximumoperation hat diese Variable jedoch den Wert des Index, bei dem das Minimum bzw. Maximum angenommen wird. Diese Variable wird eigentlich außer bei Minimum und Maximum nur noch bei den `break`-Operatoren `*`, `and` und `or` unbedingt benötigt, ist aber der Vollständigkeit halber für alle Operatoren definiert.

8.2 Wiederholte Operationen

In diesem Abschnitt werden die einzelnen mehrfachen Operationen genauer beschrieben. Das folgende Musterprogramm kann als Beispiel verwendet werden.

```
LINK
```

```
    write "Testen von For-Schleifen mit Operatoren", newline;
    integer "Durchlauf bis ", Zahl;
    integer "ausser ", Nicht;
  DO
    write "Summe (+: Lauf) ergibt ",
    for Lauf:=1 to Zahl start 100 unless Lauf==Nicht compute
      (+: Lauf)
    ,", Index = ", index, newline;

    write "Produkt (*: Lauf) ergibt ",
    for Lauf:=-2 to Zahl break unless Lauf==Nicht compute
      (*: Lauf)
    ,", Index = ", index, newline;

    write "Minimum (min: Lauf) ergibt ",
    for Lauf:=1 to Zahl unless Lauf==Nicht compute
      (min: Lauf)
    ,", Index = ", index, newline;

    write "Maximum (max: Lauf) ergibt ",
    for Lauf:=1 to Zahl unless Lauf==Nicht compute
      (max: Lauf)
    ,", Index = ", index, newline;

    write "And (*: Lauf<=4) ergibt ",
    for Lauf:=1 to Zahl break unless Lauf==Nicht compute
      (and: Lauf<=4)
    ,", Index = ", index, newline;

    write "Or (*: Lauf<=4) ergibt ",
    for Lauf:=1 to Zahl break unless Lauf==Nicht compute
      (or: Lauf<=4)
    ,", Index = ", index, newline;

    write "# (*: Lauf<=4) ergibt ",
    for Lauf:=1 to Zahl unless Lauf==Nicht compute
      (#: Lauf<=4)
    ,", Index = ", index, newline;
  DONE
```

8.2.1 Wiederholte Addition

Die wiederholte Addition `compute (+: <ausdruck>)` berechnet die Summe der Ausdrücke. Die `break`-Anweisung ist nicht erlaubt. Die Variable `index` gibt die Anzahl der Schleifendurchläufe an, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt werden. Die `start`-Anweisung initialisiert die Summationsvariable mit einem entsprechenden Wert. Implizit ist dieser Wert 0. Ist der Ausdruck hinter der `start`-Anweisung vom Typ `real`, so wird entsprechend eine `real`-Variable zur Summation verwendet; sonst wird eine Variable jenes Typs verwendet, der aus den Summanden erschlossen werden kann.

8.2.2 Wiederholte Multiplikation

Die wiederholte Multiplikation `compute (*: <ausdruck>)` berechnet das Produkt der Ausdrücke. Die `break`-Anweisung ist hier zugelassen; sie bricht die Multiplikation ab, wenn das Produket den Wert null hat. Die Variable `index` gibt die Anzahl der Schleifendurchläufe an, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt werden. Die `start`-Anweisung initialisiert die Summationsvariable mit einem entsprechenden Wert. Implizit ist dieser Wert 1. Ist der Ausdruck hinter der `start`-Anweisung vom Typ `real`, so wird entsprechend eine `real`-Variable zur Summation verwendet; sonst wird eine Variable jenes Typs verwendet, der aus den Summanden erschlossen werden kann.

8.2.3 Wiederholte Minimum-Operation

Die wiederholte Minimum-Operation `compute (min: <ausdruck>)` berechnet den kleinsten der Ausdrücke, wobei zur Zeit nur numerische Werte zugelassen sind. Die `break`-Anweisung und die `start`-Anweisung sind hier nicht zugelassen. Die Variable `index` gibt die Nummer des Schleifendurchlaufs an, an dem die Variable das erste Mal den minimalen Wert angenommen hat, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt werden.

8.2.4 Wiederholte Maximum-Operation

Die wiederholte Maximum-Operation `compute (max: <ausdruck>)` berechnet den größten der Ausdrücke, wobei zur Zeit nur numerische Werte zugelassen sind. Die `break`-Anweisung und die `start`-Anweisung sind hier nicht zugelassen. Die Variable `index` gibt die Nummer des Schleifendurchlaufs an, an dem die Variable das erste Mal den maximalen Wert angenommen hat, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt werden.

8.2.5 Anzahl-Operation

Die Anzahl-Operation `compute (count: <ausdruck>)` berechnet die Anzahl von wahren Ausdrücken. Die `break`-Anweisung und die `start`-Anweisung sind hier nicht zugelassen. Die Variable `index` gibt die Anzahl der Schleifendurchläufe an, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt wird.

8.2.6 Wiederholte Disjunktion

Die wiederholte Disjunktion `compute (or: <ausdruck>)` liefert einen wahren Wert, wenn mindesten ein Ausdruck wahr ist. Die `start`-Anweisung ist hier nicht zugelassen. Die `break`-Anweisung unterbricht den Ablauf, wenn das erste Mal ein Zwischenausdruck den Wert wahr hat. Die Variable `index` gibt die Anzahl der Schleifendurchläufe an, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt wird.

8.2.7 Wiederholte Konjunktion

Die wiederholte Konjunktion `compute (and: <ausdruck>)` liefert einen wahren Wert, wenn alle Ausdrücke wahr sind. Die `start`-Anweisung ist hier nicht zugelassen. Die `break`-Anweisung unterbricht den Ablauf, wenn das erste Mal ein Zwischenausdruck den Wert falsch hat. Die Variable `index` gibt die Anzahl der Schleifendurchläufe an, wobei eine falsche `if`- bzw. eine wahre `unless`-Bedingung ebenfalls als Durchlauf gezählt wird.

9 Standardfunktionen

In vielen Fällen werden bestimmte Funktionen immer wieder verwendet. Ein typisches Beispiel ist die Maximum-Funktion, die den größten von zwei oder mehr Zahlenwerten berechnet. Solche Funktionen sind teilweise aus der Mathematik bekannt, teilweise jedoch erst mit der Informatik eingeführt worden. So wird hier beispielsweise auch das Maximum zweier Zeichenketten definiert, wofür es entsprechende Funktionen gibt. Die in ***FirstStep*** verwendeten Standardfunktionen werden in diesem Kapitel eingeführt.

9.1 Ausgabeprozeduren

Die Ausgabe wird in ***FirstStep*** durch die Prozedur `write` durchgeführt, die zum einen in der Regel beliebig viele Parameter hat, die zum anderen in verschiedene Ausgabemedien gelenkt werden können.

Hinter dem Prozedurnamen `write` kann eine beliebig lange Liste von Ausdrücken stehen, die jeweils durch ein Komma oder eine Textkonstante getrennt werden

```
write Wert , 22+Wert ", " Wert:3 " und " pi:6:3, newline;
```

Hinter dem Prozedurnamen `write` kann ein oder kein Schlüsselwort stehen. Steht dort kein Schlüsselwort, so wird die Ausgabe auf das Terminal des Benutzers gelenkt. Steht dort das Schlüsselwort `on`, gefolgt von einem Dateinamen, so wird die Ausgabe in diese Datei gelenkt; steht hinter `on` eine Textvariable, so wird die Ausgabe auf diesen Text durchgeführt, steht hinter `on` eine Verbindungsvariable (vom Typ `connection`), so wird die Ausgabe eines Texts auf diese Verbindung durchgeführt. Danach können ein oder mehrere Parameter folgen, deren Werte auf das jeweilige Ausgabemedium ausgegeben werden. Es ist jeweils darauf zu achten, daß Dateien bzw. Verbindungen geöffnet sind. Textvariable werden automatisch angelegt. In der Regel ist die Menge der auszugebenden Werte unbeschränkt.

```
write 27:4, 3.1415:5:2, Wahrheitswert:true, newline;
write Wahrheitswert:1, Wahrheitswert:T, Wahrheitswert:true, newline;
write ErsterWert ",“ ZweiterWert ",“ DritterWert ;
write on Datei Text1;
read from Verbindung Text1;
read from TextVariable WertInt, WertReal, Zeichen;
```

9.2 Eingabeprozeduren

Die Eingabe wird in ***FirstStep*** durch die Prozedur `read` durchgeführt, die in der Regel einen Parameter hat, der von verschiedenen Ausgabemedien gelesen werden kann.

```
read Text1;
read from Datei Text1;
read from Verbindung Text1;
read from TextVariable WertInt, WertReal, Zeichen;
```

Hinter dem Prozedurnamen `read` kann ein oder kein Schlüsselwort stehen. Steht dort kein Schlüsselwort, so wird die Eingabe von dem Terminal des Benutzers gelesen. Steht dort das Schlüsselwort `from`, gefolgt von einem Dateinamen, so wird die Eingabe von dieser Datei gelesen; steht hinter `from` eine Textvariable, so wird die Eingabe von diesem Text durchgeführt; steht hinter `from` eine Verbindungsvariable (vom Typ `connection`), so wird die Eingabe eines Texts von dieser Verbindung durchgeführt. Danach kann ein Parameter folgen, dessen Wert von dem jeweiligen Ausgabemedium eingelesen wird; beim Lesen von einer Textvariablen sind auch mehrere Eingabewerte zulässig. Es ist jeweils darauf zu achten, daß Dateien bzw. Verbindungen geöffnet sind, und daß die Textvariablen genügen Text haben, der gelesen werden kann.

Beim Lesen auf eine Textvariable kann die Länge des zu lesenden Texts angegeben werden, indem zum einen das Schlüsselwort `read text` verwendet wird und zum anderen hinter der Textvariablen eines oder zwei der Schlüsselwörter `to`, `upto` oder `until` folgen; hinter diesen Schlüsselwörtern steht eine Zahl, welche die Anzahl der zu lesenden Zeichen angibt, oder ein Zeichen, bis zu dem gelesen werden soll, oder statt des Zeichens eines der Schlüsselwörer `tab`, `newline` oder `endOfText`. Weitere Einzelheiten findet man auch in 13.3 unten.

```
read text Text1 to AnzahlZeichen;
```

```
read text from Datei Text1 upto '=';
read text from Datei Text1 to Anzahl until endOfText;
```

FirstStep erzeugt in der Regel einen Text, wenn einer einzulesen ist. Es kann jedoch von der Compilerimplementierung abhängen, ob dieser Text beliebig lang sein kann (beispielsweise gibt es Grenzen von 4000 Zeichen).

9.3 Arithmetische Operatoren

Ein Operator, der Zahlen verknüpft, wird als arithmetischer Operator bezeichnet. In ***FirstStep*** gibt es die in der Mathematik üblichen arithmetischen Operatoren, die in der gleichen Weise wie in der Mathematik als Infixoperatoren bzw. Vorzeichenoperatoren verwendet werden. Diese werden hier kurz zusammengestellt.

9.3.1 Zahlenkonstante

In ***FirstStep*** können die Konstanten `pi==3.14159265358979` und `e==2.7182818` verwendet werden.

9.3.2 Elementare Operatoren

`+`, `-`, `*`, `/` bzw. `div` zur Addition, Subtraktion und Multiplikation von ganzen Zahlen (1,2,3,…), reellen Zahlen (0.1, -1.25 , 199.99, …) und komplexen Zahlen ((1.0, 4.5), (-1.0, 0.0), …). Der Divisionsoperator ist `/` mit Ausnahme der ganzzahligen Division, die mit `div` angegeben wird; das Ergebnis ist somit eine ganze Zahl, wenn beide Operatoren ganze Zahlen sind.

9.3.3 Transzendente Operatoren

Die folgenden Operatoren werden in ***FirstStep*** im Gegensatz zur Mathematik als Funktionen geschrieben, um die Eindeutigkeit zu wahren. Es gibt die folgenden Exponentiationsoperatoren

```
22 ** 3          ==          22 hoch 3
sqrt (3)         ==          3 hoch 1/2 (Quadratwurzel)
cbrt (3)         ==          3 hoch 1/3 (Dritte Wurzel)
exp (3)          ==          e hoch 3
log(3)           ==          ln(3) == log zur Basis e (3)
```

```
log10(3)          ==              log(3) == log zur Basis 10 (3)
log2(3)           ==              lb(3) == log zur 2 (3)
logBase(3,5)      ==              log zur Basis 5 (3)
```

Da diese Funktionen teilweise nur definiert sind, wenn die Operanden nicht negativ sind, hat der Programmier entsprechende Vorsicht walten zu lassen. Die Operanden werden sämtlich in reelle Werte umgewandelt, ehe die entsprechenden Operationen durchgeführt werden.

9.3.4 Trigonometrische Operatoren

Die folgenden Operatoren werden in ***FirstStep*** im Gegensatz zur Mathematik als Funktionen geschrieben, um die Eindeutigkeit zu wahren. Es gibt die trigonometrischen Operatoren

```
sin(3.5)               ==          sin(3,5)
cos(3.0)               ==          cos(3)
tan(3.7)               ==          tan(3,7)
asin(3) == arcsin      ==          arcsin(3)
asin(3) == arcsin      ==          arcsin(3)
asin(3) == arcsin      ==          arcsin(3)
```

Da diese Funktionen teilweise nur definiert sind, wenn die Operanden in einem beschränkten Bereich liegen, hat der Programmier entsprechende Vorsicht walten zu lassen, da sonst ein Programmabbruch folgen könnte.

9.4 Modulo

Für zwei ganze Zahlen ergibt der zweistellige Infixoperator `mod` den Rest den man erhält, wenn man den ersten Operanden durch den zweiten dividiert.

```
33 mod 10 == 3;
```

9.5 Ordnung

Die skalaren Typen wie Zeichen oder Aufzählungen haben in der Regel eine Ordnung, die durch `order(Wert)` berechnet werden kann. Das Ergebnis ist eine ganze Zahl. Die Umkehrung dieser Funktion wird durch `succ` erreicht.

9.6 Nachfolger und Vorgänger

Die zweistelligen Infixoperatoren `succ` und `pred` bestimmen den Nachfolger bzw. Vorgänger einer Aufzählung.

```
type Tage == values Sonntag, Montag, Dienstag, Mittwoch,
                            Donnerstag, Freitag, Samstag;
Sonntag succ 1 == Montag;
Sonntag succ 2 == Dienstag;
Freitag pred 3 == Dienstag;
```

Sind Zyklen bzw. Schwellwerte definiert (4.1), so können die Nachfolgewerte entsprechend größer gewählt werden, und das Programm liefert dennoch die korrekten Ergebnisse; ansonsten müssen sie innerhalb der gewählten Bereiche liegen, da sonst ein Laufzeitfehler angezeigt und das Programm beendet wird.

9.7 Minimum und Maximum

Diese Operatoren bilden das Minimum bzw. Maximum einer Liste von Operatoren gleichen Typs. Der Typ kann eine `integer`- oder `real`-Zahl sein, ein Zeichen oder ein Text.

```
Ergebnis := min(1.5, 24, Zahl, 34 + Zahl - 5 );
Ergebnis := max(1.5, 24, Zahl, 34 + Zahl - 5 );
```

9.8 Absolutbetrag

Der Operator `|...|` berechnet den Absolutbetrag von Zahlenwerten (d.h. den jeweils positiven Wert bzw. den Abstand vom Nullpunkt bei komplexen Zahlen), die Ordnung eines Zeichens oder die Länge eines Texts.

```
| -3 | == 3
| -3.5 | == 3.5
| (3.0, 4.0) | == 5.0
| ‘A‘ | == 33
| “Hallo“ | == 5
```

9.9 Schwellwert und Zyklus

Die Funktionen `gauge` bzw. `cycle` berechnen zu drei Parametern den jeweiligen Wert innerhalb des Bereichs, der durch den ersten und den letzten Wert angegeben wird.

```
cycle( 1, 3, 4 ) == 3
cycle( 1, 5, 4 ) == 1
cycle( Montag, Donnerstag, Mittwoch ) == Montag
gauge( 1, 3, 4 ) == 3
gauge( Leer, Voll, Mittel ) == Mittel; -- Leer<Mittel<Voll;
gauge( 1, 5, 4 ) == 4
```

Der Typ des Ergebnisses ist jeweils der Typ der Parameter, wobei nur Parameter gleichen Typs erlaubt sind, also ganze oder reelle Zahlen oder Aufzählungen.

9.10 Zufallszahlen

Zufallszahlen können durch entsprechende Variable erzeugt werden. Die Zufallszahlen werden nach einem bestimmten Verfahren berechnet, so daß ihnen ein Anfangswert zugeordnet werden muß. Danach wird eine stets gleiche Folge von Zufallszahlen im Bereich von 0 bis 1 (ausschließlich der 1) generiert.

```
declare randomSequence Zufall := 33;
do
  Wert := random ( Zufall );
  Wert := random;
```

Wird die Zufallsfolge initialisert (im Beispiel durch `Zufall`), so wird eine eigene Folge von Zufallszahlen generiert, die von dem ersten Wert, einer ganzen Zahl, abhängt; ohne Parameter wird stets die gleiche Folge von Zufallszahlen in einem Programm generiert.

10 Einheiten

Eine Zahl bedeutet in der Regel nicht nur eine Quantität, sondern eine physikalische Größe; beispielsweise sind 7 m sieben Mal ein Meter, wobei Meter die Basislängeneinheit ist. In Programmiersprachen werden meistens nur Zahlen geschrieben, auch wenn Ihnen eigentlich eine physikalische Bedeutung zukommt. Die Verwendung von Einheiten hat jedoch mehrere Vorteil; so können physikalische oder technische Umrechnungsfaktoren einfach eingefügt werden; der Compiler kann überprüfen, ob Zahlenwerte richtig miteinander verrechnet werden, und man kann auch Einfluß auf die interne Auflösung von Zahlenwerten nehmen. In ***FirstStep*** werden Einheiten nur bei Variablen und Konstanten, nicht jedoch bei der Übergabe von Werten an Funktionen bzw. der Wertrückgabe verwendet.

10.1 Definition von Einheiten

Einheiten werden durch ihre Beziehung zueinander sowie durch ihre Wertedarstellung im Rechner dargestellt.

```
DEFINE
  unit 1 s == 1/60 min == real 22;
  unit 1 m == 100 cm == real 1;
  unit 1 kg == real 123;
  unit 1 kg*m/s/s == 1 N;
  unit 1 N*m/s == 1 W;
```

In diesem Beispiel wird der Wert „10 Sekunden“ durch die `real`-Zahl 220.0 dargestellt, da der Wert „1 Sekunde“ durch die `real`-Zahl 22.0 dargestellt wird. Auf diese Weise kann man sich die für die jeweiligen Zwecke die günstigste Zahlenbereichsauflösung auswählen.

Abgeleitete Einheiten wie Minuten (sechzig Sekunden) oder Zentimeter (Hundertstel Meter) können ebenfalls aus den Basiseinheiten, welches jeweils die erste Einheit in einer Zeile ist, ausgedrückt werden. Zusammengesetzte Einheiten wie die Krafteinheit Newton (N) oder die Leistungseinheit Watt (W) können ebenfalls definiert werden; sie stehen allein in einer Zeile zusammen mit ihrer Definition.

Sollen Einheiten verwendet werden, so werden Konstante als Zahl, gefolgt von einem entsprechenden Einheitenausdruck, geschrieben. Variable werden bei der Deklaration mit einer entsprechenden initialisiert.

```
declare
  real Ort := 0 m , Geschwindigkeit := 0 m/s, DeltaT := 0.4 s;
  real FederKonstante := 1 N/m, Masse := 1 kg, Kraft := 0 N;
  real Leistung := 1 W;
```

Der Compiler kontrolliert, ob während einer Zuweisung korrekte Einheiten erzeugt und entsprechenden Variablen zugewiesen werden, da es beispielsweise keinen Sinn macht, der einer Variablen, die Zeit speichert, eine Länge zuzuordnen. Auf diese Weise kann die Bedeutung von Zahlen wesentlich besser kontrolliert werden.

```
do
  for Zeit := 0.0 s to 40 s by DeltaT do
    Kraft := if Zeit == 0.8 s
             then 10 N
             else 0 N
             end if;
    Ort, Geschwindigkeit :=        --  Berechnet dynamische
      Ort + Geschwindigkeit *Dt,   --  Groessen nach den
      Geschwindigkeit*0.5,          --  physikalischen Kraftgesetzen
  ...
  done for Zeit;
```

In dem letzten Beispiel wäre die Zuweisung

```
Ort := Ort + Geschwindigkeit;
```

natürlich falsch, da eine Geschwindigkeit (in `m/s`) nicht mit einem Länge (in `m`) addiert werden kann.

Statt der `real`-Zahlen können auch `integer`-Zahlen verwendet werden. Wegen des beschränkten Zahlenbereichs lassen sich auf diese Weise einfach Festpunktzahlen einführen, indem beispielsweise jeder Zahlenwert in Hundertsteln gespeichert wird.

```
DEFINE
  unit 1 PFG == 1/100 DM == integer 100;
```

Es sei darauf hingewiesen, daß Einheiten weder Variablenbezeichner noch Schlüsselwörter sind; das bedeutet, daß Einheiten in ***FirstStep*** auch nicht wie solche geschrieben werden dürfen. Für Einheiten darf man in ***FirstStep*** somit nur jene Bezeichner verwenden, die entweder keine Kleinbuchstaben enthalten oder nur aus Kleinbuchstaben bestehen und natürlich auch keine Schlüsselwörter sind: `m`, `s`, `cm`, `W`, `DM` aber nicht beispielsweise `Pfg`, `Km` oder `Std`. Diese kleine Einschränkung sollte aber den Programmierer, der technische Modelle erstellt, durchaus nicht entmutigen, mit Einheiten zu rechnen. Seine Programme werden dadurch sehr viel übersichtlicher und sicherer.

11 Prozeduren und Funktionen

Sollen gewisse Operationen immer wieder an verschiedenen Stellen ausgeführt werden, so lassen sich dafür Funktionen oder Prozeduren einsetzen. Diese dienen zugleich der strukturierten Programmierung, um komplexe Anweisungsfolgen zusammenzufassen.

11.1 Definition von Prozeduren und Funktionen

Eine **Prozedur** (*procedure*) benötigt einen Namen, unter dem sie aufgerufen wird, eine Parameterliste, die die aktuellen Werte angibt, mit denen die Prozedur ausgeführt werden soll, sowie die Folge von Anweisungen, die eigentliche Funktionalität. Eine **Funktion** (*function*) benötigt zusätzlich Werte, die an die aufrufende Stelle zurückgegeben werden. Im folgenden behandeln wir nur die Funktionen; die Prozeduren werden analog verwendet, außer daß keine Rückgabewerte existieren. ***FirstStep*** gestattet, daß auch Funktionen wie Prozeduren behandelt werden. Beispiele für Prozedur- bzw. Funktionsdefinitionen sind

```
DEFINE
  function integer Funk ( << integer Zahl >> ) ==
  do
    write "Funky ",Zahl,lf;
    return 2*Zahl;
  done function Funk;

  procedure Proz (real Zahl) ==
  do
    write "Prozy = ",Zahl,lf;
  done procedure Proz;
DECLARE
  ....
DO
```

```
    Wert := FunkVar(44);
    Proz(55);
  DONE
```

Die Funktion wird also genau so definiert, wie sie verwendet wird. Nach dem Schlüsselwort `function` wird die Liste der Rückgabetypen angegeben, danach der Funktionsname, schließlich die Parameterliste in Klammern. Das Zeichen == soll anzeigen, daß die Funktion gleich dem folgenden Anweisungsblock ist. Werte werden durch die Anweisung `return` mit folgender Parameterliste, die nicht geklammert werden darf, zurückgegeben.

Innerhalb einer Funktion können die üblichen Bereiche eines Blocks, nämlich `disclose`, `define`, `declare` und schließlich in jedem Fall `do` angegeben werden; die Funktionsdefinition (*function definiton*) wird mit den Schlüsselwörtern `done function Funktionsname` abgeschlossen. Funktionen können somit in ihrem `define`-Teil auch „lokale“ Funktionen besitzen, die nur innerhalb der Funktion bekannt sind. Ebenso können Datensätze oder Klassen usw. definiert werden.

11.2 Parameterübergabe

Beim Aufruf einer Funktion müssen die formalen Parameter, die bei der Funktionsdefinition aufgelistet sind, durch aktuelle Werte, die beim Aufruf der Funktion angegeben werden, ersetzt werden. ***FirstStep*** kennt Werteparameter und Referenzparameter.

Werteparameter (*call by value*) werden ohne weitere Kennzeichnung angegeben. Sie berechnen zunächst den Wert eines Ausdrucks und kopieren diesen dann auf einen lokalen Parameter, der innerhalb der Funktion wie eine Variable verwendet werden kann. Da dieses auch für komplexe Parameter wie Felder oder Datensätze gilt, sollte dieser Übergabemechanismus dort nur angewendet werden, wenn er wirklich benötigt wird. Für einfache Typen wie Zahlen oder Zeichen sollten besser möglichst Werteparameter verwendet werden.

Referenzenparameter (*call by reference*) übergeben nur einen Verweis auf das Objekt, welches als aktueller Parameter angegeben wird. Dieses bedeutet zum einen, daß ein Objekt und kein Wert als aktueller Parameter angegeben werden muß. Es dürfen also weder Konstante wie `123` oder `'C'` noch Ausdrücke wie `Wert+Zahl` angegeben werden, da diese keine Referenz besitzen. Zum zweiten kann es natürlich passieren, daß sich diese Werte nach Ausführung (und sogar schon während der Ausführung) der Funktion geändert haben können. Referenzparameter werden durch eine Klammerung `<<...>>>` angegeben, sowohl bei der Definition der Funktion als auch beim Aufruf! Dieses unterscheidet ***FirstStep*** von den meisten anderen Programmiersprachen, wo der Aufruf stets gleich aussieht, ohne daß der Programmierer u.U. weiß, daß er statt eines Werte- einen Referenzparameter übergeben hat. Da Referenzparameter keine Kopie von Objekten anlegen, sind sie relativ „preiswert“. Sie sollten also möglichst bei großen Objekten immer angegeben werden.

Wird ein formaler Referenzparameter als Werteparameter aufgerufen, also ohne <<<. . .>>>, so wird er wie ein Werteparameter behandelt, d.h. es wird eine lokale Kopie dieses Objekts angelegt. Dadurch können mit dem gleichen Mechanismus verschiedene Arten von Übergabemechanismen von dem Programmierer bewußt gewählt werden.

11.3 Werterückgabe

Werte, die eine Funktion als Ergebnis zurückgibt, werden in der Regel als Werteparameter übergeben. Natürlich können auch Referenzen auf Objekte übergeben werden, wobei hier die Referenz der Wert ist. Ansonsten müssen auch komplexe Objekte an die aufrufende Stelle kopiert werden, was stets mit einem gewissen Overhead verbunden ist. Daher wird empfohlen, nur einfache Werte durch Funktionen zurückzugeben, während komplexe Werte durch Referenzparameter übergeben werden sollten.

11.4 Funktionsaufrufe

Eine Funktion wird aufgerufen (*function call*), indem ihr Name, gefolgt von der Liste der Parameter, aufgeschrieben wird. Gibt eine Funktion einen Wert zurück, so kann sie in der Position eines Ausdruck stehen, als z.B. in einer Summe oder einem Produkt. Gibt sie mehrere Werte gleichzeitig zurück, so muß sie alleine rechts von einem Wertzuweisungszeichen := stehen; links von diesem Zeichen muß eine entsprechende Liste von Variablen stehen, die die Werte aufnehmen können.

Eine alternative Aufrufmethode für eine Funktion wird als Infixnotation bezeichnet. Dazu kann der Funktionsname hinter dem ersten Parameter und vor dem zweiten Parameter aufgelistet werden; mögliche weitere Parameter würden hinter dem zweiten Parameter, durch Kommata getrennt aufgelistet werden. Um derartige Ausdrücke übersichtlich zu halten, sollten sie möglichst alleine stehen oder sonst geklammert werden.

Prozeduraufrufe stehen stets in der Position einer Anweisung; ansonsten werden sie genauso aufgerufen wie Funktionen.

11.5 Funktionsvariable

Funktionen können Variablen zugewiesen werden, die dann jeweils die Funktionalität der letzten Zuweisung ausführen. Dabei wird ein Typname definiert, der die Art (Funktion oder Prozedur), gegebenenfalls die Typen der Rückgabeparameter sowie eine Liste der Parameter umfaßt, einschließlich Variablennamen, die jedoch keine semantische Bedeutung haben; sie sollten zur Dokumentation angegeben werden.

```
DEFINE
  type FunkTyp == function integer ( << integer Nummer >> );
  type ProzTyp == procedure (real Zummi);
  function integer Funk ( << integer Zahl >> ) ==
  do
    ...
  done function Funk;

  function integer Funk2 ( << integer Zahl >> ) ==
  do
    ...
  done function Funk2 ;

  procedure Proz (real Zahl) ==
  do
    ...
  done procedure Proz;

  procedure Proz2 (real Zahl) ==
  do
    ...
  done procedure Proz2;

  type DatTyp == record integer RecWert;
                        FunkTyp RecFunk;
                 end record;
DECLARE
  FunkTyp FunkVar := Funk;
  ProzTyp ProzVar := Proz;
  array FunkTyp FunkFeld[1 : 10];
  ...
DO
  Wert := FunkVar(44);
  Proz(55);
  ProzVar(66);
  ProzVar := Proz2;

  Daten.RecFunk := Funk2;
```

```
  Wert := Daten.RecFunk(Wert) ;

  FunkFeld[10] := Funk;

DONE
```

Die Zuweisung erfolgt in gewohnter Manier. Es ist auch möglich, Funktionsvariablen in Feldern oder Datensätzen abzulegen und bei Datensätzen direkt auszuführen. Felder gestatten dieses wegen der damit verbundenen unübersichtlichen Notation nicht.

```
Feld[Index](Parameter) -- nicht erlaubt!!
```

12 Klassen und Objekte

Klassen sind beschreiben Merkmale und Funktionalitäten eines Systems. Entsprechend umfassen Klassen die gleichen Objekte wie Funktionen, wobei jedoch Klassen die jeweiligen Werte permanent halten, nicht flüchtig wie Funktionen, die nur während ihrer Ausführung Daten speichern. Dafür müssen Klassen zunächst instanziiert werden und können erst danach Werte speichern, verändern oder wiedergeben.

12.1 Definition von Klassen in ***FirstStep***

Eine Klasse wird in ***FirstStep*** ähnlich wie eine Prozedur definiert.

```
class KlasseMitAllem(integer Wert) ==
  disclose
    KlasseBasis object;
  declare
    integer InWert;
    FunkTyp Funktion := AussenFunktion;
  do
    InWert := 100*Wert;
    Funktion(898);
    Funktion := AussenFunktion2;
    Funktion(989);
  done class KlasseMitAllem;
```

Eine Klasse ähnlich wie eine Prozedur nicht physisch (als Speicher) vorhanden, sondern wird erst bei ihrem Aufruf, wir sagen hier Instanziierung erzeugt. Im Gegensatz zu einer Prozedur bleibt jedoch die Instanz einer Klasse nach ihrer Instanziierung erhalten. Auf diese kann dann wie auf einen Datensatz durch die Punktnotation zugegriffen werden.

Eine Klasse besteht aus einer öffentlichen Klasse, auf die jeder zugreifen kann, sowie optional eine private, auf die nicht zugegriffen werden kann.

```
private class KlasseMitAllem == disclose ...
```

Sämtliche Variablen oder Funktionen, die in einer privaten Klasse definiert werden, können nur von Ausdrücken innerhalb der öffentlichen oder privaten Klasse referenziert werden. Auf diese Weise kann die öffentliche Klasse als Schnittstelle zu einer privaten Klasse aufgefaßt werden.

Innerhalb der `disclose`-Abteilung kann eine andere Klasse vererbt werden. Sämtliche in der anderen Klasse definierten Bezeichner sind auch in der abgeleiteten Klasse mit entsprechendem Typ bekannt.

Einer Funktionsvariable innerhalb einer Klasse kann eine beliebige Funktion mit gleichem Parameterwerten zugewiesen werden. Man beachte, daß die globalen Werte stets aus der statischen Umgebung einer Funktionsdefinition gewählt werden und nicht aus der Umgebung der Funktionsvariablen!

Klassen werden instanziiert, indem das Schlüsselwort `object` verwendet wird. Zusätzlich ist eine Liste von Parametern anzugeben, welche die jeweiligen aktuellen Werte einer Klasse enthält.

```
declare
->KlasseMitAllem KlasseA :- KlasseMitAllem object (12,13,14,15);
```

Als Parameter sind sämtliche Parameter, auch jene der Basisklassen, anzugeben, so daß die Anzahl der Parameter größer sein kann als die Anzahl der Parameter in der eigentlich instanziierten abgeleiteten Klassen. Mit diesen Parametern wird nur einmal bei der Instanziierung der Anweisungsteil einer Klasse ausgeführt.

Klassen können einer Referenz auf einer ihrer Basisklassen zugewiesen werden. Um zu entscheiden, was für eine dynamische Struktur eine Klasse ist, können die Operatoren `is` und `in` verwendet werden.

```
if KlasseA is KlasseMitAllem -- KlasseA ist genau KlasseMitAllem
then ... ;
if KlasseB in KlasseMitAllem -- KlasseMitAllem ist Basis von KlasseB
then ... ;
```

13 Dateien

Um Daten in ***FirstStep*** dauerhaft abzuspeichern, werden sie in Dateien gespeichert. In ***FirstStep*** enthalten Dateien stets Folgen von ASCII-Zeichen. Es ist nicht vorgesehen, andere Datentypen abzuspeichern.

13.1 Deklaration von Dateien

Dateien werden in ***FirstStep*** folgendermaßen deklariert:

```
file Datei, DateiName, NochEineDatei;
```

Nach dieser Deklaration existiert ein Objekt, welches Text in beliebiger Länge aufnimmt, indem es diesen aneinander hängt und ihn gegebenenfalls wieder zurückgibt.

13.2 Schreiben auf Dateien und Anhängen an Dateien

Um eine Datei zu beschreiben, muß die Datei zunächst zum Schreiben geöffnet werden, wozu ihr ein Name zu geben ist, unter dem sie vom Betriebssystem verwaltet und gespeichert wird; dann können beliebige Texte, Zahlen usw. auf die Datei geschrieben werden, wie es mit dem `write`-Befehl möglich ist, wobei lediglich der Zusatz `on Datei` nötig ist. Nachdem sämtliche Texte auf die Datei geschrieben wurden, sollte diese mit dem Befehl `close Datei` geschlossen werden.

```
open Datei to write on "Ausgabe.dat";
write on Datei "Eins = ", 1, newline;
write on Datei "Zwei = ", 2, newline;
close Datei;
```

Man beachte, daß die Datei "`Ausgabe.dat`" auf jeden Fall neu erzeugt wird. Soll an eine existierende Datei Information angehängt werden, so ist diese mit dem Befehl

```
open Datei to append to "Ausgabe.dat";
```

zu öffnen.

13.3 Lesen von Dateien

Um ein Datum von einer Datei zu lesen, muß die Datei zum Lesen geöffnet werden. Dann kann durch den Befehl `read from Datei ...` von der Datei gelesen werden. Hierbei ist darauf zu achten, daß Texte in der Regel definiert sind als Folgen von Zeichen, gefolgt von einem Leerzeichen bzw. Zeilenende (`newline`).

```
open Datei to read from "Ausgabe.dat";
for Lauf to 2 do
  read from Datei Text1, Text2, Zahl;       -- Lesen von Datei
  write "Gelesen: ",Text1, Text2, Zahl, lf; -- Ausgabe auf TTY
done for Lauf;
close Datei;
```

Allerdings können auf differenziertere Weise Daten von einer Datei gelesen werden. Dazu gibt es die Befehle:

```
read text from Datei Text1 to Lauf;
read text from Datei Text1 upto '=';
read text from Datei Text1 to Lauf until '=';
```

Diese `read text` Befehle haben einen oder zwei Parameter, die jeweils durch eines des Schlüsselwörter `to`, `upto` oder `until` eingeleitet werden. Einer dieser Parameter kann eine Zahl oder ein Zahlenausdruck, der andere ein Zeichen sein. Die Zeichen können eine Zeichenkonstante (‘`A`‘,‘`B`‘,‘`C`‘, ...), eine Zeichenvariable oder eines der Schlüsselwörter `newline`, `tab` oder `endofText` sein. Entsprechend werden höchstens die Anzahl von Zeichen gelesen, aber nur solange, bis das nächste gelesene Zeichen das jeweils angegebene ist. Dabei bedeutet `newline`, daß nur bis zum Ende der Zeile gelesen wird, `tab` daß bis zu ein Tabulatorzeichen gelesen wird, und `endofText` daß die gesamte Datei bis zum Ende gelesen werden soll. Dieses Zeichen können natürlich auch in eine Datei geschrieben werden:

```
write to Datei "Text" , tab , "Letzter Text" , endofText ;
```

14 Externe Module

FirstStep-Programme beschreiben die jeweils benötigte Funktionalität in Form von Klassen, Funktionen, Datenstrukturen und Werten. Häufig werden jedoch in verschiedenen Programmen die gleichen Funktionalitäten benötigt, beispielsweise für graphische Oberflächen oder für Datenbanken. Um extern definierte Objekte in ***FirstStep*** einzubinden gibt es die externen Module.

14.1 Definition eines externen Moduls

Ein externes Modul wird durch die Anweisung

```
<<< Definitionen und Deklarationen für dieses Modul >>>
DO
  save "ModulName";
```

erzeugt. Damit wird eine eigene Datei mit dem Namen `ModulName` erzeugt, die in einer komprimierten Form die Information aus den Definitionen und Deklarationen dieses Programms enthält; der Inhalt des Hauptprogramms (zwischen `DO ... DONE`) wird nicht übernommen. Er kann und sollte für Testzwecke bei der Entwicklung der Module verwendet werden.

Ein derart erzeugtes Modul kann durch die Anweisung

```
DISCLOSE
  extern "ModulName", "ModulName2", "ModulName3",;
DEFINE
  ...
```

in ein anderes Programm übernommen werden. Die Funktionalitäten sind dann dort gleichfalls vorhanden, d.h. die gleichen Bezeichner für Klassen, Funktionen, Variable usw. bekannt. Der Programmierer kann diese Bezeichner genau so verwenden, als hätte er sie zusammen mit den jeweiligen Definitionen und Deklarationen in sein Programm geschrieben. Sind mehrere Module vorhanden, so werden entsprechend sämtliche Definitionen und Deklarationen der verschiedenen Module in das Programm eingebunden.

Besitzen verschiedene Objekte den gleichen Namen, so wird der Namenskonflikt in der Reihenfolge der Module, wie sie in das Programm eingebunden werden, gelöst. Damit ist `Name`, wenn es in dem aktuellen Programm vorkommt, bekannt, und sämtliche Bezeichner `Name` in den Modulen nicht (über ihren Namen) erreichbar. Ist in dem obigen Beispiel `Name1` in `ModulName` und `ModulName2` vorhanden, so ist in dem Programm nur der Bezeichner aus `ModulName` verfügbar, nicht jedoch der aus `ModulName2`.

Die erzeugten Module enthalten die jeweilige Information in einer komprimierten Form. Der Programmierer kann diese zwar lesen, darf sie jedoch nicht verändern, da hier jedes Zeichen, auch beispielsweise Leerzeichen, zählt. Eine Spezifikation des Inhalts der Moduldateien soll hier nicht gegeben werden.

15 Kommunikation zwischen Prozessen

Heutige Programme werden häufig nicht auf einem, sondern auf mehreren Rechnern ausgeführt. Damit diese Programme Daten austauschen können, müssen entsprechende Hilfsmittel vorgesehen werden. In ***FirstStep*** lassen sich Daten mit Hilfe des Client-Server-Konzepts zwischen verschiedenen Rechensystemen austauschen, die über das ***Internetworking Protokoll*** (**IP**) miteinander verbunden sind.

15.1 Definition einer Client-Verbindung

Das folgende Beispiel soll die Programmierung eines „Clients“ verdeutlichen.

```
DECLARE
  connection Client;                       -- Definition einer Verbindung
  text Ergebnis;
  integer NeuerPort := 4747;
DO
  write "Abfragetext = ";
  read text Ergebnis to lf;
  connect Client to "localHost" via NeuerPort; -- Öffnet Client-Verbindung
  write on Client Ergebnis;                -- Sendet Daten über die Verbindung
  read from Client Ergebnis;               -- Wartet auf Datum und liest Text
  read from Ergebnis NeuerPort;            -- Wandelt Text in ganze Zahl
```

Nachdem eine Verbindung durch das Schlüsselwort `connection` definiert worden ist, hier mit dem frei wählbaren Bezeichner `Client`, wird diese Verbindung geöffnet, wozu die IP-Adresse des empfangenden Rechners und die sog. Portnummer dieses Rechners angegeben werden müssen. Die IP-Adresse des Servers erfahren sie von Ihrem Systemadministrator; hier wurde die sogenannte ***Local-Host-Adresse*** verwendet, mit der die

IP-Adresse des gleichen Rechners gemeint ist, auf der das Programm arbeitet. Die Portnummer muß eine Zahl größer als 1024 sein!

Nach diesen Vorbereitungen kann über diese Verbindung beliebig viel Information in Form von Texten, Zahlen, Zeichen oder logischen Werten zwischen Sender und Empfänger ausgetauscht werden. Durch den `write`-Befehl kann auf die Verbindung ein Text geschrieben werden; durch den `read`-Befehl kann entsprechend von dieser Verbindung ein Text gelesen werden, vorausgesetzt der Server sendet einen solchen. Statt Texten können auch ganze, reelle oder komplexe Zahlen sowie logische Werte und Wahrheitswerte geschickt werden. Jeweils muß der Kommunikationspartner allerdings wissen, um welchen Typ es sich hierbei handelt.

15.2 Definition einer Server-Verbindung

Das folgende Beispiel soll die Programmierung eines „Servers“ verdeutlichen.

```
DECLARE
  connection Server;                -- Definition einer Verbindung
  text Anfrage;
  integer PortNummer := 4747;
DO
  connect Server via PortNummer;    -- Öffnet Server-Verbindung
  read from Server Anfrage;         -- Liest einen Text von der Verbindung
  Anfrage := "Ihre Anfrage '" + Anfrage + "' ehrt uns sehr.";
  write on Server Anfrage;          -- Schreibt einen Text auf die Verbindung
  close Server;                     -- Schließt die Verbindung
```

Eine Server-Verbindung wird ebenfalls durch das Schlüsselwort `connection` deklariert. Beim Öffnen eines Servers wird nur die Portnummer angegeben, da der Rechner, von dem die Verbindung gelesen werden soll, der eigene Rechner ist. Jetzt kann beliebig häufig Text beliebiger Länge von der Verbindung gelesen bzw. auf diese geschrieben werden. Will der Server z.B. eine neue Portnummer öffnen, so kann er die Verbindung schließen, was vor Beendigung des Programms auf jeden Fall getan werden sollte.

16 Prozesse

Programme lassen sich auch sinnvoll auf einem Rechner in mehreren sogenannten Prozessen ausführen. Diese können beispielsweise bei mehreren Prozessoren parallel ablaufen und somit schneller sein, wenngleich der Nutzen in Form von Rechenzeitgewinn häufig nicht sehr groß ist.
In ***FirstStep*** lassen sich Prozesse von einem laufenden Programm aus starten und mit diesen kommunizieren. Den Prozessen kann zu Anfang eine Liste von Parametern übergeben werden und nach Beendigung kann eine Prozeß Werte entgegennehmen. Der erste Prozeß wird meistens als Wurzel- oder Rootprozeß bezeichnet; ein Prozeß, der andere Prozeß erzeugt wird als Eltern- oder Parentprozeß bezeichnet und ein erzeugter Prozeß wird als Kind- oder Childprozeß bezeichnet. Childprozesse können natürlich auch wieder zu Parentprozessen werden.

16.1 Erzeugen und Beenden von Prozessen

Das folgende Beispiel soll die Programmierung eines „Parent- und Childprozesses“ verdeutlichen.

```
LINK
  integer "First process starts with number ",Nummer;
  integer "Highest number of processes to be created = ", Anzahl;
DECLARE
  integer Process1,Process2,Process3, ReturnProcess, Status;
  integer NummerChild1, NummerChild2, NummerChild3;
DO
  write "Here starts process number ",Nummer, newline;
  if Nummer*3-1 <= Anzahl then
     create process Process1 execute "first.out"
          with 3*Nummer-1, Anzahl return NummerChild1;
```

```
    if Nummer*3 <= Anzahl then
      create process Process2 execute "first.out"
             with 3*Nummer, Anzahl return NummerChild2;
    end if;
    if Nummer*3+1 <= Anzahl then
      create process Process3 execute "first.out"
             with 3*Nummer+1, Anzahl return NummerChild3;
    end if;
    write "Process ",Nummer,": Child processes started!", newline;
    for Lauf to 3 do
      await process ReturnProcess status Status;
      if ReturnProcess==Process1 then
        write Nummer,": Child ",NummerChild1," finished!", newline;
      elseif ReturnProcess==Process2 then
        write Nummer,": Child ",NummerChild2," finished!", newline;
      elseif ReturnProcess==Process3 then
        write Nummer,": Child ",NummerChild3," finished!", newline;
      else
        write "No Process to wait for!",lf;
      endif;
    done for Lauf;

    write "Process ",Nummer,": All Childs finished!",lf;

  end if;

  if Nummer > 1 then
    exit process status 2 return Nummer;
  else
    write "Root process finished!", newline;
  end if;

DONE
```

Die folgende Anweisung erzeugt einen Prozeß, der unter dem Namen „`first.out`“ als lauffähiges ***FirstStep*** oder C- Programm vorliegen muß. Die Variable „`Process2`“ erhält nach diesem Befehl einen Wert, anhand dessen dieser Prozeß später identifiziert werden kann.

```
create process Process2 execute "first.out"
             with 3*Nummer, Anzahl return NummerChild2;
```

Hinter dem Schlüsselwort `with` wird eine Liste von Parametern ausgegegen. Dieses müssen stets Werteparameter sein, da ein Prozeß nicht auf die lokalen Werte zugreifen kann. Hier können Zahlen, Text, Zeichen oder logische Werte stehen. Wird der erzeugt

Prozeß beendet, so muß er einen oder mehrere Werte zurückgeben, die auf der hinter `return` stehenden Variablenliste abgelegt wird. Da mehrere Prozeß gestartet werden können, sind diese Variablen für jeden Prozeß gesondert anzugeben; andernfalls würden die Werte gegebenenfalls überschrieben werden.

Sobald eine Prozeß beendet wird, muß er durch die Anweisung des Beispiels

```
exit process status 2 return Nummer;
```

die Werte, die für die jeweilige `return`-Parameterliste vorgesehen sind, übergeben. Außerdem kann er einen sogenannten `status` senden, anhand dessen ein Childprozeß dem Parentprozeß mitteilen kann, ob die Ausführung des Prozesses erfolgreich beendet wurde. Der Parentprozeß wartet mittels der Anweisungen

```
await process ReturnProcess status Status;
```

auf den ersten fertig werdenden Prozeß. Anhand des Werts der Variablen `ReturnProcess` kann er feststellen, welcher Prozeß fertig wurde, und dessen Parameter übernehmen und bearbeiten. Außerdem kann er durch den Status feststellen, ob die Ausführung erfolgreich beendet wurde.

Der Childprozeß erhält die Werte, die der Parentprozeß übergeben hat, durch die `Link`-Liste. Nachdem der Childprozeß also gestartet ist, hat er in den jeweiligen Variablen (im Beispiel `Nummer` und `Anzahl`) genau jene Werte, die der jeweilige Parentprozeß in der `with`-Liste angegeben hat.

16.2 Diskussion des Prozeßbeispiels

Das obige Beispiel läßt sich jetzt einfach erklären. Zunächst muß beim Starten des Rootprozesses die Nummer dieses Prozesses (in der Regel 1) und die Anzahl zu erzeugender Prozesse angegeben werden. Jeder Prozeß meldet seinen Start mit der ersten `write`-Anweisung.

Da jeder Parentprozeß bis zu drei Childprozesse erzeugt, werden die Nummerd der Childprozesse des Parentprozesses N nach der Formel: 3*N-1, 3*N, und 3*N+1 berechnet.

Wenn die Nummer mindestens eines zu erzeugenden Prozesses nicht größer als gefordert ist, wird die erste `if`-Anweisung ausgeführt. Zunächst wird der Prozeß mit der kleinsten Nummer erzeugt, dann wird abgefragt, ob auch die anderen Prozesse noch erzeugt werden sollen. Nachdem mindestens ein Prozeß erzeugt wurde, wird eine Schleife dreimal durchlaufen, die jeweils auf einen terminierenden Childprozeß wartet, dessen

Rückgabewerte bearbeitet, hier lediglich ausgibt, und dann auf den nächsten terminierenden Prozeß, soweit er von diesem Parentprozeß erzeugt wurde, erwartet. Wurden weniger als drei Prozesse gestartet, so meldet das System entsprechend einen Fehler, fährt aber in der Ausführung fort.

Nachdem sämtliche Childprozesse eines Parentprozesses terminiert sind, terminiert auch der Parentprozeß, wobei lediglich der Rootprozeß nur eine entsprechende Meldung ausgibt.

16.3 Kommunikation zwischen Prozessen

Auch zwischen Parent- und Childprozessen können Nachrichten ausgetauscht werden. Dieses geschieht durch die Schreibe- und Leseanweisungen

```
write to parent Text;
read from parent Text;
read from process ProcessName Wert;
write to process ProcessName "Hallo, wie geht es denn heute?";
```

also praktisch genauso wie beim Schreiben und Lesen auf Verbindungen, Texte, Dateien oder den Standardmedien. Es sind lediglich die Schlüsselwörter `parent` anzugeben, wenn der Childprozeß dem Parentprozeß eine Nachricht schicken will, und `process ProcessName`, wenn ein Parentprozeß seinem Childprozeß mit dem Namen `ProcessName` eine Nachricht schicken will. Wieder müssen sich Parent- und Childprozeß über den Typ der jeweiligen Variablen einig sein.

17 Namens- und Sachverzeichnis

P

R

S

T

Ü

V

W

Z