Einführung in die Programmierung
(inkl. Geschichte der Programmiersprachen)

Dirk Louis

Peter Müller

1 Inhaltsverzeichnis

2 Einführung in die Programmierung *(inkl. Geschichte der Programmiersprachen)*

Nehmen Sie sich ein wenig Zeit, lehnen Sie sich entspannt zurück und beginnen Sie zu schmökern. Den Rechner können Sie ruhig ausgeschaltet lassen. Es geht hier nicht darum, Sie programmieren zu lehren; es geht darum, Sie mit den wichtigsten Fachbegriffen und den grundlegenden Konzepten moderner Programmiersprachen vertraut zu machen.

Leser, die vornehmlich an den Ausführungen zur Geschichte der Programmiersprachen interessiert sind, finden diese in den "Geschichte"-Abschnitten am Ende der einzelnen Unterkapitel.

Icon NOTE

2.1 Programme, Prozessoren und Programmiersprachen

Früher, so etwa in der Mitte des 20. Jahrhunderts, als es noch keine Programmiersprachen gab, war alles viel einfacher. Programmieren bedeutete damals, eine Reihe von abzuarbeitenden Befehlen aufzuschreiben, diese in den Computer einzufüttern und sie vom Rechner ausführen zu lassen. Das Ganze war so leicht, dass man dazu nicht einmal Programmierer oder Informatiker brauchte, meist überließ man das Programmieren gewöhnlichen Mathematikern.

Oder hatte es doch andere Gründe, dass sich damals so viele Mathematiker als Programmierer versuchten? Auf den zweiten Blick muss man feststellen, dass nicht alles in der Frühzeit der Computertechnologie so paradiesisch einfach war, wie oben angedeutet. So verstanden die Computer nur einen begrenzten Satz elementarer Befehle, aus denen der Programmierer "komplexere" Anweisungen erst mühsam aufbauen musste; weiterhin mussten die Befehle binär, d.h. als eine Folge von Nullen und Einsen, codiert werden, denn nur so konnte der Rechner sie verarbeiten; und die ersten Programmierer waren deshalb vornehmlich Mathematiker, weil zum einem die Informatik damals ein aufstrebendes Teilgebiet der Mathematik war[1] und zum anderen die Mathematiker zu den ersten

[1] "Informatik" ist eine Zusammensetzung aus "Information" und "Mathematik".

Nutznießer gehörten, die den Computer für komplexe, zeitaufwendige Berechnungen und Simulationen verwendeten.

Seitdem hat sich vieles geändert. Die Informatik ist zu einer eigenen Fachrichtung aufgestiegen, die Mathematiker müssen sich den Computer mit anderen Wissenschaftlern, Betriebswirtschaftlern und privaten Anwendern teilen und die Rechner selbst sind immer kleiner[2] und leistungsfähiger geworden. Das eigentliche Rechenwerk oder "Gehirn" des Rechners, der **Prozessor**, ist heute ein Chip von nur noch wenigen Quadratzentimetern.

2.1.1 Prozessor und Maschinencode

Wie die ersten Rechenmaschinen versteht der moderne Prozessor lediglich einen begrenzten Satz elementarer Befehle. Aus diesen Befehlen baut der Programmierer seine Programme auf. Dies ist aber gar nicht so leicht, denn die Befehle, die der Computer, d.h. der Prozessor des Computers, versteht, beschränken sich weitgehend auf die Verschiebung von Daten und die Ausführung einfacher elementarer Rechenschritte.

Um zum Beispiel den Prozessor anzuweisen, eine Zahl, die an einer bestimmten Stelle im Arbeitsspeicher (RAM) abgelegt ist, um 3 zu erhöhen, sind drei Befehle nötig:

```
Hole den Inhalt der Arbeitsspeicherzelle 3FA66501 und kopiere ihn in den
Prozessorregister EAX.[3]

Nehme den Inhalt aus Register EAX und addiere den Wert 3 dazu.

Hole den Inhalt des Prozessorregisters EAX und speichere ihn in der
Arbeitsspeicherzelle 3FA66501.
```

Natürlich können die Befehle nicht in obiger Form an den Prozessor geschickt werden, denn der kann ja nur Bits, sprich Folgen von Nullen und Einsen, verarbeiten. Was macht der Programmierer also? Mit säuerlicher Miene besorgt er sich Handbuch und Referenz zu seinem Prozessor, um darin nachzuschlagen, wie er die einzelnen Befehle maschinenlesbar codieren muss. Für die Intel-Pentium-Prozessoren findet er die

---

[2] ENIAC, die erste vollelektronische Rechenmaschine, füllte noch eine ganze Halle.

[3] Register sind der Speicher des Prozessors. Der Intel-Prozessor enthält 8 allgemeine Register, in denen zu verarbeitende Daten zwischengespeichert werden können, sowie eine Reihe weiterer Register für spezielle Aufgaben (etwa den Befehlsregister, in dem die Adresse des nächsten auszuführenden Befehls steht).

notwendigen Informationen beispielsweise in der "Instruction Set"-Referenz, die von der Webseite `http://developer.intel.com/design/pentium4/manuals` heruntergeladen werden kann.

Icon STOPP

Jeder Prozessor definiert seinen eigenen Satz von Befehlen. So gibt es beispielsweise unter den ca. 500 Befehlen, die der Pentium 4-Prozessor kennt, etliche die einem 486er- oder einem 286er-Prozessor gänzlich unbekannt sind. Auch die Codierung der Befehle kann für verschiedene Prozessoren unterschiedlich sein. Die Pentium-Prozessoren basieren beispielsweise auf einer 32-Bit-Architektur und können daher auch Befehle mit Operanden (Zahlen, Adressen u.a.) verarbeiten, die 32 Bit groß sind. Ein 286er-Prozessor, der im Wesentlichen mit einer 16-Bit-Architektur arbeitet, kann angesichts solcher Befehle nur passen. Und wenn Sie einen Intel-Prozessor gar mit einem SPARC-Prozessor von Sun vergleichen, werden Sie kaum noch Übereinstimmungen finden.

Das Befehlsformat moderner Prozessoren ist recht kompliziert. Wir reduzieren es daher für die weiteren Ausführungen auf das Wesentliche:

- einen kurzen *OpCode*, der angibt, welcher Befehl auszuführen ist, und
- ein oder zwei *Operanden*, die durch den Befehl verarbeitet werden.

Für die Intel-Prozessoren lautet der OpCode des Befehls, der Daten aus einer Speicherzelle in das Register EAX schreibt, 1010 0001. Üblicherweise übernehmen die Kopierbefehle zwei Operanden, einen für das Ziel und einen für die Quelle, doch da es für das Kopieren nach EAX einen eigenen Befehl gibt, muss nur noch die Adresse der Speicherzelle angegeben werden. Die hexadezimale Speicheradresse 3FA66501 ist in Binärcode gleich 00111111101001100110010100000001, womit sich für den Maschinenbefehl folgende Binärdarstellung ergibt:

```
10100001 00111111101001100110010100000001
```

Icon NOTE

Beachten Sie, dass hier unterschiedliche Arten von Codierungen Verwendung finden. Der OpCode ist vom Prozessor fest vorgegeben (d.h. der Prozessorhersteller hat beim Design des Prozessors entschieden, welche Bitfolgen für welche Befehle stehen). Positive ganze Zahlen und Adressen werden dagegen einfach vom Dezimal- ins Binärsystem umgerechnet. Wieder andere Codierungen regeln, wie Gleitkommazahlen (3,24, 0,003) oder Zeichen (Buchstaben, Ziffern etc.) als Folge von Nullen und Einsen darzustellen sind. Grundsätzlich gilt: Um vom Prozessor verarbeitet werden zu können, müssen Befehle und Daten in Bits codiert werden.

In unserem vereinfachten Format sehen die Maschinenbefehle für die oben beschriebene Addition wie folgt aus:

```
10100001 00111111101001100110010100000001

10000011 1100000 00000000000000000000000000000011

10100011 00111111101001100110010100000001
```

Diese Befehle in einen Editor einzutippen und als Datei abzuspeichern, erzeugt allerdings noch kein Programm. Wenn Sie eine 0 oder eine 1 in einen Editor eingeben, versteht dieser die "0" bzw. die "1" nicht als Bits, sondern als die Ziffern 0, respektive 1. Folglich speichert er Ihre Maschinenbefehle nicht als binäre Befehle, sondern als Ziffernfolge ab. Und selbst wenn Sie den Editor dazu bringen könnten, die Zahlenfolgen als echten Maschinencode abzuspeichern, erhielten Sie damit noch kein ausführbares Programm. Dieses muss nämlich noch mit einem Startcode versehen und in ein spezielles Dateiformat (unter Windows das PE-Format) gebracht werden, damit das Betriebssystem weiß, wie das Programm zu laden und auszuführen ist.

Sollten Sie jetzt das Gefühl haben, dass diese Art der Programmierung viel zu kompliziert und absolut untragbar ist, so kann ich Ihnen nur beipflichten. Und die ersten Programmierer mussten es wohl ähnlich empfunden haben, denn es dauerte nicht lange, bis die Maschinencodeprogrammierung – erst durch Assembler, später dann auch durch die höheren Programmiersprachen – zurückgedrängt und abgelöst wurde.

2.1.2 Assembler

Programme als eine Folge von Maschinenbefehlen zu schreiben, ist äußerst mühselig und extrem fehleranfällig. Wie schnell hat man eine 0 getippt, wo eigentlich eine 1 stehen sollte. Natürlich ist so etwas verzeihlich. Programmierer sind eben auch nur Menschen. Wer hätte dafür kein Verständnis? Vermutlich der arme Tropf, dem die Aufgabe zugefallen ist, die falsche Null in dem Heuhaufen aus Nullen und Einsen aufzuspüren.

Zum Programmieren gehört aber nicht nur das Niederschreiben der Befehle. Zuerst muss der Programmierer überlegen, welche Befehle der Rechner in welcher Reihenfolge abarbeiten soll, damit das Programm die gestellte Aufgabe erfüllt. Im Grunde ist dies die entscheidende Arbeit des Programmierers, und sie wird durch den binären Maschinencode nicht gerade erleichtert.

"So, den Inhalt der Speicherzelle XXX habe ich in ein Register geschrieben. Jetzt muss ich die Zahl 3 addieren. Hmm, in welchem Register habe ich den Inhalt der Speicherzelle noch einmal abgelegt? EAX, EBX oder doch EDX? Schnell mal in der vorangehenden Zeile nachschauen. Ah ja, der OpCode lautet 10100001, das heißt, ich habe die Daten wohin kopiert????"

Sicher, wir könnten diesem gestressten Programmierer helfen und ihm verraten, dass seine Daten in EAX stehen, doch es lohnt die Mühe nicht. Wer heute noch in Maschinencode programmiert, an den ist jede Hilfe verschwendet. Das soll nun aber nicht heißen, dass die Programmierung auf Maschinenebene nicht auch ihre Vorzüge und Reize hätte. Ganz im Gegenteil! Wenn nur nicht das leidige Codieren und Rechnen mit den binären Bits wäre.

In den Fünfzigern des letzten Jahrhunderts kamen die Programmierpioniere auf die Idee, ihre Programme nicht in binär codierten, sondern in symbolischen Maschinenbefehlen zu entwickeln. Die Befehle bekamen kurze, aber leicht zu merkende Namen (die so genannten Mnemonics) und die Operanden wurden direkt als Register, Zahlen oder Adressen angegeben. Die oben vorgestellten Maschinenbefehle zum Addieren könnten in symbolischer Form wie folgt lauten:

```
MOV  EAX, [3FA66501]
ADD  EAX, 3
MOV  [3FA66501], EAX
```

wobei MOV und ADD Mnemonics für die Maschinenbefehle sind, EAX für das Register EAX steht und [3FA66501] die Speicheradresse 3FA66501 bezeichnet.

Anfangs wurden die symbolischen Befehle auf Papier niedergeschrieben und später, wenn das Programm fertig war, manuell vom Programmierer in Maschinencode (d.h. in Nullen und Einsen) übertragen. Es dauerte jedoch nicht lange, und es gab die ersten Programme, die dem Menschen diese lästige und stupide Arbeit abnahmen: die **Assembler**.

Heute bezeichnet man sowohl die Programmierung mit symbolischen Maschinenbefehlen als auch das zugehörige Übersetzerprogramm als Assembler.

Warum die Programmierung in Assembler so schwer fällt

In Assembler entspricht jeder symbolische Befehl genau einem Maschinenbefehl. Gegenüber der Programmierung mit binär codierten Maschinenbefehlen ist dies zweifelsohne eine gewaltige Erleichterung, doch es ist immer noch eine sehr maschinennahe Programmierung, die ein

gründliches Studium der Arbeitsweise des Prozessors voraussetzt und den Menschen zwingt, wie eine Maschine zu denken – was dem Programmieren nicht immer zuträglich ist.

Programmieren als Form der Problemlösung

Programmieren ist letztlich nichts anderes als eine spezielle Form des Problemlösens. Der Programmierer bekommt eine Aufgabe gestellt, die er mit Hilfe oder in Gestalt eines Programms lösen soll. Zu diesem Zweck entwirft der Programmierer, im Kopf oder auf Papier, einen Ablaufplan, der Schritt für Schritt zur gewünschten Lösung führt. Der Ablaufplan für ein Programm zur Berechnung von Mittelwerten könnte beispielsweise wie folgt aussehen:

1. Lies alle Werte ein.
2. Summiere die Werte auf.
3. Teile die Summe durch die Anzahl der Werte.
4. Gebe das Ergebnis aus.

Derartige Schritt-für-Schritt-Anleitungen zur Lösung bestimmter Probleme werden in der Informatik als **Algorithmen** bezeichnet.

Der nächste Schritt auf dem Weg zum fertigen Programm besteht nun darin, die Anweisungen des Algorithmus in Quellcode umzusetzen. Für den Assembler-Programmierer heißt dies, die Anweisungen in Assembler-Befehle zu übersetzen. Dass dies nicht so einfach ist und viel Einfühlungsvermögen in die Arbeitsweise des Rechners erfordert, liegt auf der Hand. Erschwerend hinzu kommt, dass die Programmierung in der Regel kein geradliniger Prozess ist, der vom Entwurf über den Quellcode zum fertigen Programm führt. Meist muss der Programmierer den Quellcode mehrfach überarbeiten und korrigieren. Dabei reift sein Verständnis für das Programm und er erkennt Schwächen und Fehler im Entwurf/Algorithmus, an die er zuvor nie gedacht hätte. Also wird der Entwurf nochmals geändert und die Änderungen in den Quellcode eingearbeitet. Schließlich ist das Programm soweit gediehen, dass es getestet werden kann. Hat der Programmierer Glück, sind nur noch ein paar kleine Fehler auszumerzen, hat er Pech, erweist sich das Programm als viel zu langsam oder als extrem bedienerunfreundlich und das gesamte Programm muss neu überdacht werden. Wie auch immer, der Programmierer muss ständig im Kopf die Anweisungen des Algorithmus in Quellcode und den Quellcode in algorithmische Anweisungen übersetzen. Dies fällt naturgemäß umso schwerer, je unterschiedlicher die Sprachen sind, in denen Algorithmus und Quellcode formuliert werden. Und bei der Assembler-Programmierung, wo unsere natürliche Sprache, in der die Algorithmen verfasst werden, auf die Assemblersprache mit ihren

symbolischen Maschinenbefehle trifft, ist dieser Unterschied besonders groß.

Bereits früh in der Geschichte der Programmierung gab es daher Bestrebungen, diese Kluft zu verkleinern und die "Sprache des Quellcodes", wenn schon nicht der menschlichen Sprache so doch wenigstens der menschlichen Denkweise anzunähern.

2.1.3 Compiler, Interpreter und höhere Programmiersprachen

Zwei wichtige Schritte auf dem Weg zu einer menschengerechteren und damit produktiveren Programmierung waren

- die Einführung von Variablen und
- die Bündelung mehrerer Maschinenbefehle zu höheren, für den Menschen besser verständlichen Anweisungen.

Variablen sind letztlich nichts anderes als symbolische Speicheradressen.

Programme verarbeiten Daten: Zahlen, Zeichen, Texte, Bilder, Sound und so weiter. Die eigentliche Datenverarbeitung findet im Prozessor statt. Da aber nicht alle Daten auf einmal verarbeitet werden können, muss das Programm Daten für die spätere Weiterverarbeitung irgendwo ablegen können. Als Zwischenspeicher für die Daten dient dabei der Arbeitsspeicher des Rechners, der RAM[4].

Der Arbeitsspeicher

Den Arbeitsspeicher können Sie sich als ein Gebilde aus etlichen Tausend Speicherzellen vorstellen, die durchweg nummeriert sind. Die Nummern der Speicherzellen bezeichnet man auch als Speicheradressen. Werden Daten im Arbeitsspeicher abgelegt, belegen sie dort je nach Größe eine oder mehrere aufeinander folgende Speicherzellen. Die Adresse der ersten Zelle ist die Adresse der Daten.

In den Maschinenbefehlen aus den vorangehenden Abschnitten wurden die Speicheradressen, von denen Daten geladen oder in die Daten geschrieben werden sollten, stets als explizite (hexadezimale oder binäre) Adressen

[4] Der Code des Programms, also die binär codierten Maschinenbefehle, werden übrigens ebenfalls in den Arbeitsspeicher geladen.

angegeben. Dies hat zwei gravierende Nachteile: Der Programmierer muss sich merken oder gesondert notieren, welche Daten an welchen Adressen stehen ("Wert 1 steht in FF01, Wert 2 in FF05, die Anzahl der Werte in EF03 und die Summe in EF5A"). Und er ist selbst für die korrekte Speicherbelegung verantwortlich, muss also beispielsweise aufpassen, dass er die Summe nicht in einen Speicherbereich schreibt, in dem schon die einzelnen Werte stehen.

Hier bringt das Konzept der Variablen, das schnell auch in Assembler Einzug fand, Abhilfe. Statt numerischer Speicheradressen vergibt der Programmierer Namen und überlässt dem Übersetzerprogramm (im Falle der Assembler-Programmierung also dem Assembler) die Zuordnung dieser Namen zu echten Speicheradressen. Statt nichts sagender Zahlen kann der Programmierer nun Namen auswählen, die auf Art und Zweck der gespeicherten Daten hinweisen, beispielsweise `alter`, `preis`, `anzahlWerte`.

Icon INFO

Das Beziehungsgefüge aus Name, zugehörigem Speicherort und darin abgelegten Daten wird als Variable bezeichnet. Im alltäglichen Sprachgebrauch wird die Variable aber auch häufig mit ihrem Wert, d.h. mit den in ihr abgelegten Daten, gleichgesetzt. Wenn ein Programmierer also davon spricht, dass er die Variable A zur Variablen B addiert, so meint er damit, dass der Wert in der Variablen A zu dem Wert der Variablen B hinzuaddiert wird.

Anweisungen sind Befehle höherer Programmiersprachen.

Was höhere Programmiersprachen sind? Wenn Sie Programme in binär codiertem Maschinenbefehlen schreiben, müssen Sie sich ganz tief bis auf die Maschinenebene herab begeben. Wenn Sie in Assembler programmieren, schalten Sie ein Übersetzerprogramm, den Assembler, dazwischen, der Ihnen erlaubt symbolische Maschinenbefehle zu verwenden. Sie programmieren aber immer noch auf Maschinenebene. Höhere Programmiersprachen erlauben Ihnen dagegen, auf einer höheren Ebene zu programmieren – also nicht mehr mit Maschinenbefehlen, sondern mit Befehlen, die für Menschen besser verständlich und lesbar sind. Um diese Befehle auch begrifflich von den Maschinenbefehlen klar abzugrenzen, spricht man von Anweisungen.

Eine Anweisung, die die Zahl `4` zu dem Wert der Variablen `preis` addiert, könnte beispielsweise lauten:

```
preis = preis + 4;
```

Wie in Assembler bedarf es natürlich auch hier eines passenden Übersetzerprogramms, das diese Art von Anweisungen in Maschinencode

übersetzt. Eine einfache 1:1-Übersetzung von Anweisung in Maschinenbefehl ist allerdings nicht mehr möglich, vielmehr bedarf es mehrerer Maschinenbefehle um eine solche Anweisung auszuführen – und auch einer neuen Klasse von Übersetzerprogrammen. Genauer gesagt gibt es sogar zwei Klassen von Übersetzern für höhere Programmiersprachen: Compiler und Interpreter.

> Ein **Compiler** liest den gesamten Quellcode des Programms (oder Programmmoduls) ein, analysiert ihn und übersetzt ihn dann in Maschinencode. Dieser Maschinencode kann mit Hilfe eines weiteren Programms, des **Linkers**, mit anderem Maschinencode verbunden und in ein ausführbares Programm verwandelt werden (beispielsweise eine Windows-EXE-Datei). Zum Starten des Programms wird die ausführbare Programmdatei aufgerufen.
>
> Ein **Interpreter** liest den Quellcode Zeile für Zeile ein. Jede Zeile wird sofort übersetzt und direkt zur Ausführung an den Prozessor weiter gereicht. Zum Ausführen des Programms muss also nur der Quelltext des Programms an den Interpreter übergeben werden.

Für jede höhere Programmiersprache gibt es einen eigenen Compiler oder Interpreter. Wortschatz und Grammatik der Programmiersprache sind in der Spezifikation der Sprache niedergeschrieben und im Compiler (oder Interpreter) implementiert. Die Sprache wird also genauso gut durch den Compiler (Interpreter) wie durch ihre Spezifikation definiert. Oder um es noch drastischer zu formulieren: Wenn Sie ein Programm geschrieben haben, das sich mit dem zugehörigen Compiler nicht übersetzen lässt, nutzt es Ihnen gar nichts, wenn Sie belegen können, dass Ihr Quelltext exakt den Regeln der Sprachspezifikation folgt. Letzte Instanz ist in so einem Fall der Compiler. Sie können sich bei dem Compiler-Hersteller beschweren, aber wenn Sie Ihr Programm übersetzen möchten, müssen Sie den Regeln folgen, die der Compiler implementiert[5].

Icon NOTE

Die obige Unterscheidung von Compiler und Interpreter beschreibt die traditionellen Grundprinzipien, die mit Kompilation und Interpretation verbunden sind. Obwohl diese Grundprinzipien nach wie vor gelten, können moderne Compiler und Interpreter in einzelnen Punkten davon

[5] Ärgerlich ist es, wenn es zu einer Programmiersprache mehrere Compiler (bzw. Interpreter) gibt, die die Syntax und Grammatik der Sprache jeder für sich ein wenig verändern oder eigenmächtig interpretieren. In so einem Fall schafft jeder dieser Compiler (Interpreter) einen eigenen, inkompatiblen Sprachdialekt (wie dies Microsoft beispielsweise mit der Einführung seines Visual J++-Compilers anstrebte).

abweichen. (Der Java-Compiler erzeugt beispielsweise weder echten Maschinencode noch ausführbare EXE-Dateien. Dazu später mehr.)

2.1.4 Geschichte: Vom Maschinencode zu den höheren Programmiersprachen

Die erste höhere Programmiersprache dürfte die von dem deutschstämmigen Wissenschaftler Konrad Zuse bereits Mitte der Vierziger entwickelte Sprache Plankalkül gewesen sein. Doch Plankalkül war eine Sprache ohne Computer, und Zuse konnte seiner Arbeit in den schweren Jahren des zweiten Weltkrieges und den Wirren der Nachkriegszeit nur bedingt nachgehen. So kam es, dass Plankalkül erst 1972 der Öffentlichkeit vorgestellt wurde. Doch da waren Zuse und Plankalkül bereits von den Entwicklungen in USA eingeholt worden.

Bis in die Vierziger waren "Computer" rein mechanische Rechenmaschinen[6]. Dann kam die große Zeit der elektronischen Rechenmaschinen. Zwar gab es noch keine Transistoren und erst recht keine Chips oder integrierte Schaltkreise, doch Vakuumröhren und Relais taten auch ihren Dienst – sie brauchten lediglich ein wenig mehr Platz. ENIAC[7], die erste vollelektronische Rechenmaschine, füllte denn auch eine ganze Halle. ENIAC war vielseitig einsetzbar, musste aber für jede neue Berechnung neu verdrahtet werden, d.h., die einzelnen Einheiten mussten neu zusammengestellt und verschaltet werden.

Inspiriert von ENIAC beschäftige sich der Mathematiker John von Neumann, der nach der Machtübernahme Hitlers nach USA ausgewandert war, intensiver mit der Theorie der Rechenmaschinen. Er formulierte eine Reihe von Ideen und Konzepten, die unsere Vorstellung von einem Computer nachhaltig geprägt haben:

- Ein Computer sollte eine einfache, feststehende Struktur haben, die es ihm gestattet, unterschiedliche Programme auszuführen, ohne dass dazu Änderungen an der Hardware vorgenommen werden müssen. Kurz gesagt: Hard- und Software sollten getrennt werden.

---

[6] Bekannt geworden sind vor allem die Hollerith-Maschinen, die seit 1890 für Volkszählungen und statistische Erhebungen verwendet wurden. Noch in den Dreißigern machte IBM-Chef Thomas Watson gute Geschäfte mit Hollerith-Maschinen, die er an Nazi-Deutschland verkaufte.

[7] ENIAC ist die Abkürzung für "Electronic Numerical Integrator And Calculator".

- Programmcode und Daten sollten zusammen im Rechner gespeichert werden.
- Die Programmausführung sollte jederzeit unterbrochen und an einer anderen Stelle des Programms fortgesetzt werden können (Grundlage für Sprünge, Verzweigungen, Schleifen und Funktionsaufrufe).

Die ersten "von Neumann"-Rechner wurden Ende der Vierziger fertig gestellt und in binärer Maschinensprache programmiert. Sie verwendeten in der Regel 16-Bit-Befehle: vier Bits für den Befehlscode, 12 Bit für eine einzelne Adressenangabe. Typische Berechnungen umfassten in etwa 200 Befehle, d.h. der Programmierer musste 3200 Nullen und Einsen via Lochkarte, Tastatur oder Band in den Rechner eingeben! Kein Wunder also, dass es nicht lange dauerte, bis sich neue, fortschrittlichere Formen der Programmierung durchsetzten.

Bereits 1950 begann der Siegeszug der Assembler-Programmierung. Einer der ersten Rechner, der in Assembler programmiert wurde, war der EDSAC[8], ein in England von Maurice V. Wilke entwickelter "Von Neumann"-Rechner. Der EDSAC unterstützte gerade einmal 18 Maschinenbefehle, im Vergleich zu den ca. 500 Befehlen eines modernen Intel-Rechners also eine verschwindend geringe Zahl, aber genug um den Programmierer mit Befehlen zum Addieren, Subtrahieren und Multiplizieren, zum Ein- und Ausgeben, Laden und Speichern von Daten sowie ersten Befehlen zur Steuerung des Programmflusses zu erfreuen.

Nahezu zeitgleich reifte in der Mathematikern Grace Hopper die Idee für den ersten Compiler. Hopper, die 1943 in die United States Naval Reserve eintrat und es bis zum Konteradmiral brachte (einen höheren Dienstgrad konnten Frauen damals in der U.S. Navy nicht erreichen), war Programmiererin der ersten Stunde. In den letzten Jahres des zweiten Weltkriegs arbeitete sie in Harvard an dem Mark I Calculator, einem der letzten großen elektromechanischen Rechner[9], den sie programmieren sollte. Aus irgendeinem Grund arbeitete der Mark I nicht korrekt. Hopper fand den Fehler: Es war eine Motte (engl. Bug), die sich in einem Relais verfangen hatte. Seitdem bezeichnet man Fehler in Programmen als "Bugs" und das Ausmerzen von Fehlern als "Debuggen".

---

[8] EDSAC ist die Abkürzung für "Electronic Delay Storage Automatic Calculator".

[9] der 1946 fertig gestellte ENIAC war das vollelektronische Pendant zum Marc I.

Hopper sagte später einmal, dass sie den Compiler nur aus Faulheit entwickelt hätte (und weil sie hoffte, dass aus Computer-Programmierern wieder Mathematiker würden). Aus Faulheit? Warum nicht? Die Aussicht auf Arbeitserleichterung dürfte der Motor hinter vielen Erfindungen gewesen sein.

Die erste weltweit erfolgreiche höhere Programmiersprache war FORTRAN[10]. FORTRAN wurde von John Backus bei IBM für die 700er-Serie entwickelt und speziell auf die Bedürfnisse von Mathematikern, Naturwissenschaftlern und Ingenieuren abgestimmt. 1957 wurde der FORTRAN-Compiler für den 704er fertig gestellt.

2.2 Eine eigene Programmiersprache?

Der beste Weg, sich mit den grundlegenden Elementen und Konzepten moderner Programmiersprachen vertraut zu machen, besteht zweifelsohne darin, eine eigene Programmiersprache zu entwickeln. Da die Verwirklichung eines solchen Unterfangens allerdings sehr viel Zeit kosten würde (und Stoff für ein eigenes Buch wäre), begnügen wir uns mit dem *Entwurf einer eigenen Programmiersprache*, den wir Schritt für Schritt ausarbeiten und fortführen, bis am Ende eine Programmiersprache steht, die – wer hätte es gedacht – der Programmiersprache Java sehr ähnlich sein wird und die wir daher *Bali*[11] nennen.

2.2.1 Zielsetzung

Jede höhere Programmiersprache entsteht im Spannungsfeld zweier entgegengesetzter Anforderungen:

- Auf der einen Seite soll sie für den Menschen leicht verständlich sein.
- Auf der anderen Seite muss sie von einem Übersetzerprogramm in Maschinencode übertragen werden können.

Es gibt sicherlich mehrere Wege, eine Programmiersprache zu entwickeln, die beiden Anforderungen genügt. Ein denkbarer Ansatz wäre, als Ausgangspunkt eine Sprache zu wählen, die eine der gestellten Forderungen von vorneherein erfüllt – also entweder die eigene Muttersprache, die zumindest von allen Programmierern gleicher

[10] FORTRAN steht für "formula translator".

[11] Bali ist eine kleine Nachbarinsel von Java.

Nationalität verstanden wird, oder Assembler, das sich wie oben ausgeführt besonders leicht in Maschinencode übersetzen lässt – und diese Sprache dann so weit zu verändern, dass sie auch das jeweils zweite Kriterium erfüllt. Beide Möglichkeiten scheiden jedoch aus verschiedenen Gründen für uns aus. Um die eigene Muttersprache für ein Übersetzerprogramm verständlich zu machen, müssten wir sie so stark vereinfachen, dass im Endeffekt ein monströses Regelwerk von Ausschlussregeln und Positiv-/Negativlisten entsteht, welches für den Menschen viel schwieriger zu erlernen sein dürfte als eine neue Sprache. Die Ableitung einer höheren Programmiersprache aus Assembler wäre schon eher denkbar, setzt aber entsprechende Assembler-Kenntnisse voraus, über die wir nicht verfügen.

Wir wählen daher einen grundsätzlich anderen Ansatz: Wir stellen uns einfache Programmieraufgaben und überlegen, welche Konstrukte wir in die Sprache einführen müssen, um die gestellten Aufgaben lösen zu können. Die Konstrukte versuchen wir so zu wählen, dass sie unsere beiden Grundanforderungen genügen. Erfreulicherweise ist dies gar nicht so schwierig, denn der scheinbare Widerspruch zwischen den Anforderungen löst sich auf, wenn die Programmiersprache nicht abgeleitet, sondern von Grund auf neu erdacht wird. Dann sind Mensch wie Übersetzerprogramm nämlich in der gleichen Situation, dass sie die Sprache "erlernen" müssen und dies fällt beiden umso leichter, umso konsequenter wir uns beim Sprachdesign an folgende Richtlinien halten

- *kleiner Wortschatz*
- *einfache Syntax*
- *klare Semantik*

und da Programmierzeit kostbar und Programmierer faul sind, fügen wir noch hinzu:

- die Tipparbeit soll sich in Grenzen halten[12].

[12] Um die Schreibarbeit klein zu halten, werden wir viele Symbole verwenden, die aus einzelnen Zeichen bestehen – wie zum Beispiel das "+" zur Kennzeichnung einer Addition oder die geschweiften Klammern "{" und "}" zur Gruppierung von Anweisungen. Java verwendet die gleichen Symbole, jedoch nicht aus Rücksicht auf die meist nur mäßigen Tippkünste der Programmierer, sondern um die eigene Syntax der Syntax von C++ anzupassen und so C++-Programmierern den Umstieg zu erleichtern. C++ wiederum übernahm die Syntax C und C, das zu einer Zeit entwickelt wurde, als Arbeits- und Festplattensspeicher kostbare und rare Güter waren, verwendete die Symbole unter anderem um die Dateigröße der

Anforderungskatalog für den ersten Entwurf

Für den ersten Entwurf unserer Programmiersprache *Bali* beschränken wir uns auf die grundlegenden Aufgaben, die jede Programmiersprache unterstützen muss:

- Datenrepräsentation und -verwaltung
- Datenbearbeitung
- Einlesen und Ausgeben von Daten

Und

- Kommentierung des Quelltextes

Nach Fertigstellung des Entwurfs sollten wir in *Bali* erste Programme schreiben können – beispielsweise ein Programm, mit dem man das hierzulande veraltete, in England und Amerika aber immer noch gebräuchliche Längemaß Fuß in Zentimeter umrechnen kann. Der Algorithmus für dieses Programm könnte wie folgt aussehen:

1. Lese die Länge in Fuß über die Tastatur ein.
2. Rechne die Länge in Zentimeter um (Umrechnungsfaktor 30,48).
3. Gebe die Länge in Zentimeter aus.

2.2.2 Erster Entwurf: die Daten

Programme verarbeiten Daten: Zahlen, Texte, Bilder, Töne, Adressen – die Liste könnte beliebig fortgesetzt werden. Angesichts dieser unüberschaubaren Vielzahl an Datentypen, versuchen wir erst gar nicht, alle erdenklichen Datentypen zu unterstützen, sondern konzentrieren uns im ersten Entwurf unserer Programmiersprache *Bali* auf vier einfache, für die Programmierung aber äußerst bedeutsame Datentypen: ganze Zahlen, reelle Zahlen (mit Nachkommastellen), einzelne Zeichen und Texte (sprich Aneinanderreihungen mehrerer Zeichen).

Programmquelltexte klein zu halten. Das klingt vielleicht seltsam, ist aber einfach zu erklären: C sollte kompiliert werden. Dazu ist es erforderlich, dass der Compiler die komplette Quelltextdatei in den Arbeitsspeicher lädt. Während des Kompilierens muss der Arbeitsspeicher also das Betriebssystem, das Compiler-Programm, die Quelltextdatei und die vom Compiler angelegten Hilfsdatenstrukturen aufnehmen. Aus heutiger Sicht kein Problem, doch auf Rechnern, die wie der IBM-XT gerade einmal über 64 KByte Arbeitsspeicher verfügten, ein echter Engpass.

Für jeden dieser Datentypen werden wir jetzt festlegen, wie seine Werte im Quelltext des Programms dargestellt werden können.

Werte, die direkt im Quelltext stehen, bezeichnet man übrigens als Konstanten oder – um ganz exakt zu sein – als **Literale**. Wichtig ist, dass das Übersetzerprogramm, wenn es den Quelltext einliest und in Maschinencode übersetzt, die Literale an ihrem Format erkennen und sowohl untereinander als auch von anderen Sprachelementen (die wir noch einführen werden) unterscheiden kann.

Zahlen-Literale

Aus der Mathematik sind Ihnen eine ganze Reihe von Zahlendarstellungen bekannt: 14, 13,456, 10^{-3}, 1/3 und so weiter. Zwei dieser Zahlendarstellungen wollen wir in unsere Sprachspezifikation übernehmen, wobei wir das Format (die Schreibweise) zum Teil ein wenig verändern:

```
123
```

```
123.4
```

Das erste Format ist für ganz normale ganzzahlige Werte.

Das zweite Format, `123.4`, ist für Zahlen mit Nachkommastellen.

Etwas befremdlich ist der Punkt zur Abtrennung der Nachkommastellen. Als Deutscher würde man hier ein Komma erwarten und den Punkt zur Kennzeichnung der Tausenderstellen verwenden, wie z.B. in 1.000.000,50 €. Die Engländer und Amerikaner machen es jedoch genau umgekehrt, und da das Englische so etwas wie die Ziehmutter aller Programmiersprachen ist, werden in Programmiersprachen die Nachkommstellen meist mit einem Punkt eingeleitet. Um keine unnötige Verwirrung zu stiften, schließen wir uns dieser Konvention an.

Negative Zahlen werden einfach durch Voranstellung eines Minuszeichens gebildet:

```
-123
```

Ganzzahlen werden in der Informatik als **Integer**, Zahlen mit Nachkommastellen als **Fließ-** oder **Gleitkommazahlen** bezeichnet.

Icon INFO

Zeichen- und Zeichenfolgen-Literale

Einzelne Zeichen kennzeichnen wir dadurch, dass wir Sie in Hochkommata setzen:

```
'z'
```

Icon NOTE

Das einzelne Hochkommata befindet sich auf der deutschen Tastatur über dem Teppichsymbol #.

Zeichenfolgen, die auch **Strings** genannt werden, setzen wir in Anführungszeichen:

```
"string"
```

Variablen

Den Literalen oder Konstanten stehen die Variablen gegenüber. Wie Sie bereits wissen (bzw. aus Abschnitt 2.1.3 wissen sollten), gehören zu einer Variablen ein Name und ein eigener Speicherbereich, in dem ein Wert abgelegt werden kann. Über den Variablennamen kann der Wert abgefragt oder durch einen neuen Wert ersetzt werden.

Selbstverständlich soll es auch in unserer Programmiersprache *Bali* möglich sein, Variablen einzurichten. Die Frage ist nur wie? Die Lösung liegt in einer vernünftigen Aufgabenteilung: der Programmierer vergibt den Namen und das Übersetzerprogramm reserviert und verwaltet den Speicherbereich.

Wir legen fest: Variablennamen dürfen vom Programmierer frei gewählt sein, so lange er darauf achtet, dass die Namen nur aus Buchstaben, Ziffern und Unterstrichen bestehen und nicht mit einer Ziffer beginnen.

Diese wenigen, aber wichtigen Regeln beschneiden den Programmierer kaum in der freien Wahl seiner Variablennamen und sorgen gleichzeitig dafür, dass das Übersetzerprogramm die Variablennamen von den Literalen und anderen Sprachelementen (die wir noch einführen werden) unterscheiden kann.

Gültige Variablennamen sind:

```
einevar
eineVar
laengeInZentimeter
x
Punkt_zahl
```

Nicht erlaubt sind dagegen:

```
3facherWert             // beginnt mit Ziffer
sauerstoffgehalt_in_%   // enthält ungültiges Zeichen %
```

Trifft das Übersetzerprogramm auf einen neuen Variablennamen, reserviert es Speicher für die Variable. Um sich merken zu können, welche Variablen bereits definiert sind und mit welchen Speicheradressen sie verbunden sind,

legt das Übersetzerprogramm intern eine Tabelle an, in der es Name und Speicheradresse aller erzeugten Variablen einträgt.

Wie die meisten Programmiersprachen unterscheidet *Bali* zwischen Groß- und Kleinschreibung. `eineVar` und `einevar` sind demnach unterschiedliche Variablen!

Icon STOPP

Wir haben nun festgelegt, wie in *Bali* Literale (Konstanten) aussehen und wie Variablen erzeugt werden können. Jetzt müssen wir noch einen Weg vorgeben, wie der Programmierer mit den Literalen und Variablen arbeiten kann.

2.2.3 Erster Entwurf: die Anweisungen

Sinn und Zweck einer Variablen ist die Aufbewahrung von Werten. Die erste Anweisung, die wir einführen, ist daher die Zuweisung von Werten an Variablen.

Zuweisungen

Das allgemeine Format einer Zuweisung soll wie folgt aussehen:

```
VAR = WERT;
```

Links steht also der Name der Variablen, die den Wert aufnehmen soll, rechts der Wert, der in der Variablen abgelegt wird.

Dazwischen steht das Gleichheitszeichen, das dem Übersetzerprogramm anzeigt: der Wert rechts soll in die Variable links geschrieben werden. Wir hätten auch ein anderes Symbol wählen können, beispielsweise einen Pfeil <-, aber das Gleichheitszeichen ist schneller einzutippen und sicherlich genauso ausdrucksstark. (Symbole, die wie das Gleichheitszeichen, festlegen, was mit Daten zu geschehen hat, nennt man **Operatoren**.)

Und das Semikolon? Es soll dem Übersetzerprogramm die Analyse des Quelltextes erleichtern, indem es das Ende der Anweisung anzeigt. (Zeichen, die den Quelltext strukturieren, nennt man **Satzzeichen**.)

Somit sind wir jetzt in der Lage, in *Bali* einer Variablen `demo` den Wert `34` zuzuweisen:

```
demo = 34;
```

Die gleiche Syntax kann auch verwendet werden, um einer Variablen den Wert einer anderen Variablen zuzuweisen:

```
var1 = 1;
var2 = var1;
```

Die erste Anweisung erzeugt die Variable `var1` und weist ihr den (literalen) Wert `1` zu. Die zweite Anweisung erzeugt die Variable `var2` und weist ihr den Wert der Variablen `var1` zu. Danach steht in beiden Variablen der Wert `1`.

Weitere Operatoren

Als Nächstes legen wir fest, wie Daten verarbeitet werden können.

Für das von uns angestrebte Umrechnungsprogramm ist es beispielsweise erforderlich zwei Werte, die Längenangabe in Fuß und den Umrechnungsfaktor, miteinander zu multiplizieren. Dazu benötigen wir eine spezielle Syntax, die eben diese Multiplikation erlaubt. Um die Multiplikation zu kennzeichnen, könnten wir ein spezielles Schlüsselwort einführen, beispielsweise `multiplizieren`, das angibt, was mit den beiden Werten geschehen soll. Doch wozu die mühselige Tipparbeit, wo doch die Verwendung der üblichen mathematischen Symbole so nahe liegt. Wir greifen daher auf den *-Operator zurück und legen die Syntax für die Multiplikation wie folgt fest:

```
OP * OP
```

wobei OP für Operand steht.

Trifft das Übersetzerprogramm auf diesen Ausdruck berechnet er das Ergebnis der Operation. Aufgabe des Programmierers ist es, dem Übersetzerprogramm noch mitzuteilen, was mit dem Ergebnis geschehen soll. Meist wird er es in einer Variablen speichern:

```
VAR = OP * OP;
```

Icon INFO

Eine Kombination aus Operatoren und Operanden (Literale, Variablen) bezeichnet man in der Programmierung als **Ausdruck**. Ausdrücke können ausgerechnet werden und liefern einen Ergebniswert. Sie können daher überall dort in einem Programmquelltext stehen, wo ein Wert erwartet wird – beispielsweise auf der rechten Seite einer Anweisung (oder auch als Operand in einem anderen Ausdruck). Im einfachsten Fall besteht ein Ausdruck direkt aus einem Wert (sprich aus einem Literal oder einer Variablen).

Neben dem *-Operator für die Multiplikation definieren wir noch weitere Operatoren für die anderen Grundrechenarten:

```
OP + OP          // Addition
OP - OP          // Subtraktion
```

```
OP * OP          // Multiplikation
OP / OP          // Division
```

So weit, so gut, doch es gibt da ein Problem. Wenn als Operanden beliebige Werte zugelassen werden, könnte jemand auf die Idee kommen, in seinem Quelltext folgende Anweisung einzubauen:

```
eineVar = "Hallo " * 3;
```

Was der Programmierer damit zu bezwecken versucht, lässt sich erahnen. Er will den String `"Hallo "` vermutlich verdreifachen, so dass in `eineVar` der Wert `"Hallo Hallo Hallo "` abgelegt wird. Doch ist dies möglich?

Grundsätzlich ist alles möglich, was wir in der Sprachspezifikation vorsehen. Nicht möglich ist es jedoch, Operatoren, die für einen Datentyp gelten, auf andere Datentypen zu übertragen. Mit anderen Worten: Operatoren sind datentypspezifisch. Der Grund hierfür liegt auf der Hand: Was eine Operation bedeutet und wie sie durchzuführen ist, hängt von der Art und der Beschaffenheit der Daten ab.

Für jeden Operator, die wir in unsere Sprache einführen, müssen wir also angeben, für welchen Datentyp dieser Operator gültig ist und was er bewirkt. Der Programmierer weiß dann, wie er den Operator verwenden kann, und das Übersetzerprogramm weiß, wie es die Operationen in Maschinencode übersetzen muss.

`+`, `-`, `*` und `/` sind in *Bali* für die Ganzzahlen und Gleitkommazahlen definiert und implementieren die Grundrechenarten.

Für Strings definieren wir nur einen Operator, den wir ebenfalls durch das +-Zeichen symbolisieren. Mit diesem Operator können zwei Strings aneinander gehängt werden:

```
gruss = "Hallo " + "Programmierer!"
```

Diese Anweisung speichert in der Variablen `gruss` den String `"Hallo Programmierer!"`.

Die Aneinanderreihung von Strings wird in der Informatik als **Konkatenation** bezeichnet.

Icon INFO

Dass wir für die Addition von Zahlen und die Konkatenation von Strings das gleiche Operatorsymbol verwenden, ist durchaus statthaft, führt aber zu einer Uneindeutigkeit, die wir auflösen müssen. Welcher Operator ist gemeint, wenn ein String und eine Zahl mit dem +-Operator verbunden werden? Wir legen fest, dass in diesen Fällen stets der String-Operator

gemeint ist und dass das Übersetzerprogramm die Zahl in einen String umwandeln soll, bevor es dann beide Strings aneinander hängt.

```
punktzahl = 1200;
text = "Ihre Punktzahl: " + punktzahl;
```

In `text` steht danach der String `"Ihre Punktzahl: 12000"`.

Icon NOTE

Den *-Operator definieren wir nicht für Strings. Zwar gibt es Programmiersprachen, die einen Operator zur Vervielfältigung von Strings vorsehen (beispielsweise in Perl; der Operator lautet dort `x`), doch in Anlehnung an unser großes Vorbild Java verzichten wir in *Bali* darauf.

Bleibt noch der Datentyp der einzelnen Zeichen.

In Java sind für diesen Typ die gleichen Operatoren wie für die Zahlen definiert. Das mag verwundern, lässt sich aber damit erklären, dass die Zeichen intern durch Zahlen codiert werden. Doch mit diesem Thema werden wir uns erst in der Zwischenbilanz beschäftigen, wenn wir dem Übersetzerprogramm bei seiner Arbeit über die Schultern schauen. Für unsere eigene Sprache legen wir einfach fest, dass es keine Operatoren für Zeichen gibt.

Ein- und Ausgabe

"Alle Programme verarbeiten Daten, nützliche Programme kommunizieren sie auch."

Wir sind mittlerweile soweit, dass wir in *Bali* ein Programm schreiben können, das eine vorgegebene Länge in Fuß in die entsprechende Länge in Zentimeter umrechnet. Für eine Länge von 12 Fuß würde das Programm dann beispielsweise wie folgt aussehen:

```
laengeInZM = 12 * 30.48;
```

Gäbe es einen passenden Compiler oder Interpreter für *Bali*, könnten wir das Programm auch ausführen lassen, doch viel Freude hätten wir wohl nicht daran, denn das Programm behält den berechneten Wert für sich. Was fehlt, ist schlichtweg irgendeine Form der Ausgabe, die uns über das Ergebnis in Kenntnis setzt.

Ausgabe

Grundsätzlich gibt es zwei wichtige Formen der Ausgabe: die Ausgabe auf den Bildschirm und die Ausgabe in eine Datei. Wir entscheiden uns für die Unmittelbarere, die Ausgabe auf den Bildschirm, und nehmen diese direkt in die Sprache auf. Die Syntax für die Ausgabe soll lauten:

```
out STRING;
```

`out` erklären wir zum **Schlüsselwort** der Sprache[13], d.h. im Gegensatz zu den Variablennamen, die der Programmierer nach Bedarf einführt, gehört es zum festen Wortschatz der Sprache (und des Übersetzerprogramms). Direkt hinter dem Schlüsselwort `aus` folgt der String, der ausgegeben werden soll.

Das verbesserte Programm lautet jetzt

```
laengeInZM = 12 * 30.48;
out laengeInZM;
```

Wird das Programm so ausgeführt, erscheint auf dem Bildschirm die Ausgabe:

```
365.76
```

Wir können die Ausgabe noch etwas benutzerfreundlicher gestalten und dem Ergebniswert einen erläuternden Text voranstellen. Wir brauchen dafür nicht einmal eine zweite `aus`-Anweisung, es genügt der Einsatz des Stringoperators `+`.

```
laengeInZM = 12 * 30.48;
out "Laenge in Zentimeter: " + laengeInZM;
```

Ausgabe:

```
Laenge in Zentimeter: 365.76
```

Bildschirmausgaben und Konsolenanwendungen

Die meisten PC-Benutzer, vor allem Windows- oder KDE-Anwender, sind daran gewöhnt, dass die Programme als Fenster auf dem Bildschirm erscheinen. Dies erfordert aber, dass das Programm mit dem Fenstermanager des Betriebssystem kommuniziert und spezielle Optionen und Funktionen des Betriebssystems nutzt. Unsere einfachen *Bali*-Programme tun dies nicht; sie erzeugen keine Fenster und können diese folglich auch nicht als Schnittstelle zum Benutzer verwenden.

Wo also erscheinen die Ausgaben des Programms?

Programme, die keine eigenen Fenster erzeugen, bezeichnet man als Konsolenprogramme. Die Konsole ist eine spezielle Umgebung, die dem Programm vorgaukelt, es lebe in der guten alten Zeit, als es noch keine Window-Systeme gab und immer nur ein Programm zur Zeit ausgeführt werden konnte. Dieses Programm konnte dann uneingeschränkt über alle

[13] Die meisten Programmiersprachen entlehnen ihre Schlüsselwörter dem Englischen, daher "out" und nicht "aus".

Ressourcen des Rechners verfügen – beispielsweise die Tastatur, das wichtigste Eingabegerät, oder auch den Bildschirm, das wichtigste Ausgabegerät. Der Bildschirm war in der Regel in den Textmodus geschaltet, wurde also nicht aus Pixelreihen, sondern aus Textzeilen aufgebaut.

Unter Windows heißt die Konsole MS-DOS-Eingabeaufforderung oder auch nur Eingabeaufforderung und kann je nach Betriebssystem über START/PROGRAMME oder START/PROGRAMME/ZUBEHÖR aufgerufen werden.

Konsolenprogramme werden meist von der Konsole aus aufgerufen, d.h., am Eingabeprompt (Ende der untersten Textzeile der Konsole) wird der Name des Programms eingetippt und mit der (Enter)-Taste abgeschickt. Die Ausgaben des Programms erscheinen Zeile für Zeile darunter (sind die Zeilen der Konsole voll geschrieben, werden sie nach oben gescrollt). Eingaben über die Tastatur werden in der aktuellen Zeile der Konsole angezeigt und nach Drücken der (Enter)-Taste an das Programm weitergeleitet.

Eingabe

Trotz Ausgabe ist der Nutzwert unseres Programms immer noch recht beschränkt, denn das Programm kann nur die eine Länge von 12 Fuß in Zentimeter umrechnen. Wir könnten das Programm erweitern und nacheinander die Längen von 1 bis 20 Fuß in Zentimeter umrechen und ausgeben, doch ändert dies nichts an dem eigentlichen Manko des Programms: Es verarbeitet nur eigene Daten, die als Literale im Quelltext stehen. Für manche Programme mag dies ausreichen, für unser Programm nicht. Unser Programm entspricht erst dann den gestellten Anforderungen, wenn es die Längenangabe, die von Fuß in Zentimeter umgerechnet werden soll, vom Benutzer entgegen nimmt. Analog zur Ausgabe definieren wir daher noch eine Syntax für das Einlesen von Daten über die Tastatur:

```
in VAR;
```

Die `in`-Anweisung liest eine einzelne Eingabe von der Tastatur ein und speichert sie in einer Variablen. Wie `out` ist auch `in` ein Schlüsselwort der Sprache.

Mit Hilfe der `in`-Anweisung ist es nun ein Leichtes, das Programm wie gewünscht fertig zu stellen:

```
out "Geben Sie eine Länge in Fuss ein";
in laengeInFuss;
laengeInZM = laengeInFuss * 30.48;
out "Laenge in Zentimeter: " + laengeInZM;
```

Wenn das Programm von der Konsole aus gestartet wird, gibt es zuerst einen Text aus, der den Benutzer informiert, welche Art Eingabe von ihm erwartet wird. Danach wartet die `in`-Anweisung, bis der Anwender die gewünschte Längenangabe über die Tastatur eintippt und durch Drücken der (Enter)-Taste abschickt. Den eingegebenen Wert speichert das Programm in der Variablen `laengeInFuss`. Danach folgt die Umrechnung in Zentimeter und die Ausgabe des Ergebnisses.

Ein Punkt macht mir jedoch noch Kopfzerbrechen. Ebenso wie auf dem Textbildschirm der Konsole nur Zeichen ausgegeben werden, können über die Tastatur nur Zeichen und Zeichenfolgen eingelesen werden. Wenn der Benutzer unseres Programm für die umzurechnende Länge 19 eingibt, kommt in der Variablen `laengeInFuss` also nicht die Zahl 19, sondern der String "19" an. An sich wäre das kein Problem, wenn nicht in der Zeile darunter der Wert der Variablen `laengeInFuss` mit 30,48 multipliziert würde. Für Strings ist der *-Operator aber nicht definiert. Was also soll das Übersetzerprogramm tun?

Der *-Operator ist für die Multiplikation von Zahlen definiert. Wenn seine Operanden Strings sind, soll das Übersetzerprogramm die Strings in Zahlen umwandeln. Geht dies nicht, etwa weil die Eingabe des Benutzers keinem Zahlenformat entspricht, dass das Übersetzerprogramm versteht (z.B. "1.000.000" oder "drei" oder "fast 3"), so soll das Programm die Übersetzung abbrechen.

Icon NOTE

Die in diesem Abschnitt vorgestellten Ein- und Ausgabe-Befehle `in` und `out` gibt es in Java nicht! Ein- und Ausgabe über Tastatur und Bildschirm sind in Java nicht in die Sprache integriert, sondern werden als gesonderte Funktionalität über die Java-Standardbibliothek zur Verfügung gestellt. Der Grund hierfür ist, dass Ein- und Ausgabe stark plattformspezifisch sind (also von Hardware und Betriebssystem abhängen) und Sprachentwickler grundsätzlich bemüht sind, plattformspezifische Konstrukte aus der Sprache herauszuhalten.

2.2.4 Erster Entwurf: Kommentare und Whitespace

Die *Bali*-Sprachspezifikation ist nun soweit ausgearbeitet, dass wir unser erstes vernünftige Programm schreiben konnten. Wir haben uns viel Gedanken darüber gemacht, wie Daten im Code repräsentiert und verarbeitet werden können; wir haben peinlich darauf geachtet, dass die Syntaxformen der Anweisungen und die Formate der Literale eindeutig sind und vom Übersetzerprogramm korrekt identifiziert und verarbeitet

werden können; vollkommen vernachlässigt haben wir dagegen die Frage, wie wir den Programmierer dabei unterstützen können, gut lesbaren, verständlichen Quellcode zu schreiben. Dies wollen wir nun schleunigst nachholen

Kommentare

Die Anweisungen höherer Programmiersprachen sind zweifelsohne besser verständlich als Assemblerbefehle. Doch wenn Sie einmal in die Verlegenheit geraten, einen umfangreicheren Programmquelltext überarbeiten zu müssen, den Sie selbst vor längerer Zeit geschrieben haben, werden Sie feststellen, wie schwer es manchmal fällt, den Sinn einer Anweisung oder einer Gruppe von Anweisungen herauszufinden, die Ihnen zum Zeitpunkt der Niederschrift doch vollkommen klar war. Um diesem Gedächtnisschwund entgegenzuwirken und allgemein den Programmierern die Einarbeitung in fremde Quelltexte zu erleichtern, gestatten wir in *Bali* die Kommentierung der Quelltexte.

Der Programmierer soll bei der Kommentierung seiner Quelltexte keinerlei Beschränkungen unterworfen sein, d.h., Kommentare dürfen von beliebigem Umfang und Inhalt sein und an jeder beliebigen Stelle im Quelltext sein.

Das Übersetzerprogramm soll die Kommentare vollkommen ignorieren. Dazu muss es aber wissen, wo ein Kommentar anfängt und wo er endet. Genau diese Kriterien legen wir nun fest und unterscheiden danach zwischen zwei Formen:

- Die erste Form des Kommentar beginnt mit der Zeichenfolge // und endet am Ende der aktuellen Zeile.
- Die zweite Form beginnt mit /* und endet mit */.

`/* */`-Kommentare werden meist für mehrzeilige Bemerkungen verwendet, die über den Anweisungen stehen, die sie kommentieren.

`//`-Kommentare enthalten meist kurze, einzeilige Informationen, die vor oder neben den betreffenden Anweisungen stehen.

Whitespace oder die Bedeutung der Leere

Ohne groß darüber nachzudenken, haben wir in der Syntax der *Bali*-Anweisungen Leerzeichen zur Trennung der einzelnen Teile der Anweisung verwendet.

```
VAR = OP * OP;

out STRING;
```

Und als wäre es eine Selbstverständlichkeit, haben wir die Anweisungen unseres Programms nicht nur mit dem Semikolon, sondern auch noch mit einem Zeilenumbruch abgeschlossen, so dass jede Anweisung in einer eigenen Zeile stand.

```
out "Geben Sie eine Länge in Fuss ein";
in laengeInFuss;
...
```

Leerzeichen und Zeilenumbruch gehören zusammen mit Tabulator und Seitenvorschub zu den Zwischenraumzeichen, die in der Programmierung unter dem Begriff **Whitespace**[14] zusammengefasst werden. Die Whitespace-Zeichen erfüllen im Programmquelltext zwei wichtige Aufgaben:

- sie trennen Sprachelemente und
- sie können zur übersichtlicheren Gestaltung des Quelltextes beitragen.

So müssen z.B. in der `in`-Anweisung das Schlüsselwort `in` und der Variablenname durch Whitespace getrennt werden.

```
in laengeInFuss;
```

Würden beide zusammengeschrieben (`inlaengeInFuss`), würde das Übersetzerprogramm das Schlüsselwort `in` als Teil des Variablennamens interpretieren.

Welches und wie viele Whitespace-Zeichen zwischen Schlüsselwort und Variablenname stehen, ist jedoch egal – zumindest aus Sicht der Sprache und des Übersetzerprogramms. Für die Lesbarkeit des Quelltextes ist an dieser Stelle das Leerzeichen zu empfehlen; ein Tabulator würde die beiden Wörter zu weit auseinander schieben, ein Zeilenumbruch würde gar die Anweisung auseinander reißen.

Vor und hinter einem Satzzeichen muss kein Whitespace stehen. Das Satzzeichen übernimmt in diesem Fall die Aufgabe des Trennzeichens. Ob Sie also `in laengeInFuss;` oder `in laengeInFuss ;` schreiben – beides ist korrektes *Bali*.

Icon NOTE

In Java fungieren neben den Whitespace-Zeichen und den Satzzeichen auch noch Operatoren als Trennzeichen. So kann man zur Addition zweier Variablen `a` und `b` in Java sowohl `a + b` als auch `a+b` schreiben.

[14] Zu Deutsch "weißer Raum".

Optional ist der Whitespace beispielsweise zwischen zwei Anweisungen, denn diese werden ja bereits durch das Semikolon getrennt. Es ist also durchaus erlaubt, in Bali mehrere Anweisungen hintereinander zu schreiben:

```
out "Geben Sie eine Länge in Fuss ein";in laengeInFuss;
```

Übersichtlicher ist es in der Regel jedoch – Sie werden mir da sicher zustimmen –, wenn jede Anweisung in einer eigenen Zeile steht, die Anweisungen also durch einen Zeilenumbruch getrennt werden. Und wenn man Anweisungen gruppieren möchte, tippt man einfach zwei Zeilenumbrüche hintereinander ein.

Das fertige Programm

Kommentiert und mit Whitespace formatiert sieht unser Umrechnungsprogramm wie folgt aus:

```
/* Programm zur Umrechnung von Längenangaben
   von Fuss in Zentimeter
*/
out "Geben Sie eine Länge in Fuss ein";
in laengeInFuss;

laengeInZM = laengeInFuss * 30.48;

out "Laenge in Zentimeter: " + laengeInZM;
```

2.2.5 Zwischenbilanz

Sie haben nun bereits eine ganze Menge über die Programmierung, über Variablen und Operatoren, über Ausdrücke und Anweisungen oder auch die Eindeutigkeit von Syntaxformen und die Lesbarkeit von Quelltexten gelernt. Trotzdem gibt es noch einige Punkte nachzutragen. Die meisten dieser Punkte haben mit der Arbeit des Übersetzerprogramms zu tun, so dass wir diesem einfach einmal bei der Arbeit, sprich bei der Übersetzung unseres Beispielprogramms zuschauen.

Wie bereits erwähnt, bestehen Programmquelltexte aus einer Folge einfacher Textzeichen, die als Textdatei abgespeichert und zur Umwandlung in Maschinencode an ein Übersetzerprogramm übergeben werden. Das Übersetzerprogramm liest die Zeichen ein und beginnt mit der Quelltextanalyse, die drei Schritte umfasst:

- die *lexikalische Analyse*, bei der das Übersetzerprogramm die einzelnen "Wörter" identifiziert, aus der der Quelltext aufgebaut ist,

- die *syntaktische Analyse*, die prüft, ob die einzelnen Wörter entsprechend der Grammatik zu korrekten Sätzen zusammen gestellt wurden,
- die *semantische Analyse*, die die Bedeutung der Wörter und Sätze erfasst und die eigentliche Grundlage für die nachfolgende Codeerzeugung ist.

Ein Interpreter führt diese drei Schritte (grundsätzlich) für jede Zeile eines Programms aus, ein Compiler vollzieht jeden Schritt nur einmal für das gesamte Programm.

Wenn wir also untersuchen wollen, wie ein Übersetzerprogramm mit unserer Sprachspezifikation und unserem Programmquelltext zurecht kommt, müssen wir zuerst entscheiden, ob *Bali* interpretiert oder kompiliert werden soll. Keine leichte Entscheidung, doch wenn ich mir unseren Programmquelltext so betrachte, scheint mir die klare Abfolge der Anweisungen prädestiniert für die Verarbeitung durch einen Interpreter[15].

Der Ausgangspunkt

Wir steigen just zu dem Zeitpunkt in die Analyse ein, da der Interpreter sich anschickt, die dritte Anweisung (`laengeInZM = ...`) zu übersetzen. Was ist bis dahin geschehen?

Der Programmquelltext wurde eingetippt und als Textdatei abgespeichert. Diese wurde dem Interpreter übergeben, um von diesem Anweisung für Anweisung übersetzt und ausgeführt zu werden.

Wo beginnt das Programm?

Doch mit welcher Anweisung beginnt eigentlich das Programm und in welcher Reihenfolge sollen die Anweisungen ausgeführt werden?

Die Antwort lautet natürlich, dass die Anweisungen beginnend mit der ersten Quelltextzeile von oben nach unten ausgeführt werden sollen.

Icon NOTE

So trivial und abwegig, wie die Frage nach dem Beginn des Programms zu sein scheint, ist sie übrigens nicht, denn in modernen Programmiersprachen stehen die Anweisungen nicht mehr nur brav untereinander gereiht im Quelltext, sondern sind verteilt und in Funktionen oder Methoden organisiert (dazu später mehr).

---

[15] Ganz so leicht wie hier angedeutet, sollte man sich die Entscheidung zwischen Interpreter und Compiler nicht machen, denn sie prägt die Sprache nachhaltig. Verantwortliche Sprachentwickler entscheiden sich daher bereits früh für Compiler, Interpreter oder – wie im Falle von Java – eine Zwischenform.

Die erste Anweisung – der darüber stehende Kommentar wird vom Interpreter ignoriert – lautete:

```
out "Geben Sie eine Länge in Fuss ein";
```

Anders als kleine Zahlenliterale oder einzelne Zeichen können Strings nicht direkt als Operanden in Maschinenbefehle eingebaut werden; sie müssen zuerst in den Arbeitsspeicher geschrieben werden. Der Interpreter erzeugt also Maschinencode, der für den String Speicher reserviert und die Zeichen des Strings in diesen Speicherbereich schreibt. Danach führt er die Maschinenbefehle aus, die den abgespeicherten String auf die Konsole ausgeben.

In der zweiten Anweisung taucht eine Variable auf.

```
in laengeInFuss;
```

Der Interpreter reserviert Speicher für die Variable. Weiter führt er die Maschinenbefehle aus, die eine Tastatureingabe einlesen, und speichert die Eingabe in der Variablen. Die Anweisung ist damit korrekt ausgeführt und übersetzt, doch der Interpreter macht noch mehr.

Für den Fall, dass die Variable später noch einmal benutzt wird, merkt er sich den Namen der Variablen und die Adresse ihres Speicherbereiches. Beide Informationen trägt er in eine interne Tabelle ein (die so genannte **Symboltabelle**), mit deren Hilfe er über alle Namen ("Symbole") Buch führt, die vom Programmierer eingeführt werden.

Die nächste auszuführende Anweisung ist

```
laengeInZM = laengeInFuss * 30.48;
```

Die lexikalische Analyse

Vor der lexikalischen Analyse besteht der restliche Quelltext für den Interpreter aus einer Folge unverständlicher, nichts sagender Zeichen – als handele es sich um lauter Xe, deren eigentliche Bedeutung im Dunkeln liegt:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
```

Um aus dieser Zeichenflut die bedeutungstragenden Wörter und Zeichenfolgen, die so genannten **Token** herauszuspüren, sucht der Compiler nach Trennzeichen, sprich nach Whitespace- und Satzzeichen (in Java auch Operatoren), Kommentare ignoriert er. Trifft er auf das Ende einer Anweisung bricht er ab. In unserem Fall wäre dies hinter dem die Anweisung abschließenden Semikolon:

```
xxxxxxxxxx x xxxxxxxxxxxx x xxxx;
```

Im nächsten Schritt bestimmt der Compiler, um welche Art von Wörtern (Schlüsselwörter, Namen, Literale, Operatoren oder Satzzeichen) es sich bei den einzelnen Token handelt und ob diese richtig geschrieben sind.

Das erste Wort, `laengeInZM`, erkennt der Compiler als Name. Er schlägt in seiner Symboltabelle nach und erkennt, dass dieser Name neu ist. Also legt er eine neue Variable dieses Namens an: Er reserviert den Speicher und trägt Name und Speicheradresse in die Tabelle ein.

Als Nächstes folgen der Zuweisungsoperator (=) und ein weiterer Name. Der Interpreter schlägt den Namen in seiner Symboltabelle nach und wird fündig: Die Variable existiert bereits, er muss also keinen Speicher reservieren.

Die letzten Token werden als Multiplikationsoperator, Gleitkommaliteral und abschließendes Semikolon identifiziert.

Die syntaktische Analyse

Der Interpreter hat nun alle Token erfasst, identifiziert und verifiziert. Der logische nächste Schritt ist, zu überprüfen, ob die Wörter (Token) auch zu korrekten Sätzen (Anweisungen) zusammengesetzt wurden.

In Token formuliert, lautet die Anweisung:

```
VAR = VAR * LITERAL;
```

Da sowohl Variablennamen als auch Literale als Operanden verwendet werden dürfen, gehorcht diese Anweisung dem Satzkonstrukt `VAR = OP * OP;`. Sie ist also syntaktisch korrekt.

Die semantische Analyse

Jetzt muss der Interpreter nur noch analysieren, was die Anweisung

```
laengeInZM = laengeInFuss * 30.48;
```

bedeutet und wie sie in Maschinencode zu übersetzen ist.

Am `=`-Operator erkennt er, dass es sich um eine Zuweisung handelt, d.h., der rechte Teil soll der Variablen auf der linken Seiten als Wert zugewiesen werden. Dazu muss der Interpreter zuerst berechnen, für welchen Wert der Ausdruck auf der rechten Seite der Anweisung steht. Und genau hier stößt er (und wir mit ihm) auf ein gravierendes Problem!

Um den Ausdruck zu berechnen, muss der Wert der Variablen `laengeInFuss` mit dem Literal `30.48` gemäß der Definition des *-Operators verrechnet werden. Klingt an sich einfach, ist es aber nicht, denn die Bedeutung des *-Operators hängt vom Datentyp der Operanden ab! Wir

erinnern uns: Zwei Zahlen (Integer- oder Gleitkomma) werden einfach multipliziert, ist ein Operand ein String, wird dieser in eine Zahl umgewandelt (geht dies nicht, wird die Programmausführung abgebrochen), für zwei String-Operanden ist der Operator überhaupt nicht definiert.

Der Interpreter muss jetzt also entscheiden, welche Operation zu den Datentypen der Operanden passt, doch genau dies ist ihm nicht möglich, denn woher soll er wissen, welche Art von Daten in `laengeInFuss` steht? *Wir* wissen, dass in der Variablen ein String, nämlich die Tastatureingabe aus der vorangehenden Anweisung gespeichert ist. Doch der Interpreter weiß dies nicht. Er kann den Datentyp auch nicht durch einen Blick in den Speicher der Variablen eruieren, denn dort findet er nur eine Folge von Nullen und Einsen. Diese codieren zwar den Wert der Variablen, doch sie geben keinen Hinweis darauf, wie die Ausgangsdaten aussahen. Ergo: Ohne den Datentyp der abgespeicherten Daten zu kennen, bestehen diese nur noch aus einer (scheinbar) beliebigen Folge von Nullen und Einsen, 00110101001110 ..., denen der Interpreter vollkommen ratlos gegenüber steht.

Um das Problem zu lösen, müssen wir dem Interpreter eine Art Gedächtnis verleihen, in dem die Informationen über die Datentypen bewahrt werden können. Dieses "Gedächtnis" könnte beispielsweise die Symboltabelle sein. Jedes Mal, wenn in einer Variablen ein Wert abgespeichert wird, trägt der Interpreter in der Symboltabelle ein, welche Art von Wert diese Variable danach beinhaltet.

Überprüfen wir das Verfahren für unser Beispielprogramm:

In der zweiten Anweisung wird eine Tastatureingabe eingelesen und in `laengeInFuss` gespeichert. Der Interpreter legt die Variable wie gehabt an und trägt zusätzlich in der Symboltabelle ein, dass die Variable aktuell einen String enthält (Eingaben sind immer Strings).

In der dritten Anweisung wird der Wert der Variablen benötigt. Der Interpreter schlägt den Datentyp in der Symboltabelle nach. Da die Variable einen String enthält, erzeugt der Interpreter Maschinencode, der den String in eine Zahl umwandelt, diese mit `30.48` multipliziert und das Ergebnis in der Variablen `laengeInZM` speichert. In der Symboltabelle trägt der Interpreter ein, dass die Variable `laengeInZM` jetzt einen Gleitkommawert enthält).

Wenn die Anweisung vollständig übersetzt und ausgeführt ist, fährt der Interpreter mit der nächsten Anweisung fort.

2.3 Drei Kernfragen moderner Programmierung

Obwohl unsere Programmiersprache *Bali*, syntaktisch durchaus schon Ähnlichkeit mit Java aufweist, erinnert ihre Konzeption – Interpreter als Übersetzer, Programmquelltexte aus reinen Anweisungen, Programmstart mit erster Anweisung in Quelltext, untypisierte Variablen, die nach Bedarf vom Interpreter angelegt werden – mehr an ein frühes Basic. Damit dies nicht so bleibt, erweitern wir *Bali* um

- Kontrollanweisungen
- typisierte Variablen
- Funktionen

2.3.1 Zweiter Entwurf: Mehr Kontrolle durch Kontrollanweisungen

Bisher wurden unser *Bali*-Programme streng sequentiell, d.h. Anweisung für Anweisung ausgeführt. Es gibt jedoch Situationen und Programmieraufgaben, die ohne die Möglichkeit, die strikt sequentielle Abfolge aufzureißen und umzubiegen, nicht oder nur sehr schlecht gelöst werden können. Wir führen daher in Bali folgende Konstrukte ein

- bedingte Ausführung,
- Verzweigung
- Schleife

Die bedingte Ausführung

Sie möchten ein Programm schreiben, das zu einer eingegeben Zahl die Fakultät[16] berechnet. Da die Fakultät sehr schnell ansteigt – 10! ist bereits gleich 3 628 800 – und nur für positive Zahlen definiert ist, müssen Sie einen Weg finden, wie Sie verhindern, dass das Programm nur Zahlen zwischen, sagen wir, 0 und 20 verarbeitet.

---

[16] Die Fakultät einer Zahl n ist das Produkt der ersten n natürlichen Zahlen. Für n = 0 ist die Fakultät als 1 definiert:

```
0! = 1
n! = n * (n-1) * (n-2) * ... * 1
```

Zuerst schreiben Sie eine Ausgabe, die den Benutzer des Programms anweist, nur Werte zwischen 0 und 20 einzugeben:

```
out "Geben Sie eine positive Zahl kleiner 20 ein: ";
```

So sinnvoll diese Maßnahme ist, hinreichend ist sie keineswegs. Ob aus Versehen, Schusseligkeit, Neugier oder Bösartigkeit – immer wieder kommt es vor, dass Benutzer falsche Eingaben an ein Programm schicken. Wenn Sie diese falschen Eingaben nicht abfangen, wird ihr Programm vermutlich abstürzen oder falsche Ergebnisse liefern.

Icon STOPP

Anders als Autoren von Programmierlehrbüchern sollten verantwortungsbewusste Programmierer Benutzereingaben stets auf Gültigkeit und korrekte Formatierung prüfen!

Wie können Sie verhindern, dass die Fakultät für Zahlen größer 20 berechnet wird?

Zuerst müssen Sie die Eingabe des Benutzers einlesen. Dann prüfen Sie, ob die eingegebene Zahl größer als 20 ist. Wenn dies der Fall ist (und nur dann), setzen Sie die Zahl zurück auf, sagen wir, 5. Anschließend wird die Fakultät der Zahl berechnet.

Der Ansatz klingt vernünftig, doch er ist in *Bali* nicht zu realisieren, denn es fehlen der Sprache sowohl die Möglichkeit, Werte zu vergleichen ("eingegebene Zahl größer 20"), als auch die Möglichkeit, eine Anweisung ("Zahl auf 5 setzen") nur dann auszuführen, wenn eine bestimmte Bedingung ("Zahl größer 20") eintritt.

Vergleiche

Um Vergleiche zu unterstützen, führen wir in Bali vier Vergleichsoperatoren ein:

```
OP == OP        // Gleichheit
OP != OP        // Ungleichheit
OP > OP         // größer gleich
OP < OP         // kleiner gleich
```

Icon STOPP

Beachten Sie, dass der Gleichheitsoperator aus zwei Gleichheitszeichen besteht, da das einfache Gleichheitszeichen ja schon die Zuweisung symbolisiert.

Die Vergleichsoperatoren sind für alle unsere Datentypen definiert, müssen jedoch vom Interpreter unterschiedlich, dem Datentyp entsprechend, übersetzt werden. Während Zahlen anhand ihres Betrags zu vergleichen sind, werden einzelne Zeichen und Strings lexikographisch, d.h. nach dem Alphabet, verglichen.

Ein neuer Datentyp

Wie alle Operationen liefern auch Vergleiche einen Ergebniswert. Das Ergebnis eines Vergleichs soll jedoch nur zwei Werte annehmen können: wahr oder falsch. Vergleiche sind aus Sicht des Programms also Aussagen.

Wenn Sie im Programm schreiben

```
zahl > 20
```

dann versteht der Interpreter dies als Aussage:

"Der Wert von `zahl` ist größer als 20"

Indem der Interpreter den aktuellen Wert von `zahl` mit dem Literal `20` vergleicht, stellt er fest, ob die Aussage wahr oder falsch ist.

Für die Wahrheitswerte von Aussagen führen wir einen neuen Datentyp in die Sprache ein: den Booleschen Datentyp[17]. Wie gesagt, gibt es zu diesem Datentyp nur zwei Werte, wahr und falsch, die wir im Programmcode durch die Schlüsselwörter `true` und `false` symbolisieren. (`true` und `false` sind also die Literale des Booleschen Datentyps.)

Die if-Bedingung

Um eine Anweisung in Abhängigkeit von einer Bedingung, sprich dem Wahrheitswert einer Aussage, auszuführen oder zu überspringen, führen wir eine neue Syntax ein:

```
if (Bedingung)
   Anweisung;
```

Das Schlüsselwort `if` leitet die Syntax ein. Dann folgt in runden Klammern die Aussage, die in der Regel aus einem Vergleich besteht. Darunter (oder dahinter) folgt die Anweisung, die von der `if`-Bedingung kontrolliert wird.

Zur Verdeutlichung ein kleines Programmfragment:

```
in zahl;

if (zahl > 20)
   out "Nur Eingaben kleiner 20";

out zahl * zahl;
```

Wenn der Benutzer eine Zahl kleiner oder gleich 20 eingibt, erhält er als Ausgabe das Quadrat der Zahl. Wenn er eine Zahl größer 20 eingibt, erscheint auf dem Bildschirm zuerst die Mahnung "Nur Eingaben kleiner 20" und dann das Quadrat der eingegebenen Zahl. "Hmm", sagt sich da der

---

[17] Benannt nach dem englischen Mathematiker George Boole, dem Begründer des algebraischen Kalküls in der Logik (Boolesche Algebra).

gewitzte Benutzer, "Wozu auf Zahlen kleiner 20 beschränken, wenn das Programm auch größere Zahlen verarbeitet."

Hier besteht Handlungsbedarf. Damit wir mit der `if`-Bedingung mehrere Anweisungen kontrollieren können, legen wir fest, dass Anweisungen, die zusammen in geschweiften Klammern `{}` stehen, als Gruppe oder **Anweisungsblock** gelten und überall stehen können, wo eine einzelne Anweisung steht.

Das verbesserte Programm setzt Eingaben größer 20 auf 5 zurück:

```
in zahl;

if (zahl > 20)
   {
   out "Nur Eingaben kleiner 20";
   zahl = 5;
   }

out zahl * zahl;
```

Die Verzweigung

Wir könnten das Programm zur Berechnung von Quadratzahlen auch so schreiben, dass das Quadrat nur für Eingaben kleiner oder gleich 20 berechnet wird und für Zahlen größer 20 die Fehlermeldung erscheint.

Am einfachsten lässt sich dies mit einer Verzweigung implementieren, d.h. das Programm soll in Abhängigkeit vom Wert einer Bedingung einen von mehreren Anweisungsblöcken ausführen.

Hierzu erweitern wir die Syntax der `if`-Bedingung um einen (optionalen) `else`-Zweig:

```
if (Bedingung)
   Anweisung A;
   else
   Anweisung B;
```

Diese Konstruktion kann man wie folgt lesen:

"Wenn die Bedingung erfüllt ist, dann führe die Anweisung A aus, überspringe den `else`-Zweig mit der Anweisung B und fahre mit der nächsten Anweisung hinter der `if`-`else`-Konstruktion fort. Wenn die Bedingung nicht erfüllt ist, überspringe die Anweisung A, führe den `else`-Teil mit der Anweisung B aus, und fahre mit der nächsten Anweisung hinter der `if`-`else`-Konstruktion fort. "

Selbstverständlich können auch hier anstelle der einfachen Anweisungen Anweisungsblöcke stehen.

Mit einer Verzweigung sähe das Quadratzahlenprogramm so aus:

```
in zahl;

if (zahl > 20)
   out "Nur Eingaben kleiner 20";
   else
   out zahl * zahl;
```

Die Schleife

Schleifen dienen dazu, eine Anweisung oder einen Anweisungsblock mehrfach hintereinander ausführen zu lassen. Die meisten Programmiersprachen kennen mehrere Syntaxformen für Schleifen, die nach den einleitenden Schlüsselwörtern als `for`-Schleife, `while`-Schleife usw. bezeichnet werden. In Bali unterstützen wir nur einen einzigen Schleifentyp, die `while`-Schleife, was nicht weiter schlimm ist, da alle anderen Schleifenformen auf die `while`-Schleife zurückgeführt werden können.

```
while (Bedingung)
  Anweisung
```

Diese Syntax besagt, dass die Anweisung (in der Praxis ist es meist ein ganzer Anweisungsblock) so lange wiederholt ausgeführt wird, wie die Bedingung erfüllt ist.

Typisch für Schleifen ist die Einrichtung einer **Schleifenvariable**, über die man kontrolliert, wie oft die Schleife ausgeführt wird. Dazu wird die Schleifenvariable vor Eintritt in die Schleife auf einen Anfangswert gesetzt, in der Schleife inkrementiert (um eins erhöht) oder in irgendeiner anderen Weise verändert und vor jedem neuen Schleifendurchgang getestet.

```
loop = 1;
while ( loop < 10 )
  {
  out loop * loop; // Berechnung u. Ausgabe der Quadrate
  loop = loop + 1; // Schleifenvariable erhoehen
  }
```

Die Schleifenvariable heißt in diesem Beispiel sinnigerweise `loop` (englisch für Schleife). In der Praxis verwendet man jedoch meist kürzere Namen wie `i`, `j` oder `n`. Die Schleife selbst dient dazu, die ersten neun Quadratzahlen zu berechnen und auszugeben.

Die Schleife wird so lange ausgeführt, wie der Wert der Variablen `loop` kleiner 10 ist. Da der Anfangswert der Variablen 1 ist und `loop` bei jedem Schleifendurchgang (**Iteration**) um 1 erhöht wird, wird `loop` nach genau neun Schleifendurchgängen den Wert 10 beinhalten. Die Schleifenbedingung ist dann nicht mehr erfüllt und die Schleife wird

beendet. Das Programm wird mit der nächsten Anweisung unter der Schleife fortgeführt.

Interessant ist, dass die Schleifenvariable hier nicht nur zur Kontrolle der Schleife, sondern auch zur Berechnung der Quadratzahlen verwendet wird:

```
out loop * loop;
```

Diese Doppelnutzung ist statthaft, oft sogar notwendig, doch müssen Sie aufpassen, dass es Ihnen nicht wie Truffaldino, dem "Diener zweier Herren" geht, und Sie die beiden Aufgaben nicht mehr auseinander halten können.

Wichtig ist:

- Wenn Sie die Schleifenvariable für Berechnungen in der Schleife verwenden, dann darf sich der Wert der Schleifenvariablen dadurch nicht ändern (Ausnahmen bestätigen die Regel). In obiger Schleife ist diese Forderung erfüllt, denn in `out loop * loop` wird `loop` zur Berechnung der Quadratzahl verwendet, der Wert von `loop` ändert sich jedoch nicht.
- Der Wert der Schleifenvariable muss sich so ändern, dass die Bedingung der Schleife irgendwann nicht mehr erfüllt ist. Ansonsten hätten Sie eine **Endlosschleife** programmiert, die ewiglich ausgeführt wird.

Die Erhöhung des Werts einer Variablen um 1 wird in der Programmierung als **Inkrement**, die Verminderung um 1 als **Dekrement** bezeichnet.

Icon INFO

Und das Programm zur Berechnung der Fakultät? Es könnte wie folgt aussehen:

```
in zahl;
if (zahl < 0)
   {
   out "Nur Eingaben zwischen 0 und 20";
   out "Berechnet wird 5!\n";
   zahl = 5;
   }
if (zahl > 20)
   {
   out "Nur Eingaben zwischen 0 und 20";
   out "Berechnet wird 5!\n";
   zahl = 5;
   }

fakultaet = 1;
loop = zahl;
while(loop > 1)
   {
```

```
    fakultaet = fakultaet * loop;
    loop = loop - 1;
    }

out "Fakultaet von " + zahl + ": " + fakultaet;
```

Nachdem die beiden `if`-Bedingungen sichergestellt haben, dass in `zahl` wirklich nur eine Zahl zwischen 0 und 20 steht, berechnet die anschließende `while`-Schleife die Fakultät. Die Schleifenvariable `loop` wird mit dem Wert von `zahl` initialisiert und in der Schleife dekrementiert, bis sie gleich 1 ist. In jeder Schleifeniteration wird der aktuelle Wert von `fakultaet` mit der Schleifenvariable multipliziert und das Ergebnis wieder in `fakultaet` gespeichert. Am klarsten wird die Arbeitsweise der Schleife, wenn man sich ansieht, wie sich der Wert der Schleifenvariable und der Variablen `fakultaet` von Iteration zu Iteration ändert.

Iteration	**fakultaet (nach Iteration)**	**loop (nach Iteration)**
0.	1	5
1.	5	4
2.	20	3
3.	60	2
4.	120	1

Tabelle 2.1: Schleifenausführung für `zahl` gleich 5

Für die Eingaben 0 und 1 wird die Schleife nicht ausgeführt. Da 0! und 1! aber beide gleich 1 sind und die Variable `fakultaet` vor der Schleife auf 1 gesetzt wird, liefert das Programm auf für diese Eingaben die korrekte Fakultät.

2.3.2 Zweiter Entwurf: Segen (und Fluch) der Datentypisierung

Mittlerweile ist *Bali* den Kinderschuhen entwachsen und es ist an der Zeit einen weiteren wichtigen Entwicklungssprung zu wagen: Bali soll zu einer kompilierten Sprache werden!

Für den Benutzer hat dies verschiedene Vorteile. Zum einem braucht er keinen Interpreter, um die Programme ausführen zu können. Zum anderen wird die Ausführungsgeschwindigkeit der Programme wesentlich beschleunigt, da die Programme als Maschinencode vorliegen (EXE-Dateien) und direkt ausgeführt werden können (und nicht erst nach und

nach in Maschinencode übersetzt und durch Vermittlung des Interpreters ausgeführt werden).

Für uns Entwickler hat die Umstellung ebenfalls Vorteile. Beispielsweise werden die Programme nicht mehr als für jeden lesbaren Quelltext an die Benutzer und Kunden ausgeliefert, sondern als binäre EXE-Datei. Der Quelltext und damit die in das Programm investierte Entwicklungsarbeit werden dadurch vor Nachahmern und illegaler Verwertung geschützt.

Die Umstellung bringt aber auch Nachteile mit sich. Da Maschinencode prozessorspezifisch ist, d.h. nur von einem bestimmten Prozessor (oder einer Prozessorfamilie) ausgeführt werden kann, sind die erstellten EXE-Dateien nicht portabel – können also nicht zwischen Rechnern mit inkompatiblen Prozessoren und Betriebssystemen ausgetauscht werden. (Wir werden uns dieser Problematik am Ende dieses Kapitels noch detaillierter zuwenden.)

Am unmittelbarsten betreffen uns aber die Konsequenzen, die das Sprachdesign betreffen.

Sehen Sie sich dazu folgenden Bali-Code an, wobei `n` eine beliebige natürliche Zahl enthält, deren Wert wir nicht kennen (sie könnte beispielsweise von Benutzer eingegeben worden sein):

```
eineVar = 3;

if (n > 10)
   eineVar = "Hallo";

zweiteVar = eineVar + n;
```

Welcher Wert wird hier in `zweiteVar` abgespeichert?

Nun, das hängt von dem Wert von `n` ab. Wenn `n` kleiner `10` ist, enthält `eineVar` die Zahl `3` und in `zweiteVar` wird das Ergebnis der mathematischen Addition von `eineVar` und `n` gespeichert. Ist `n` jedoch größer als 10, wird in `eineVar` der String `"Hallo"` gespeichert. Der `+`-Operator in der nachfolgenden Anweisung wird daher als Konkatenationsoperator gedeutet und in `zweiteVar` wird ein String gespeichert, der sich aus dem String von `eineVar` und dem in einen String umgewandelten Wert von `n` zusammensetzt.

Fazit: Welchem Datentyp der Wert von `eineVar` in dem Ausdruck `eineVar + n` angehört, kann erst zur **Laufzeit**, nach Ausführung der `if`-Bedingung, bestimmt werden. Von dem Datentyp des Werts hängt aber ab, ob das `+`-Zeichen für den mathematischen oder den String-Operator steht und davon hängt wiederum ab, welcher Wert in `zweiteVar` abgespeichert wird.

Ein Interpreter kann diesen Quelltext ohne Probleme übersetzen, da er den Code Schritt für Schritt übersetzt und ausführt. Wenn er zu der ominösen "Addition" kommt, muss er den Datentyp von `eineVar` nur in seiner Symboltabelle nachschlagen. Wurde die `if`-Bedingung erfüllt, ist dort als Datentyp String eingetragen, sonst Integer.

Ein Compiler muss aber den ganzen Quelltext vorab übersetzen. Zu diesem Zeitpunkt, der **Kompilierzeit**, steht der Wert von `n` aber nicht fest. Folglich kann der Compiler auch nicht entscheiden, welche Art von Wert in `eineVar` steht und welche Operation er für den `+`-Operator ausführen soll. Der Compiler kann diesen Quelltext nicht übersetzen!

Ein vernichtendes Urteil! Ist dies das Ende unserer Träume von einer kompilierten Bali-Sprache? Mitnichten.

Analysieren wir noch einmal, wo genau das Problem liegt: Operationen sind datentypspezifisch. Daher müssen zum Zeitpunkt der Übersetzung die Datentypen der Operanden feststehen. Für Literale ist dies unkritisch, da ihr Datentyp am Format abgelesen werden kann. Variablen können jedoch im Laufe eines Programms beliebige Werte verschiedener Datentypen annehmen. Die Datentypen ihrer Werte sind daher nur zur Laufzeit bestimmbar. Der Compiler ist aber darauf angewiesen, dass sämtliche Datentypinformationen bereits zur Kompilierzeit vorliegen.

Einführung typisierter Variablen

Das Problem liegt also letztlich darin, dass die Variablen im Laufe des Programms Werte unterschiedlicher Datentypen annehmen können.

Wie wäre es nun, wenn wir genau dies unterbinden, wenn wir festlegen, dass eine Variable einem festen Datentyp angehört und nur Werte dieses Datentyps aufnehmen kann? Dann könnte der Compiler den Datentyp der Variable zusammen mit Name und Speicheradresse in seiner internen Symboltabelle eintragen und dort bei Bedarf jederzeit nachschlagen. Sehr gut, das ist die Lösung. Überlegen wir uns also, wie wir das Konzept typisierter Variablen in Bali einführen.

Zuerst müssen wir klären, wie eine Variable einem Typ zugeordnet wird.

Variablen müssen deklariert werden

Bisher ergab sich der Datentyp der Variablen aus dem Datentyp der zugewiesenen Werte. Wann immer einer Variablen ein Wert zugewiesen wurde, trug der Interpreter den Datentyp des Werts als Datentyp der Variable in die Symboltabelle ein. Was liegt also näher als dieses Verfahren in abgewandelter Form zu übernehmen und zu statuieren, dass der Datentyp einer Variablen bei der ersten Zuweisung eines Werts festgelegt und danach nicht mehr geändert werden kann? Vorsicht! Mit

dieser Vorschrift könnten Sie typisierte Variabel in eine interpretierte Sprache einführen, nicht aber in eine kompilierte. Betrachten wir dazu eine leicht abgewandelte Form des letzten Beispiels:

```
if (n > 10)
   eineVar = "Hallo";

eineVar = 3;
zweiteVar = eineVar + n;
```

Angenommen die Variable `eineVar` taucht in diesen Zeilen zum ersten Mal im Programm auf. Welche Zeile enthält dann die erste Verwendung, die den Datentyp festlegen soll? Wieder hängt die Antwort vom Wert der Variablen `n` ab und kann vom Compiler nicht gegeben werden.

Wir brauchen eine klarere, eine eindeutigere Typfestlegung. Wir fordern daher, dass in Bali jede Variable vorab deklariert werden muss. Eine Deklaration soll aus dem Datentyp und dem Namen der Variablen bestehen. Für die Angabe des Datentyps richten wir folgende Schlüsselwörter ein:

- `int` für Integer-Werte
- `float` für Gleitkommawerte
- `char` für einzelne Zeichen
- `boolean` für Boolesche Werte
- `string` für Strings

Gültige Variablendeklarationen wären beispielsweise:

```
int zaehler;
float quotient;
string ausgabeText;
```

Fortan kann der Programmierer nur noch mit Variablen arbeiten, die definiert sind. Versucht er, neue Variablen wie bisher direkt zu verwenden, quittiert der Compiler dies bei der Übersetzung des Quelltextes mit einer Fehlermeldung.

```
zahl = 3;             // Fehler: zahl nicht definiert
quadr = zahl * zahl;  // Fehler: quadr nicht definiert
```

Korrekt wäre:

```
int zahl;
int quadrat;

zahl = 3;
quadr = zahl * zahl;
```

In gleicher Weise schmettert der Compiler Versuche ab, in einer Variablen einen Wert abzuspeichern, der nicht zum Datentyp der Variable passt:

```
int zahl;
zahl = 3.4;          /* Fehler: zahl ist vom Typ int,
                        zugewiesen wird aber ein
                        Gleitkommaliteral */
```

Korrigierte Version:

```
float zahl;
zahl = 3.4;
```

Positive Folgen der Typisierung

- Keine Fehler durch falsch geschriebene Namen

  Wenn Sie vergessen, eine Anweisung durch ein Semikolon abzuschließen, ist dies meist nicht weiter schlimm. Ein solcher Fehler wird vom Übersetzer (Interpreter wie Compiler) meist bemerkt und mit einer Fehlermeldung quittiert.

  Wenn Sie aber einen Variablennamen falsch schreiben, können Sie froh sein, wenn Sie mit einem Compiler arbeiten.

  ```
  in prozentsatz;
  out guthaben * prozenstatz;
  ```

  Ein Interpreter, der Variablen nach Bedarf erzeugt, wird `prozenstatz` für eine neue Variable halten, die er diensteifrig anlegt und deren Wert er mit `guthaben` multipliziert. Welchen Wert `prozenstatz` hat? Entweder weist der Interpreter der neu angelegten Variablen einen Standardwert zu, beispielsweise 0, oder er interpretiert einfach das zufällige Bitmuster, das er in der neu angelegten Variable vorfindet, als Wert. Auf jeden Fall wird der Wert nicht dem Wert in `prozentsatz` entsprechen und das Programm wird falsche Ergebnisse liefern.

  Ein Compiler wird dagegen sofort feststellen, dass der Name `prozenstatz` nicht deklariert ist und eine Fehlermeldung ausgeben.

- Einfachere und effizientere Speicherverwaltung

  Auch wenn wir in höheren Programmiersprachen mit Integern, Gleitkommazahlen, Strings und anderen Daten arbeiten, dürfen wir nicht vergessen, dass alle diese Daten vom Übersetzer in Bitfolgen codiert werden. Für jeden Datentyp gibt es dazu ein eigenes Codierungsverfahren: für Integer beispielsweise das 2n+1-Komplement, für Gleitkommazahlen die IEEE 754, für Zeichen den Unicode, Strings werden meist als Folgen von Zeichen codiert. Für die

elementaren Datentypen (Integer, Gleitkomma, Zeichen, Boolean, nicht aber String!) liefern diese Codierungsverfahren Bitfolgen fester Größe.

Steht der Datentyp einer Variable von vorneherein fest, kann der Übersetzer dem Datentyp entnehmen, wie viel Speicher er für die Variable reservieren muss. Dieses sehr effiziente und speicherschonende Verfahren ist typisch für Compiler.

Interpreter, die Datentypwechsel gestatten, müssen den benötigten Speicher hingegen von vorneherein so berechnen, das er für alle Datentypen ausreicht, oder dynamisch, d.h. zur Laufzeit, bei jedem Datentypwechsel neuen Speicher reservieren.

- Übersichtliche Zusammenfassung der Variablendeklarationen

  Variablendeklarationen müssen nicht an dem Ort stehen, wo die Variable erstmalig verwendet wird. Sie müssen der Verwendung lediglich vorangestellt sein. Der Programmierer kann dies dazu nutzen, wichtige Variablen am Anfangs des Programms zusammenzuziehen. Die Programme werden dadurch meist übersichtlicher und besser verständlich. Lediglich lokal benötigte Hilfsvariablen, beispielsweise Schleifenvariablen, werden vor Ort deklariert.

"Negative" Folgen der Typisierung

Wirklich negative Folgen hat die Typisierung nicht. Im Grunde genommen war Bali ja auch schon zuvor typisiert – ich denke da etwa an die datentypspezifische Formate der Literale oder die datentypspezifischen Operationen. Was das alte Bali von dem neuen Bali unterscheidet, ist daher weniger die Frage "Typisiert oder nicht" als vielmehr die Stärke der Typisierung.

Interpreter-Sprachen sind in der Regel eher schwach typisiert. Dies erleichtert die Programmierung und beugt unter Umständen unerwünschten Programmabstürzen vor. Dafür können sich schneller Fehler durch unachtsame Programmierung mit den Daten einschleichen.

Kompilierte Sprachen sind dagegen meist stark typisiert, d.h. alle Elemente der Sprache, die mit Daten zu tun haben, sind datentypspezifisch. Daten eines bestimmten Datentyps können nur mit den für diesen Typ definierten Operationen bearbeitet werden und dürfen nur an solchen Stellen im Programm auftauchen, an denen ihr Datentyp erlaubt ist.

```
int zahl;

in zahl;  // Fehler: Laut Bali-Spezifikation muss auf
          // das Schlüsselwort in eine String-Variable
```

```
        // folgen. Korrekt wäre als die Variablen-
        // deklaration "string zahl;"
```

Die strenge Abgrenzung der Datentypen, die vom Compiler überwacht wird, führt zu besseren, weil sichereren Programmen. Sie kann sich aber auch als extrem störend und nervtötend erweisen– dann nämlich, wenn der Programmierer darauf angewiesen ist, einen Wert von einem Datentyp in einen anderen umzuwandeln. Betrachten wir dazu folgendes Programm:

```
string zahl;
int quadrat;

in zahl;
quadrat = zahl * zahl;
out quadrat;
```

In der vorletzten Zeile versucht der Programmierer, zwei Strings (`zahl`) miteinander zu multiplizieren. Ein rotes Tuch für jeden anständigen Compiler. Und als wäre dies nicht Provokation genug, will der Programmierer dann noch in der Zeile darunter einen `int`-Wert über `out` ausgeben, obwohl `out` laut Bali-Spezifikation nur für Strings definiert ist. Der Compiler schüttelt den Kopf und weigert sich den Quelltext so zu übersetzen. Der Programmierer muss nachbessern.

Ursache für die ganze Ungemach sind die Standardeingabe und die Standardausgabe. Der Programmierer möchte das Quadrat einer eingegebenen Zahl berechnen, könnte sich also an sich ganz auf die Arbeit mit einem Zahlen-Datentyp konzentrieren, doch Standardein- und -ausgabe verarbeiten nur Strings und ziehen ihm einen Strich durch die Rechnung. So muss die als String eingelesene Eingabe (beispielsweise "12") vor der Berechnung des Quadrats in eine Zahl (12) umgewandelt und das Ergebnis (144) muss vor der Ausgabe als String ("144") aufbereitet werden. Kurz gesagt: Es müssen Wege gefunden werden, wie Werte unterschiedlicher Datentypen ineinander umgewandelt werden können.

Typumwandlungen

Schwach typisierte Programmiersprachen – hierzu gehören die meisten interpretierten Sprachen – zeichnen sich dadurch aus, dass sie viele Typumwandlungen automatisch vornehmen. In obigem Beispiel würde der Interpreter also die Stringeingabe von der Tastatur in die Variable `zahl` einlesen und, wenn der Wert von `zahl` in der nachfolgenden Anweisung mit sich selbst multipliziert werden soll, automatisch schließen, dass der Programmierer eine Typumwandlung in einen Zahlen-Datentyp wünscht.

Um keine Missverständnisse aufkommen zu lassen. Es geht hier nicht darum, den Typ der Variablen `zahl` zu ändern. Lediglich der Typ des

Icon STOPP

Wertes in `zahl` soll umgewandelt werden, d.h. der Übersetzer holt den String aus `zahl`, wandelt ihn in eine Zahl um, und führt mit dieser temporär erzeugten Zahl die gewünschte Multiplikation aus. Danach steht in `quadrat` die berechnete Quadratzahl und in `zahl` immer noch der eingelesene String!

Apropos Missverständnisse. Wenn hier die Rede davon ist, dass der Übersetzer die Werte von Variablen ausliest, Operationen ausführt oder sonstige Programmanweisungen ausführt, so ist dies natürlich nicht zu wörtlich zu nehmen. In Wirklichkeit erzeugt der Übersetzer lediglich die Maschinenbefehle, die all diese Aktionen ausführen.

Auch eine stark typisierte Sprache muss Typumwandlungen zulassen. Die Frage ist nur, welche Typumwandlung in welcher Form zugelassen wird. Drei Kategorien sind hier zu unterscheiden:

1. Typumwandlungen, die der Übersetzer automatisch vornimmt
2. Typumwandlungen, die der Übersetzer nach expliziter Aufforderung vornimmt
3. Typumwandlungen, die von der Sprache nicht unterstützt werden und daher vom Programmierer selbst programmiert werden müssen

Die letzte Form der Typumwandlung ist natürlich immer möglich. Sehr, sehr streng typisierte Sprachen würden nur diese Form der Typumwandlung zulassen, doch – glücklicherweise – ist nicht einmal die strengste typisierte Sprache so streng. Die meisten Sprachen, auch die stärker typisierten, bieten für häufig benötigte Typumwandlungen zwischen den Standarddatentypen (Integer, Gleitkomma, Zeichen ...) Umwandlungen der ersten oder zweiten Kategorie an. In die dritte Kategorie fallen damit Typumwandlungen, die der Übersetzer nicht leisten kann oder die vom Sprachdesigner als zu riskant oder zu exotisch betrachtet werden, um sie direkt zu unterstützen.

In die zweite Kategorie fallen zumeist Umwandlungen, die der Übersetzer vornehmen kann, die aber mit Risiken verbunden sind. Zu diesen Risiken gehören

- Informationsverlust (wird beispielsweise die Gleitkommazahl 3.2 in die Ganzzahl 3 umgewandelt, geht der Nachkommateil verloren) und
- Fehleranfälligkeit (Ob die Umwandlung eines Strings in eine Zahl fehlerfrei vonstatten geht, hängt vom Inhalt des Strings. Strings, deren Inhalt einem gültigen Zahlenliteral entspricht, können in der Regel fehlerfrei in Zahlen umgewandelt werden. Strings, die sonstige

Zeichenfolgen enthalten ("drei", "hallo") können dagegen nicht umgewandelt werden).

In solchen Fällen verlangt die Sprache, dass der Programmierer durch eine besondere Syntax anzeigt, dass er sich über die Risiken der Typumwandlung im Klaren ist und diese trotzdem vornehmen will. Unter Umständen muss er zudem Code aufsetzen, der etwaige Fehler, die durch die Typumwandlung ausgelöst werden, abfängt.

Die erste Kategorie ist Typumwandlungen vorbehalten, die der Übersetzer vornehmen kann, die keine Fehler produzieren und die in der Regel zu keinem Informationsverlust führen.

In Bali wollen wir es so halten, dass alle Umwandlungen zwischen den von uns eingeführten Typen in die zweite Kategorie fallen. Die Syntax für die Typumwandlung soll so aussehen, dass der Programmierer vor dem umzuwandelnden Wert in runden Klammern den gewünschten Zieltyp angibt:

```
(typ) wert
```

Lediglich die Umwandlung von einem beliebigen Typ in einen String soll automatisch durchgeführt werden.

Das Programm zur Berechnung der Quadratzahlen sieht dann so aus:

```
string zahl;
int quadrat;

in zahl;
quadrat = (int) zahl * (int) zahl;
out quadrat;
```

2.3.3 Zweiter Entwurf: Modularisierung durch Funktionen

Funktionen sind ein von vielen Programmiersprachen angebotenes Hilfsmittel zur Modularisierung des Programmcodes.

Warum empfiehlt es sich, Programmcode zu modularisieren? Erstens wird der Programmquelltext übersichtlicher. Wenn Sie umfangreicheren Code von einigen Dutzend Zeilen, Anweisung für Anweisung aufsetzen, werden Sie irgendwann große Schwierigkeiten haben zu verstehen, was in Ihrem Programm eigentlich vorgeht (noch schwieriger dürfte dies für andere Programmierer sein, die Ihr Programm später eventuell überarbeiten und warten müssen). Mit Hilfe von Funktionen können Sie größere Programme in Teilprobleme auflösen.

Nehmen wir an, Sie möchten mehrere Zahlenwerte einlesen und daraus den Mittelwert berechnen. Statt den Code direkt Anweisung für Anweisung herunterzuschreiben, können Sie den Code in die folgenden drei Teilprobleme auflösen:

```
Werte einlesen

Mittelwert berechnen

Ergebnis ausgeben
```

Nachdem Sie dies getan haben, schreiben Sie für jedes der drei Teilprobleme eine eigene Funktion, die das Teilproblem bearbeitet. Danach brauchen Sie nur noch nacheinander die drei Funktionen aufzurufen.

Der zweite Grund, der für die Auslagerung von Code in Funktionen spricht, ist, das man Funktionen sehr gut wiederverwerten kann. Eine einmal definierte Funktion kann man nämlich an jeder beliebigen Stelle aufrufen. Stellen Sie sich vor, Sie müssen in Ihrem Programm an verschiedenen Stellen eine recht komplizierte mathematische Formel berechnen. Ohne Funktionen müssten Sie an jeder Stelle, an der die Formel berechnet werden soll, die Anweisungen zur Berechnung der Formel neu aufsetzen. (Sie können den Code natürlich kopieren, doch wenn Sie später im Code einen Fehler bemerken, haben Sie immer noch das Problem, nachträglich sämtliche Stellen aufsuchen und den Fehler beheben zu müssen). Mit Funktionen können Sie eine Funktion zur Berechnung der Formel schreiben und brauchen diese dann nur noch an den betreffenden Stellen aufzurufen.

Eine Funktion ist im Grunde nichts anderes als ein Anweisungsblock, der mit einem Namen (dem Funktionsnamen) versehen ist, damit man ihn von beliebigen Stellen des Programms aus aufrufen kann. Ebenso wie Variablen fordern wir, dass in Bali Funktionen vor der Verwendung definiert werden müssen. Unser erster Syntaxentwurf sieht wie folgt aus

```
funktionsname()
  {
  Anweisung(en);
  }
```

Der Funktionsname ist frei wählbar. An dem abschließenden Klammernpaar kann der Übersetzer erkennen, dass es sich um die Definition einer Funktion und nicht einer Variablen handelt.

Weiter unten im Programm könnte diese Funktion dann wie folgt aufgerufen werden:

```
...
funktionsname();
...
```

Der Funktionsaufruf bewirkt, dass die Programmausführung in die Funktionsdefinition springt und die dort definierten Anweisungen abgearbeitet werden. Ist das Ende der Funktion erreicht, kehrt die Programmausführung wieder in die Zeile des Aufrufs zurück und der Code wird nach dem Aufruf fortgesetzt.

Icon NOTE

Vielleicht fragen Sie sich, wieso es nicht zum erneuten Funktionsaufruf kommt, wenn die Programmausführung nach Beendigung der Funktion in die Zeile ihres Aufrufs zurückkehrt. Die Antwort auf dieses scheinbare Paradoxon liegt in der Übersetzung des Quelltextes in Maschinencode. Wenn der Compiler die Anweisung mit dem Funktionsaufruf übersetzt, erzeugt er eine ganze Reihe von Maschinenbefehlen, von denen der eigentliche Sprung in die Funktion nur einer ist. Nach Beendigung der Funktion springt die Programmausführung zu dem Maschinenbefehl, der *unter* dem Aufruf folgt – ein wiederholter Aufruf wird somit vermieden. Auf der Quelltextebene gehört dieser Maschinenbefehl jedoch noch zur Anweisung mit dem Funktionsaufruf, so dass wir davon sprechen, dass die Programmausführung in die Zeile des Aufrufs zurückkehrt.

Datenfluss in und aus Funktionen

Funktionen wären nicht sehr hilfreich, wenn es nur darum ginge, einen ausgelagerten Anweisungsblock auszuführen. Darum gestatten Funktionen es auch, Werte vom Aufrufer (d.h. dem aufrufenden Code) entgegenzunehmen und Ergebnisse an den Aufrufer zurückzuliefern.

Betrachten wir einmal die folgende Funktion zur Berechnung der Fakultät.

```
fakultaet()
  {
  string zahl;
  int fak;
  int loop;

  in zahl;      // zahl, deren Fakultät berechnet wird
  fak = 1;
  loop = (int) zahl;
  while(loop > 1)
    {
    fak = fak * loop;
    loop = loop - 1;
    }
  out fak;      // Ausgabe des Ergebnisses
  }
```

Diese Funktion nimmt über die Tastatur eine Zahl entgegen, berechnet die Fakultät dieser Zahl und gibt das Ergebnis auf den Bildschirm aus. Wenn

wir also weiter oben statuierten, dass eine Funktion ein Teilproblem löst, so löst diese Funktion gleich drei Teilprobleme. Und genau dies steht einer allgemeinen Nutzung der Funktion entgegen. Viel besser wäre es, die Funktion würde sich auf ihre eigentliche Aufgabe, die Berechnung der Fakultät, konzentrieren. Die zu verarbeitenden Daten sollte sie vom Aufrufer entgegen nehmen und diesem sollte sich auch die Verarbeitung des Ergebnisses überlassen. Wie aber kann die Funktion Daten vom Aufrufer beziehen, wie ihm Daten zurückliefern?

Datenfluss in die Funktion

Um den Datenaustausch zwischen Funktion und aufrufendem Code zu ermöglichen, müssen wir die Syntax der Funktionsdefinition erweitern. Zuerst führen wir das Konzept der **Parameter** ein. Parameter sind Variablen, die in den runden Klammern hinter dem Funktionsnamen definiert werden.

```
funktionsname(int zahl)
  {
   ...
  }
```

Der Programmierer kann beliebig viele Parameter definieren. Mehrere Parameter werden durch Komma getrennt.

```
funktionsname(int param1, float param2)
  {
   ...
  }
```

Für die Funktion sind die Parameter ganz normale Variablen, jedoch mit einer Besonderheit: Wenn die Funktion aufgerufen wird, übergibt der Aufrufer jedem Parameter der Funktion einen Wert – beispielsweise ein Literal oder den Wert einer Variablen. So kann der Aufrufer der Funktion die Daten übergeben, die diese verarbeiten soll.

```
funktionsname(3, eineVar);            // Funktionsaufruf
```

Die Werte, die beim Funktionsaufruf an die Parameter übergeben werden, nennt man **Argumente**.

Icon NOTE

Datenfluss aus der Funktion

Damit die Funktion auch Daten zurückliefern kann, führen wir das Schlüsselwort `return` ein.

```
return wert;
```

Der aufrufende Code kann diesen **Rückgabewert** entgegen nehmen und weiterverarbeiten. Wie? Indem er ihn an eine Variable zuweist oder in einen Ausdruck einbaut. Der Funktionsaufruf repräsentiert dabei den Rückgabewert (ebenso wie eine Variable in einem Ausdruck den in ihr gespeicherten Wert repräsentiert):

```
ergebnis = funktionsname(3);

ergebnis = funktion1() * funktion(2);
```

Fehlt nur noch der Datentyp des Rückgabewerts. Dieser muss unbedingt in der Funktionsdefinition angegeben werden, damit der Compiler sicherstellen kann, dass der Rückgabewert vom aufrufenden Code korrekt, d.h. seinem Typ entsprechend, weiterverarbeitet wird.

Wir legen fest, dass der Typ des Rückgabewerts in der Funktionsdefinition vor dem Funktionsnamen anzugeben ist. Eine vollständige Funktionsdefinition sieht in Bali damit folgendermaßen aus:

```
Rückgabetyp Funktionsname(Parameterliste)
  {
  Anweisungen;
  }
```

Ergänzend gilt:

- Funktionen, die keine Parameter definieren, hängen an den Funktionsname einfach ein leeres Paar runder Klammern an:

  ```
  int eineFunk()
  ```

- Funktionen, die keinen Rückgabewert zurückliefern, geben als "Rückgabetyp" das Schlüsselwort `void` an:

  ```
  void andereFunk(int param)
  ```

Das folgende Codefragment definiert eine allgemein verwendbare Funktion zur Berechnung der Fakultät. Darunter sehen Sie Code, der die Funktion in einer Schleife aufruft, um die Fakultäten der Zahlen 0 bis 10 auszugeben:

```
int fakultaet(int zahl)
  {
  int fak;

  fak = 1;
  while(zahl > 1)
    {
    fak = fak * zahl;
    zahl = zahl - 1;
    }

  return fak;
  }
...
int n;
int f;

n = 0;
while (n < 11)
  {
  f = fakultaet(n);
```

```
out "Fakultät von " + n + ": " + f;
n = n + 1;
}
```

Wieder ist die Bali-Spezifikation um ein Element reicher geworden. Wer in Bali mit Funktionen arbeiten möchte, kann die Syntax für Definition und Aufruf in der Spezifikation nachlesen. Doch damit ist das Thema "Funktionen" nicht erschöpft. Wie zuvor die Einführung der typisierten Variablen zieht auch die Einführung der Funktionen eine Reihe von Fragen nach sich.

Programmausführung

Bisher bestand der Quelltext eines Bali-Programms aus einer Abfolge von untereinander geschriebenen Anweisungen, die von oben nach unten übersetzt und ausgeführt wurden.

Nun aber hat der Programmierer die Freiheit, Funktionen zu definieren, und dies zwingt uns sowohl über die Organisation des Quelltextes als auch die Abfolge, in der die Anweisungen ausgeführt werden, neu nachzudenken.

Wie sollen die Bali-Quelltexte jetzt organisiert werden? Zwei Möglichkeiten bieten sich an:

- A) Der Quelltext besteht aus einem Hauptanweisungsteil und einer Reihe beliebig vieler Funktionsdefinitionen. Wenn das Programm gestartet wird, werden die Anweisungen im Hauptanweisungsteil der Reihe nach ausgeführt. Steht in einer Anweisung ein Funktionsaufruf werden die Anweisungen der Funktion ausgeführt (wobei die Funktion wiederum andere Funktionen aufrufen kann), danach kehrt die Programmausführung zum aufrufenden Code zurück. Wenn die Anweisungen im Hauptanweisungsteil abgearbeitet sind, wird das Programm beendet. Dieses System verwendet zum Beispiel die Sprache Pascal.
- B) Der Hauptanweisungsteil wird ebenfalls in eine Funktionsdefinition eingefasst. Damit der Übersetzer diese spezielle Funktion, mit der die Programmausführung beginnen soll, erkennt, erhält sie einen speziellen, vorgegebenen Namen – beispielsweise `main()`. Dieses System verwendet zum Beispiel die Sprache C.

Wir folgen dem Beispiel von C und legen fest, dass Bali-Programme aus beliebig vielen Funktionsdefinitionen bestehen, von denen genau eine den Namen `main()` tragen muss.

```
/* Programm, das die Fakultät der Zahlen 0 bis 10
   berechnet
 */
```

```
// Funktion zur Berechnung der Fakultät
int fakultaet(int zahl)
  {
  int fak;

  fak = 1;
  while(zahl > 1)
    {
    fak = fak * zahl;
    zahl = zahl - 1;
    }

  return fak;
  }

// Hauptfunktion
void main()
  {
  int n;
  int f;

  n = 0;
  while (n < 11)
    {
    f = fakultaet(n);
    out "Fakultät von " + n + ": " + f;
    n = n + 1;
    }
  }
```

Gültigkeitsbereich und Lebensdauer von Variablen

Als unsere Quelltexte noch aus einer einzigen Folge von Anweisungen bestanden, war klar, dass Variablen nach ihrer Definition in allen nachfolgenden Anweisungen verwendet werden konnten. Jetzt aber sind die Anweisungen auf Funktionen verteilt und damit stellt sich die Frage: "Kann eine Variable, die in einer Funktion `funkA()` deklariert ist, auch in einer Funktion `funcB()` verwendet werden?"

Zugegeben, die Vorstellung frei und unbekümmert von jeder beliebigen Funktion aus auf die Variablen anderer Funktionen zugreifen zu können, hat etwas Bestechendes, doch wir sollten uns mit dieser Vorstellung gar nicht erst zu sehr anfreuen, denn die Folge wäre ein unkontrollierbares Chaos.

Rufen wir uns noch einmal ins Gedächtnis, wozu wir die Funktionen eingeführt haben. Wir wollten den Quellcode modularisieren. Nicht durch willkürliche Aufteilung, sondern indem wir dem Programmierer ein Mittel an die Hand gaben, wie er Teilaufgaben separat lösen kann. Dieses Mittel waren die Funktionen.

Wir können den Programmierern nicht vorschreiben, wozu und wie sie die Funktionen verwenden, aber im Idealfall sollte jede Funktion ein Softwarebaustein sein, der

- eine genau abgegrenzte Aufgabe löst
- selbst möglichst autark arbeitet und
- über eine klar definierte Schnittstelle zur Außenwelt (Aufrufer)

verfügt.

Wenn wir zulassen, dass eine Funktion auf Variablen einer anderen Funktion zugreifen, verschwimmt die Schnittstelle zur Außenwelt, die Funktionen verlieren ihre Autarkie und es kommt zu gegenseitigen Abhängigkeiten, die oft nur schwer zu durchschauen sind.

Wir legen daher klar und deutlich fest:

"Die in einer Funktion `f()` definierten Variablen (einschließlich der Parameter) sind nur in der Funktion selbst existent und verwendbar. Keine andere Funktion – auch nicht Funktionen, die von der Funktion `f()` aufgerufen werden – können auf diese Variablen zugreifen."

In der Programmierung spricht man in diesem Zusammenhang vom **Gültigkeitsbereich** (englisch "scope") einer Variablen.

Icon INFO

In Funktionen definierte Variablen werden üblicherweise als "**lokale Variablen**" bezeichnet und den "globalen Variablen" gegenübergestellt. Globale Variablen gibt es beispielsweise in C, C++, JavaScript und vielen anderen Sprachen. Sie werden außerhalb jeder Funktion definiert und können von allen Funktionen, die später definiert sind, verwendet werden. In Java gibt es keine globalen Variablen, weswegen wir in Bali ebenfalls darauf verzichten.

Die letzte Frage, die wir klären müssen, betrifft die Lebensdauer der Variablen. Bislang wurden die Variablen bei ihrer Deklaration erzeugt – d.h. mit Speicher verbunden – und erst am Programmende aufgelöst – d.h. der Speicher wurde wieder freigegeben. Doch soll dies auch jetzt noch gelten, da die Variablen alle in Funktionen definiert werden?

Wann werden die Variablen denn überhaupt gebraucht? Doch nur während der Zeit, da die Funktion ausgeführt wird. Es liegt also nahe, die Lebensdauer lokaler Variablen auf diese Zeit zu begrenzen. Jedes Mal, wenn eine Funktion aufgerufen wird, erzeugt der Compiler Maschinencode, der die lokalen Variablen der Funktion im Speicher anlegt, und immer wenn, die Funktion beendet wird, erzeugt der Compiler Maschinencode, der den belegten Speicher wieder freigibt.

Dieses Verfahren schont die Ressource Arbeitsspeicher, kostet aber (ein klein wenig) zusätzliche Ausführungszeit.

Funktionen und der Stack

Wenn Sie den Abschnitt zur "Programmausführung" aufmerksam gelesen haben, wird es Ihnen sicherlich nicht schwer fallen, herauszufinden, in welcher Reihenfolge die Funktionen und Anweisungen des folgenden Programms ausgeführt werden und wie die Ausgabe des Programms aussieht.

```
void funkB()
  {
  out "Beginn von funkB()";
  out "Ende von funkB()";
  }

void funkA()
  {
  out "Beginn von funkA()";
  funkB();
  out "Ende von funkA()";
  }

void main()
  {
  out "Beginn von main()";
  funkA();
  funkB();
  out "Ende von main()";
  }
```

Das Programm beginnt mit der `main()`-Funktion, führt die erste `out`-Anweisung aus und springt in die Funktion `funkA()`. Hier wird wiederum die erste `out`-Anweisung ausgeführt und danach in die Funktion `funkB()` gewechselt. `funkB()` wird vollständig ausgeführt. Danach kehrt das Programm in `funkA()` zurück, führt die zweite `out`-Anweisung aus und springt zurück zu `main()`. `main()` ruft jetzt direkt `funkB()` auf. Die Funktion und ihre beiden `out`-Anweisungen werden abgearbeitet, dann kehrt das Programm zum zweiten Mal zurück zu `main()`, um die letzte Ausgabe auf den Bildschirm zu schreiben und das Programm zu beenden.

Ausgabe

```
Beginn von main()
Beginn von funkA()
Beginn von funkB()
Ende von funkB()
Ende von funkA()
Beginn von funkB()
```

```
Ende von funkB()
Ende von main()
```

Haben Sie die richtige Ausgabe herausgefunden? Wie? Die Aufgabe war zu leicht? Dann schauen Sie sich das Programm bitte noch einmal an und legen Sie ihr besonderes Augenmerk auf die Funktion `funkB()`. Woher weiß die Funktion, wenn sie ausgeführt wird, in welche Anweisung sie nach Beendung zurückspringen soll?

Zermartern Sie sich nicht den Kopf. Die Aufgabe ist wirklich knifflig und die Lösung ohne spezielle Kenntnisse in Assembler und auf dem Gebiet der Speicherverwaltung nicht zu finden. Trotzdem werden wir dem Problem nachgehen, denn es führt uns zu einem ebenso wichtigen wie lehrreichen Modell zur Datenverwaltung: dem Stack.

Wenn der Compiler den Quelltext der Funktion `funkB()` übersetzt, erzeugt er nicht nur den Maschinencode für die beiden `out`-Anweisungen, sondern hängt auch noch Code zur Beendigung der Funktion an. Wie dieser Code im Einzelnen aussieht, soll uns nicht weiter interessieren. Wichtig ist, dass der Code einen Befehl enthält, der dafür sorgt, dass das Programm an der Stelle fortgesetzt wird, an der die Funktion aufgerufen wurde.

Nun kann aber die Funktion von verschiedenen Stellen des Programms aus aufgerufen werden. (Die Funktion `funkB()` wird z.B. sowohl aus `funkA()` als auch aus `main()` aufgerufen.) Da es folglich nicht möglich ist, das Ziel für den Rücksprung direkt anzugeben, verfällt der Compiler auf einen Trick. Er nennt der Funktion einen bestimmten Speicherbereich (eine Art Postfach) und sagt, "Wenn es soweit ist, dass Du beendet wirst, dann schau in diesen Speicherbereich. Die Adresse, die Du darin findest, ist die Adresse des Maschinenbefehls, mit dem das Programm fortgesetzt werden soll."

Jetzt muss der Compiler nur noch dafür sorgen, dass die Funktion, wann immer sie ausgeführt wird, stets die richtige Rücksprungadresse im Postfach findet. Da sich die Rücksprungadresse nach dem Ort des Aufrufs richtet, bietet es sich an, die Übersetzung des Funktionsaufrufs mit der Hinterlegung der Rücksprungsadresse zu verbinden. Und genau dies tut der Compiler.

Soweit die grundsätzliche Vorgehensweise, nun zu den tatsächlichen Abläufen.

Der Funktionsaufruf im Detail

Für jeden Funktionsaufruf erzeugt der Compiler Maschinencode, der

- die Programmausführung vom Funktionsaufruf zur ersten Anweisung der aufgerufenen Funktion umlenkt,

- Speicher für die Parameter der Funktion reserviert und die Werte der Argumente in die Parameter kopiert,
- Speicher für die lokalen Variablen bereitstellt,
- und die Rücksprungadresse hinterlegt.

Eine ganze Menge Arbeit, und wenn die Funktion beendet wird, muss der Compiler alles wieder rückgängig machen. Zudem muss er den Überblick behalten, welche Funktionen noch darauf warten, beendet zu werden, welche Funktion gerade ausgeführt wird und welche Variablen zu welchem Funktionsaufruf gehören, damit nicht Funktion A unerlaubterweise auf die Variablen von Funktion B zugreift oder der zweite Aufruf von Funktion C die Variablen für den ersten Aufruf von C überschreibt.

Man könnte vermuten, dass der Verwaltungsaufwand hierfür enorm sein muss, doch weit gefehlt: Dank des genialen Rückgriffs auf eine in der Programmierung weit verbreiteten Datenstruktur, den **Stack**, kann der Compiler ohne große Mühe sicherstellen, dass die Funktionsaufrufe in der korrekten Reihenfolge abgearbeitet werden und jeder Funktionsaufruf mit seinem ganz privaten Satz von lokalen Variablen arbeitet.

Das Stack-Modell

Das Stack-Modell gehört zu einer Gruppe von Modellen, die beschreiben, wie Daten auf einer höheren Ebene verwaltet werden können.

Der Begriff Stack stammt aus dem Englischen und heißt übersetzt Stapel – womit der besondere Charakter dieses Modells schon angedeutet wird. Wie für einen Stapel Teller gilt: Neue Elemente (Teller, Daten, was auch immer) können nur auf dem oberen Ende des Stapels abgelegt werden, und heruntergenommen werden die Elemente auch nur von oben, also in der umgekehrten Reihenfolge, in der sie auf dem Stapel abgelegt wurden. Man bezeichnet dies auch als das LIFO-Prinzip. LIFO ist ein Akronym für Last-In-First-Out, d.h. das zuletzt abgelegte Elemente wird als erstes wieder heruntergenommen.

(Das Pendant zum Stack ist die Queue, oder Schlange, die nach dem FIFO-Prinzip arbeitet: Das Element, das zuerst in die Schlange geht, kommt als erstes wieder heraus.)

Die Idee ist, dass der Compiler einen bestimmten Teil des Arbeitsspeichers als Stack-Speicher nutzt und in diesem Speicher die Informationen für die Funktionsaufrufe stapelt.

Für jeden Funktionsaufruf reserviert der Compiler im Stack einen eigenen Speicherblock, einen so genannten **Stackrahmen**. In diesem Block legt der Compiler sämtliche Informationen ab, die mit dem Funktionsaufruf verbunden sind (Parameter, Variablen, Rücksprungadresse etc.).

Solange eine Funktion abgearbeitet wird, bleibt ihr Stackrahmen auf dem Stack liegen. Wird aus einer Funktion heraus eine weitere Funktionen aufgerufen, wird deren Stackrahmen auf den Stackrahmen der aufrufenden Funktion draufgepackt. Wird eine Funktion beendet, wird ihr Stackrahmen vom Stack entfernt.

Ganz oben auf dem Stack liegt also immer der Stackrahmen der aktuell ausgeführten Funktion. Darunter liegen die Stackrahmen der Funktionen, die zuvor aufgerufen und noch nicht beendet wurden – in der umgekehrten Reihenfolge ihres Aufrufs.

Icon INFO

Jeder Funktionsaufruf ist mit der Einrichtung und späteren Auflösung eines Stackrahmens verbunden. Aus diesem Grunde kostet die Ausführung einer Funktion stets mehr Laufzeit und Speicher, als die alleinige Ausführung der Anweisungen in der Funktion kosten würde. Die Mehrkosten für den Funktionsaufruf bezeichnet man als **Function Overhead**.

Wie kann man einen Funktionsaufruf vom Stack entfernen?

Der Rechner unterstützt das Stack-Konzept durch zwei Register (meist BP und SP genannt). Das Register BP enthält die Anfangsadresse des aktuellen Stackrahmens, das Register SP die aktuelle Spitze des Stacks, sprich das Ende des aktuellen Stackrahmens.

Beim Aufruf einer neuen Funktion wird zuerst der Inhalt von BP, also die Anfangsadresse des noch aktuellen Stackrahmens auf den Stack geschrieben. Dann wird der Inhalt von SS in BP kopiert – die Stackspitze wird zum Anfang des neuen Stackrahmens. Während die lokalen Variablen der aufgerufenen Funktion im neuen Stackrahmen angelegt werden, wächst dieser nach unten (der Stack wächst immer von oben nach unten). Wurde die Funktion beendet, wird ihr Stackrahmen, d.h. der Bereich zwischen BP und SS aufgelöst. Dazu wird der Inhalt von BP einfach nach SS kopiert (Stackspitze wird zurückgesetzt) und die Anfangsadresse des vorangehenden Stackrahmens vom Stack nach BP geschrieben.

Da lokale Variablen in dem Stackrahmen ihrer Funktion abgelegt werden, wird ihr Speicherbereich bei Verlassen der Funktion zusammen mit dem Stackrahmen freigegeben. Aus diesem Grunde sind lokale Variablen nur während der Ausführung ihrer Funktion gültig.

Neben dem Stack gibt es noch zwei weitere Speicherbereiche, die für Programme reserviert werden: der Code-Speicher und der Heap.

Icon INFO

Im Code-Speicher werden die Maschinenbefehle des Programms abgelegt. (Falls Sie sich weiter oben gefragt haben, was genau die Rücksprung*adresse* ist, es ist die Adresse der Speicherzellen, in der der betreffende Maschinenbefehl steht.)

Auf dem Heap werden Daten abgespeichert, die – aus verschiedenen Gründen – nicht im Stack abgelegt werden sollen. In Bali gibt es derzeit allerdings nur lokale Variablen, so dass wir den Heap gar nicht nutzen. Im Abschnitt "Objektorientierung" wird sich dies ändern.

Wir sind in diesem Abschnitt weit in die Theorie der Funktionsaufrufe eingedrungen. Wenn Sie dabei nicht allen Ausführungen folgen konnten, so ist dies kein Grund sich zu grämen. Wichtig ist, dass Sie verstanden haben, welche Vorteile die Modularisierung durch Funktionen bringt und wie die Programmierung mit Funktionen in der Praxis aussieht (Definition, Aufruf, Datenaustausch). Die zugehörigen Hintergrundinformationen können Sie später noch einmal nachlesen, wenn Sie selbst mit Funktionen programmieren.

2.3.4 Geschichte: Von der imperativen zur strukturierten Programmierung

Die erste erfolgreiche höhere Programmiersprache war FORTRAN Auf FORTRAN folgten alsbald eine Reihe weiterer Programmiersprachen: COBOL (1960), ALGOL (1958-60), BASIC (1960) und andere. Obwohl diese Sprachen keine direkten Abkömmlinge von FORTRAN waren, übernahmen sie die von FORTRAN geprägte, sehr mathematische Vorstellung davon, wie ein Programm funktioniert: Auf der einen Seite gibt es Daten (Zahlen oder Text), auf der anderen Seite Operationen, die auf diese Daten angewendet werden. Programme bestehen aus einer Reihe von Anweisungen, in denen Daten durch Operatoren verändert werden. Die Anweisungen werden grundsätzlich nacheinander ausgeführt, der Programmablauf kann aber durch Sprünge, Verzweigungen und Schleifen gesteuert werden. Man bezeichnet dies als das Paradigma der *imperativen* Programmierung.

Neben der imperativen Programmierung gibt es noch drei weitere Programmiermodelle: die funktionale, die logische und die objektorientierte Programmierung.

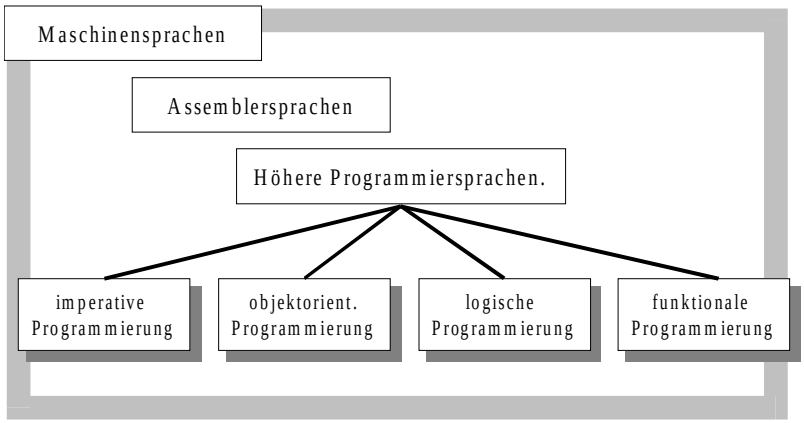

Abbildung 2.1: Paradigmen der höheren Programmiersprachen

Die Paradigmen der *funktionalen* und der *logischen* Programmierung entstammen dem Bereich der künstlichen Intelligenz. LISP (1958) ist eine funktionale, PROLOG (1972) eine logische Sprache.

Das Paradigma der *objektorientierten* Programmierung stellt nicht wie die imperative Programmierung die Daten, sondern das Konzept der Objekte in den Mittelpunkt. Die ersten objektorientierten Programmiersprachen, Simula und Smalltalk, wurden Ende der Sechziger, Anfang der Siebziger entwickelt und führten neben den imperativen Programmiersprachen ein eher bescheidenes Schattendasein. Wer hätte damals vermutet, dass ihnen einmal die Zukunft gehören würde?

Das Zeitalter der strukturierte Programmierung

Dass die ersten objektorientierten Sprachen kein großer Erfolg waren, hatte sicherlich mehrere Gründe. Im Vergleich zu den imperativen Sprachen wiesen die objektorientierten eine relativ flache Lernkurve auf, und die Programme waren im Umfang größer, in der Ausführungsgeschwindigkeit langsamer – was damals natürlich weitaus schwerer wog als heute, da die Prozessorleistung in Gigahertz und die Speicherkapazität in Gigabyte gemessen wird. Zudem waren FORTRAN und COBOL schon länger auf dem Markt und einfach bekannter.

Doch nicht alle waren mit den bestehenden imperativen Sprachen zufrieden. Ihr größtes Manko war das Fehlen von Unterroutinen (Funktionen), ohne die es nicht möglich war, Teilaufgaben modular zu lösen. War in einem Programm die gleiche Aufgabe, wie z.B. die Berechnung eines Mittelwerts, an verschiedenen Stellen zu lösen, musste der Programmierer den Code kopieren. Das Programm wurde dadurch

länger, unleserlicher und schwieriger zu debuggen (Fehler im kopierten Code mussten in allen Kopien korrigiert werden).

ALGOL (1959) war die erste Sprache, die dem Bedürfnis nach mehr Modularität und Strukturiertheit Rechnung trug und das Konzept der Unterroutinen einführte. Teilprobleme konnten fortan in Form von Unterroutinen gelöst werden. Sammlungen von Unterroutinen für immer wiederkehrende Aufgaben konnten als "Bibliotheken" weitergegeben und wiederverwertet werden. Geringfügige Laufzeitverluste, die mit dem Aufruf der Unterroutinen verbunden waren (Function Overhead), wurden angesichts dieser Vorteile gern in Kauf genommen.

Die Strukturierung des Codes durch Unterroutinen und Anweisungsblöcke wurde nach und nach von allen imperativen Programmiersprachen übernommen – aus den imperativen wurden die *strukturierten* oder *prozeduralen* Programmiersprachen, deren Primus die 1972 entwickelte Sprache C wurde.

Der Siegeszug von C

Trotz der schrittweisen Verbesserungen blieben die alt eingesessenen imperativen Programmiersprachen jedoch für bestimmte Aufgaben, beispielsweise die Entwicklung von Betriebssystemen oder Compilern, ungeeignet. So wurde das Betriebssystem UNIX für den DEC-PDP-7-Computer noch 1970 in Assembler programmiert. Die Weiterentwicklung und Portierung des Betriebssystems war jedoch so mühselig, dass Ken Thompson und Dennis Ritchie eine eigene, besonders schnelle und maschinennahe, höhere Programmiersprache entwickelten: C.

C war nie für die Software-Entwicklung gedacht. Es war auf die PDP-11-Architektur ausgerichtet und sollte mehr den Compiler-Herstellern das Leben erleichtern als den Software-Entwicklern. Die Maschinennähe, die Mächtigkeit, die gute Portabilität, die Schnelligkeit der erzeugten Programme und schließlich die Verbreitung in Zusammenhang mit UNIX machten C jedoch bald zu einer der erfolgreichsten Programmiersprachen überhaupt. Ein aus heutiger Sicht kaum nachvollziehbarer Siegeszug begann, der Ende der Siebziger durch die Markteinführung des "Personal Computers" noch verstärkt wurde.

Der PC revolutionierte die Software-Entwicklung. Privatleute und mittelständische Unternehmen waren nicht an mathematischen Kalkulationen und wissenschaftlichen Simulationen interessiert. Gefragt waren vielmehr Programme zur Textverarbeitung, Tabellenkalkulation, Datenbanken und natürlich... Spiele.

Die ersten PCs, ob nun Apple, IBM-PC, Atari oder Commodore, waren eher bescheiden ausgestattet. Der erste IBM-PC verfügte gerade einmal über 64 KByte Arbeitsspeicher[18] und war mit 2 MHz getaktet. Arbeitsspeicher, Festplattenspeicher und Prozessorzeit waren damals kostbare Güter, mit denen jeder Programmierer sparsam und verantwortlich umgehen musste. Daran änderte auch die zunehmende Leistungsfähigkeit der PCs nicht, denn freiwerdende Kapazitäten wurden sofort durch immer komplexere Software aufgebraucht. So bescherte der PC vor allem zwei Kategorien von Sprachen einen Boom: einfachen Lehrsprachen für angehende Programmierer (Basic und Pascal) und leistungsfähige Sprachen, mit denen man schnelle, schlanke Programme schreiben konnte (C und Assembler). Die objektorientierten Sprachen galten dagegen weiterhin als zu kompliziert und zu langsam. Doch die Zeit sollte für sie arbeiten.

2.4 Objektorientierung

"Programme verarbeiten Daten". Nirgends schlägt sich dieser Satz deutlicher nieder als im Paradigma der imperativen Programmierung. Wer eine imperative Programmiersprache erlernt, der hört zuerst von den verschiedenen Daten, die es gibt, und dann von den Operationen und Funktionen, mit denen er diese Daten verarbeiten kann. Auf der einen Seite die Daten, auf der anderen Seiten die Operationen – nichts ist klarer, nichts einfacher und nichts verdirbt den angehenden Programmierer schneller für andere Paradigmen wie z.B. die objektorientierte Programmierung. Sind wir schon verdorben?

Wer objektorientiert programmieren möchte, der sollte auch lernen, objektorientiert zu denken. Objektorientiert zu denken, bedeutet aber, sich von Begriffen wie Daten und Operationen auf Daten zu lösen und Programme stattdessen als Sammlungen von Objekten zu begreifen, die in unserem Auftrag Probleme lösen und Aufgaben erledigen. Also eine vollkommene Abkehr von der imperativen Programmierung? Mitnichten, denn letztlich verarbeiten auch objektorientierte Programme Daten und so wundert es nicht, dass die meisten objektorientierten Sprachen irgendwo

[18] Wenn Sie ein neues Word-Dokument anlegen und ohne Text einzugeben direkt abspeichern, belegt allein dieses "leere" Dokument gut ein Drittel dieses Werts. (Je nach verwendeter Dokumentvorlage können sogar Werte erreicht werden, die den Arbeitsspeicher des ersten IBM-PC gesprengt hätten... und dabei müsste der Arbeitsspeicher ja noch Betriebssystem und Textprogramm aufnehmen!)

eine Brücke zwischen den Objekten auf der einen Seite und den Daten/Operationen auf der anderen Seite schlagen. Meist sieht dies so aus, dass die Objekte[19] mit den typischen Elementen der imperativen Programmierung implementiert werden.

Fazit: Grundkenntnisse in der imperativen Programmierung sind auch für die objektorientierte Programmierung unerlässlich. Ein guter objektorientierter Programmierer muss jedoch verstehen, dass er seine Programme auf zwei Ebenen schreibt: einer höheren Ebene, auf der er mit Objekten programmiert, und einer niedrigeren, auf der er mit imperativen Elementen die von ihm benötigten Objekte (Klassen) definiert.

Wenden wir uns zuerst der niederen Ebene zu.

2.4.1 Eigene Datentypen

Bisher gibt es in Bali fünf Datentypen, mit denen der Programmierer arbeiten kann: zwei Zahlenformate (`int` und `float`), einen Zeichentyp (`char`), einen Booleschen Typ (`boolean`) und einen String-Typ (`string`). Was aber, wenn der Programmierer andere Daten verarbeiten möchte – beispielsweise Adressen?

Sicherlich, wir könnten für diese Fälle noch einen Datentyp `address` in die Spezifikation aufnehmen, aber dann wendet sich ein anderer Programmierer an uns und hätte noch gerne einen Datentyp für komplexe Zahlen und ein dritter würde gerne mit Vektoren bearbeiten... Sehen wir es ein: Wir können unmöglich alle denkbaren Datentypen direkt in die Spezifikation aufnehmen. Trotzdem müssen wir dem Programmierer Mittel an die Hand geben, wie er mit jeder beliebigen Art von Daten arbeiten kann.

Die Lösung dieses Dilemmas liegt darin, dem Programmierer die Definition eigener Datentypen zu gestatten.

Wie könnte eine Datentypdefinition aussehen? Mit Definitionen der Art "`datentyp Adresse;`" ist es sicherlich nicht getan. Schließlich sollen später ja auch Variablen dieses Datentyps angelegt werden ("`Adresse adrPeter;`") und dann muss der Compiler wissen, wie viel Speicher er für diese Variable reservieren muss, wie er die Werte des Datentyps in Bitfolgen codieren kann und welche Operationen er für Variablen dieses Typs zulassen darf. All diese Informationen in ein festes, für den Compiler

[19] Eigentlich müsste es heißen "die Klassen der Objekte", doch dazu in Kürze mehr.

verständliches Format zu kleiden, dürfte allerdings unmöglich zu sein – es sei denn, man führt die Datentypdefinition auf bekannte Elemente zurück. Diese "bekannten Elemente" sind die elementaren Datentypen.

Wir legen fest: "In Bali können neue Datentypen als Kombination von Variablen bekannter Datentypen erzeugt werden". Bekannte Datentypen sind die elementaren Datentypen und alle Datentypen, die der Programmierer bereits zuvor nach diesen Regeln definiert hat. Die zugehörige Syntax soll von dem Schlüsselwort `type` eingeleitet werden. Danach folgen der frei wählbare Typname und in geschweiften Klammern die einzelnen Variablen.

```
type Adresse
  {
  string vorname;
  string name;
  string strasse;
  int hausnummer;
  int plz;
  string stadt;
  }
```

Ist der neue Typ definiert, kann der Programmierer Variablen dieses Typs erzeugen:

```
int eineFunktion()
  {
  Adresse kunde1;
  Adresse kunde2;
  ...
```

Eine neue Terminologie

Plötzlich haben wir es mit zwei Arten von Variablen zu tun: lokalen Variablen und Variablen, die Teil eines selbst definierten, komplexen Typs sind. Um beide begrifflich besser auseinander halten zu können, bezeichnen wir die Variablen aus der Typdefinition als **Felder**. Die Variable `kunde1` aus obigem Codefragment verfügt demnach über sechs Felder (`vorname`, `name`...).

Programmieren mit komplexen Datentypen, Teil 1

Für den Zugriff auf die Felder führen wir eine neue Syntax ein:

```
varname.feldname
```

Zuerst wird die (komplexe) Variable angegeben, dann wird mit einem Punkt das Feld angehängt, auf das zugegriffen werden soll.

Jetzt können wir den Namen von `kunde1` festsetzen oder die Stadt von `kunde1` in `kunde2` kopieren:

```
kunde1.name = "Meier";
kunde2.stadt = kunde1.stadt;
```

Beachten Sie, dass der Compiler dieser Operationen nicht nur übersetzt, sondern auch auf Korrektheit und Typenverträglichkeit hin überprüft. Der Variablendeklaration entnimmt er, dass `kunde1` vom Typ `Adresse` ist. Er schlägt daher in der Typdefinition von `Adresse` nach und kontrolliert, ob Variablen vom Typ `Adresse` über ein Feld `name` verfügen und ob dieses vom Typ `string` ist.

Nachdem wir nun spezifiziert haben, wie der Programmierer auf die Felder einer komplexen Variable zugreifen kann, müssen wir uns noch einmal den komplexen Variablen selbst zuwenden. Welche Operationen sollen für sie definiert sein? Kann man sie einander zuweisen? Kann man sie vergleichen? Sollen wir vielleicht sogar die arithmetischen Operatoren für sie definieren?

Wert und Referenztypen

Die Zuweisung sollten, nein müssen, wir auch für komplexe Variablen unterstützen. Wieso?

Erstens wird der Programmierer zweifelsohne auch einmal komplexe Variablen komplett zuweisen wollen:

```
tmpKunde = kunde1;
```

Zweitens gibt es in den Programmen versteckte, unumgängliche Zuweisungen. Wo diese Zuweisungen versteckt sind? In den Funktionsaufrufen.

Angenommen eine Funktion `demo()` übernimmt als Argument eine `Adresse`-Variable, bearbeitet sie in irgendeiner Weise und liefert sie dann als Ergebniswert zurück:

```
Adresse kunde;
Adresse erg;
...
erg = demo(kunde);
```

Hier finden in der letzten Zeile zwei versteckte Zuweisungen von Variablen des Typs `Adresse` statt. Beim Aufruf der Funktion `demo()` übergibt der Aufrufer die Variable `kunde`. Die Funktion besitzt demnach einen passenden Parameter gleichen Typs. Wie der Parameter heißt, ist irrelevant. Wichtig ist, dass der Compiler im Stackrahmen der Funktion Speicher für den Parameter reserviert und dann den Wert des Arguments (`kunde`) in den Speicher des Parameters kopiert. Mit anderen Worten: der Wert des Arguments wird dem Parameter zugewiesen.

Die zweite Zuweisung tritt zumindest im Quelltext offen zu Tage: es ist die Abspeicherung des Rückgabewerts der Funktion. Nicht ganz so offensichtlich sind dagegen die Konsequenzen, die sich ergäben, wenn komplexe Variablen nicht zugewiesen werden könnten. Dann könnte diese Variablen nicht als Rückgabewerte von Funktionen dienen, es könnten keine Parameter komplexer Typen definiert werden. Letztendlich wäre das ganze Konzept der Funktionen in Frage gestellt.

Es kann also nicht darum gehen zu entscheiden, ob Zuweisungen komplexer Variablen erlaubt werden, sondern nur noch darum, was diese Zuweisungen bedeuten.

```
kunde1 = kunde2;
```

Eine Zuweisung bedeutet immer, dass der Wert des Ausdrucks auf der rechten Seite (hier der Wert der Variablen `kunde2`) in den links angegebenen Speicherbereich (hier der Speicher der Variablen `kunde1`) kopiert wird. Was jedoch ist der Wert einer komplexen Variable? Natürlich das, was in ihrem Speicherbereich abgelegt ist, und im Falle komplexer Variablen sind dies die Werte der einzelnen Felder. Bei der obigen Zuweisung werden also die Feldwerte von `kunde2` in die korrespondierenden Felder von `kunde1` kopiert. Halt, halt, HAAALT!

So geht es nicht! Bedenken Sie doch, welche Konsequenz dies für die Funktionsaufrufe hätte! Jeder Übergabe eines Arguments ist gleichbedeutend mit der Zuweisung an den entsprechenden Parameter. Das kostet Zeit und Speicher, selbst wenn es sich um Daten elementarer Datentypen handelt. Daten komplexer Datentypen sind aber meist viel umfangreicher, d.h. die Kosten für das Kopieren der Werte von den Argumenten in die Parameter können um ein Vielfaches höher liegen!

Wenn wir nicht möchten, das Bali-Programme, die viel mit Funktionen arbeiten (was ja nur wünschenswert ist), in der Ausführung unerträglich langsam werden, müssen wir einen anderen Weg finden.

Referenzen

Wo genau liegt das Problem? In den Variablen? Nein. In dem Umfang der Daten komplexer Datentypen? Auch nicht. Das Problem liegt darin, dass wir die Daten direkt in dem Speicherbereich der Variablen ablegen! Nun kann man uns dafür keinen Vorwurf machen, denn so sind wir es ja von den elementaren Datentypen gewohnt. Doch was für die elementaren Datentypen gut ist, ist für die komplexen Datentypen beileibe nicht billig.

Das Zauberwort, das uns hier Lösung verspricht, lautet: **Referenz**. Statt die Daten direkt in der Variablen zu speichern, legen wir sie irgendwo im Speicher ab und verwahren in der Variablen lediglich eine Referenz (einen Verweis) auf den Speicherbereich mit den Daten.

Wie sieht dies im Detail aus?

Wenn Sie eine lokale Variable von einem komplexen Datentyp deklarieren, reserviert der Compiler zwei Speicherbereiche: Im Stack reserviert er wie gehabt Speicher für die Variable. Da in der Variablen statt der Daten nur die Referenz auf die Daten abgelegt wird, ist der Speicherbedarf der Variablen recht klein; er entspricht der Größe einer Adresse im Arbeitsspeicher. Den Speicher für die eigentlichen Daten reserviert der Compiler nicht auf dem Stack, sondern im so genannten Heap. Die Anfangsadresse dieses Speicherbereichs trägt er in die lokale Variable ein.

Fassen wir noch einmal zusammen:

- Variablen repräsentieren Daten.
- Für Variablen elementarer Datentypen werden die Daten direkt in der Variablen gespeichert. Der Wert einer Variablen ist gleich den in ihr gespeicherten Daten (eine Zahl, ein Zeichen, ein Boolescher Wert).
- Für Variablen komplexer Datentypen werden die Daten nicht direkt in der Variablen gespeichert, sondern irgendwo im Heap, wo sie ein so genanntes Speicherobjekt bilden. Der Wert einer Variablen ist die Anfangsadresse dieses Speicherbereichs.

Icon INFO

Datentypen, deren Daten nicht in den Variablen gespeichert werden, bezeichnet man auch als **Referenztypen** und stellt sie den **Werttypen** gegenüber. In Bali – wie in Java – sind alle komplexen Datentypen Referenztypen und alle elementaren Datentypen sind Werttypen.

Wird jetzt einer Variablen eines Referenztyps der Wert einer anderen Variablen gleichen Typs zugewiesen, kopiert der Compiler nicht die Daten des Speicherobjekt, sondern lediglich den Verweis. Danach gibt es zwei Variablen, die auf ein und dasselbe Speicherobjekt verweisen!

Dies hat Konsequenzen für die Programmierung.

Wenn Sie einer Variablen A eine andere Variable B gleichen Typs zuweisen, kopieren Sie den Wert der Variablen B. Handelt es sich um Variablen eines Werttyps haben Sie Daten kopiert und können danach beide Kopien unabhängig voneinander bearbeiten. Handelt es sich dagegen um Variablen eines Referenztyps haben Sie Referenzen kopiert und bearbeiten über beide Variablen dasselbe Objekt. Das folgende Codefragment soll dies verdeutlichen. Der Datentyp `Adresse` sei dabei wie oben definiert.

```
demo(Adresse adr, int z)
  {
  adr.name = "Neuer Name";
```

```
  z = 1;
  }

void main()
  {
  Adresse kunde;
  int zahl;

  kunde.name = "Heinz";
  zahl = 444;

  demo(kunde, zahl);

  out kunde.name;
  out zahl;
  }
```

Wie lautet die Ausgabe dieses Programms? `kunde` ist eine Variable eines Referenztyps. Beim Aufruf der Funktion `demo()` wird also die Referenz auf das Speicherobjekt in `adr` kopiert. Über diese Referenz greift die Funktion `demo()` auf das Speicherobjekt zu und ändert den Namen. Wenn später die Funktion `main()` auf das Objekt zugreift und den Wert des `name`-Felds ausgibt, steht in diesem der String "Neuer Name". Ganz anders sieht die Verarbeitung des `int`-Arguments aus. Beim Funktionsaufruf wird der Wert von `zahl` in das Argument `z` kopiert. Beide Variablen, `zahl` und `z`, besitzen danach den Wert 444. Dann weist die Funktion ihrer lokalen Variablen `z` den Wert `1` zu. Der Wert der `main()`-Variablen `zahl` bleibt davon unberührt. Die Ausgabe des Programms lautet daher:

```
"Neuer Name"
444
```

Lebensdauer von Heap-Objekten

Einen Aspekt der Speicherverwaltung für Referenztypen haben wir noch nicht angesprochen: die Lebensdauer. Wie Sie bereits gelernt haben, existieren lokale Variablen nur so lange, wie ihre Funktion ausgeführt wird, weil der Compiler den Speicher für die Variablen auf dem Stack reserviert. Für Variablen von Werttypen bedeutet dies, dass mit der Variablen auch die Daten verschwinden. Für Variablen von Referenztypen existieren die Daten dagegen als unabhängiges Speicherobjekt auf dem Heap, und es stellt sich die Frage, ob dieses Speicherobjekt zusammen mit seiner lokalen Variablen aufgelöst wird. (Der Compiler könnte vor Auflösung der lokalen Variablen dafür sorgen, dass der Speicherbereich, auf den die Variable verweist, freigegeben wird.)

Nun, sie wird es nicht. Warum er dies nicht tut, wird schon an obigem Beispiel klar. Wenn die Funktion `demo()` aufgerufen wird, wird die Referenz von `kunde` in den Parameter `adr` kopiert. Beide Referenzen

verweisen auf das gleiche Speicherobjekt. Schließlich wird `demo()` wieder beendet. Angenommen das Speicherobjekt, auf das `adr` verweist, würde jetzt aufgelöst. Dann würde `main()` in der `out`-Anweisung, die unter dem Funktionsaufruf folgt, versuchen über `kunde` auf ein Speicherobjekt zuzugreifen, dass es nicht mehr gibt. Das Programm würde bei der Ausführung abstürzen.

Um derartige Fehler zu verhindern, könnte man einmal erzeugte Objekte einfach über die gesamte Ausführungszeit des Programms bestehen lassen. Dies wäre allerdings eine ziemliche Speicherverschwendung und könnte sogar dazu führen, dass der Speicher des einen oder anderen Rechners bei Ausführung des Programms komplett voll geschrieben wird. Die Objekte sollen also aufgelöst werden. Die Frage ist nur wann? Natürlich wenn sie nicht mehr benötigt werden. Und wann werden sie nicht mehr benötigt? Diese Frage kann im Grunde nur der Programmierer beantworten. Wir führen daher ein weiteres Schlüsselwort ein. Mit

```
delete varname;
```

kann der Programmierer das Speicherobjekt löschen, auf das `varname` verweist.

Es sei gleich darauf hingewiesen, dass diese Syntax äußerst fehleranfällig ist. Zum einem entstehen Speicherlecks, wenn der Programmierer vergisst, nicht mehr benötigte Objekte zu löschen, zum anderen – und dies ist noch gefährlicher – entstehen gravierende Fehler, wenn der Programmierer ein Objekt löscht, auf das er später versehentlich noch einmal zuzugreifen versucht, weil es in seinem Programm noch Variablen gibt, die Referenzen auf das ehemalige Objekt enthalten.

Icon STOPP

In Java gibt es obige Syntax daher nicht, die Auflösung der Speicherobjekte wird anders gehandhabt.

Programmieren mit komplexen Datentypen, Teil 2

Das Umstand, dass in Bali alle komplexe Datentypen Referenztypen sind, hat Konsequenzen für die Programmierung mit den Variablen dieser Datentypen.

- Die Zuweisung ist definiert, kopiert aber nicht die Speicherobjekte, sondern nur die Verweise.
- Von den Vergleichsoperatoren sind nur `==` und `!=` definiert. Sie testen, ob zwei Variablen die gleiche Referenz enthalten (sprich auf ein und dasselbe Speicherobjekt verweisen).

Andere Operationen sind für Referenztypen nicht definiert.

Angesichts einer derart dürftigen Unterstützung durch die Sprache, stellt sich die Frage, ob sich mit den komplexen Datentypen überhaupt sinnvoll programmieren lässt? Wie soll man mit Variablen programmieren, die sich weder kopieren noch vernünftig miteinander vergleichen lassen? Wie soll man mit Daten arbeiten, für die es keine datenspezifischen Operationen gibt?

Indem man die gewünschten Operationen selbst codiert!

Selbst geschriebene Operationen? Bedeutet dies, dass wir dem Programmierer gestatten, neue Operatorsymbole in die Sprache einzuführen oder bestehende Operatorsymbole – etwa das +-Zeichen – für komplexe Datentypen umzudefinieren? Nein, tut mir leid. Es gibt zwar Sprachen, bei denen dies möglich ist (namentlich C++), doch Bali – ebenso wie Java – gehören nicht zu ihnen. Unsere Lösung ist viel prosaischer: Der Programmierer muss die "Operationen" aufdröseln und mit Hilfe der zur Verfügung stehenden Anweisungen implementieren – wobei vor allem die `.`-Syntax für den Zugriff auf die Felder der Speicherobjekte hilfreich ist.

Möchte der Programmierer beispielsweise ein Speicherobjekt richtiggehend kopieren – statt nur die Referenz auf das Objekt zu duplizieren –, muss er die einzelnen Felder des Datentyps 1:1 von dem Original in die Kopie übertragen. Statt

```
// gegeben:  adrOrg vom Typ Adresse
Adresse adrKopie;
adresseKopie = adrOrg;                  // kopiert nur die Referenz
```

muss er also schreiben:

```
// gegeben:  adrOrg vom Typ Adresse
Adresse adrKopie;
adrKopie.vorname = adrOrg.vorname;
adrKopie.name = adrOrg.name;
adrKopie.strasse = adrOrg.strasse;
...
```

Praktisch veranlagte Programmierer werden die "Operationen" für ihre Referenztypen, zumindest die häufiger benötigten, in Form von Funktionen implementieren:

```
Adresse kopieren(Adresse adrOrg)
  {
   Adresse adrKopie;
   adrKopie.vorname = adrOrg.vorname;
   adrKopie.name = adrOrg.name;
   adrKopie.strasse = adrOrg.strasse;
   ...
   return adrKopie;
  }
```

Die obige Funktion übernimmt als Argument die Referenz des zu kopierenden Adresse-Objekt. Dann definiert sie eine lokale Adresse-Variable `adrKopie` (für die der Compiler ein neues Adresse-Objekt im Speicher anlegt) und kopiert die Feldwerte von `adrOrg` in die Felder des `adrKopie`-Objekts. Zum Abschluss liefert sie die Referenz auf `adrKopie` an den Aufrufer zurück. (Die lokale Variable `adrKopie` wird nach Beendigung der Funktion aufgelöst, doch das Speicherobjekt, auf das `adrKopie` verwies, bleibt auf dem Heap bestehen und kann über die zurückgelieferte Referenz angesprochen werden.)

```
// wenn eineAdr und andereAdr vom Typ Adresse sind,
// macht der folgende Aufruf eineAdr zu einer Kopie von andereAdr
eineAdr = kopieren(andereAdr);
```

2.4.2 Von den Daten zu den Objekten

Zu einer leistungsfähigen Programmiersprache gehört ein schlüssiges Konzept zur Programmierung mit selbst definierten, komplexen Datentypen. Bali verfügt jetzt über ein solches Konzept – ein durchaus befriedigendes Konzept, das allerdings die Einheit von Daten und typspezifischen Operationen, wie wir sie von den elementaren Datentypen her kennen, aufhebt.

Lassen Sie uns daher überlegen, wie wir Daten und Operationen auch für die komplexen Datentypen wieder zusammenbringen können. Betrachten Sie dazu folgenden Code:

```
type Kreis
  {
  double radius;
  int    x;          // x-Koordinate des Mittelpunkts
  int    y;          // y-Koordinate des Mittelpunkts
  }

double umfang(double r)
  {
   return 2 * r * 3.14159;
  }

double flaeche(double r)
  {
   return r * r * 3.14159;
  }

void main()
  {
   Kreis k;
   string eingabe;

   out "Radius eingeben: ";
```

```
  in eingabe;
  k.radius = (double) eingabe;

  out " Umfang: " + umfang(k.radius));
  out " Fläche: " + flaeche(k.radius));
}
```

In diesem Programm gibt es auf der einen Seite die Funktionen `umfang()` und `flaeche()` und auf der anderen Seite die Variable `k` vom Typ `Kreis`. Das Programm selbst dient dazu, Umfang und Fläche von Kreisen zu berechnen. Der Anwender muss dazu den Radius des zu berechnenden Kreises eingeben und das Programm liefert dann den Umfang und die Fläche. Klingt alles recht vernünftig und durchdacht, und doch... irgend etwas stimmt hier nicht.

Liegt es an dem komplexen Typ `Kreis`? Eigentlich wird dieser doch gar nicht benötigt. Es wäre absolut ausreichend, eine `double`-Variable `radius` zu definieren, in dieser die Benutzereingabe zu speichern und die Variable dann als Argument an `umfang()` und `flaeche()` zu übergeben. Trotzdem ist die Definition des Typs `Kreis` durchaus gerechtfertigt. Zum einem entspricht es der Logik des Programms, dessen Zweck es schließlich ist, Fläche und Umfang von Kreisen (und nicht Radien) zu berechnen. Zum anderen kann das Programm so leichter erweitert werden (beispielsweise zur Berechnung von Abständen oder der Erkennung von Überlappungen zwischen mehreren Kreisen).

Nein, das Problem liegt an anderer Stelle, genauer gesagt bei den Funktionen `umfang()` und `flaeche()`. Diese sind zwar so implementiert, dass Sie nur für Kreise korrekte Werte liefern, können aber mit beliebigen `double`-Werten aufgerufen werden. Sie könnten also hingegen, eine Typ Quadrat mit einem `double`-Feld `seitenlaenge` definieren und dann umfang(`qu.seitenlaenge`) zur Berechnung des Umfang des Quadrats aufrufen. Natürlich erhalten Sie dann als Ergebnis nicht den Umfang des Quadrats, sondern den Umfang eines Kreises, dessen Radius gleich der Seitenlänge des Quadrats ist, und natürlich werden Sie einwenden, dass Ihnen ein solcher Fehler nicht passieren wird. Doch bedenken Sie, dass dies nur ein sehr kleines und gut überschaubares Beispiel ist. In der Praxis werden Sie es in der Regel mit viel umfassenderen Programmen zu tun haben und Sie werden immer wieder auf bestehenden Code zurückgreifen, über dessen korrekte Verwendung Sie womöglich nur unzureichend informiert sind, weil der Code von einem anderen Programmierer stammt, weil es solange her ist, dass Sie den Code geschrieben haben, weil Sie es versäumt haben, die Dokumentation zu lesen. Unter diesem Aspekt wäre es schon sehr hilfreich, wenn der Compiler sicherstellen würde, dass eine Funktion (wie z.B. `umfang()`), die für die Bearbeitung eines bestimmten Datentyps (Kreisen) geschrieben

wurden, auch nur für Variablen dieses Datentyps aufgerufen werden kann – so wie der Operatoren auch nur auf die Datentypen angewandt werden können, für die sie definiert sind! Haben Sie dazu eine Idee?

Richtig, statt eines `double`-Parameters sollten die Funktionen einen Parameter desjenigen Datentyps definieren, den Sie bearbeiten – in unserem Beispiel also eines `Kreis`-Typs:

```
type Kreis
  {
  double radius;
  int    x;          // x-Koordinate des Mittelpunkts
  int    y;          // y-Koordinate des Mittelpunkts
  }

  double umfang(Kreis kreis)
    {
     return 2 * kreis.radius * 3.14159;
    }

  double flaeche(Kreis kreis)
    {
     return kreis.radius * kreis.radius * 3.14159;
    }
  }

void main()
  {
   Kreis k;
   string eingabe;

   out "Radius eingeben: ";
   in eingabe;
   k.radius = (double) eingabe;

   out " Umfang: " + umfang(k));
   out " Fläche: " + flaeche(k));
  }
```

Sollte jetzt irgend jemand irrtümlich versuchen, eine der Funktionen mit einem Argument eines falschen Typs aufzurufen:

```
type Quadrat
  {
  double seitenlaenge;
  int    x;          // x-Koordinate der linken oberen Ecke
  int    y;          // y-Koordinate der rechten oberen Ecke
  }

...

Quadrat qu;
...
umfang(qu));     // Fehler!
```

erkennt der Compiler bereits zur Übersetzungszeit, dass die Funktion nicht für die Verarbeitung von Argumenten dieses Typs definiert wurde und meldet einen Fehler.

Wir haben es nun mit Hilfe der Parameter der Funktion geschafft, wieder eine Beziehung zwischen Operationen (Funktionen) und Daten herzustellen. Noch aber ist dies lediglich die triviale Beziehung die zwischen jedweden Daten und Funktionen, die diese Daten verarbeiten, automatisch entsteht. Unser Ziel aber ist es, Funktionen zu schreiben, die semantisch eher Operatoren ähneln. Analysieren wir dazu kurz, was die Operatoren überhaupt auszeichnet:

1. Zu jedem Datentyp gibt es, einen festen Satz Operatoren, der nicht erweitert werden kann. (+, -, *. /, =, ==, < etc. für Integer-Typen.)
2. Operatoren operieren automatisch auf Ihren Operanden, d.h., den Daten, die links und womöglich auch rechts von ihnen stehen.
3. Für verschiedene Datentypen können gleichnamige Operatoren definiert sein, die jedoch jeweils typspezifische Aktionen ausführen. (+ für Numerische Typen bewirkt Addition, + für Strings bewirkt Konkatenation.)

Schritt für Schritt werden wir nun "OpFunktionen" in Bali einführen, die für neu definierte, komplexe Datentypen wie Operationen wirken.

Wir teilen die Funktionen also in zwei Kategorien ein:

Icon INFO

- "normale" Funktionen, wie wir Sie bisher kennen, und
- "OpFunktionen", die komplexen Datentypen angehören und Operatoren ähneln.

(Leser mit Vorkenntnissen in objektorientierter Programmierung werden wahrscheinlich bereits ahnen, dass diese "OpFunktionen" nichts anderes als die Methoden von Klassen sind.)

Schritt 1 – Typspezifische Funktionen

Welche Operatoren für einen elementaren Datentyp definiert sind, gibt die Sprachspezifikation vor. Welche "Operatoren" für einen komplexen Datentyp definiert sind, kann die Sprachspezifikation nicht vorgeben, dies muss sie dem Programmierer überlassen, der den Datentyp neu definiert. Die Sprache kann aber festlegen, dass diese "Operatoren" zusammen mit dem Datentyp definiert werden müssen. Die zugehörige Syntax könnte beispielsweise wie folgt aussehen:

```
type Typname
  {
  TYP feld1;
  TYP feld2;

  TYP opFunktion1(PARAMETER)
    {
     ANWEISUNGEN
    }
  TYP opFunktion2(PARAMETER)
    {
     ANWEISUNGEN
    }
  }
```

Diese Syntax hat drei Vorteile:

- Die OpFunktionen können nicht mehr über den Quelltext (womöglich gar über mehrere Quelletextdateien) verstreut werden, sondern stehen gebündelt an der Stelle, an der auch der Typ definiert wird. Wenn ein Programmierer also mal vergisst, welche OpFunktionen für einen Typ definiert sind oder wie eine bestimmte OpFunktion implementiert ist, weiß er sofort, wo er nachschauen muss.
- Programmierer, die einen Typ definiert haben, können den Quelltext kompilieren und das Kompilat anderen Programmierern zur Verfügung stellen. Diese können den Typ so verwenden, als hätten sie ihn selbst in ihrem Programm definiert (wie dies im Detail geht, ist von Programmiersprache zu Programmiersprache verschieden und braucht uns hier nicht weiter zu interessieren), da sie aber den Quelltext der Typdefinition nicht vorliegen haben, können Sie die Typdefinition nicht ändern. Insbesondere können Sie also OpFunktionen weder löschen noch hinzufügen – so wie es Punkt 1 forderte (fester Satz an Operationen).
- Und selbst wenn der Programmierer, der den Typ definiert hat, Kompilat und Quelltext weitergibt, so wissen die anderen Programmierer doch, dass er sich bei der Implementierung des Typs seine Gedanken gemacht hat und sie die Typdefinition nicht ändern sollten. Sie können normale Funktionen schreiben, die Variablen des Typs als Argumente übernehmen und bearbeiten (so wie sie Funktionen schreiben können, die elementare Daten als Argumente übernehmen), aber sie werden sich hüten, neue OpFunktionen zu definieren, die Implementierung bestehender OpFunktionen zu verändern oder Felder zu streichen.

Schritt 2 – Eigene Aufrufsyntax

Für den Zugriff auf die Felder eines komplexen Typs haben wir eine besondere Syntax eingeführt. Zuerst wird eine Variable des Typs definiert, dann kann man über den Variablennamen und den Punktoperator auf die Felder des Typs zugreifen:

```
EinKomplexerTyp eineVar;

eineVar.feld1 = ...
```

Die gleiche Syntax soll auch für den Aufruf von OpFunktionen gelten:

```
eineVar.opFunktion1();
```

Dank dieser Form des Aufrufs, können wir darauf verzichten, die zu bearbeitende Variable als Parameter an die OpFunktion zu übergeben: sie steht ja bereits vor der OpFunktion. Und in der OpFunktionsdefinition greifen wir einfach direkt (ohne vorangestellten Variablennamen) auf die Felder des Typs zu. Die Definition für den Typ `Kreis` sähe dann beispielsweise so aus:

```
type Kreis
  {
  double radius;
  int    x;          // x-Koordinate des Mittelpunkts
  int    y;          // y-Koordinate des Mittelpunkts
  }

  double umfang()
    {
     return 2 * radius * 3.14159;
    }

  double flaeche()
    {
     return radius * radius * 3.14159;
    }
  }
```

Beachten Sie, dass `umfang()` und `kreis()` nunmehr keinen Parameter vom Typ `Kreis` definieren und direkt auf das Feld `radius` zugreifen.

Wenn Sie nun für zwei `Kreis`-Variablen den Umfang berechnen wollen, schreiben Sie:

```
Kreis k1;
k1.radius = 1;
Kreis k2;
k2.radius = 2;
...
out k1.umfang();  // Ausgabe 6.28318
out k2.umfang();  // Ausgabe 12.56636
```

Um diese vereinfachte Syntax für Definition und Aufruf von OpFunktionen möglich zu machen, lassen wir den Übersetzer ein wenig Magie betreiben. Konkret weisen wir den Übersetzer an, dass er Definitionen und Aufrufe von OpFunktionen intern ergänzt:

- Er soll die Parameterliste um einem Parameter von dem betreffenden Typ erweitern und in der Implementierung allen Feldnamen diesen Parameter voranstellen.
- Er soll die Aufrufe intern so umschreiben, dass er die Variable, für die die OpFunktion aufgerufen wird, als Argument an den intern hinzugefügten Parameter übergibt.

Der Übersetzer würde den obigem Code also intern wie folgt "umschreiben":

```
type Kreis
  {
  double radius;
  int    x;          // x-Koordinate des Mittelpunkts
  int    y;          // y-Koordinate des Mittelpunkts
  }

  double umfang(Kreis k)
    {
     return 2 * k.radius * 3.14159;
    }

  double flaeche(Kreis k)
    {
     return k.radius * k.radius * 3.14159;
    }
  }
...
Kreis k1;
k1.radius = 1;
Kreis k2;
k2.radius = 2;
...
out umfang(k1);  // Ausgabe 6.28318
out umfang(k2);  // Ausgabe 12.56636
```

Icon NOTE

Es ist zwar nur von marginaler Bedeutung, aber ich möchte Sie doch noch auf die formale Ähnlichkeit zwischen dem Aufruf einer OpFunktion und dem Einsatz eines Operators hinweisen. Angenommen Sie haben eine OpFunktion namens `demo()`, die für den Typ `TA` definiert ist und als Parameter einen `double`-Wert definiert. (Der Umstand, dass der Übersetzer intern für die OpFunktionen einem Parameter des betreffenden Typs definiert, bedeutet ja nicht, dass die OpFunktionen keine sonstigen Parameter besitzen könnten.) Ein Aufruf von `demo()` sähe dann beispielsweise wie folgt aus:

```
TA a;
a.demo(3.14);
```

Trotz unterschiedlicher Symbolik entspricht dies vom Aufbau durchaus dem Einsatz eines Operators mit zwei Operanden:

```
Op1 operator Op2;
```

Schritt 3 – Überladung

Wo sinnvoll, besitzen unterschiedliche Datentypen durchaus gleiche Operatoren. So ist der +-Operator nicht nur für `int`, sondern auch für `double` definiert. Die Art der Operation, die der Operation durchführt, ist dabei jeweils die gleiche (Addition), die konkrete Durchführung ist typspezifisch (`double`- und `int`-Werte werden auf Maschinenebene unterschiedlich dargestellt und müssen daher auch mit unterschiedlichen Maschinenbefehlen addiert werden).

In gleicher Weise ist die Berechnung des Umfangs nicht nur für Kreise, sondern beispielsweise auch für Quadrate sinnvoll. Da der Umfang eines Kreises aber nach einer anderen Formel berechnet wird als der Umfang eines Quadrats, ist klar, dass der Programmierer hierfür zwei eigene Funktionen schreiben muss. Aber ist es auch notwendig, dass diese unterschiedliche Namen haben?

Grundsätzlich gilt natürlich die Regel, dass alle vom Programmierer eingeführten Namen (Bezeichner) eindeutig sein müssen, denn schließlich muss der Übersetzer die eingeführten Elemente ja auseinander halten können. Einfache Programmiersprachen, wie Bali bisher auch, fordern daher rigoros, dass alle Bezeichner eindeutig sein müssen (mit dem einzigen Zugeständnis, dass für Variablen in unterschiedlichen Gültigkeitsbereichen Namen wieder verwendet werden dürfen).

Der Programmierer muss dann für Funktionen, auch wenn diese "gleichartige" Operationen oder Berechnungen durchführen, unterschiedliche Namen vergeben:

```
int quadrierenFuerInteger(int zahl)
  {
   return zahl * zahl;
  }

double quadrierenFuerDouble(double zahl)
  {
   return zahl * zahl;
  }
```

Besonders hässlich ist dies, bei OpFunktionen:

```
type Kreis
  {
  ...
  double umfangKreis()
    {
     return 2 * radius * 3.14159;
    }
  }

type Quadrat
  {
  ...
  double umfangQuadrat()
    {
     return 4 * seitenlaenge;
    }
  }
```

Sprachen, die die Definition komplexer Typen mit Feldern und OpFunktionen unterstützen, erlauben es daher für Funktionen (jedweder Art) Namen mehrfach zu verwenden – vorausgesetzt, dem Compiler stehen sonstige Informationen zur Verfügung, anhand deren er entscheiden kann, welche Funktion für welchen Funktionsaufruf ausgeführt werden soll. Diese zusätzlich Information liefern in der Regel die Parameter, genauer gesagt ihre Anzahl, ihre Typen und ihre Reihenfolge. Funktionen, die sich im Namen oder in der Parameterliste unterscheiden, können auch vom Compiler unterschieden werden.

Typ, Anzahl und Reihenfolge der Parameter plus dem Namen der Funktion werden zusammen als **Signatur** der Funktion bezeichnet.

Icon INFO

```
int quadrieren(int zahl)
  {
   return zahl * zahl;
  }
double quadrieren(double zahl)
  {
   return zahl * zahl;
  }
```

Trifft der Übersetzer im Quelltext nun auf einen Aufruf von `quadrieren()`:

```
double wert = 12.0;
...
out quadrieren(wert);
```

schaut er zuerst nach, ob es eine Funktion namens quadrieren gibt. Da er sogar zwei Funktionen dieses Namens findet, ermittelt er Anzahl und Typ der im Aufruf übergebenen Argumente (hier ein einziger `double`-Wert) und prüft, ob es unter den `quadrieren`-Funktionen eine (und zwar genau eine) gibt, deren Parameter zu den Argumenten passen. Auf diese Weise gelangt er zu der zweiten `quadrieren`-Definition und führt diese aus.

Reale Übersetzer arbeiten meist etwa effizienter. Der C++-Compiler erweitert beispielsweise die Namen der Funktionen um Kürzel, die die Parameterliste kodieren – so dass er intern wieder mit eindeutigen Namen arbeiten kann.

Icon INFO

Auch OpFunktionen verschiedener Typen dürfen nun gleich lautende Namen haben:

```
type Kreis
  {
  ...
  double umfang()
    {
     return 2 * radius * 3.14159;
    }
  }


type Quadrat
  {
  ...
  double umfang()
    {
     return 4 * seitenlaenge;
    }
  }
```

Beim Aufruf:

```
Kreis k;
k.radius = 3;
Quadrat q;
q.radius = 2;
...
out umfang(k);
out umfang(q);
```

wird ja die Variable, über die der Aufruf erfolgt, intern als Argument übergeben. Folglich kann der Übersetzer die aufzurufenden OpFunktionen anhand ihrer Parameter unterscheiden und genau die OpFunktion aufrufen, die zum Typ der Variable passt.

Die Definition mehrerer Funktionen gleichen Namens bezeichnet man in der Programmierung als **Überladung**.

Icon INFO

Klassen und Objekte

Es ist nun an der Zeit, die technische Spezifikation unserer komplexen Datentypen philosophisch zu untermauern. Tatsächlich stellen unsere komplexen Datentypen mittlerweile schon ziemlich genau das dar, was man in der objektorientierten Programmierung unter einer Klasse versteht.

Im Gegensatz zur klassischen Art der Programmierung, bei der der Programmierer auf der einen Seite die Daten identifiziert, die er verarbeiten möchte, und auf der anderen Seite die Funktionen schreibt, die er zur Bearbeitung der Daten benötigt, propagiert die objektorientierte Programmierung die Idee, dass der Programmierer die ihm gestellten Aufgaben mit Hilfe von Objekten löst. Die Objekte der objektorientierten Programmierung sind dabei nichts anderes als die "Werte" der Variablen unserer komplexen Datentypen, die wir fortan in Anlehnung an die objektorientierte Terminologie als Klassen bezeichnen und mit dem Schlüsselwort `class` (statt mit `type`) definieren wollen. Die "OpFunktionen" bezeichnen wir von nun an – wie in der objektorientierten Programmierung üblich – als Methoden.

Objektorientierte Terminologie

Eine **Klasse** ist ein mit `class` eingeleitete Typdefinition. Die internen Variablen einer Klasse nennt man **Feldern**, die internen Funktionen **Methoden**.

Ein **Objekt** ist ein Speicherobjekt, das auf Grundlage einer Klassendefinition angelegt wird. Es besitzt von jedem Feld der Klasse eine Kopie, in der es seine objektspezifischen Werte speichern kann. Objekte von Klassen werden auch als **Instanzen** bezeichnet.

In den **Variablen eines Klassentyps** wird lediglich eine Referenz (siehe Abschnitt 2.4.1,"Wert und Referenztypen") auf das Objekt im Speicher abgelegt. Variablen von Klassentypen werden zur Unterscheidung von anderen Variablen manchmal auch **Objektvariablen** genannt oder man identifiziert sie einfach mit den Objekten, auf die sie verweisen.

Angenommen ein Software-Entwickler soll ein Programm schreiben, das zwei Adressen vergleicht und rückmeldet, ob die Adressen auf den gleichen Namen eingetragen sind. Wie das Programm die Adressdaten einliest, ob über die Eingabe, eine Datei oder das Internet, ist dabei

unwichtig. Uns interessiert, wie der Programmierer – mit den Mitteln unserer Programmiersprache Bali – die Daten verwaltet und vergleicht.

Vielleicht handelt es sich bei dem Software-Entwickler um einen Programmierer der alten Schule, der den Begriff der Datenverarbeitung noch wörtlich nimmt. Dann wird er auf der einen Seite Variablen definieren, in denen er die Adressdaten speichert und auf der anderen Seite eine Funktion zum Vergleichen der beiden Adressen:

```
// Funktion zum Vergleichen der Adressen
boolean adressenGleich(string vn1, string nn1, string vn2, string nn2)
  {
  if(vn1 == vn2)
    {
    if (nn1 == nn2)
      {
      return true;
      }
    }
    else
    {
      return false;
    }
  }

void main()
  {
  // Variablen für 1. Adresse
  string vornameAdr1;
  string nameAdr1;
  string strasseAdr1;
  int hausnummerAdr1;
  int plzAdr1;
  string stadtAdr1;

  // Variablen für 2. Adresse
  string vornameAdr2;
  string nameAdr2;
  string strasseAdr2;
  int hausnummerAdr21;
  int plzAdr2;
  string stadtAdr2;

   // Adressdaten einlesen
   ...
   // Adressen vergleichen
   boolean gleich = false;
   gleich = adressenGleich(vornameAdr1, nameAdr1,
                           vornameAdr2, nameAdr2);
   if (gleich == false)
     {
     out "Unterschiedliche Namen in Adressen";
     }
     else
     {
```

```
    out "Gleiche Namen in Adressen";
    }
  }
```

Vielleicht ist der Programmierer aber auch bereits mit der objektorientierten Programmierung vertraut und möchte deren Konzepte nutzen.

Dann wird er sich zuerst überlegen, mit welchen Objekten er in seinem Programm arbeiten möchte. In unserem Beispiel ist die Antwort schnell gefunden, da es nur eine Art von Objekten gibt: die Adressen.

Um Adressen-Objekte erzeugen zu können, benötigt der Programmierer eine passende Klassendefinition. Er überlegt daher,, ob ihm bereits eine solche Klasse zur Verfügung steht (vielleicht hat er selbst schon einmal eine solche Klasse geschrieben – etwa als Bestandteil eines Programm zur Adressenverwaltung – oder er hat entsprechende Klassen in Form einer Klassenbibliothek käuflich erworbenen). Findet er keine passende Klasse muss er die Klasse selbst definieren. Er überlegt sich, welche Felder und Methoden die Klasse enthalten muss und kommt zu folgender Klassendefinition:

```
class Adresse {
  string vorname;
  string name;
  string strasse;
  int hausnummer;
  int plz;
  string stadt;

  boolean istGleich(Adresse andereAdresse)
    {
    if(vorname == andereAdresse.vorname)
      {
      if (name == andereAdresse. name)
        {
        return true;
        }
      }
      else
      {
        return false;
      }
    }
  }
```

Ist die Klasse für die Adressen-Objekte erst einmal definiert, ist der Rest des Programms schnell geschrieben. Der Programmierer erzeugt Objekte der Klasse und nutzt deren Methoden, um die gestellten Aufgaben zu erledigen:

```
void main()
  {
  Adresse adr1;          // 1. Adressenobjekt erzeugen
  Adresse adr2;          // 2. Adressenobjekt erzeugen

   // Adressdaten einlesen
   ...

   // Adressen vergleichen
   boolean gleich = false;
   gleich = adr1.istGleich(adr2);
   if (gleich == false)
     {
     out "Unterschiedliche Namen in Adressen";
     }
     else
     {
     out "Gleiche Namen in Adressen";
     }
  }
```

2.4.3 Zugriffsschutz und Objekterzeugung

In der objektorientierten Programmierung wird viel von dem "Zustand" der Objekte geredet. Der momentane Zustand eines Objekts ist dabei nichts anderes als die zusammen genommen aktuellen Werte aller seiner Felder.

Warum ist dieser Zustand so wichtig?

Zuerst einmal natürlich weil er für die Daten des Objekts steht. Wenn Sie die Daten einer Adresse in die Felder eines Adresse-Objekt speichern, repräsentiert das Objekt diese Adresse und sein Zustand ist die aktuelle Adresse.

Für den Programmierer ist aber noch ein anderer Aspekt wichtig: Wie kann sich der Zustand eines Objekts ändern und wie kann man sicherstellen, dass das Objekt keine falschen Zustände annimmt.

In Bali gibt es zwei Wege, den Zustand eines Objekts zu ändern:

- Die Methoden der Klasse können den Feldern Werte zuweisen.

```
class Demo
  {
   int feld;

   void aendern()
    {
    feld = 2;
    }
  }
```

- Funktionen oder Methoden anderer Klassen, die über ein Objekt verfügen (in Form eines Parameters oder einer lokalen Variable), können den Feldern auf dem Weg über die Objektvariable Werte zuweisen.

```
void main()
  {
  Demo obj;
  obj.feld = 3;
  }
```

Für die weitere Betrachtung wollen wir zwischen dem Programmierer, der die Klasse definiert (dem *Autor*), und dem Programmierer, der die Klasse benutzt, indem er Objekte der Klasse erzeugt und mit diesen arbeitet (der *Benutzer*), unterscheiden. In der Praxis sind Autor und Benutzer zwar häufig in der Person eines Programmierers vereint, doch dies spricht nicht gegen eine Trennung der damit verbundenen Aufgabenbereiche: Der Autor ist dafür verantwortlich, dass die Klasse korrekt implementiert ist und möglichst einfach und sicher verwendet werden kann. Der Benutzer arbeitet mit der Klasse, wozu er sich über die Bedeutung ihrer Felder und Methoden informiert (die genaue Implementierung der Methoden muss ihn dabei aber nicht weiter interessieren, er ruft die Methoden lediglich auf).

Zurück zur Zustandsänderung und unserer Klasse `Adresse`:

```
class Adresse {
  string vorname;
  string name;
  string strasse;
  int hausnummer;
  int plz;
  string stadt;

  ...
  }
```

Das Feld für die Postleitzahl ist vom Typ `int`. Obwohl Postleitzahlen stets positiv sind, wäre es also auch möglich, `plz` eine negative Zahl zuzuweisen.

Dem Autor sollte ein solcher Fehler nicht passieren. Wie aber kann man verhindern, dass ein Benutzer, beispielsweise ein Amerikaner, der sich unter `plz` nichts vorstellen kann, `plz` einen negativen Wert zuweist?

Objektschutz durch Zugriffsspezifizierer

Zu diesem Zweck führen wir zwei Zugriffsspezifizierer namens `public` (öffentlich) und `private` (privat) ein. Der Autor kann jedem Element einer Klasse genau einen dieser Zugriffsspezifizierer zuweisen. Auf

Elemente, die als `public` deklariert sind, kann auf dem Weg über Objektvariablen zugegriffen werden, auf Elemente, die als private deklariert sind, nicht. Mit anderen Worten: `private`-Elemente sind nur für die Methoden derselben Klasse zugänglich.

Will der Autor die Felder seiner Klasse schützen (und sicherstellen, dass die Objekte der Klasse nur korrekte Zustände annehmen können), deklariert er alle Felder als `private` und stellt für jede in Frage kommende Zustandsänderung eine passende `public`-Methode zur Verfügung:

```
class Adresse {
  private string vorname;
  private string name;
  private string strasse;
  private int hausnummer;
  private int plz;
  private string stadt;

  public void setPLZ(int neuePLZ)
    {
      if (neuePLZ > 0)            // PLZ nur ändern, wenn neuer Wert
         plz = neuePLZ;           // eine positive Zahl ist
    }
  ...
  }
```

Icon NOTE

Die meisten objektorientierten Programmsprachen kennen drei oder mehr verschiedene Zugriffsspezifizierer und weisen Klassenelemente, für die der Programmierer keinen Zugriffsspezifizierer angegeben hat, einen Standardzugriffsspezifizierer zu. In Bali fordern wir, dass der Autor allen Elementen explizit einen Zugriffsspezifizierer zuteilt.

Objektinitialisierung mit Konstruktoren

Damit der Benutzer sinnvoll mit einem Objekt arbeiten kann, muss es ihm möglich sein, den Feldern des Objekts individuelle Anfangswerte zuweisen zu können. Wenn alle oder auch nur einzelne Felder des Objekts private sind, ist dies aber nicht mehr möglich – zumindest nicht durch direkten Zugriff auf die Felder. Der Autor könnte als Ersatz für jedes Feld eine `public` set-Methode zur Verfügung stellen, doch wäre die Objektinitialisierung dann recht unbequem und unter Umständen mit unnötig vielen Methodenaufrufen verbunden. Aus diesem Grunde postulieren wir, dass jede Klasse in Bali über eine spezielle Methode zur Objektinitialisierung verfügen muss. Diese Methode soll den gleichen Namen tragen wie die Klasse, keinen Rückgabetyp haben (nicht einmal

`void`) und automatisch bei jeder Objekterzeugung aufgerufen werden. Wir nennen Sie **Konstruktor**:

```
class Adresse {
  private string vorname;
  private string name;
  private string strasse;
  private int hausnummer;
  private int plz;
  private string stadt;

  public Adresse(string vn, string n, string str, int hn,
                 int p, string sta)
    {
      vorname    = vn;
      name       = n;
      strasse    = str;
      hausnummer = hn;
      if (p > 0)
         plz     = p;
         else
         plz     = 0;
      stadt      = sta;
    }
  ...
  }
...
void main()
  {
  Adresse adr("Manfred", "Mustermann", "Gartenweg", 12,
              85321, Großstadt);
  ...
  }
```

Dem Benutzer dient der Konstruktor dazu, den erzeugten Objekten Anfangswerte zuzuweisen. Die eigentliche Aufgabe des Konstruktors ist allerdings weiter gefasst: Er soll das Objekt in einen korrekten Anfangszustand setzen. Welche Arbeiten hierzu erforderlich sind, hängt von der Klasse und der Einschätzung des Autors ab.

Für eine einfache Klasse `Punkt`, die lediglich über zwei `public`(!) Felder `x` und `y` verfügt, könnte der Konstruktor beispielsweise sogar leer bleiben:

```
class Punkt
  {
  public int x;
  public int y;

  public Punkt()
    {
    }
  }
```

Ansonsten gehören zum typischen Aufgabenbereich eines Konstruktors:

- Initialisierung von Feldern mit Werten, die der Benutzer als Argumente an den Konstruktor übergibt.
- Initialisierung von Feldern, für die der Benutzer keine Argumente an den Konstruktor übergeben hat. (Nicht immer sind alle Felder einer Klasse auch für den Benutzer von Bedeutung. Viele Klasse besitzen beispielsweise Feldern, die nur der internen Implementierung der Klasse dienen, und mit denen der Benutzer gar nichts zu tun hat. Für diese Felder würde der Konstruktor natürlich keine Parameter definieren. Eine andere Möglichkeit ist, dass der Autor durch Überladung zwei oder mehrere Konstruktoren definiert hat: einen, über den der Benutzer sämtlichen Feldern Anfangswerte zuweisen kann und weitere, die nur für bestimmte Felder Argumente übernehmen und den anderen Feldern Standardwerte zuweisen.)
- Sonstige Initialisierungsarbeiten erledigen (beispielsweise Felder mit Daten aus Dateien füllen, eine Internetverbindung herstellen, Bilder laden etc.).

2.4.4 Statische Klassenelemente

Zur Zeit ist Bali eine hybride Programmiersprache, in der sich strukturierte und objektorientierte Programmierung mischen, d.h., es steht dem Bali-Programmierer frei, entweder mit Daten und Funktionen zu arbeiten oder Klassen und Objekte zu erzeugen.

Gegen diese Vermischung ist grundsätzlich nichts einzuwenden (C++ ist ein Beispiel für eine äußerst erfolgreiche und leistungsfähige Programmiersprache, die sowohl die strukturierte als auch die objektorientierte Programmierung gestattet). Tatsache ist aber, dass viele Programmierer – vor allem Anfänger und halbprofessionelle Programmierer –, sofern Sie die Wahl haben, vorzugsweise datenorientiert statt objektorientiert programmieren.

In professionellen Anwendungen geht der Trend aber ganz klar in Richtung Objektorientierung, da objektorientierter Code sicherer, besser zu pflegen, und leichter wiederzuverwenden ist. Unser letzter Schritt soll daher darin bestehen, Bali von einer hybriden in eine rein objektorientierte Programmiersprache zu verwandeln.

Dazu verbannen wir zuerst alle nicht-objektorientierten Konzepte aus der Sprache. Fortan:

- gibt es in Bali keine Funktionen mehr (d.h. Funktionen treten nur noch als Methoden in Klassendefinitionen auf).

- gibt es keine `type`-Definitionen, sondern nur noch `class`-Definitionen (d.h. es gibt nur noch zwei Arten von Datentypen: elementare Typen und Klassen).
- können Variablen nur als Parameter, als lokale Variablen in Methoden oder als Felder von Klassen definiert werden (entspricht weitgehend dem Ist-Zustand, da wir auf globale Variablen, siehe Abschnitt 2.3.3, "Gültigkeitsbereich und Lebensdauer von Variablen") ja sowieso schon verzichtet haben).

Mit anderen Worten: Bali-Quelltext bestehen jetzt nur noch Klassendefinitionen!

Wenn ein Bali-Programm nur aus Klassendefinitionen besteht, was wird dann aus der Startfunktion `main()`? Sie wird natürlich zu einer Startmethode:

```
class HauptklasseDesProgramms
  {
  ..

  void main()
    {
    ...
    }
  }
```

Jedes Bali-Programm darf nur eine einzige Klasse mit einer `main()`-Methode beinhalten.

Wie aber, wird `main()` aufgerufen? Da es sich um eine Methode handelt, muss Sie über ein Objekt ihrer Klasse aufgerufen werden. Dieses Objekt müsste in Code erzeugt werden, der vor `main()` ausgeführt wird. Dies ist jedoch ein Widerspruch in sich, da das Programm ja erst mit `main()` beginnen soll!

Um dieses Problem zu lösen, müssen wir über ein Hintertürchen doch wieder nicht-objektorientierte Konzepte zulassen. Konkret führen wir das Schlüsselwort `static` ein und postulieren, dass Klassenelemente, die als `static` deklariert sind, nicht den Objekten der Klasse angehören, sondern eigenständig und global verfügbar sind. Der Zugriff erfolgt direkt über `Klassennamen.Elementname`.

Indem die `main()`-Methode als `static` deklariert wird, kann sie vom Betriebssystem aufgerufen werden, ohne dass dazu ein Objekt der Klasse benötigt wird:

```
class HauptklasseDesProgramms
  {
```

```
  ..

  static void main()
    {
    ...
    }
  }
```

`main()` ist nicht der einzige Anwendungsbereich für das Schlüsselwort `static`. Trotz aller Vorteile, die die Arbeit mit Objekten mit sich bringt, gibt es immer wieder auch Aufgaben, die besser mit "Funktionen" als mit Objekten zu lösen sind. Ein typisches Beispiel hierfür sind die mathematischen Funktionen wie Sinus, Kosinus, Potenz, Wurzel, Fakultät etc. Diese als Methoden eines Objekts zur Verfügung zu stellen, ist wenig sinnvoll. Denkbar wäre es. Sie könnten beispielsweise für die Werte, die die Methoden verarbeiten, Felder einrichten und die Methoden auf diesen Feldern operieren lassen.

Icon NOTE

```
class MathObjekt
  {
  public double wert1;
  public double wert2;

  ...

  public MathObjekt(double w1, double w2)       // Konstruktor
    {
    wert1 = w1;
    wert2 = w2;
    }
  public double potenz()
    {
     // liefert wert1 hoch wert2 zurück
    }
  }
```

Letzten Endes komplizieren Sie dadurch aber nur unnötig den Aufruf. Um beispielsweise 2 hoch 3 zu berechnen, müsste der Benutzer ein Objekt erzeugen, den Feldern die gewünschten Parameter zuweisen und dann `potenz()` aufrufen:

```
MathObjekt obj(2, 3);
out obj.potenz;
```

Logischer ist es in solchen Fällen, die Klasse als Funktionssammlung zu konzipieren und die Methoden als `static` zu deklarieren:

```
class MathObjekt
  {
  public MathObjekt()       // leerer Konstruktor
    {
    }

  ...
```

```
  public static double potenz(double basis, double exponent)
    {
     // liefert wert1 hoch wert2 zurück
    }
}
```

Aufruf:

```
out MathObjekt.potenz(2, 3);
```

2.4.5 Geschichte: der objektorientierten Programmierung

Simula und Smalltalk

Die erste objektorientierte Programmsprache war Simula, genauer gesagt Simula 67 (1967).

Simula I (1965), die erste Version, hatte allerdings noch wenig mit Objektorientierung zu tun. Simula I baute auf ALGOL auf und sollte die Berechnung von Simulationen vereinfachen. Zu diesem Zweck führte die Sprache das Prinzip der Aktivitäten und Prozesse ein. Der Programmierer deklarierte verschiedene Aktivitäten und erzeugte dann Prozesse, die diese Aktivitäten ausführten.

Für die zweite Version, Simula 67, wurde das Konzept der Aktivitäten und Prozesse ausgebaut. Die Änderungen entwickelten eine gewisse Eigendynamik, und plötzlich hatte man eine leistungsfähige neue Sprache, die sich keineswegs nur für Simulationen eignete. Aus den Aktivitäten wurden Klassen, aus den Prozessen Objekte, und Simula wurde die erste objektorientierte Programmiersprache.

Anfang der Siebziger ging aus Simula die Sprache Smalltalk hervor. Wie Simula 67 unterstützt Smalltalk die einfache Vererbung, teilweise abstrakte Klassen, das Überschreiben von Methoden. Darüber hinaus wies Smalltalk einige interessante, objektorientierte Eigenheiten auf:

- Alle Attribute (Felder einer Klasse) sind private, alle Methoden public. (Spätere objektorientierte Programmiersprachen wie z.B. Java werden dies auflockern und es dem Programmierer überlassen zu entscheiden, welche Zugriffsrechte für die einzelnen Felder und Methoden gelten sollen.)
- Es gibt bereits eine Ausnahmebehandlung (Exception Handling)

- Der Compiler erzeugt virtuellen Bytecode, der auf die verschiedensten Rechner portiert und dort von passenden Interpretern ausgeführt werden kann (ein Konzept, das von Java übernommen wurde).
- Objekte werden dynamisch reserviert, der Zugriff auf die Objekte erfolgt über Referenzen. Objekte, auf die keine Referenzen verweisen, werden von der automatischen Speicherbereinigung, der "Garbage Collection" entsorgt (ein Konzept, das ebenfalls von Java übernommen wurde).
- Smalltalk führte das Konzept der Nachrichten ein: Soll ein Objekt eine bestimmte Aktion ausführen (eine seiner Methoden aufrufen), "sendet man dem Objekt eine Nachricht". Es gab spezielle Syntaxformen für das Senden von Nachrichten mit einer verschiedenen Anzahl von Argumenten. (In C++ und Java wurde aus dem "Senden von Nachrichten" das Aufrufen von Methoden; die mit Smalltalk eingeführte Terminologie, das Objekte auf Nachrichten agieren, ist in der objektorientierten Literatur aber noch heute weit verbreitet.)

C++ und das Zeitalter der objektorientierten Programmiersprachen

Die ersten objektorientierten Programmiersprachen waren kein großer Erfolg. Dies änderte sich jedoch schlagartig, als Ende der Siebziger der Däne Bjarne Stroustrup C++ entwicklelte. Stroustrup arbeitete zu dieser Zeit in Cambridge an seiner Doktorarbeit über Systemsoftware für verteilte Systeme. Zu diesem Zweck schrieb er einen Simulator, der die Ausführung von Software auf einem verteilten System simulierte. Diesen Simulator schrieb er natürlich in... Simula 67. Stroustrup war begeistert von dieser objektorientierten Sprache. Zum einem konnte er seine Ideen vom Aufbau des Simulators nahezu 1:1 in objektorientierten Code umsetzen (wir erinnern uns: die objektorientierte Programmierung ist der menschlichen Denkweise näher als die imperative), zum andern weil er mit Freude feststellte, dass die Fehlerrate des Programms nicht überproportional mit dem Umfang des Programms anstieg (wie dies bei der imperativen Programmierung üblicherweise der Fall ist). Simula hatte nur zwei Nachteile: der Simula-Linker wurde irgendwann nicht mehr mit dem immer größer werdenden Programm fertig und die Laufzeit des Programms wurde immer schlechter (wir erinnern uns: objektorientierte Programme hatten den Ruf, langsamer in der Ausführung zu sein).

Stroustrup schloss seine Doktorarbeit schließlich ab[20] und wechselte zu den Bell Laboratories in New Jersey, wo er in den Achtzigern die Sprache C

[20] Den Simulator programmierte er neu in BCPL, der Vorgängersprache von C.

um objektorientierte Konzepte erweiterte. So entstand aus C und Stroustrups positiven Erfahrungen mit Simula die Sprache C++ (1985).

C++ ist eine Multiparadigmen-Sprache, denn in C++ kann man sowohl rein imperativ als auch objektorientiert programmieren. Stroustrup wollte nicht nur auf C aufbauen, er wollte alten C-Code auch weiterhin unterstützen, sprich C++ sollte (nahezu vollständig) abwärtskompatibel zu C sein. Letztlich dürfte es genau diese Abwärtskompatibilität gewesen sein, die C++ so erfolgreich machte, denn sie erlaubte den vielen Tausenden C-Programmierern weltweit reibungslos auf C++ umsteigen zu können – ohne bestehende C-Implementierungen gänzlich neu schreiben zu müssen.

C++ und die objektorientierte Programmierung waren bald in aller Munde. Wer Augen hatte zu sehen, der deutete die Zeichen der Zeit: Die Programme wurden immer komplexer, ihre Entwicklung und Wartung immer kostspieliger. Die Grenzen des Wachstums waren für imperativ programmierte Anwendungen erreicht, der Kollaps stand kurz bevor. Das Paradigma der objektorientierten Programmierung versprach dagegen kürzere Entwicklungszeiten, leichtere Wartung, bessere Wiederverwendung. Auf der anderen Seite wurden die PCs immer schneller, die Speicherkapazitäten immer luxuriöser, die Programmiererlöhne immer teurer. Niemand wollte mehr wochenlang einen Assemblerprogrammierer beschäftigen, um 5% Ausführungsgeschwindigkeit herauszuholen, wenn zwei Monate nach Erscheinen der Software ein neuer PC herauskommt, der doppelt so schnell ist, wie die vorangehende Generation. Plötzlich war es allen klar: die Achtziger würden das Zeitalter der objektorientierten Programmierung einläuten.

Nun, C++ wurde ein voller Erfolg und die objektorientierte Programmierung erfuhr durch C++ einen gewaltigen Popularitätsschub, doch das Jahrzehnt der objektorientierten Programmierung wurden die Achtziger nicht. Viele Programmierer verwendeten mittlerweile einen C++-Compiler, doch sie programmierten nicht objektorientiert. Zwar nutzen sie die vordefinierten Klassen aus der Standardbibliothek, weil sich mit diesen viel besser und sicherer programmieren ließ als mit den alten C-Funktionen, doch definierten sie keine eigene Klassen. Große Teile bestehenden C-Codes wurden nicht auf C++ umgestellt (beispielsweise das Microsoft-Betriebssystem Windows), Mängel und Fehler in der Software, die bei konsequent objektorientierter Programmierung vielleicht aufgespürt worden wären, wurden aus Kostengründen in Kauf genommen.

Der Weg zu Java (1995)

Die Sprache Java hieß ursprünglich eigentlich Oak, der Legende nach wegen der großen Eiche, die James Gosling von seinem Büro bei Sun Microsystem aus sah. Oak war für die Programmierung von Software für" intelligente", sprich mit Chips ausgestattete, Anwenderelektronik gedacht. Die Lancierung von Oak scheiterte, doch Sun erkannte das Potenzial der Sprache für Internet und World Wide Web und machte aus Oak Java.

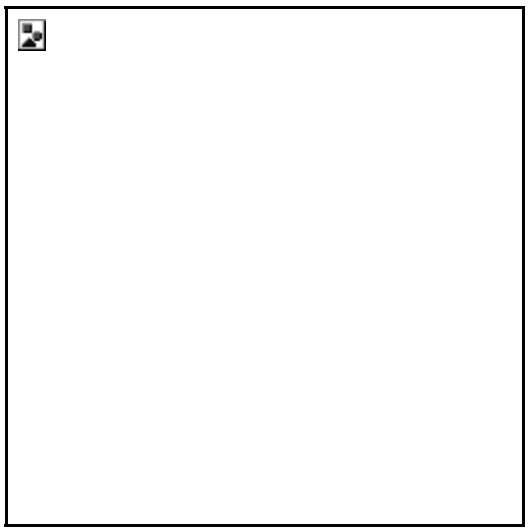

Abbildung 2.2: Der Weg von FORTRAN zu Java

Die Java-Syntax ist eng an C++ angelehnt. Die Entwickler hätten Java natürlich auch nach Smalltalk oder Eiffel modellieren können, doch wäre dann wohl kaum jemand auf die Sprache aufmerksam geworden. Die Anlehnung an C++ hingegen gewährleistete, dass die Sprache von der großen Gemeinde der C++-Programmierer positiv und interessiert aufgenommen wurde.

Java ist im Gegensatz zu C++ keine Multiparadigmen-Sprache, sondern rein objektorientiert. Zudem wurden verschiedene komplizierte, Fehler provozierende Sprachelemente verworfen. So wurde auf Zeiger verzichtet, an ihre Stelle tritt eine automatisch Speicherverwaltung. Die Mehrfachvererbung (eine Klasse wird von mehr als einer Basisklassen abgeleitet), zugunsten des besser beherrschbaren Konzept der Schnittstellenableitung aufgegeben. Templates, ein ebenso kompliziertes wie leistungsfähiges Element von C++, wurden ersatzlos gestrichen.

Java wird wie Smalltalk zu virtuellem Bytecode kompiliert, der auf jedem Rechner mit installiertem Java-Interpreter ausgeführt werden kann – für verteilte Internet-Anwendungen und Applets (kleine Java-Anwendungen, die in Webseiten eingebettet sind) ein absolutes Muss.

Icon NOTE

Würden Java-Anwendungen in prozessorspezifischen Maschinencode übersetzt, könnten sie nur auf Rechner mit passendem Prozessor ausgeführt werden. Angenommen Sie würden auf diese Weise ein Applet für die Intel/Windows-Plattform erstellen und in Ihrem Web veröffentlichen, dann würden Sie ohne Zweifel in kürzester Zeit eine große Zahl begeisterter EMail-Zuschriften von Webbesuchern erhalten, die keinen Intel/Windows-Rechner haben, aber trotzdem ganz gerne ihr Applet sehen würden. Der Trick besteht daher darin, die Umwandlung in Maschinencode einem Interpreter zu überlassen, der auf dem jeweiligen ausführenden Rechner installiert ist. Damit die Umwandlung nicht zu langsam ist, wird der Java-Code vorab in einen maschinencodeähnlichen, aber prozessorunabhängigen Bytecode übersetzt und dieser Bytecode als Applet in die Webseite eingebettet.

Zu guter Letzt sei noch darauf hingewiesen, dass es nur einen Java-Dialekt gibt. Für den Java-Programmierer hat dies den Vorteil, dass er seinen Java-Code mit jedem beliebigen Java-Compiler in Bytecode übersetzen kann.

Ende der Neunziger versuchte Microsoft zwar mit Visual J++ einen eigenen Java-Dialekt zu schaffen und seine Kunden an diesen Dialekt zu binden, doch wurde dieses Unterfangen per Gericht verboten. Seit 2001 läuft nun mit C# und J# ein weiterer Versuch, Java Konkurrenz zu machen.