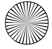
Kap02.fm Seite 81 Dienstag, 7. September 2010 2:21 14

2 Grundlagen der Programmierung

In diesem Kapitel bereiten wir die Grundlagen für ein systematisches Programmieren. Wichtigstes Ziel ist dabei die Herausarbeitung der fundamentalen Konzepte von Programmiersprachen. Wir benutzen bereits weitgehend die Syntax von Java, obwohl in dieser Sprache die Trennlinien zwischen einigen grundlegenden Konzepten von Programmiersprachen, wie z.B. zwischen *Ausdruck* und *Anweisung* nicht mehr so deutlich zu erkennen sind, wie bei der vor allem aus didaktischen Erwägungen konzipierten Sprache Pascal. Gelegentlich stellen wir aber den Java-Notationen die entsprechende Pascal-Syntax gegenüber, auch um zu zeigen, dass nicht immer das aus wissenschaftlicher Sicht bessere Konzept sich auch in der Praxis durchsetzt. Insbesondere was die Syntax einer Sprache angeht, hat Java bewusst an die Sprache C angeknüpft, vor allem, weil für viele C-Programmierer damit der initiale Aufwand, in eine andere Sprache umzusteigen, gering war.

Obwohl wir also Java als Vehikel für die Vorstellung der wichtigsten Programmiersprachenkonzepte benutzen und obwohl wir auch schon zeigen, wie man die vorgestellten Konzepte möglichst einfach testen kann, bleibt eine umfassende Einführung in Java dem folgenden Kapitel vorbehalten.

Wir beginnen mit einer Erläuterung der Begriffe *Spezifikation*, *Algorithmus* und *Abstraktion*. Der Kern einer Programmiersprache, *Datenstrukturen*, *Speicher*, *Variablen* und fundamentale *Kontrollstrukturen* schließt sich an. Einerseits ist unsere Darstellung praktisch orientiert – die Programmteile kann man sofort ausführen – andererseits zeigen wir, wie die Konzepte exakt mathematischen Begriffsbildungen folgen. Der Leser erkennt, wie zusätzliche Kontrollstrukturen aus dem Sprachkern heraus definiert werden, wie Typkonstruktoren die mathematischen Mengenbildungsoperationen nachbilden und wie Ojekte und Klassen eine systematische Programmentwicklung unterstützen.

Zusätzlichen theoretischen Konzepten, *Rekursion* und *Verifikation*, sind jeweils eigene Unterkapitel gewidmet. Der eilige Leser mag sie im ersten Durchgang überfliegen, ein sorgfältiges Studium lohnt sich aber in jedem Fall. Rekursion gibt dem mathematisch orientierten Programmierer ein mächtiges Instrument in die Hand. Ein Verständnis fundamentaler Konzepte der Programmverifikation, insbesondere von *Invarianten*, führt automatisch zu einer überlegteren und zuverlässigeren Vorgehensweise bei der Programmentwicklung.

Kap02.fm Seite 82 Dienstag, 7. September 2010 2:21 14

2.1 Programmiersprachen

Die Anweisungen, die wir dem Computer geben, werden als Text formuliert, man nennt jeden solchen Text ein *Programm*. Programme nehmen Bezug auf vorgegebene Datenbereiche und auf Verknüpfungen, die auf diesen Datenbereichen definiert sind. Allerdings, und das ist ein wichtiger Aspekt, können innerhalb eines Programmes nach Bedarf neue Datenbereiche und neue Verknüpfungen auf denselben definiert werden.

Der Programmtext wird nach genau festgelegten Regeln formuliert. Diese Regeln sind durch die sogenannte Grammatik einer *Programmiersprache* festgelegt. Im Gegensatz zur Umgangssprache verlangen Programmiersprachen das exakte Einhalten der Grammatikregeln. Jeder Punkt, jedes Komma hat seine Bedeutung, selbst ein kleiner Fehler führt dazu, dass das Programm als Ganzes nicht verstanden wird.

In frühen Programmiersprachen standen die verfügbaren Operationen eines Rechners im Vordergrund. Diese mussten durch besonders geschickte Kombinationen verbunden werden, um ein bestimmtes Problem zu lösen. Moderne *höhere Programmiersprachen* orientieren sich stärker an dem zu lösenden Problem und gestatten eine abstrakte Formulierung des Lösungsweges, der die Eigenarten der Hardware, auf der das Programm ausgeführt werden soll, nicht mehr in Betracht zieht. Dies hat den Vorteil, dass das gleiche Programm auf unterschiedlichen Systemen ausführbar ist.

Noch einen Schritt weiter gehen so genannte *deklarative* Programmiersprachen. Aus einer nach bestimmten Regeln gebildeten mathematischen Formulierung des Problems wird automatisch ein Programm erzeugt. Im Gegensatz zu diesen problemorientierten Sprachen nennt man die klassischen Programmiersprachen auch *befehlsorientierte* oder *imperative* Sprachen.

Zu den imperativen Sprachen gehören u.a. *BASIC*, *Pascal*, *C*, *C++* und *Java*, zu den deklarativen Sprachen gehören z.B. *Prolog*, *Haskell* und *ML*. Allerdings sind die Konzepte in der Praxis nicht streng getrennt. Die meisten imperativen Sprachen enthalten auch deklarative Konzepte (z.B. Rekursion), und die meisten praktisch relevanten deklarativen Sprachen beinhalten auch imperative Konzepte. Kennt man sich in einer imperativen Sprache gut aus, so ist es nicht schwer eine andere zu erlernen, ähnlich geht es auch mit deklarativen Sprachen. Der Umstieg von der imperativen auf die deklarative Denkweise erfordert einige Übung, doch zahlt sich die Mühe auf jeden Fall aus. Deklarative Sprachen sind hervorragend geeignet, in kurzer Zeit einen funktionierenden Prototypen zu erstellen. Investiert man dagegen mehr Zeit für die Entwicklung, so gelingt mit imperativen Sprachen oft eine effizientere Lösung.

2.1.1 Vom Programm zur Maschine

Programme, die in einer höheren Programmiersprache geschrieben sind, können nicht unmittelbar auf einem Rechner ausgeführt werden. Sie sind anfangs in einer Textdatei gespeichert und müssen erst in Folgen von Maschinenbefehlen übersetzt werden. *Maschinenbefehle* sind elementare Operationen, die der Prozessor des Rechners unmittelbar ausführen kann.

Kap02.fm Seite 83 Dienstag, 7. September 2010 2:21 14

Wie wir bereits im vorigen Kapitel gesehen haben, beinhalten sie zumindest Befehle, um

- Daten aus dem Speicher zu lesen,
- elementare arithmetische Operationen auszuführen,
- Daten in den Speicher zu schreiben,
- die Berechnung an einer bestimmten Stelle fortzusetzen (Sprünge).

Die Übersetzung von einem Programmtext in eine Folge solcher einfacher Befehle (auch *Maschinenbefehle* oder *Maschinencode* genannt), wird von einem *Compiler* durchgeführt. Das Ergebnis ist ein *Maschinenprogramm*, das in einer als *ausführbar* (engl. *executable*) gekennzeichneten Datei gespeichert ist.

Eine solche ausführbare Datei muss noch von einem *Ladeprogramm* in den Speicher geladen werden, und kann erst dann ausgeführt werden. Ladeprogramme sind im Betriebssystem enthalten, der Benutzer weiß oft gar nichts von deren Existenz. So sind in den Windows-Betriebssystemen ausführbare Dateien durch die Endung „.exe“ oder „.com“ gekennzeichnet. Tippt man auf der Kommandozeile den Namen einer solchen Datei ein und betätigt die „Return“-Taste, so wird die ausführbare Datei in den Hauptspeicher geladen und ausgeführt.

2.1.2 Virtuelle Maschinen

Die Welt wäre einfach, wenn sich alle Programmierer auf einen Rechnertyp und eine Programmiersprache einigen könnten. Man würde dazu nur einen einzigen Compiler benötigen. Die Wirklichkeit sieht anders aus. Es gibt (aus gutem Grund) zahlreiche Rechnertypen und noch viel mehr verschiedene Sprachen. Fast jeder Programmierer hat eine starke Vorliebe für eine ganz bestimmte Sprache und möchte, dass seine Programme auf möglichst jedem Rechnertyp ausgeführt werden können. Bei n Sprachen und m Maschinentypen würde dies $n \times m$ viele Compiler erforderlich machen.

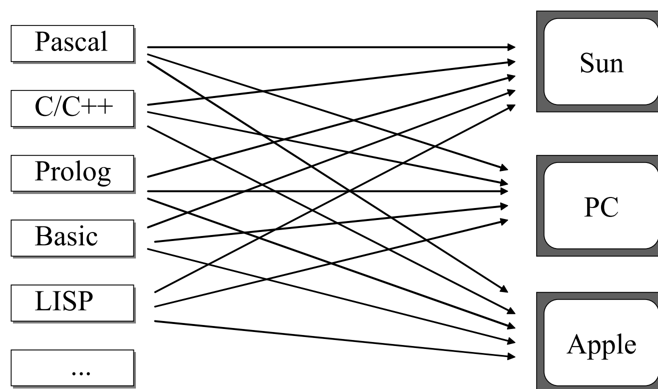

Abb. 2.1: *$n \times m$ viele Compiler*

Kap02.fm Seite 84 Dienstag, 7. September 2010 2:21 14

Schon früh wurde daher die Idee geboren, eine *virtuelle Maschine V* zu entwerfen, die als gemeinsames Bindeglied zwischen allen Programmiersprachen und allen konkreten Maschinensprachen fungieren könnte. Diese Maschine würde nicht wirklich *gebaut*, sondern man würde sie auf jedem konkreten Rechner *emulieren*, d.h. nachbilden. Für jede Programmiersprache müsste dann nur *ein* Compiler vorhanden sein, der Code für V erzeugt. Statt $n \times m$ vieler Compiler benötigte man jetzt nur noch n Compiler und m Implementierungen von V auf den einzelnen Rechnertypen, insgesamt also nur $n + m$ viele Übersetzungsprogramme – ein gewaltiger Unterschied.

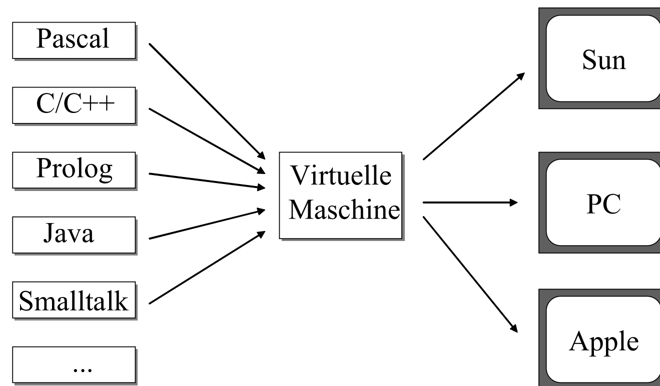

Abb. 2.2: *Traum: Eine gemeinsame virtuelle Maschine*

Leider ist eine solche virtuelle Maschine nie zu Stande gekommen. Neben dem Verdacht, dass ihr Design eine bestimmte Sprache oder einen bestimmten Maschinentyp bevorzugen könnte, stand die begründete Furcht im Vordergrund, dass dieses Zwischenglied die Geschwindigkeit der Programmausführung beeinträchtigen könnte. Außerdem verhindert eine solche Zwischeninstanz, dass spezielle Fähigkeiten eines Maschinentyps oder spezielle Ausdrucksmittel einer Sprache vorteilhaft eingesetzt werden können.

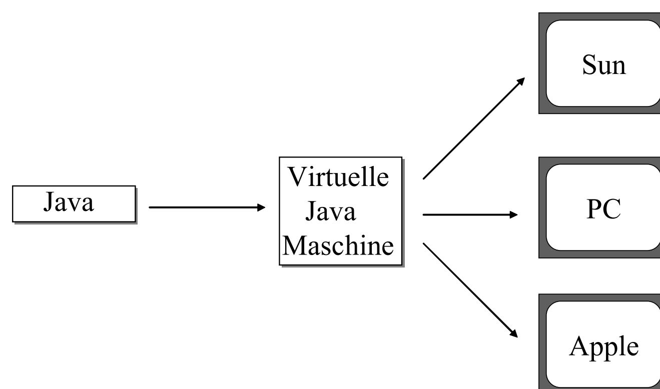

Abb. 2.3: *Realität: Virtuelle Java-Maschine*

Kap02.fm Seite 85 Dienstag, 7. September 2010 2:21 14

Im Zusammenhang mit einer festen Sprache ist das Konzept einer virtuellen Maschine jedoch mehrfach aufgegriffen worden – jüngst wieder in der objektorientierten Sprache *Java*, der das ganze nächste Kapitel gewidmet sein wird. Ein Java-Compiler übersetzt ein in Java geschriebenes Programm in einen Code für eine *virtuelle Java-Maschine* (JVM). Auf jeder Rechnerplattform, für die eine Implementierung für diese virtuelle Java-Maschine verfügbar ist, wird das Programm dann lauffähig sein. Weil man also bewusst auf die Ausnutzung besonderer Fähigkeiten spezieller Rechnertypen verzichtet, wird die Sprache *plattformunabhängig.* Eine virtuelle Maschine allein für Windows Betriebssysteme, die dafür aber eine Zwischensprache für eine großes Spektrum von Hochsprachen bereitstellt – von Visual Basic über C++ bis C# (Microsoft's Java Konkurrent) – bietet die *.NET-Plattform* von Microsoft.

2.1.3 Interpreter

Ein *Compiler* übersetzt immer einen kompletten Programmtext in eine Folge von Maschinenbefehlen, bevor die erste Programmanweisung ausgeführt wird. Ein *Interpreter* dagegen übersetzt immer nur eine einzige Programmanweisung in ein kleines Unterprogramm aus Maschinenbefehlen und führt dieses sofort aus. Anschließend wird mit der nächsten Anweisung genauso verfahren. Interpreter sind einfacher zu konstruieren als Compiler, haben aber den Nachteil, dass ein Befehl, der mehrfach ausgeführt wird, jedesmal erneut übersetzt werden muss.

Grundsätzlich können fast alle Programmiersprachen compilierend oder interpretierend implementiert werden. Trotzdem gibt es einige, die fast ausschließlich mit Compilern arbeiten. Dazu gehören *Pascal*, *Modula*, *COBOL*, *Fortran*, *C*, *C++* und *Ada*. Andere, darunter *BASIC*, *APL*, *LISP* und *Prolog*, werden überwiegend interpretativ bearbeitet. Sprachen wie *Java* und *Smalltalk* beschreiten einen Mittelweg zwischen compilierenden und interpretierenden Systemen – das Quellprogramm wird in Code für die virtuelle Java- bzw. Smalltalk-Maschine, so genannten *Bytecode,* compiliert. Dieser wird von der virtuellen Maschine dann interpretativ ausgeführt. Damit ist die virtuelle Maschine nichts anderes als ein Interpreter für Bytecode.

2.1.4 Programmieren und Testen

Ein Programm ist ein Text und wird wie jeder Text mit einem Textverarbeitungsprogramm erstellt und in einer Datei gespeichert. Anschließend muss es von einem Compiler in Maschinencode übersetzt werden. Üblicherweise werden während dieser Übersetzung bereits Fehler erkannt. Die Mehrzahl der dabei entdeckten Fehler sind so genannte *Syntaxfehler.* Sie sind Rechtschreib- oder Grammatikfehlern vergleichbar – man hat sich bei einem Wort vertippt oder einen unzulässigen Satzbau (*Syntax*) verwendet. Eine zweite Art von Fehlern, die bereits beim Compilieren erkannt werden, sind *Typfehler.* Sie entstehen, wenn man nicht zueinander passende Dinge verknüpft – etwa das Alter einer Person zu ihrer Hausnummer addiert oder einen Nachnamen an einer Stelle speichert, die für eine Zahl reserviert ist. Programmiersprachen unterscheiden sich sehr stark darin, ob und wie sie solche Fehler erkennen. Syntaxfehler kann man sofort verbessern und dann einen erneuten Compilierversuch machen. Sobald das Programm fehlerlos compiliert wurde, liegt es als Maschinenprogramm vor und kann testweise ausgeführt werden.

Kap02.fm Seite 86 Dienstag, 7. September 2010 2:21 14

Auch ein fehlerfrei compiliertes Programm kann noch Fehler enthalten.

- *Laufzeitfehler* entstehen, wenn beispielsweise zulässige Wertebereiche überschritten werden, wenn durch 0 dividiert oder die Wurzel einer negativen Zahl gezogen wird. Laufzeitfehler können i.A. nicht von einem Compiler erkannt werden, denn der konkrete Zahlenwert, mit dem gearbeitet wird, steht oft zur *Compilezeit* nicht fest, sei es, weil er von der Tastatur abgefragt oder sonstwie kompliziert errechnet wird.
- *Denkfehler* werden sichtbar, wenn ein Programm problemlos abläuft, aber eben nicht das tut, was der Programmierer ursprünglich im Sinn hatte. Denkfehler können natürlich nicht von einem Compiler erkannt werden.

Einen Fehler in einem Programm nennt man im englischen Jargon auch *bug*. Das Suchen und Verbessern von Fehlern in der Testphase heißt konsequenterweise *debugging*. Laufzeitfehler und Denkfehler können bei einem Testlauf sichtbar werden, sie können aber auch alle Testläufe überstehen. Prinzipiell gilt hier immer die Aussage von E. Dijkstra:

Durch Testen kann man die Anwesenheit, nie die Abwesenheit von Fehlern zeigen.

Dennoch werden bei den ersten Tests eines Programms meist Fehler gefunden, die dann einen erneuten Durchlauf des Zyklus *Editieren – Compilieren – Testen* erforderlich machen. Die Hoffnung ist, dass dieser Prozess zu einem Fixpunkt, dem korrekten Programm, konvergiert.

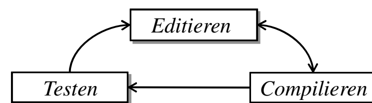

Abb. 2.4: *Zyklus der Programmentwicklung*

2.1.5 Programmierumgebungen

Interpretierende Systeme vereinfachen die Programmerstellung insofern, als die Compilationsphase entfällt und auch kleine Programmteile interaktiv getestet werden können, sie erreichen aber nur selten die schnelleren Programmausführzeiten compilierender Systeme. Außerdem findet bei vielen interpretierenden Systemen keine Typüberprüfung statt, so dass Typfehler erst zur Laufzeit entdeckt werden.

Einen Kompromiss zwischen interpretierenden und compilierenden Systemen stellte als erstes das *Turbo-Pascal* System dar. Der in das Entwicklungssystem eingebaute Compiler war so schnell, dass der Benutzer den Eindruck haben konnte, mit einem interpretierenden System zu arbeiten. Für fast alle Sprachen gibt es heute ähnlich gute „integrierte Entwicklungssysteme“ (engl.: *integrated development environment – IDE*), die alle zur Programmerstellung notwendigen Werkzeuge zusammenfassen:

- einen Editor zum Erstellen und Ändern eines Programmtextes,
- einen Compiler bzw. Interpreter zum Ausführen von Programmen,
- einen *Debugger* für die Fehlersuche in der Testphase eines Programms.