Einführung in die Informatik

Klaus Knopper

09.11.2004

Programmiersprachen

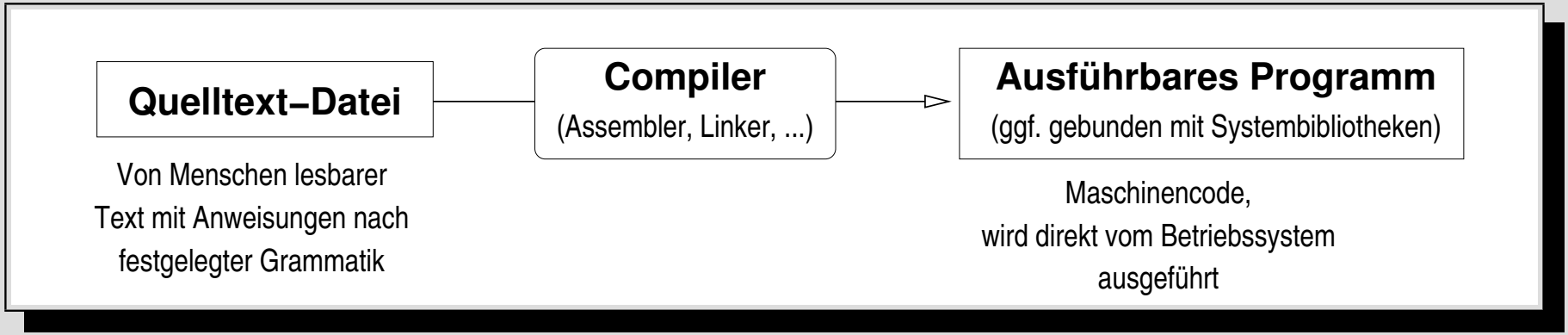
Eine Programmiersprache ist eine formale Sprache zur Darstellung (Notation) von Computerprogrammen. Sie vermittelt dem Computersystem durch von einem Menschen geschriebenen Text genaue Angaben zu einer Kette von internen Verarbeitungsschritten, beteiligten Daten und deren Struktur in Abhängigkeit von internen oder externen Ereignissen.

Da digitale Computer intern nur die Werte 0 und 1 verarbeiten, wäre es nach heutigen Maßstäben extrem umständlich und mühsam, die vielen Formen der Informationsverarbeitung als Binärzahlen einzugeben.

Idee: Hilfswerkzeuge benutzen.

Compiler

Wird ein Programmtext als Ganzes übersetzt, spricht man in Bezug auf den Übersetzungsmechanismus von einem **Compiler**. Der Compiler ist selbst ein Programm, welches als Dateneingabe den menschenlesbaren Programmtext bekommt und als Datenausgabe den Maschinencode liefert, der direkt vom Prozessor verstanden wird (zum Beispiel Objectcode, EXE-Datei) oder der in einer Laufzeitumgebung in einer „virtuellen Maschine“ (zum Beispiel JVM oder .NET) ausgeführt wird.



Interpreter

Wird ein Programmtext Schritt für Schritt übersetzt und der jeweils übersetzte Schritt sofort ausgeführt, spricht man von einem **Interpreter**. Interpretierte Programme laufen meist langsamer als kompilierte.

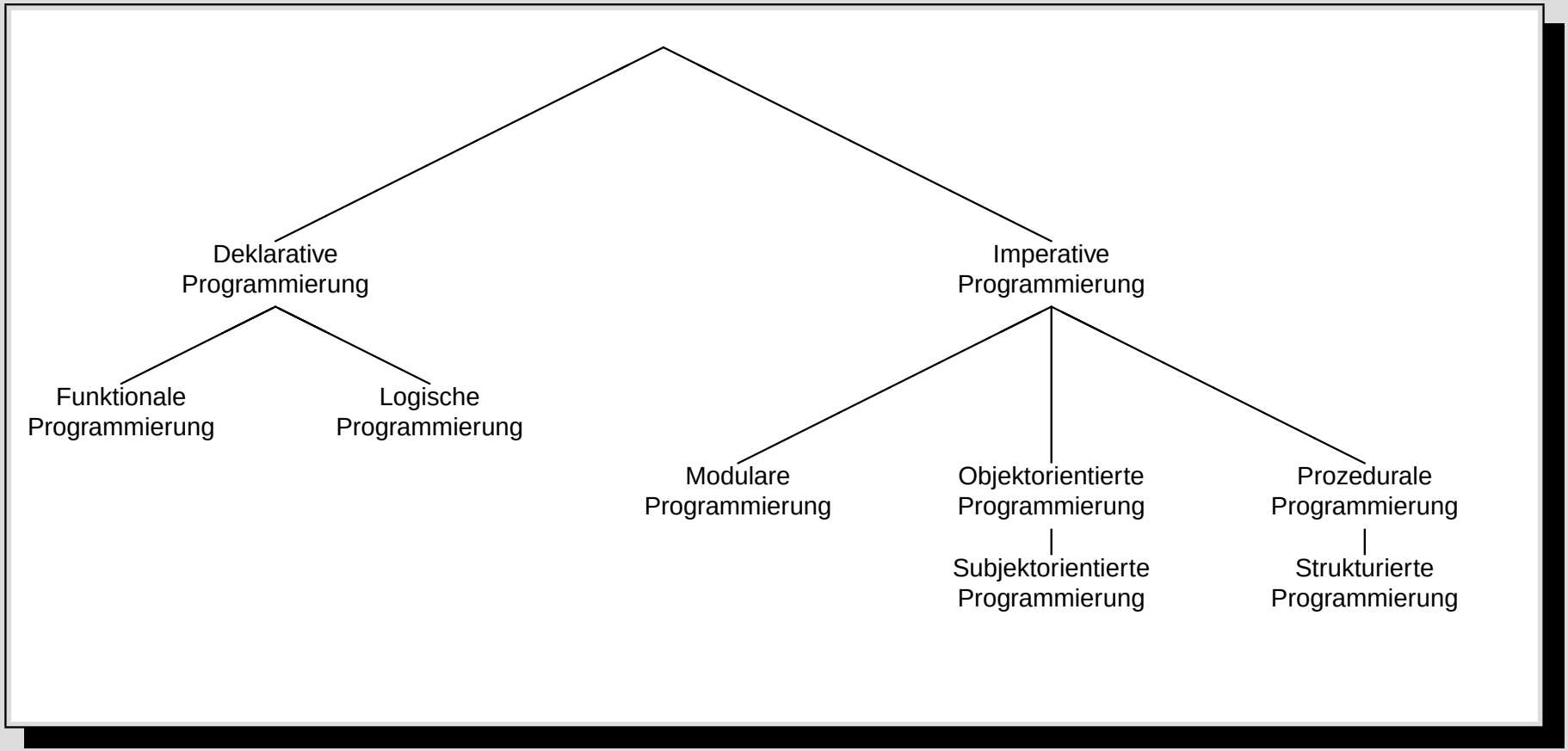
Programm, Programmcode, Quelltext

Eine logische Abfolge von Befehlen in einer Programmiersprache nennt man allgemein **Programm**, **Programmcode** oder **Quelltext**. (Quelltext betont besonders die Lesbarkeit). Dieser wird von Programmierern verfasst.

Programmierparadigma

Ein **Programmierparadigma** ist das einer Programmiersprache oder Programmiertechnik zugrundeliegende Prinzip. Das Wort **Paradigma** kommt aus dem Griechischen und bedeutet Beispiel, Vorbild, Muster oder auch Abgrenzung.

Programierparadigmen



Diese Darstellung ist bei weitem nicht vollständig!

Beispiel

Die Programmiersprache **Java** folgt den Paradigmen der **objektorientierten** und der **attributorientierten** Programmierung, die Programmiersprache **C++** ermöglicht **objektorientierte** Programmierung und **prozedurale** Programmierung, beide Sprachen entsprechen dem Paradigma der **imperativen** Programmierung und ermöglichen **generische** Programmierung.

Programmierparadigmen . . .

- . . . bestimmen den Aufbau des Quelltextes,
- . . . sind die Philosophie des Programmierens,
- . . . müssen vom Programmierer verstanden werden.
- . . . können in Programmiersprachen kombiniert werden.

Frage: Wie charakterisieren sich nun die verschiedenen Programmierparadigmen?

Deklarative Programmierung

Bei der **deklarativen Programmierung** wird nicht beschrieben, wie etwas zu geschehen hat bzw. was zu tun ist, sondern nur welches Ergebnis gewünscht ist. Es sollte also nicht mehr der Lösungsweg programmiert werden, sondern nur noch angegeben werden, welches Ergebnis gewünscht wird.

Als Verwirklichungen dieses Grundgedankens sind die **funktionale Programmierung** und die **logische Programmierung** hervorgegangen. Wir betrachten hier lediglich kurz die Prinzipien der funktionalen Programmierung.

Funktionale Programmierung

- Ein funktionales Programm ist eine Abbildung von Eingabedaten auf Ausgabedaten.
- In einem funktionalen Programm wird die *Reihenfolge der Berechnungsschritte* in der Regel *nicht* festgelegt.
- Funktionale Programmiersprachen erlauben es, Funktionen (wie Daten) als Argumente und Rückgabewerte anderer Funktionen zu behandeln. Dadurch ist es einfach, Operatoren auf Funktionen zu definieren. Dies macht funktionale Programme oft kürzer und abstrakter, erfordert jedoch oft eine Umgewöhnungszeit für Programmierer, die den funktionalen Programmierstil nicht gewohnt sind.

strict evaluation vs. lazy evaluation

Funktionale Sprachen kann man auch nach ihrer Auswertungsstrategie unterscheiden: Bei strenger Auswertung (engl. eager bzw. **strict evaluation**) werden Ausdrücke sofort ausgewertet. Dem gegenüber steht die Bedarfsauswertung (engl. **lazy evaluation**), bei der Ausdrücke erst ausgewertet werden, wenn deren Wert in einer Berechnung benötigt wird. Dadurch lassen sich z.B. unendlich große Datenstrukturen (die Liste aller natürlicher Zahlen, die Liste aller Primzahlen, etc.) definieren und bestimmte Algorithmen vereinfachen sich.

Typisierung

Weiterhin kann man funktionale Sprachen einteilen in **dynamisch** und **statisch typisierte Sprachen**, die sich dadurch ergeben, dass die Typprüfung während der Laufzeit bzw. während der Übersetzungszeit stattfinden kann.

Eine Typisierung ist vergleichbar mit der Festlegung von Definitions– und Wertebereich bei mathematischen Funktionen.

Algorithmen und funktionales Programmieren

Der Verzicht auf zerstörerische Zuweisungen führt dazu, das etliche klassische Algorithmen und Datenstrukturen, die regen Gebrauch von dieser Möglichkeit machen, so nicht für funktionale Sprachen verwendet werden können und man nach neuen Lösungen suchen muss.

„Auch wenn die meisten dieser Bücher [über Datenstrukturen und Algorithmen] behaupten, das sie unabhängig von einer bestimmten Programmiersprache sind, so sind sie leider nur sprachunabhängig im Sinne Henry Fords: Programmierer können jede Programmiersprache benutzen, solange sie imperativ ist.“
– Chris Okasaki

Funktionale Programmiersprachen

- LISP
- Scheme
- OCaml
- Haskell
- Erlang
- XSLT

Haskell

- Haskell ist eine rein funktionale Programmiersprache (erlaubt somit ausschließlich das funktionale Programmierparadigma).
- Haskell unterstützt lazy evaluation.
- Haskell ist ein Interpreter.
- Haskell unterstützt statische Typisierung.

Die folgenden Beispiele wurden mit der Haskell-Implementierung `hugs` erstellt.

Auswerten von Ausdrücken

Das Berechnungsmodell von Haskell ist denkbar einfach: ein gegebener Ausdruck wird zu einem Wert (auch: Normalform) reduziert.

```
Prelude> 5+2+3
10
Prelude> sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
1
```

Wir sagen: ’`sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3`’ reduziert zu ’1’.

Rekursive Funktionen

Rekursion bedeutet Selbstbezüglichkeit (von lateinisch recurrere = zurücklaufen). Sie tritt immer dann auf, wenn etwas auf sich selbst verweist. Ein rekursives Element muss nicht immer direkt auf sich selbst verweisen (direkte Rekursion), eine Rekursion kann auch über mehrere Zwischenschritte entstehen. Rekursion kann dazu führen, dass merkwürdige Schleifen entstehen. So ist z.B. der Satz „Dieser Satz ist unwahr“ rekursiv, da er von sich selber spricht. Eine etwas subtilere Form der Rekursion (indirekte Rekursion) kann auftreten, wenn zwei Dinge gegenseitig aufeinander verweisen. Ein Beispiel sind die beiden Sätze: „Der folgende Satz ist wahr“ „Der vorhergehende Satz ist nicht wahr“. Die Probleme beim Verständnis von Rekursion beschreibt der Satz: „Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen“.

Beispiel: Kaninchenpopulation (1)

Der Mathematiker Fibonacci stieß bei der einfachen mathematischen Modellierung des Wachstums einer Kaninchenpopulation nach folgender Vorschrift:

1. Zu Beginn gibt es ein Paar neugeborener Kaninchen.
2. Jedes neugeborene Kaninchenpaar wirft nach 2 Monaten ein weiteres Paar.
3. Anschließend wirft jedes Kaninchenpaar jeden Monat ein weiteres.
4. Kaninchen leben ewig und haben einen unbegrenzten Lebensraum.

. . .

Kaninchenpopulation (2)

. . . auf die nach ihm benannte rekursive Fibonacci-Folge:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n+2) &= f(n) + f(n+1) \end{aligned}$$

Kaninchenpopulation (3)

Zur Bestimmung der Elemente der Folge läßt sich nun folgende mathemethische Funktion fib : $IN^+ \rightarrow IN$ ableiten:

$$\text{fib}(n) = \begin{cases} 0 & \text{falls} \quad n = 0 \quad \text{(Rekursionsanfang)} \\ 1 & \text{falls} \quad n = 1 \quad \text{(Rekursionsanfang)} \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{falls} \quad n \geq 2 \quad \text{(Rekursionsschritt)} \end{cases}$$

In Haskell lässt sich die Fibonacci-Folge folgendermaßen berechnen:

```
fib :: Integer -> Integer
fib(0) = 0
fib(1) = 1
fib(n+2) = fib(n) + fib(n+1)
```

Zusammenfassung

Programmieren mit Funktionen bedeutet:

- Einfache Funktionen werden zu komplexen Funktionen zusammengesetzt, diese werden wiederum zu komplexeren Funktionen zusammengesetzt und so weiter.
- Am Schluss existiert eine Funktion, die das Ergebnis liefert.
- Wer mathematische Funktionen beschreiben kann, kann auch mit Funktionen programmieren.
- Quellcode ist sehr übersichtlich und in der Regel sehr kurz.

Fazit funktionale Programmiersprachen

- Funktionale Programmiersprachen haben in der Regel einen hohen Speicherverbrauch beim Interpretieren der Programme. Die Ausführungs-Geschwindigkeit ist in der Regel langsam.
- Programmieren mit Funktionen wird auch tatsächlich hin und wieder in der Praxis eingesetzt, z.B. bei Intel (Mikroprozessor-Verifikation)

Ausblick

Nach dem funktionalen Programmierparadigma wird noch das imperative Programmierparadigma vorgestellt. Imperative Programmiersprachen basieren auf der genauen Festlegung, welcher Befehl oder welche Anweisung wann ausgeführt wird. Wesentlich ist dabei die *Reihenfolge* von Anweisungen, und Kontrollstrukturen, die den Ablauf durch Bedingungen steuern. Hierzu wird die Programmiersprache C eingeführt.

Im weiteren Verlauf der Vorlesung wird auch noch das objektorientierte Programmierparadigma vorgestellt. Dieses Paradigma wird mit Hilfe der Programmiersprache Java praktisch begleitet.