Mit Stellenanzeigen

iX DEVELOPER

Digitale Ausgabe
€ 8,99

1/2013

Gewinnspiel:
Notebook zu gewinnen

Zum Download:

Über 8 GByte Software für Entwickler

Intel Parallel Studio XE 2013 Update 1, Visual Studio 2012 für Web, Windows 8 und Desktop, Eclipse Orion 1.0, Cloudera's Distribution Including Hadoop (CDH) 4.0.1, 37 Episoden des SoftwareArchitekTOUR-Podcasts

Sponsored Software:
Intel Corp.,
h. o. Computer

Programmieren heute

Programmiersprachen im Wandel:

Funktional vs. objektorientiert

Professionelles JavaScript

CoffeeScript, TypeScript, Dart

App-Entwicklung:

Design Patterns für Mobile

WinRT als Evolution von .NET

Big Data und moderne Datenverarbeitung:

Apache Hadoop, NoSQL, NewSQL, In-Memory

IT-Hypes unter der Lupe:

GPU Computing mit CUDA und OpenCL

Agile IT mit DevOps

Präsentiert von: heise Developer www.heise-developer.de

NEU!

Windows Server 2012

Standard Edition für **19,99 €** im Monat

webtropia.com

Brandneuer HP ProLiant DL 360e Gen 8 Server

Auf Wunsch kostenlose Virtualisierungs-Lösung mit vmware vSphere 5

	Enterprise-Server S 2.0	Enterprise-Server M 2.0	Enterprise-Server L 2.0
CPU	1 x intel XEON E5504	1 x Intel XEON E5504	1 x Intel XEON E5-2420 (neu)
Leistung	1 x 4 x 2,0 GHz	1x 4 x 2,0 GHz	1 x 6 x 1,9 GHz mit HT
CPU-Ausbau bis zu	2 x 4 x 2,4 GHz mit HT	2 x 4 x 2,4 GHz mit HT	2 x 6 x 2,4 GHz mit HT
RAM	8 GB DDR3 ECC	16 GB DDR3 ECC	32 GB DDR3 ECC
RAM Ausbau bis zu	32 GB DDR3 ECC	192 GB DDR3 ECC	384 GB DDR3 ECC
Festplatten (7.200 rpm)	2 x 500 GB Enterprise-Edition	2 x 1.000 GB Enterprise-Edition	2 x 1.000 GB Enterprise-Edition
Festplatten-Ausbau bis zu	4 x 2.000 GB Enterprise-Edition	4 x 2.000 GB Enterprise-Edition	4 x 3.000 GB Enterprise-Edition
Optionaler Raid Controller	✓	✓	✓
KVM over IP enthalten	✓	✓	✓
Anbindung	1.000 MBit Flatrate	1.000 MBit Flatrate	1.000 MBit Flatrate
VLANs möglich	✓	✓	✓
IPv4 Subnetz (/29) inkl.	✓	✓	✓
IPv6 Subnetz (/64) inkl.	✓	✓	✓
Extras	250 GB Backup-Speicher, Monitoring, Reset- und Rescue-System		
Vertragslaufzeit	1 Monat	1 Monat	1 Monat
Monatsgrundgebühr ab	59,99 €	89,99 €	139,99 €
Einrichtungsgrundgebühr	0,00 €	0,00 €	0,00 €

Jetzt informieren und bestellen Tel.: 0211 / 545 957 - 330 www.webtropia.com

DENIC.de

Parallels PLATINUM PARTNER

Microsoft CERTIFIED SPLA MEMBER

powered by hp

Vielfalt erwünscht

Programmiersprachen bekommen allem Anschein nach wieder eine größere Bedeutung, nachdem für einige Zeit der Fokus vieler vermehrt auf funktionsreichen Tools und Frameworks lag. Das liegt sicherlich auch daran, dass nach langer Entwicklung innerhalb kurzer Zeit neue Sprachversionen der verbreiteten Programmierplattformen C, C++ und Java erschienen sind. Die lange Entwicklungszeit nicht nur bei diesen „Großen" war aber einer der Gründe, warum auf einmal neue verheißungsvolle Sprachen die Bühne betreten haben. Für die einen dauerte es zu lange, bis ein als wichtig erachtetes Sprach-Feature oder ein den Umständen angemessener Umgang mit Anforderungen – etwa durch das Aufkommen der Mehrkernsysteme – in den Sprachstandard einzog. Andere vermissten grundsätzliche Konzepte wie den funktionalen Programmierstil in ihren bis dato präferierten Sprachen.

Im Umfeld der Java-Plattform dürfen hier Scala, Groovy, JRuby und Clojure bereits als arriviert gelten, neuerdings hinzugekommen sind das von Red Hat entwickelte Project Ceylon und JetBrains' Kotlin, die wiederum aggressiv gegen die zuerst genannten Sprachen positioniert werden. Im Umfeld der C/C++-Programmierung haben vor allem D und Go die Aufmerksamkeit auf sich gezogen. Während es um D mittlerweile wieder still geworden ist, taucht das von Google entworfene Go regelmäßig in der Berichterstattung auf und gewinnt nicht nur an Interesse, sondern auch an Verbreitung. Letztlich ist jedoch allen Emporkömmlingen anzusehen, woher sie stammen, und daran ist abzulesen, dass die im Artikel auf Seite 8 skizzierte Basisarbeit in den früheren Jahrzehnten keine so schlechte gewesen sein kann.

Die neue Konkurrenz ist jedoch insofern zu begrüßen, dass viele der in den „neuen" Sprachen erarbeiteten Ideen über kurz oder lang in den „Muttersprachen" landen. Scala und natürlich Clojure sind implizit mit dem Einsatz der aus der funktionalen Programmierung stammenden Closures vertraut. Diese ziehen nun nach mehrmaligem Anlauf endlich (als Project Lambda) in Java 8 ein. Der Einfluss der Konkurrenz oder der Alternativen lässt sich auch bei der Weiterentwicklung von JavaScript beziehungsweise ECMAScript beobachten, in deren Version 6 bereits in CoffeeScript und TypeScript umgesetzte Ideen wie Klassen und Klassenvererbung landen sollen. Dessen ungeachtet ist JavaScript so etwas wie der neue unerwartete Stern am Programmiersprachenhimmel. Microsoft hat es zum „first-class citizen" in Visual Studio 2012 ernannt – die Windows Store Apps lassen sich wohl bevorzugt in dieser Skriptsprache schreiben. Selbst in der Entwicklung für mobile Geräte hat sich das Dreigespann aus HTML5, CSS und JavaScript zur dritten ernst zu nehmenden Plattform gemausert.

Nun heißt das nicht, dass der Entwickler gleich sieben Programmiersprachen in sieben Wochen erlernen muss,[1] vielleicht reicht es schon, sich für jedes Jahr gewissenhaft eine neue Sprache vorzunehmen, wie es Eberhard Wolff, Autor in diesem Sonderheft, zu einem früheren Zeitpunkt geschrieben hat. Das vierte „Programmieren heute"-Sonderheft mag als Anreiz dienen, sich mit der einen oder anderen Sprache eingehender zu beschäftigen.

Selbiges gilt für die anderen Themen des Hefts: Beispielhaft sei der gegenwärtige Hype Big Data genannt, zu dem die Redaktion nahezu täglich neue Pressemeldungen erhält. Für Apache Hadoop, das als De-facto-Standard für Big-Data-Projekte gelten darf, haben Marktforscher prognostiziert, dass der Markt rund um das MapReduce-Framework bis 2016 auf etwas mehr als 810 Millionen US-Dollar anwachsen werde. Das entspräche einer jährlichen Steigerungsrate von rund 60 Prozent. Angesichts der weiter fortschreitenden Explosion der Datenmengen mag es also geraten sein, sich mit dem ab Seite 84 beschriebenen Hadoop-Ökosystem vertraut zu machen.

iX und *heise Developer* wünschen viel Spaß beim Vertiefen aller Trendthemen des *iX-Developer*-Sonderhefts.

ALEXANDER NEUMANN

[1] Bruce A. Tate; Seven Languages in Seven Weeks; A Pragmatic Guide to Learning Programming Languages; Pragmatic Programmers, 2010

Programmiersprachen

Mit neuen Vorgehensweisen oder dem bewussten Rückgriff auf bewährte Ideen versucht man in der Softwareentwicklung, den aktuellen Marktanforderungen besser gerecht zu werden. Der Trend macht sich auch bei den Programmiersprachen deutlich bemerkbar, vor allem bei JavaScript, das nun professionellen Ansprüchen genügt.

ab Seite 7

Big Data

Die größte Bedeutung der NoSQL-Bewegung liegt darin, den alteingesessenen Datenbanken ihre Schwachstellen aufgezeigt zu haben. Genau das hat zu viel Bewegung auf „beiden Seiten" geführt. Gemein ist ihnen das Bestreben, Anbindungen an den prominentesten Vertreter der Big-Data-Welt zu schaffen – an Apache Hadoop.

ab Seite 67

Programmiersprachen

Sprachkonzepte
Programmiersprachen im Wandel der Zeit 8

Sprachen-Design
Grundsätzliches zur Entwicklung von Programmiersprachen 12

Funktionale Programmierung
Wo Haskell und Co objektorientierten Sprachen überlegen sind 18

C++-Entwicklung
C++11 – der Sprachstandard ein Jahr später 26

Java-Entwicklung
Gegenwart und Zukunft der Java-Plattform 30

Java Virtual Machine
Programmiersprachen für die JVM im Zusammenspiel mit Java 36

Professionelles JavaScript
Skriptsprache für Industrie- und Geschäftsanwendungen 40

JavaScript-Alternativen
CoffeeScript – das bessere JavaScript? 48

TypeScript: Microsofts neue typisierte Skriptsprache 54

Dart – Googles zweite Programmiersprache 58

Serverside JavaScript
Zeitgemäße Webanwendungen mit Node.js entwickeln 62

Big Data

DBMS
NoSQL-Datenbanken ergänzen relationale Datenbank-Systeme 68

NewSQL
Relationale Datenbanken schlagen zurück 74

Hauptspeicherdatenbanken
In-Memory Computing als Treiber neuartiger Geschäftsanwendungen 78

Interview
Franz Färber im Gespräch über In-Memory Computing und SAP HANA 82

Hadoop
Das Ökosystem der MapReduce-Implementierung unter dem Apache-Dach 84

Big Data im Praxiseinsatz 88

Einblick in die nächste Generation des Apache-Frameworks 94

GUI

Design Patterns für Mobile
Wege für eine erfolgreiche mobile Oberflächenentwicklung 100

Mobilprogrammierung
Best Practices für die Entwicklung mobiler Unternehmens-Apps 105

GUIs

Apps sind speziell für den Einsatz auf mobilen Endgeräten konzipiert. Sie lassen sich im Allgemeinen gegenüber Desktop-Anwendungen einfacher bedienen und bieten grundsätzlich andere Einsatzmöglichkeiten. Doch stopp! – Spätestens mit Windows 8 halten die Prinzipien mobiler Entwicklung auch auf dem Desktop Einzug.

ab Seite 99

Parallelprogrammierung

Programmierbare Grafikkarten ermöglichen große Geschwindigkeitssteigerungen im Vergleich zu Multicore-Systemen. Da sich inzwischen GPU-Kerne in den CPUs finden, wird nach der gewöhnlichen Parallelprogrammierung das GPU Computing künftig ein fester Bestandteil der Softwareentwicklung sein.

ab Seite 127

Windows 8
Windows Store Apps – Überblick der Frontend-Techniken **112**

Interview
Brandon Bray im Gespräch zu .NET und WinRT **119**

Web-Clients
Wie HTML5 Rich Internet Applications verändert **122**

Parallelprogrammierung

Multicore
Programmieren mit Tasks und parallelen Schleifen **128**

GPU Computing
Grundlagen der GPU-Programmierung **134**

GPU-Frameworks
GPU-Programmierung in der Praxis mit CUDA **139**

Heterogene Programmierung mit OpenCL **145**

Trends

Agilität
Entwicklung und Betrieb verzahnen mit DevOps **154**

Eingebettete Systeme
Trends der Embedded-Entwicklung **159**

Spieleentwicklung
Der Unterbau von Computerspielen **164**

Unternehmensprozesse
Crowdsourcing – absehbarer Trend in der Softwareentwicklung **168**

Sonstiges

Editorial **3**

DVD-Inhalt **6**

Gewinnspiel **151**

Inserentenverzeichnis **151**

Impressum **151**

Alle Links: www.ix.de/ix1217SSS Artikel mit Verweisen ins Web enthalten am Ende einen Hinweis darauf, dass diese Webadressen auf dem Server der iX abrufbar sind. Dazu gibt man den iX-Link in der URL-Zeile des Browsers ein. Dann kann man auch die längsten Links bequem mit einem Klick ansteuern. Alternativ steht oben rechts auf der iX-Homepage ein Eingabefeld zur Verfügung.

Auf der Heft-DVD

Sponsored Software

Intel Parallel Studio XE 2013 Update 1

Die Toolsuite für die Parallelprogrammierung kombiniert Intels C++- und Fortran-Compiler, Performance- und Parallel-Bibliotheken mit den Profiling-Werkzeugen Intel Inspector XE und Intel VTune Amplifier XE. Die neue Version enthält jetzt auch Intels Parallel Advisor und unterstützt die neuen Intel-Prozessoren einschließlich Intel Xeon Phi.

Auf der Heft-DVD findet man kombinierte 32- und 64-Bit-Versionen für Windows und Linux. Diese sind identisch mit den Kaufversionen, laufen allerdings nur 30 Tage. (Hierfür ist explizit die Option „30-Tage-Eval" auszuwählen.) Die Windows-Version setzt zudem die Installation von Visual Studio 2010 voraus.

Tools

Visual Studio 2012 für Web, Windows 8 und Desktop

Microsofts Entwicklungsumgebung in den kostenlosen Express-Versionen für Web-, Desktop- und Windows-8-Entwickler als ISO-Image.

Eclipse Orion 1.0

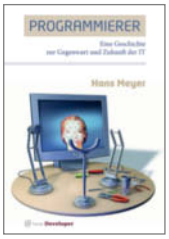
Die sich den Browser integrierende IDE eignet sich für die Entwicklung mit JavaScript, HTML und CSS.

Cloudera's Distribution Including Hadoop (CDH) 4.0.1

Die Open-Source-Sammlung umfasst die Kernkomponenten von Hadoop zur verteilten und ausfallsicheren Speicherung und Verarbeitung großer Datenmengen. Darüber hinaus sind Werkzeuge für Sicherheit, Hochverfügbarkeit und Integration in bestehende Datenlandschaften enthalten. CDH ist unter der Apache-Lizenz verfügbar und eine der verbreitetsten Hadoop-Distribution und wird für Tests, Entwicklung und den Betrieb großer Cluster verwendet. Auf der DVD befindet sich ein VMware-Image, das insbesondere Einsteigern helfen kann, sich schnell einen Überblick zu verschaffen.

Literatur

Programmierer – eine Geschichte zur Gegenwart und Zukunft der IT



Eine fiktionale Geschichte zur Softwareprogrammierung in den Formaten PDF, EPUB oder Mobi (Kindle): Genauer gesagt präsentiert der Protagonist im Rahmen einer großen CeBIT-Show seine bahnbrechenden Erfindungen in den Gebieten Künstliche Intelligenz und Robotik. Autor der 76 Seiten langen Geschichte ist Hans Meyer, ein erfahrener Softwareentwickler und IT-Projektleiter, der seine rund 40 Jahre Erfahrung in der Softwareentwicklung auf teilweise bitter-ironische, aber auch gesellschaftskritische Weise in seine Programmierergeschichte einfließen lassen hat.

SoftwareArchitekTOUR

37 Episoden des renommierten Podcasts auf heise Developer zu Themen wie:

- ROCA (Resource-oriented Client Architecture)
- leichtgewichtige Webapplikationen
- Ausbildung zum Architekten
- Rolle des Softwarearchitekten
- Business Process Management
- Webarchitekturen
- Product Line Engineering
- Cloud Computing
- Lisp
- domänenenspezifische Sprachen
- Systems Engineering
- Testing
- NoSQL
- funktionale Programmierung
- REST
- Architektur-Reviews
- Architekturdokumentation
- Entwicklung für Multicore-Systeme
- serviceorientierte Architekturen
- modellgetriebene Softwareentwicklung
- Middleware
- Patterns

Listings und Lizenzen

Die Listings zu den Heftartikeln und die Lizenzen zu den Softwarepaketen.

Hinweis für Käufer

- PDF- und iPad-Version: In der *iX*-App finden Sie einen Button zum Download des DVD-Images.
- PDF-E-Book: Folgen Sie im Browser der unter „Alle Links" angegebenen URL.

Alle Links: www.ix.de/ix1217006

Moderner Blick auf Programmiersprachen

Mit neuen Methoden und Vorgehensweisen oder dem bewussten Rückgriff auf bewährte Ideen versucht man in der Softwareentwicklung, den aktuellen Marktanforderungen besser gerecht zu werden. Der Trend macht sich auch bei den Programmiersprachen deutlich bemerkbar, mit denen sich die zunehmend komplexer werdenden Aufgaben effizienter lösen lassen.

Programmiersprachen im Wandel der Zeit	**8**
Grundsätzliches zur Entwicklung von Programmiersprachen	**12**
Wo Haskell und Co objektorientierten Sprachen überlegen sind	**18**
C++11 – ein Jahr später	**26**
Gegenwart und Zukunft der Java-Entwicklung	**30**
Programmiersprachen für die JVM im Zusammenspiel mit Java	**36**
JavaScript für Industrie- und Geschäftsanwendungen	**40**
CoffeeScript – das bessere JavaScript?	**48**
TypeScript: Microsofts neues typisiertes JavaScript	**54**
Dart – Googles zweite Programmiersprache	**58**
Node.js: Zeitgemäße Webanwendungen in JavaScript entwickeln	**62**

Programmiersprachen im Wandel der Zeit

Der Mix macht's

Frank Müller

Nach einer Zeit der vermeintlichen Ruhe ist es in den letzten Jahren in der Welt der Programmiersprachen wieder etwas unruhiger geworden. Seit 2000 sind diverse neue Sprachen angetreten, die um die Gunst der Entwickler buhlen. Doch wie neu sind deren Ideen wirklich, und was bieten sie mehr als ihre Vorgänger?

Gemessen an der Historie anderer wissenschaftlicher oder technologischer Zweige sind Programmiersprachen jung, sodass eine stürmische Entwicklung nicht ungewöhnlich erscheinen sollte. Doch so jung ist das Thema nun auch wieder nicht. Bereits in den 40er-Jahren des letzten Jahrhunderts haben sich Menschen mit der flexiblen und automatisierten Steuerung von Maschinen befasst. Das geschah anfangs in Maschinensprache, war jedoch aufwendig und fehlerträchtig. Alternativen wurden gesucht, unter anderem von Konrad Zuse. Er entwickelte auf Basis des Lambda-Kalküls [a] die erste höhere Programmiersprache Plankalkül [b]. Er konnte die Sprache zwar nie realisieren, erst 1998 wurde ein Compiler für sie gebaut, doch Plankalkül verfügte neben der linearen Ausführung von Operationen bereits über bedingte Anweisungen, Schleifen und Funktionsaufrufe. Dies alles findet sich heute in so ziemlich allen praktischen Sprachen wieder, auch wenn sich die Form unterscheidet. Andere Sprachen kamen hingegen früher zur Einsatzreife.

Zeitreise von den Anfängen ...

In den Jahren nach Zuses Pionierarbeit bewegte sich die Forschung und Entwicklung bezüglich der Programmiersprachen noch auf einem maschinennahen Niveau. Für die unterschiedlichen Systeme wurden individuelle Assemblersprachen entwickelt, die im Prinzip nur eine menschenlesbare Form der Maschinensprache darstellten. Weitere Impulse für Hochsprachen entstanden erst in den 50er-Jahren. Sie bildeten die Keimzelle für so manche Sprachen, die heute noch in Verwendung sind oder zumindest einen starken Einfluss auf die weitere Entwicklung hatten.

Den Beginn machte 1957 der Formula Translator, kurz Fortran. Der Start der Entwicklung begann 1953 auf Anregung von John W. Backus. Es sollte eine Assembler-Alternative für numerische und wissenschaftliche Programme geschaffen werden. So erhielt Fortran komplexe Zahlen als Datentyp. Die erste Version der Sprache arbeitete bereits strukturiert und brachte die logische und arithmetische Verzweigung mit *IF* mit sich. Letzteres unterscheidet das Ergebnis einer mathematischen Berechnung in die drei Bereiche Negativ, Null und Positiv. Weitere *IF*-Statements dienten der Behandlung von Exceptions. Schleifen ließen sich mit *DO* realisieren, und auch das *GOTO* war Bestandteil der Sprache.

FORTRAN II von 1958 kannte dann Unterroutinen und Funktionen, die robuste Basis imperativer Programmierung war geboren. Das Paradigma zeichnet sich dadurch aus, dass der Entwickler über Befehle sowohl die einzelnen Schritte als auch deren Reihenfolge in einem Programm vorgibt. Weitere Bestandteile sind die erwähnten Verzweigungen und Schleifen sowie Unterprogramme und Funktionen für die Mehrfachnutzung von Code-Blöcken. Es ist somit eine direkte Form der Entwicklung. Weitere wichtige Vertreter des Konzepts folgten kurze Zeit später mit den besonders auf Großrechnern verbreiteten COBOL (1960) und PL/1 (1964) oder den in den 70er-Jahren entwickelten Pascal (1971) und C (1972).

Im gleichen Jahr wie Fortran erblickte eine weitere bedeutende Sprache das Licht der Welt. Auch sie ist noch im praktischen Einsatz: Lisp. Wie bei Fortran hatte der Entwickler John McCarthy die Lösung mathematischer Probleme unter Nutzung

des Lambda-Kalküls zum Ziel. Er wählte als Notation anfangs M-Expressions, die in S-Expression übersetzt wurden. Später setzten sich jedoch die S-Expressions mit Funktionen in Polnischer Notation durch. Sie sind geliebt wie gehasst, aber auf jeden Fall elegant. Die Wurzeln der Sprache finden sich im funktionalen Paradigma, oft werden Lisp und die funktionale Programmierung synonym genannt, auch wenn Anhänger der reinen Lehre das anders sehen.

Wie der Name es andeutet, bestehen funktionale Programme aus Funktionsdefinitionen, -anwendungen und -kompositionen. Funktionen bilden selbst Vergleiche, arithmetische Operationen und Strukturen ab. Die Anwendung von Funktionen auf Konstanten oder die Komposition von Funktionsergebnissen als Eingabe für weitere Funktionen vermeidet die gefürchteten Seiteneffekte imperativer Sprachen. Ausgehend von McCarthys Sprache haben sich über die Zeit weitere Dialekte gebildet, ihre berühmtesten sind Scheme (1975) und Common Lisp (1984), beide mit unterschiedlichen Ausrichtungen. Scheme kannte bereits endrekursive Funktionen, einen eleganten Weg für rekursive Funktionen mit konstantem Speicherverbrauch. Common Lisp ist hingegen eine Multi-Paradigmen-Sprache, die mit dem Common Lisp Object System über eine Objektorientierung verfügt.

Neben diesen beiden gibt es noch diverse weitere Dialekte, die sich über die Zeit ausgeprägt haben. Weitere funktionale Sprachen sind ML (1973) und seine Sprachfamilie sowie Haskell (1990). Letztere führte mit ihrer Version 1.3 Monaden als wichtiges Konstrukt in die funktionalen Programmiersprachen ein.

... bis zur jüngsten Vergangenheit

Anfang der 70er-Jahre erblickte eine weitere Programmiersprache mit Fokus auf logische Aufgabenstellungen die Welt. Der Franzose Alain Colmerauer veröffentlichte 1972 die Programmiersprache Prolog. Die Idee der Sprache ist das Erzeugen einer Datenbasis aus einer Reihe von Fakten und Regeln. Gegen die Datenbasis kann der Anwender dann Anfragen stellen. Eine Beschreibung des Lösungswegs erfolgt nicht, das ist Aufgabe des Systems. Der Ansatz ist das genaue Gegenteil imperativer Programmierung, das Paradigma wird als deklarativ bezeichnet.

Ein interessanter Bestandteil der Sprache, den weitere übernommen haben, ist das Pattern Matching. In der Regel erfolgt die Auswahl einer Funktion über ihren Namen und die Anzahl ihrer Argumente. In einigen statisch typisierten Sprachen sind zudem die Typen der Argumente relevant. Ist die Funktion *myFunction* beispielsweise einmal mit einer Integer-Zahl und einmal mit einer Zeichenkette als einziges Argument definiert, führen *myFunction(4711)* und *myFunction("iX")* zum Aufruf der jeweils passenden Funktion.

Beim Pattern Matching spielt auch die Form der definierten Argumente eine Rolle. Sie bilden ein Muster, das auf die übergebenen Werte angewandt wird. Passt es, lassen sich benannte Bestandteile des Musters innerhalb einer definierten Regel weiter nutzen. So prüft *liste([H,T], H, T) :- true.*, ob das erste Argument interpretiert als Liste aus dem Kopfelement *H* (Head) und der restlichen Liste *T* (Tail) besteht, und gibt in diesem Fall *true* zurück. Die Technik kommt nicht nur in Prolog zum Einsatz. So berechnet sich die Länge von Listen in Haskell leicht nachvollziehbar über

```
length        :: [a] -> Integer
length []     =  0
length (x:xs) =  1 + length xs
```

Listing 1: Aktoren in Scala

```
import scala.actors.Actor
import scala.actors.Actor._

case class Inc
case class Value

class Counter extends Actor {
    var counter: Int = 0;

    def act() = {
        while (true) {
            receive {
                case Inc =>
                    counter += 1
                case Value =>
                    println("Value is "+counter)
                    exit()
            }
        }
    }
}

object ActorTest extends Application {
    val counter = new Counter
    counter.start()

    for (i <- 0 until 100000) {
        counter ! Inc
    }
    counter ! Value
}
```

Listing 2: Asynchrone Methoden in Vala

```
async void list_dir() {
    ...
}

void main() {
    var loop = new MainLoop();
    list_dir.begin((obj, res) => {
            list_dir.end(res);
            loop.quit();
        });
    loop.run();
}
```

Und auch Erlang (1988), das Joe Armstrong anfangs in Prolog entwickelte, nutzt das Pattern Matching intensiv.

Eine wichtige Entwicklung in der Welt der Programmiersprachen startete noch vor Prolog in den 60er-Jahren. Die Forschungsarbeiten von Ole-Johan Dahl und Kristen Nygaard an der Universität Oslo zum Thema Simulation führten zur Entwicklung der Programmiersprache Simula (1965). Mit der Sprache erblickte das bis heute führende Paradigma der Objektorientierung das Licht der Welt. Der Begriff ist jedoch erst später mit der von Alan Kay entwickelten Sprache Smalltalk entstanden. Bei ihm geht es um die Kapselung von Daten und Methoden, die auf diesen Daten in Klassen operieren. Die Instanzen der Klassen operieren unabhängig und tauschen untereinander Nachrichten aus, zumindest so Kays idealisierte Darstellung. In der Entwicklung sind das Methodenaufrufe auf den Instanzen. Ein weiterer wichtiger Faktor ist die Vererbung, in der Klassen die Eigenschaften anderer Klassen erben und sie nach Bedarf erweitern oder verändern. Das führt zur Polymorphie. Über die Vererbung, über Schnittstellen oder einfach nur über Methoden mit gleicher Signatur – je nach Sprache – kann das Laufzeitsystem einer Sprache entscheiden, welche Implementierung es zur Ausführung heranzieht. Diese Prinzipien nutzen die heute wichtigen Sprachen Objective-C und C++ (beide 1983), Python (1991), Java (1995) und C# (2000).

Ein Konzept, das die Idee Kays bald noch mehr aufgreift, ist das Paradigma der Nebenläufigkeit. Hier operieren Komponenten unabhängig und tauschen Daten über Nachrichten oder miteinander geteilten Speicherbereichen aus. Das lässt sich auf unterschiedliche Weise realisieren. Ist die Anzahl der nebenläufigen Stränge nicht zu groß, lassen sich seitens des Betriebssystems

bereitgestellte Threads nutzen. Zwischen ihnen geteilte Variablen sind über Synchronisationsmechanismen zu schützen, was allerdings fehlerträchtig ist und schnell zu Blockaden führen kann. Ein besseres Modell mit mehreren Tausend parallelen Ausführungssträngen nutzt daher leichtgewichtige Prozesse oder Koroutinen, denen ein interner Scheduler Rechenzeit zuordnet. Sie verwenden in der Regel den synchronen oder asynchronen Datenaustausch über Nachrichtenboxen, Ports oder Kanäle. Das oben aufgeführte Erlang ist eine derartige Sprache. In ihr lassen sich Funktionen als Prozesse mit individuellen Message Boxes starten. In der Regel arbeiten die Funktionen als endrekursive Schleifen und empfangen über den *receive*-Ausdruck Daten, die an die Prozess-ID geschickt werden. Hier greift wieder das Pattern Matching, um die empfangenen Daten der richtigen Verarbeitung zuzuordnen. Mit der zunehmenden Anzahl an Prozessoren, Kernen und Hyperthreads in heutigen Rechnern rückte dieses Paradigma zunehmend in den Fokus, da es zwar nicht ohne ein entsprechendes Softwaredesign, dafür aber recht natürlich über die zusätzlichen Ressourcen skaliert.

Angekommen in der Gegenwart

Der kurze Streifzug durch die Geschichte der Programmiersprachen verdeutlicht: Auch wenn so manche Sprache synonym mit einem Paradigma genannt wird, finden vielfach Vermischungen statt. So deckt Common Lisp neben der funktionalen die objektorientierte Programmierung ab. Und der Inbegriff der Objektorientierung, Smalltalk, verfügt mit seiner intensiven Nutzung von Closures über eine starke funktionale Natur. Java als zeitgemäße objektorientierte Sprache lässt sich ebenfalls für nebenläufige Aufgaben über die Nutzung von Threads und Synchronisierungen heranziehen, was aber eher die Domäne Erlangs ist, das wiederum stark funktional arbeitet und das Pattern Matching der deklarativen Programmierung nutzt.

Ein alter Hase der Post-2000-Sprachen ist Scala, das Martin Odersky 2003 vorgestellt hat. Sie gilt als eine Multi-Paradigmen-Sprache auf der Basis von Java und der Java Virtual Machine als Laufzeitumgebung. Ziel war es, ein „besseres“ Java zur Verfügung zu stellen. So wurden Scala viele Eigenschaften der genannten Paradigmen mitgegeben, beispielsweise Closures und List Comprehension, Typinferenz, Currying und Pattern Matching. Auch unterstützt die Sprache die Nebenläufigkeit wie Erlang mit Aktoren, kombiniert mit Software Transactional Memory (s. Listing 1). Die Flexibilität und Mächtigkeit der Sprache kommt der Vielfalt entgegen und bietet Nutzern die Möglichkeit, eine eigene speziell auf die zu lösende Aufgabe zugeschnittene Domain Specific Language (DSL) zu implementieren.

Listing 3: Goroutinen in Go

```
package main

func printer(ch chan string) {
       i := 1
       for s := range ch {
              println(i, s)
              i++
       }
}

func main() {
      ch := make(chan string)
      strings := []string{"foo", "bar", "baz"}

      go printer(ch)

      for _, str := range strings {
              ch <- str
      }
      close(ch)
}
```

Listing 4: Tasks in Rust

```
let po = comm::Port();
let ch = comm::Chan(&po);

do task::spawn {
    ...
    comm::send(ch, ~"Hello, World!");
};

let result = comm::recv(po);
```

Ohne virtuelle Maschine, aber dennoch für diverse Plattformen verfügbar, ist das 2006 vorgestellte Vala. Es wurde von Jürg Billeter und Raffaele Sandrini, damals beides Studenten an der ETH Zürich, entwickelt. Ziel war eine Sprache für das Objektsystem GObjekt, der Grundlage von Gnome. Damit ist klar, dass Vala zuerst einmal objektorientiert ist. Dazu gehören Mixins, Interfaces und Generics. Die Kontrollstrukturen sind dabei wie in Scala imperativ. Außerdem kennt Vala Delegates zur Definition von Methoden, die sich wie Objekte an weitere Methoden übergeben lassen. Auch Closures und damit das Currying sind der Sprache nicht unbekannt. Über die Anbindung von Signal-Handlern lassen sich Ereignisse verarbeiten. Ähnlich zu den Constraints in Erlang, dort im Rahmen des Pattern Matching, kennt Vala Vor- und Nachbedingungen und erlaubt so eine Entwicklung mit Verträgen. Neben Threads kommen in der Sprache asynchrone Methoden zum Einsatz.

Eine auf den ersten Blick ausschließlich imperativ anmutende Sprache stellt sich bei genauerer Betrachtung ebenfalls als ein munterer Mix heraus, wenn auch weniger vielfältig als Scala. 2009 haben die Google-Entwickler Ken Thompson, Rob Pike und Robert Griesemer die Sprache Go vorgestellt. Hier galt es ebenfalls, eine bessere Alternative zu einer bestehenden Sprache herzustellen. Bei der Vergangenheit von Thompson und Pike verwundert es nicht, dass es sich dabei um C handelt. Jedoch sollte die statisch typisierte Sprache für moderne Mehrkernrechner gewappnet sein, komfortable Konstrukte für eine hohe Produktivität bieten und typische Fehlerquellen vermeiden. Ist die generelle Syntax noch C-nah, verfügt die Sprache wie Scala und Vala über Closures und Typinferenz. Nebenläufigkeit wird über Koroutinen realisiert, in Go treffenderweise als Goroutinen bezeichnet und über die Anweisung *go* gestartet. Die Kommunikation kann zwar über Shared Memory stattfinden, in der Regel werden jedoch Channels genutzt. Die Basis hierfür legte Tony Hoare mit den von ihm entwickelten Communicating Sequential Processes [c].

Go ist zwar nicht objektorientiert, bietet jedoch die Möglichkeit, Methoden an beliebige Typen zu binden, quasi eine Objektorientierung ohne Vererbung. Dafür lassen sich Typen einbetten, eine Form der Komposition. Die Sprache verfügt zudem über Interfaces, sodass sie eine Form der Polymorphie enthält. Zu guter Letzt bietet Go auch ohne virtuelle Maschine eine Garbage Collection.

Ein Jahr nach Go wurde seitens Graydon Hoare und Mozilla Research die Sprache Rust vorgestellt. Sie ist als sichere, nebenläufige und anwendungsorientierte Sprache auf Basis von LLVM (Low Level Virtual Machine) als Laufzeitsystem konzipiert. Dabei deckt sie bewusst mehrere Paradigmen ab. Von ihrer Struktur her ist sie wie die Sprachen zuvor imperativ und wird C-Entwickler mit ihrer Notation sofort ansprechen. Rust verfügt über Closures, Typinferenz und Pattern Matching sowie eine Objektorientierung inklusive Generics und Traits. Ähnlich zu Erlang sind Variablen unveränderbar, sie lassen sich bei Bedarf aber als solche definieren. Das ist in ähnlicher Form auch in Scala realisiert. Die Nebenläufigkeit in Rust ergibt sich über ei-

Onlinequellen

[a] Fabian Nilius; Das gefürchtete Lambda-Kalkül
www.betoerend.de/dasLandHinterDemEndeDesSinns/lambda/welcome.html

[b] Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Krüger, Ludmila Scharf; Konrad Zuses Plankalkül – Seine Genese und eine moderne Implementierung
www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Genese/Genese/Genese.htm

[c] Tony Hoare; Communicating Sequential Processes (CSP)
www.usingcsp.com

ne Baumstruktur von Tasks mit Ports für die eingehende und Channels für die ausgehende Kommunikation.

Fazit

Neben den genannten Sprachen gibt es noch ein paar weitere Neuerscheinungen. Beispielsweise Clojure (2007) auf Basis der JVM, Arc (2008) und Shen (2011), alles drei Vertreter des funktionalen Lagers. Sie verfügen über unveränderliche und persistente Datenstrukturen sowie Software Transactional Memory und Agenten für Nebenläufigkeit (Clojure) oder enthalten ein vollwertiges Prolog mit dem notwendigen Pattern Matching (Shen). Dazu kommen weitere Sprachen aus dem Lager der geschweiften Klammern wie CoffeeScript (2009), Ceylon (2011) und Dart (2011). Auch sie bedienen sich im Warenlager der guten alten Sprachen und bieten zum Beispiel Closures, Comprehensions oder Mechanismen zur Nebenläufigkeit.

Betrachtet man die Entwicklungen dieses Jahrtausends, fällt bis auf wenige Ausnahmen ein Hang zum Pragmatismus auf. Der Fokus der neuen Sprachen liegt in der Regel auf einem gesunden Mix unterschiedlicher Paradigmen für eine komfortable und dennoch mächtige Softwareentwicklung auf modernen Mehrkernsystemen. Die jeweilige Umsetzung basiert dabei auf der Historie und den persönlichen Präferenzen der Köpfe hinter den Sprachen. Sie spiegelt sich vielfach deutlich in der jeweiligen Syntax, der Realisierung der Features und der Laufzeitumgebung wider. Bei allen Unterschiedlichkeiten ist dennoch interessant, wie viele und welche Eigenschaften sich in nahezu allen Kandidaten wiederfinden. Insofern ist die eigene Entscheidung für eine Sprache – neben eventueller wirtschaftlicher Betrachtungen – von persönlichen Präferenzen abhängig. (ane)

Literatur

[1] Stefan Tilkov: Renaissance der Vielfalt; Aktuelles zur Entwicklung von Programmiersprachen; *iX Developer* 1/2012, S. 117 ff.

[2] Michael Wiedeking; Babylonische Verwirrung; Eine Betrachtung zur Entwicklung von Programmiersprachen; *iX Developer* 1/2011, S. 94

Frank Müller
arbeitet als Software Engineer bei der Canonical Ltd.

Alle Links: www.ix.de/ix1217008

Leica Geosystems

Wenn es gilt, die Qualität von Flugzeugen, Windenergieanlagen oder Hochgeschwindigkeitszügen schon bei der Herstellung zu sichern, setzt die Industrie auf Messsysteme von Leica Geosystems. Im globalen Hexagon Metrology-Netzwerk sind wir auf modernste optische Lasermesssysteme zur Erfassung, Darstellung und Verarbeitung räumlicher Daten spezialisiert. Unsere Kunden vertrauen der Zuverlässigkeit und Innovationskraft unserer Marke.

Am Standort Unterentfelden suchen wir für die Entwicklung neuer High Tech-Produkte eine/n

EntwicklungsingenieurIn Embedded Software

Ihr Aufgabengebiet:
- Entwicklung Embedded Software von der Designphase bis zur Serienreife
- Zusammenarbeit in interdisziplinären und internationalen Entwicklungsteams aus den Bereichen Elektronik, Software, Messtechnik und Konstruktion

Ihr Anforderungsprofil:
- Abgeschlossenes Elektronik Studium ETH, FH oder Universität (TU) (direkt ab Studium möglich)
- Vertiefte Kenntnisse in hardwarenaher SW-Entwicklung in C#, C++ einschliesslich Multi-Threading und Laufzeitoptimierung
- Kenntnisse in DSP- oder Mikroprozessor-basierten Systemarchitekturen
- Erfahrung mit Windows CE 6 von Vorteil
- Bereitschaft zur Mitarbeit im gesamten Entwicklungsprozess von den Anforderungen über Design bis zur Implementierung
- Gute Deutsch- und Englischkenntnisse in Wort und Schrift
- Selbständiger und methodisch strukturierter Arbeitsstil
- Teamorientierte und kommunikationsfreudige Persönlichkeit

Wenn Sie sich in einer Hightechumgebung wohlfühlen, ein dynamisches und motiviertes Team sowie attraktive Anstellungsbedingungen und eine moderne Infrastruktur schätzen, dann senden Sie Ihre Bewerbung an Edith Veith, Human Resources e-Mail: jobs.metrology@leica-geosystems.com

Hexagon Metrology. Part of Hexagon

TVN GROUP
Film & TV Production

Die TVN GROUP Film & TV Production ist mit Fernsehstudios, Sendetechnik sowie Fachpersonal für Film, Fernsehen und Corporate Media an mehreren Standorten in Deutschland vertreten. Wir entwickeln für renommierte Kunden Smartphone Apps, Mobile Websites, Touch Applikationen und Games.

Zur Verstärkung unseres Teams suchen wir umgehend

Senior PHP Entwickler w/m, Standort Hannover

Ihre Aufgaben
- Betreuung der Intranet- und Webseiten auf PHP5 Basis
- Einrichtung von Schnittstellen zu externen Portalen
- Selbstständige Weiterentwicklung der Aufgabenbereiche
- Erstellung von Betriebs- und Anwenderdokumentationen

Ihr Profil
- Erfolgreich abgeschlossenes Studium im Bereich Informatik, Mathematik oder eine vergleichbare Qualifikation
- Berufserfahrung in vergleichbarer Position
- Sehr gute Kenntnisse in den Bereichen PHP5, Javascript, MYSQL
- Analytisches Denkvermögen und technisches Verständnis
- Von Vorteil: Fremdsprachen, Kenntnisse in den Bereichen Client/Server-Systeme, E-Commerce und Warenwirtschaft
- Verantwortungsbewusstsein, Kreativität und Teamfähigkeit
- Kundenorientiertes, verkaufsförderndes Verhalten
- Leistungsbereitschaft und ergebnisorientierte Arbeitsweise

Wenn Sie über die genannten Kompetenzen verfügen, möchten wir Sie kennen lernen! Wir bieten Ihnen zum nächstmöglichen Zeitpunkt eine eigenverantwortliche und innovative Arbeitsumgebung in einem kollegialen Team.

Ihre aussagekräftige Bewerbung schicken Sie uns bitte per E-Mail.

Ihr Kontakt
TVN PRODUCTION GmbH & Co. KG
Frank Schliefer
Telefon +49 511 1212-3647
E-Mail frank.schliefer@tvn.de

Grundsätzliches zur Entwicklung von Programmiersprachen

Sprachschöpfer

Michael Wiedeking

Eine Programmiersprache zu entwickeln ist nicht trivial. Das liegt weniger an der technischen Herausforderung, denn nie zuvor war es einfacher, sich mit populären Werkzeugen einen Parser generieren zu lassen. Es liegt viel mehr daran, dass man sich nur zu gut überlegen muss, wie diese Sprache aussehen soll und welche Ziele man damit zu erreichen gedenkt. Denn es gilt, allzu viele Entscheidungen zu treffen, die das Potenzial einer Sprache signifikant beeinflussen können.

Wer sich daran macht – aus welchen Gründen auch immer –, eine eigene Programmiersprache zu entwickeln, muss eines sicherlich mitbringen: Er oder sie muss Entscheidungen treffen und mit Kompromissen leben können. Eine Programmiersprache ist nämlich das Fundament aller darauf aufbauenden Programme. Das ist durchaus vergleichbar mit unserer Sprache, von der Philosoph Ludwig Wittgenstein sagt: *„Die Grenze meiner Sprache* bedeuten die Grenzen meiner Welt." Zum Beispiel gestaltet es sich relativ schwierig, Objektorientierung in einer nicht objektorientierten Sprache umzusetzen. Sicherlich gibt es Mittel und Wege, das zu tun, aber es ist dann womöglich nicht effizient genug.

Dem scheint zu widersprechen, dass die ersten Übersetzer für C++ als Präprozessoren für C-Compiler realisiert wurden. Das heißt tatsächlich, dass C (mit seinen Funktionszeigern, funktionsübergreifenden Sprüngen et cetera) sehr wohl für die Implementierung der Objektorientierung geeignet ist, aber es wäre nie jemand auf die Idee gekommen, diese freiwillig einzusetzen, weil C einen nun wirklich nicht dazu einlädt. Alles was eine Sprache nicht beherrscht, lässt sich also entweder gar nicht oder nur mit enormem Aufwand für bestimmte Aufgaben beschreiben. Eine Programmiersprache tut also gut daran, die Zieldomäne mit eigenem Vokabular und Idiomen zu unterstützen. Je spezieller die Domäne ist, desto mehr kann man sich dabei deren Vokabular bedienen; je weiter sie gefasst ist, desto allgemeiner muss auch das Vokabular sein.

Aber egal, ob der Sprachentwickler eine Domain Specific Language (DSL) für ein konkretes Problemfeld schreiben will oder eher eine General Purpose Language (GPL) schaffen möchte, mit der sich viele Anforderungen bewältigen lassen, für alle gelten die gleichen Anforderungen: Der Entwickler muss wissen, was er der Sprache abverlangen können will.

Die andere Satzlehre

Zu den am einfachsten und gleichzeitig schwierigsten festzulegenden Eigenschaften einer Sprache gehört die Syntax. Einfach sind diesbezügliche Entscheidung deswegen, weil man „nur" die äußere Form der Sprache bestimmen muss: Bezeichner, Schlüsselwörter, Satzzeichen, Operatoren et cetera. Schwierig wird das Aussehen dadurch, dass es neben den technischen Aspekten nur eine Frage des Geschmacks ist, für was man sich entscheidet. Leider scheint bei Geschmacksfragen jeder eine ei-

gene Meinung zu haben, was immer wieder zu völlig überflüssigen Streitereien führt.

Rein technisch gesehen gibt es schon bei der Wahl der Zeichen schier unbegrenzte Optionen. Aber obwohl es dafür den Unicode gibt, nutzen ihn nicht viele. Damit sind in unseren Breiten neben dem normalen Alphabet nicht nur die Umlaute zulässig, sondern auch Akzente wie $\overrightarrow{v}$. Dass sie aber kaum Anwendung finden, mag vielerlei Gründe haben, wobei einer davon sicherlich der ist, dass die meisten Anwender erst gar nicht wissen, wie man diese Sonderzeichen eingeben kann.

Java hat hier Federführendes geleistet, indem die Sprache den Unicode nicht nur für den Programmtext akzeptierte. Leider wurde dabei versäumt, das Dateiformat für Textdateien eindeutig festzulegen, sodass es gelegentlich zu Fehlern bei der Interpretation der Zeichen kommt. Darüber hinaus taucht das Problem auf, dass eine Java-Klasse eine korrespondierende Klassendatei gleichen Namens hat. Benutzt man etwa einen Umlaut in einem Klassennamen, mag das zwar noch funktionieren, sobald man aber diese Klassendateien auf ein anderes Betriebssystem kopiert, kann es passieren, dass eben diese Datei unauffindbar ist.

Da jeder spätestens ab dem ersten Schuljahr mit Mathematik konfrontiert wird, bietet es sich an, für alle mathematischen Aufgaben eine vergleichbare Notation zu wählen. So wird aus der Multiplikation $a \cdot b$ eben $a * b$, weil sich der Asteriskus zum einen auf der Tastatur findet und zum anderen eine gewisse Ähnlichkeit zum Multiplikationspunkt hat.

Bei den Vergleichsoperation tut man sich bereits etwas schwerer, etwas Passendes auf der Tastatur zu finden. Aus <, ≤, > und ≥ wird mal <, <=, > beziehungsweise >=, was vielleicht noch eingängig erscheinen mag. Bei Vergleichen mit = und ≠ tut man sich schon etwas schwerer, und so finden sich für die Ungleichheit die Varianten !=, /=, |=, =/= oder <>; was immer die Tastatur und der ASCII-Code hergeben.

Das größte Problem scheint aber die Zuweisung in imperativen Sprachen zu sein. Besonders Anfänger tun sich schwer damit, dass das, was sie als Vergleich kennen ($a = b$), in sich von C ableitenden Sprachen als Zuweisung gilt und der Vergleich selbst seltsam anmutend mit zwei Gleichzeichen geschrieben wird ($a == b$). Sprachen mit Algol-Herkunft wie Pascal verwenden deshalb zur Unterscheidung für die Zuweisung :=. Fortran nutzt für die Zuweisung zwar auch das Gleichzeichen ($a = b$), für alle Vergleiche kommen dafür aber nur Namen zum Einsatz (a *.EQ.* b oder a *.LT.* b).

Ohnehin stellt sich die Frage, ob man lieber mit Namen oder Operatoren arbeitet. Beispielsweise kann man sich unter *a mod b* vielleicht noch etwas vorstellen, derweil es mit *a % b* fast unmöglich scheint. Mit Ausnahme der Standardoperatoren +, –, • und / tut man sich schon recht schwer mit den Symbolen. In Scala etwa gibt es die Operatoren /: und :\. Wie man „leicht" sieht, handelt es sich dabei um die beiden Funktionen *foldLeft* und *foldRight*. Die imperativen Programmierer können sich unter den gesprächigeren Varianten nichts vorstellen, aber die funktionalen Entwickler wissen sehr wohl, was damit gemeint ist.

Ein spezielles Thema im Zusammenhang mit der Syntax ist das Format. Die meisten Sprachen sind formatfrei. Das heißt, dass nichts dagegen spricht, alles ohne Leerzeichen in eine einzige Zeile zu schreiben. Unter den aktuellen Sprachen verlangt einzig Python, dass ein Programm vernünftig gesetzt wird – zumindest was das Einrücken betrifft.

```
def greet(name):
      print 'Hallo', name
greet( Python')
```

Grammatikalische Spitzfindigkeiten

Selbst wenn alle Ungereimtheiten auf lexikalischer Ebene gelöst sind, kann es immer noch welche auf syntaktischer Ebene geben. So enthält die C#-Spezifikation das folgende Beispiel, das diese Probleme demonstriert:

```
F(G<A, B>(7));
```

Bei dem Ausdruck kann es sich um die Funktion *F* handeln, die zwei boolesche Parameter hat, die mit den Argumenten *G < A* und *B > (7)* aufgerufen wird. Genauso gut kann es aber auch die Funktion *F* sein, die nur einen Parameter hat, bei dem es sich um die generische Funktion *G* mit den beiden Typ-Parametern *A* und *B* handelt, die einen einzelnen Parameter hat. Die Regeln in C# sagen tatsächlich, dass Letzteres der Fall ist.

```
F(G<A, B>7);
F(G<A, B>>7);
```

Variiert man die Ausdrücke und eliminiert etwa die runden Klammern, handelt es sich um eine zweistellige Funktion *F* mit den Argumenten *G < A* und *B > 7*, was auch im zweiten Fall mit *B >> 7* gilt.

An dieser Stelle zeigt sich also, wie schwierig eine Grammatik werden kann, wenn Zeichen beziehungsweise Token für völlig unterschiedliche Bedeutungen wiederverwendet werden.

Die vorgegebene Syntax entscheidet also auch darüber, wie konsistent das Erscheinungsbild der Quelltexte ist. Beispielsweise ist es in Perl den Variablen *$v*, *@a*, *%m* und *&f* sofort anzusehen, dass es sich um ein Skalar, eine Liste, eine Hash-Map beziehungsweise eine Funktion handelt. Oder während sich in Java Instanzvariablen optional durch ein *this* anführen (*this.v*) und Klassenvariablen sowohl über die Klasse direkt (*C.v*) als auch über jedes Objekt dieser Klasse (*o.v*) ansprechen lassen, beginnen in Ruby Instanzvariablen immer mit einem „Klammeraffen" (*@v*), derweil statische Klassenvariablen mit zwei davon beginnen (*@@v*).

Bezüglich der Syntax gehen die Meinungen also weit auseinander. Für die einen ist es das A und O einer Sprache, für andere nur Tinnef. Die Syntax entscheidet allerdings darüber, ob sich ein Programm einfach lesen lässt oder nicht. Und nachdem man Programmen nachsagt, sie werden deutlich öfter gelesen als geschrieben, scheint eine sorgfältige Wahl der Syntax durchaus sinnvoll. Allerdings wären mehrere Tausend Mannjahre an Diskussionen darüber erspart geblieben, wohin denn die geschweiften Klammern zu setzen sind, wenn sich die Herren Kernighan und Ritchie nicht nur dazu durchgerungen hätten, die Klammern wie in ihrem C-Buch nach oben zu setzen, sondern auch den Compiler dazu motiviert hätten, das zu überprüfen und zu erzwingen.

Prinzipien und Pragmatisches

Neben der Syntax tritt das Problem auf, wie der Entwickler die Sprache eventuell um Funktionen erweitern kann, ohne dabei die darauf aufbauende Codebasis anfassen zu müssen. Dazu bieten sich beispielsweise Annotationen an, die etwa bei Java und C# zu finden sind. Mit ihnen kann man Konstruktionen markieren, sodass sie bei der Übersetzung (oder zur Laufzeit) zu unterschiedlichen Interpretationen führen. Beispielsweise kann man für die Zugriffskontrolle relevante Schlüsselwörter wie *public* oder *private* als Annotation umsetzen.

Ist einem ein Sachverhalt wichtig genug, kann man ihn auch als Schlüsselwort implementieren. So hat Java 50 Schlüsselwörter. Im Gegensatz dazu gibt es in C# schon 77 Schlüsselwörter und 23 Kontextschlüsselwörter. Von Typbezeichnern über Literale und Konstanten bis hin zu Annotationen ist alles dabei.

Unabhängig davon, ob 100 Schlüsselwörter zu viel des Guten sind, schafft man es in der Regel nicht, später noch ein fehlendes hinzuzufügen.

Die Einführung der Enumerationen in Java haben schön demonstriert, wie fatal ein nachträglich eingeführtes Schlüsselwort sein kann. Das dazu ausgesuchte Wort *enum* war anscheinend so beliebt und so oft in Benutzung, dass es unmöglich war, es einfach zu ergänzen. Das hat dazu geführt, dass seit seiner Einführung die Source-Version von Java-Quellen anzugeben ist, um das Schlüsselwort zu aktivieren. Diese Probleme führen auch dazu, dass man Schlüsselwörter lieber wiederverwendet, als neue einzuführen. Im Zusammenhang mit den Generics wurden in Java die Schlüsselwörter *extends* und *super* für die Varianz von Klassen wiederverwendet, in C# die Wörter *in* und *out*.

Auch auf anderer Ebene finden sich Beispiele für Eigenschaften, die sich nur schwer im Nachhinein umsetzen ließen. So findet sich sich neuerdings öfter das Prinzip, dass *alles* ein Objekt ist. Das hat Konsequenzen, denn so gibt es ja keine primitiven Datentypen mehr. Es obliegt damit dem Compiler herauszufinden, wann sich diese Objekte optimieren und in primitive Operationen transformieren lassen. Dabei ist vorteilhaft, dass der Compiler die Implementierung vollständig verbergen und im „schlimmsten" Fall auch austauschen kann.

Möchte man beispielsweise beliebig große ganze Zahlen unterstützen, lässt sich das sicherlich immer über eine geeignete Implementierung erreichen. Um dabei die Unterstützung des Prozessors nicht zu verlieren, der zumindest im Bereich eines 32-Bit- oder 64-Bit-Integer unschlagbar schnelle Maschinenbefehle anbietet, könnte man ja erst beim Überschreiten dieser 32-Bit- beziehungsweise 64-Bit-Bereiche auf die beliebig langen Zahlen zurückgreifen. Dadurch wird allerdings schon eine Addition um ein Vielfaches aufwendiger, ist doch zu unterscheiden, ob etwa zwei 32-Bit-Zahlen oder zwei beliebig große Zahlen oder eine Mischung davon zu berechnen sind. Und da sich die Größe der Zahlen zur Laufzeit nach Belieben verändern kann, gibt es zur Übersetzungszeit kaum eine Möglichkeit für den Compiler, unterstützend einzugreifen.

Das zeigt deutlich, dass eine Großzügigkeit auf der einen Seite unter Umständen gravierende Einbußen auf der anderen Seite zur Folge hat. Diese Abhängigkeit kennt man etwa auch bei Architekturen und ihren Qualitätsmerkmalen. Alles hat eben seinen Preis, und deshalb muss man sich gut überlegen, welches Attribut man zugunsten eines anderen opfert.

Konform mit den Erwartungen

Eine wichtige Frage, die sich beim Gestalten einer Programmiersprache stellt, ist die nach dem Zielpublikum. Handelt es sich bei den Anwendern um Gelegenheitsbenutzer oder Programmierexperten, soll die Sprache sowohl Anfängern oder Profis bei der Arbeit helfen, soll die Lernkurve eher steil oder flach verlaufen? Nur in diesem Zusammenhang lässt sich die Frage beantworten, welche Erwartungen an die Sprache gestellt werden.

Das nachfolgende Beispiel zeigt, dass es auch in Sprachen, die für den täglichen Gebrauch gedacht sind, zu Unterschieden zwischen dem, was die Sprache bietet, und dem, was die Entwickler erwarten, kommen kann. Beispielsweise liefern Java oder C# immer wieder „schöne" Überraschungen.

```
long i = 2 000 000 000 + 2 000 000 000;
```

Wer glaubt, dass vier Milliarden die Antwort ist, hat übersehen, dass die Konstanten nur *int*-Werte sind und das dementsprechend auch die Addition eine *int*-Addition ist – und nicht, wie einen das *long* glauben machen könnte, eine *long*-Addition. Damit wird erst das *int*-Ergebnis berechnet (–294 967 296) und dann erst dieses zu einem *long* umgewandelt.

Ganze Bücher sind damit gefüllt, was man alles falsch machen kann, wobei dabei sicherlich C die meisten Möglichkeiten bietet. Aber auch die vermeintlich sicheren Sprachen wie Java und C# bieten ausreichend Spielraum für Fehl-Erwartungen. Dazu kommen noch stiefmütterlich behandelte Themen wie Gleitkommazahlen. So ist vielen Entwicklern nicht klar, dass (und warum) diese nicht dazu geeignet sind, Geldbeträge zu speichern oder der Additionsoperator nicht wie bei den ganzen Zahlen assoziativ ist. So müsste, wenn es denn wichtig genug ist, der Compiler bei jeder kritischen Anwendung warnend eingreifen.

Qual der Wahl beim Paradigma

Immer bestimmt die Anwendungslandschaft die Wichtigkeit von Eigenschaften. Damit stellt sich die Frage, wie man seine Aufgaben zu lösen gedenkt. Auch wenn das imperative Paradigma gefällig ist, um Abläufe zu programmieren, hilft die Objektorientierung doch dabei, diese Programme besser zu strukturieren und in eigenverantwortliche Objekte zu gruppieren. Die Objektorientierung hat aber ebenfalls nicht immer gehalten, was sich der eine oder andere davon versprochen haben mag. Die Welt in Objekten zu sehen ist sicherlich hilfreich, hat aber auch gravierende Nachteile. Da die Objekte auf einem Zustand operieren, treten nämlich Nebeneffekte auf, die im Zusammenhang mit nebenläufigen Anwendungen zunehmend problematischer werden.

In der funktionalen Programmierung tauchen solche Nebeneffekte im Gegensatz dazu nie auf, da Funktionen definitionsgemäß für gleiche Eingabewerte immer das exakt gleiche Ergebnis liefern. Und das sogar unabhängig davon, in welcher Reihenfolge man die Eingabewerte berechnet – also auch wenn sie gleichzeitig berechnet werden.

Wie erwähnt lassen sich sowohl objektorientierte als auch funktionale Features in anderen Sprachen formulieren, aber erst mit einer vernünftigen Unterstützung lässt sich das selbstverständlich und effizient umsetzen. Die Wahl des Paradigmas spielt also eine entscheidende Rolle, wie Aufgaben „natürlich" gelöst werden. Viele Probleme lassen sich viel besser deklarativ als imperativ programmieren. Zum Beispiel kann man Grammatiken von Sprachen einfach in Form von Deklarationen definieren. Sicherlich lässt sich das auch oft über Bibliotheken umsetzen, aber dann sind zur Laufzeit Fehlerprüfungen durchzuführen, die normalerweise schon ein Compiler prüfen kann.

Die Liste der zur Auswahl stehenden Programmierparadigmen, für die es jeweils mehrere Programmiersprachen gibt, ist auch auszugsweise schon lang:

- imperatives Paradigma mit strukturierter, prozeduraler und modularer Programmierung und abstrakten Datentypen
- funktionales Paradigma
- deklarative Paradigmen mit logischer Programmierung und der Programmierung mit Constraints
- objektorientiertes Paradigma
- Programmieren mit Komponenten und Agenten
- aspektorientierte Programmierung
- generative Programmierung
- generische Programmierung
- Daten- und datenstromorientierte Programmierung

Nur wenige Programmiersprachen bedienen sich dabei tatsächlich mehrerer Paradigmen. Man darf nicht vergessen, dass jedes Paradigma eigene Idiome hat, die erst einmal gelernt werden

Intel® Parallel Studio XE 2013.

IHR PERFORMANCE TOOL.

Gewinnen Sie ein **Lenovo-IdeaPad**! Alle Details auf Seite 151.

Intel® Parallel Studio XE 2013 ist das Performance-Paket für schnellen C++ und Fortran-Code unter Windows und Linux. Sehr schnelle Compiler und Performance-Bibliotheken werden ergänzt durch Tools für die effiziente Parallelisierung, Optimierung und Fehlersuche. Testen Sie jetzt die neue Version 2013!

30-Tage-Testversion auf Heft-DVD

hoCOMPUTER
compiler & tools

info@hocomputer.de
+49 / 221 / 762086
shop.hocomputer.de

intel Software Elite Reseller

Weitere Informationen über Performance- und Optimierungslösungen aus der Intel Softwareproduktpalette finden Sie unter http://software.intel.com/en-us/articles/optimization-notice.
* 2012, Intel Corporation. Alle Rechte vorbehalten. Intel, das Intel Logo und Parallel Studio XE sind Marken der Intel Corporation in den USA und anderen Ländern. *Andere Marken oder Produktnamen sind Eigentum der jeweiligen Inhaber.

wollen. Außerdem ist für jede Aufgabe zu überlegen, welches Paradigma am besten geeignet ist. Wer nur einen Hammer hat, für den ist alles ein Nagel. Das gilt auch für die konkrete Auswahl der zur Verfügung stehenden Paradigmen. Wer einen ganzen Werkzeugkasten hat, ist dann vielleicht überfordert.

Auf die Schnelle

Ungeachtet der Domäne ist zu überlegen, wie sich ein Programm ausführen lässt. Eine Variante ist die Übersetzung des Quelltexts in ein ausführbares Artefakt. Dabei spielt es keine Rolle, ob dieses direkt auf der Hardware laufen kann oder eine virtuelle Maschine (VM) zum Einsatz kommt. Entscheidend ist dabei, dass in der Regel ein expliziter Übersetzungsschritt einzuleiten ist, der nur bei erfolgreicher Durchführung das Artefakt liefert, das dann explizit auszuführen ist.

Der Erfolg vieler Skriptsprachen ist dem Umstand geschuldet, dass deren Interpreter den Editier-und-Ablauf-Zyklus minimieren. Im Zusammenhang mit Webapplikationen sind etwa nur ein paar Dateien in ein Verzeichnis zu kopieren, und schon ist die Anwendung verfügbar. Im Extremfall lässt sich also eine fehlerhafte Datei einfach an Ort und Stelle korrigieren: kein Übersetzen, kein Verpacken, kein Deployment und auch sonst nichts. Sicherlich hat das auch Nachteile, aber die gefühlten Vorteile scheinen zu überwiegen.

Skriptsprachen haben immer einen besonderen Reiz, wenn es schnell gehen muss. Das haben auch die Entwickler von D oder Scala erkannt, denn deren Sprachen lassen sich sowohl in Skriptform nutzen als auch zu ausführbaren Dateien übersetzen. In D verhindert zudem ein geeigneter Caching-Mechanismus, das einmal abgelaufene Dateien – solange sie unverändert sind – erneut „übersetzt“ werden.

Darüber hinaus haben viele Skriptsprachen Eigenschaften, die in den „statischen“ Sprachen eher seltener zu finden sind. Beispielsweise liefert in Perl der Ausdruck 1 + "1" den Wert 2, denn + arbeitet dort mit Zahlen, und die Zeichenkette "1" lässt sich in eine Zahl konvertieren; 1 + "a" liefert dementsprechend 1. Viele dieser Sprachen dulden auch, dass sich einer Variablen nacheinander Werte unterschiedlicher Typen zuordnen lassen.

Ein Frage des Typs

Ein Typsystems soll sicherstellen, dass der Anwender keinen Unfug anstellen kann, den der Sprachdesigner nicht billigt – egal ob beabsichtigt oder aus Versehen. Ein Typsystem kann in der Sprache integriert sein, lässt sich aber genau so gut als externes Werkzeug anbieten. Es bestimmt, wie Typen zugeordnet und interpretiert werden: Typen lassen sich explizit angeben oder aus dem Kontext ermitteln; sie können statisch (zur Übersetzungszeit) oder dynamisch (zur Laufzeit) zugeordnet werden; Typen können streng oder eher schwach sein; Die Gleichheit von Typen lässt sich über den Namen oder die Struktur bestimmen; Typen können von anderen Größen abhängig sein. Die Möglichkeiten kann der Entwickler nach Belieben kombinieren und gegebenenfalls weiterentwickeln (Pluggable Type Systems).

Das Typsystem entscheidet unter anderem darüber, wie viele Fehler und in welchem Detaillierungsgrad zur Übersetzungszeit oder Laufzeit erkannt werden können. Unabhängig vom Typsystem kann man natürlich nicht auf Tests verzichten, aber möglicherweise sind bei einem schwächeren Typsystem bestimmte Tests verstärkt durchzuführen, um Laufzeitfehler in der Produktion zu vermeiden.

Da viele Entwickler erfahrungsgemäß speziell bei redundanter Information schreibfaul sind, ist Typinferenz ein beliebtes Feature. Dabei versucht der Übersetzer oder Interpreter so viel wie möglich eigenständig, über beteiligte Typen zu erfahren. Der Vorteil dabei ist, dass sich bei komplizierten Typen tatsächlich der vollständige Typ ermitteln lässt. Der Nachteil ist der, dass der Leser nicht mehr lesen kann, um welchen Typ es sich handelt; aber da helfen einem ja die Entwicklungsumgebungen.

Apropos redundante Information: In vielen Sprachen muss man gelegentlich explizit einen Typ zuweisen, damit der Compiler glücklich und zufrieden ist.

```
if (o instanceof C) {
    ((C) o).method0fC();
}
```

In dem Java-Beispiel ist dem Leser klar, dass nach der *instanceof*-Prüfung die Variable *o* vom Typ *C* ist. Dennoch ist für den Java-Compiler eine Typumwandlung vorzunehmen, sodass sich die Methode aus der Klasse *C* benutzen lässt. Das ist aus Sicht der neueren Sprachen einfach nicht mehr nötig und erlaubt

```
if (o instanceof C) {
    o.method0fC();
}
```

zu schreiben. Denn der Compiler, der meist ohnehin eine Flussanalyse durchführen muss, kann den Typ von *o* sicher ermitteln und so den Aufruf ohne explizite Typumwandlung erlauben.

Wie auch immer das Typsystem aussehen mag, es beantwortet nicht automatisch die Frage, welche Typen mit der Sprache ausgeliefert werden sollen. Typischerweise enthält eine Sprache immer ganze Zahlen mit den typischen Operationen. Aber ob Fließkommazahlen, komplexe Zahlen, Brüche, Vektoren, Matrizen et cetera nötig sind, ist schon fraglich. An der Stelle ist also immer zu entscheiden, was in die Sprache kommt und was in die Bibliothek – vorausgesetzt, dass es überhaupt einen Mechanismus zur Erweiterung des Funktionsumfangs gibt.

Im Dienst der Übersetzung

Wenn die Sprache definiert ist, ist noch der Übersetzer zu schreiben. Das gestaltet sich heutzutage so einfach wie nie zuvor. Werkzeuge wie ANTLR (antlr.org) oder Rats! (cs.nyu.edu/rgrimm/xtc/) erlauben es, Grammatiken zu definieren und daraus Parser generieren zu lassen, die fast eigenständig abstrakte Syntaxbäume erstellen. Frameworks wie Xtext generieren die Entwicklungsumgebung gleich mit, mit Syntaxhervorhebung und Code-Vervollständigung.

Jetzt stellt sich nur noch die Frage, auf welcher Plattform die Kompilierung der Sprache laufen soll. Die GNU Compiler Collection (GCC) etwa kann Code für quasi beliebige Prozessoren generieren. Dazu ist nur der abstrakte Syntaxbaum in die interne Repräsentation, die Register Transfer Language (RTL), zu überführen. Dann kümmern sich die Backends inklusive Optimierungen um den Rest. Wann immer es einen neuen Prozessor gibt, speziell auch für den Embedded-Bereich, kann man davon ausgehen, dass der Hersteller eine Anbindung für GCC liefert oder aber zumindest unterstützt.

Eine Alternative dazu stellt die Low Level Virtual Machine (LLVM) dar. Hier erzeugt das Frontend ähnlich wie bei der GCC Code für die virtuelle Maschine. Im Gegensatz zur GCC kann man die LLVM so konfigurieren, dass sie ausführbaren Code erzeugt oder den VM-Code erst zur Laufzeit bindet.

Natürlich kann man alles auch selbst entwickeln, allerdings ist man meist mit nur einem Prozessor mehr als ausgelastet. Und die Spitzfindigkeiten eines Prozessors sind nicht zu unterschätzen. Hier kann es eine gute Idee zu sein, Code für die JVM oder die CLR zu übersetzen. Der Bytecode ist übersichtlich, relativ gut strukturiert und sozusagen frei von Nebeneffekten. Denn er wird ja erst zur Laufzeit in ausführbaren Maschinencode übersetzt.

Sowohl die eine als auch die andere VM sind für die mitgelieferten Bibliotheken optimiert und wissen beispielsweise die SIMD-Befehle aktueller Prozessoren zu nutzen. Das hat ebenfalls einen Haken: Da die Optimierungen oft nur an die konkreten, mit der VM ausgelieferten Bibliotheken gekoppelt sind, muss die eigene Sprache, um solche Möglichkeiten ausnutzen zu können, diese Bibliotheken auch unterstützen. „Leider", muss man sagen, denn deren Schnittstellen sind oft suboptimal, zu allgemein gehalten und haben meist eine andere Struktur als die, die man sich wünschen würde. Auch braucht man noch eine Option, sich die „Altlasten" passend zu machen.

In Kotlin, einer relativ neuen JVM-Sprache, sind etwa Referenzen *R*, die *null* werden können, explizit mit *R?* als solche zu deklarieren. Damit kann der Compiler zu jedem Zeitpunkt prüfen, ob die *null* möglich und zulässig ist. In Kombination mit Java-Klassen gibt es nun aber das Problem, dass man diese Information dort nicht hat. Im ersten Schritt wurde diese klassenweise nachgetragen, aber das Unterfangen stellte sich als Fass ohne Boden heraus. Das Problem scheint aber insofern gelöst worden zu sein, dass es ein Werkzeug gibt, dass die Information analytisch aus Klassendateien extrahiert.

Kontakt zur Außenwelt

Ist die Sprache definiert und implementiert, fehlt nur noch die richtige Entwicklungsumgebung. Hier hat Java Wegweisendes hervorgebracht. Will man also eine konkurrenzfähige oder akzeptierte Sprache schaffen, kommt man nicht um die entsprechenden Editoren oder IDE-Plug-ins herum. Damit ist es aber nicht getan. Um vernünftig entwickeln zu können, braucht man eine Infrastruktur, die den Entwicklern eine Möglichkeit gibt, Rückmeldungen abzugeben, Verbesserungsvorschläge einzubringen und Erfahrungen auszutauschen. Hat man die Community aber erst mal auf seiner Seite, ist diese bereit, heilige Kriege zu führen. Genauso energisch wie Pythons Zwangseinrückung vom Rest der Welt angegriffen wird, genau so vehement verteidigt sie die Python-Gemeinde.

Bis es aber so weit ist, muss man sich immer wieder mit der „anderen" Seite auseinandersetzen. Da wird (meist anonym in ungebührlichem Ton) angeprangert, dass der Typ hinter der Variablen steht ($x : T$) statt davor ($T\ x$); die Klammern auf der gleichen Zeile stehen statt darunter; Methodennamen mit einem GroßBuchstaben beginnen statt mit einem kleinBuchstaben; Operatoren überladen werden können oder auch nicht – oder das Ganze umgekehrt.

Ist das aber mal geklärt, gibt es viele Programmierer, die sich vom Neuen angezogen fühlen und es besser zu machen wünschen, es irgendjemandem zeigen oder einfach nur mitmachen wollen. Und begabte Softwareentwickler gibt es viele. Die haben auch viele Ideen und breit gefächerte Erfahrungen. Hier tut man gut daran, ihren Ideen und Erfahrungen eine Plattform zu geben, durchaus auch, um den einen oder anderen Entwickler zeigen zu lassen, was er kann.

Unabhängig davon tut es einer Programmiersprache gut, wenn sie sich mit *einer* Person assoziieren lässt, vor allem wenn kein Unternehmen dahinter steht. Bei Java war es James Gosling, und als er Java im Zuge der Übernahme von Sun durch Oracle verlassen hatte, schien alles plötzlich nicht mehr so recht vorwärts gehen zu wollen. C# hatte Anders Hejlsberg, aber der erweckte den Eindruck, nicht ganz so nötig zu sein, weil hinter ihm die Marke Microsoft stand. So macht es Linus Torvalds richtig: Jeder darf für Linux entwickeln, was er will, aber letztlich entscheidet er, was in den Kern kommt und was nicht.

Eine Community, die keine Führung hat, kommt auch nicht so schnell zu einem oder auch mal zu keinem Ende. Ein gutes Beispiel dafür sind die Sprachen, die an ein Standardisierungskomitee übergeben wurden. Hier wird versucht, es allen rechtzumachen, was in der Regel zu politischen Irrungen und Wirrungen führt.

Fazit

Als Sprachdesigner darf man grundsätzlich machen, was man will. Möchte man aber eine Sprache schaffen, die zu den großen gehören oder als ebensolche akzeptiert werden soll, muss man Kompromisse eingehen und sicherstellen, dass die Sprache gefällt. Dazu müssen möglicherweise alle Zeichen auf der Tastatur sein, wahrscheinlich die Blöcke durch geschweifte Klammern eingeschlossen werden, sämtliche Schlüsselwörter bekannt und der Mehrwert des Alleinstellungsmerkmals ausreichend groß sein.

Unabhängig davon braucht der Sprachschöpfer eine große Community, denn viele Programmierprobleme werden über Suchmaschinen gelöst. Und die liefern nur genügend Stoff, wenn sich eine ausreichende Anzahl Entwickler damit beschäftigt. Vielleicht wäre es deshalb nicht verkehrt, die Sprache für die JVM und CLR zu schreiben. Denn damit stehen, weitverbreitete Plattformen zur Verfügung, mit denen man alles machen kann – insbesondere verschiedene problemorientierte Sprachen miteinander kombinieren.

Über die wirklich wichtigen Programmiersprachen kann man ja sagen, was man will, aber sie sind fertig. Das sollte eine Sprache bei ihrer Einführung auch sein. Und selbst wenn über die Lebensdauer der Sprachen noch die eine oder andere Änderung vorzunehmen ist, sollte das nur behutsam erfolgen, möglichst ohne die bestehende Codebasis zu gefährden. Das gilt nicht nur für den Quellcode, sondern auch für eine etwaige Binärkompatibilität. Dass sich die Programmiersprache Java so zögerlich weiterentwickelt, liegt ja nicht daran, dass es zu wenige Einfälle gäbe, sondern daran, dass unter keinen Umständen die riesige Codebasis bei einer Änderung tangiert werden darf.

Und vielleicht muss es auch nur Spaß bereiten. Yukihiro Matsumoto wollte schon immer mal eine Programmiersprache schreiben, und daraus ist Ruby geworden. Guido van Rossum brauchte eine bessere Alternative zu C und der Bourne Shell, um das Amoeba-Betriebssystem zu administrieren, und schuf Python. Larry Wall musste Log-Dateien auswerten und war erst zufrieden, als er Perl geschaffen hatte. Also wenn (nicht nur) der geneigte Leser etwas braucht, wozu die optimale Unterstützung fehlt, mache er es doch einfach – und erzähle dann darüber. Der Rest ergibt sich von ganz alleine. (ane)

Michael Wiedeking
ist Gründer und Geschäftsführer der MATHEMA Software GmbH, Veranstalter des Herbstcampus und Herausgeber des „Kaffee-Klatsch" – eines kostenlosen PDF-Magazins rund um Softwareentwicklungsthemen.

Wo Haskell und Co objektorientierten Sprachen überlegen sind

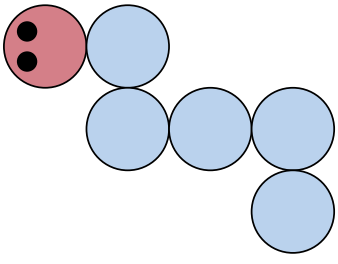
Praxisreif

Michael Sperber

Funktionale Programmierung erhebt den Anspruch, die „bessere OOP“ zu sein. Mit ihr sollen Programmierer mit geringerem Aufwand bessere Programme schreiben können: kürzer, lesbarer, zuverlässiger und schneller. Sie ist inzwischen eine ernst zu nehmende Alternative zu traditionellen objektorientierten Sprachen. Grund genug, mit konkreten Beispielen Lust auf die funktionale Programmierung zu wecken.

Funktionale Programme sehen anders aus als objektorientierte, imperative. Verdeutlicht seien die Unterschiede anhand eines in Java und in der funktionalen Sprache Haskell [a] verfassten Beispiels: der Entwicklung der Kernfunktionen einer vereinfachten Variante des Videospiels Snake. Bei ihm bewegt sich eine Schlange über den Schirm, die der Benutzer steuern kann. Sie besteht aus einem Kopf und mehreren Gliedern (s. Abb. 1). Ab und zu wächst die Schlange um ein Glied: Der Benutzer muss möglichst lange verhindern, dass sie sich selbst frisst, der Kopf also auf eines der Glieder trifft.

Die beiden Version verwenden das gleiche Datenmodell und verzichten dabei zum Beispiel auf Java- oder Haskell-spezifische Collections-Datentypen. Die folgende Einsicht ist für das Datenmodell wichtig: Wenn man bei einer Schlange mit mindestens einem Glied das letzte weglässt, bleibt immer noch (rein technisch gesehen zumindest) eine Schlange. Diese ist entsprechend der Idee entweder nur ein Kopf *SnakeHead* oder ein Schlangenglied *SnakeLink* mit einer „Restschlange“ vorne dran.



Eine Schlange im „Snake-Spiel“ mit Kopf und Gliedern (Abb. 1)

In der Datenrepräsentation – von hinten nach vorn – sieht die Schlange aus Abbildung 1 so aus:

```
SnakeLink SOUTH
SnakeLink EAST
SnakeLink EAST
SankeLink SOUTH
SnakeLink EAST
SnakeHead (0, 0) WEST
```

Der Artikel setzt Java-, aber keine Haskell-Kenntnisse voraus.

Die javanische Schlange

In Java sei das Klassendiagramm aus Abbildung 2 mit einem Interface und zwei Klassen für die zwei Schlangenformen genutzt: Beim Schlangenkopf sind für die Realisierung des Spiels die Position *pos* und die Bewegungsrichtung *dir* mitzuführen, beim Schlangenglied die Restschlange *prev* sowie die Richtung von der Restschlange zu diesem Glied *relPos*. Zunächst werden die Codegerüste für Interface und Klassen mit den kanonischen Konstruktoren erstellt – auch für die benötigten Hilfsklassen *Posn* und *Direction*, die Rümpfe folgen später (s. Listing 1).

Programmiert werden nun die zwei in der Realisierung des Videospiels benötigten Methoden, einerseits die Berechnung der absoluten Position eines Schlangenglieds, andererseits die Bewegung der Schlange in eine bestimmte Richtung. Zunächst bekommt *ISnake* zwei Methodensignaturen dafür: