Der Strichpunkt-Krieg

Jeder Programmierer weiß, dass es nur eine einzig wahre Computersprache gibt. Jede Woche eine neue.

Von Brian Hayes

Wer in aller Welt zu Gast bei Freunden sein will, hat einiges zu lernen – zum Beispiel 6912 Versionen der Frage: »Wo ist die Toilette, bitte?« Das ist die Zahl der bekannten auf der Erde gesprochenen Sprachen (siehe www.ethnologue.com).

Wer sich allen Computern der Welt verständlich machen will, steht vor einer nur geringfügig kleineren Aufgabe. Wie sagt man

```
printf("hello, world\n");
```

(hier in der Sprache C) in allen anderen Programmiersprachen? Nach einer Aufstellung von Bill Kinnersley von der Universität von Kansas gibt es deren rund 2500 Stück. In einer anderen Sammlung kommt Diarmuid Piggott von der Murdoch University in Perth (Australien) sogar auf mehr als 8500. Das ist erstaunlich, denn die menschlichen Sprachen haben sich im Lauf von Jahrtausenden entwickelt und diversifiziert, während es Computersprachen erst seit 50 Jahren gibt. Selbst nach den eher konservativen Annahmen der Kinnersley-Zählung bedeutet dies, dass seit Fortran jede Woche eine neue Computersprache erfunden wurde.

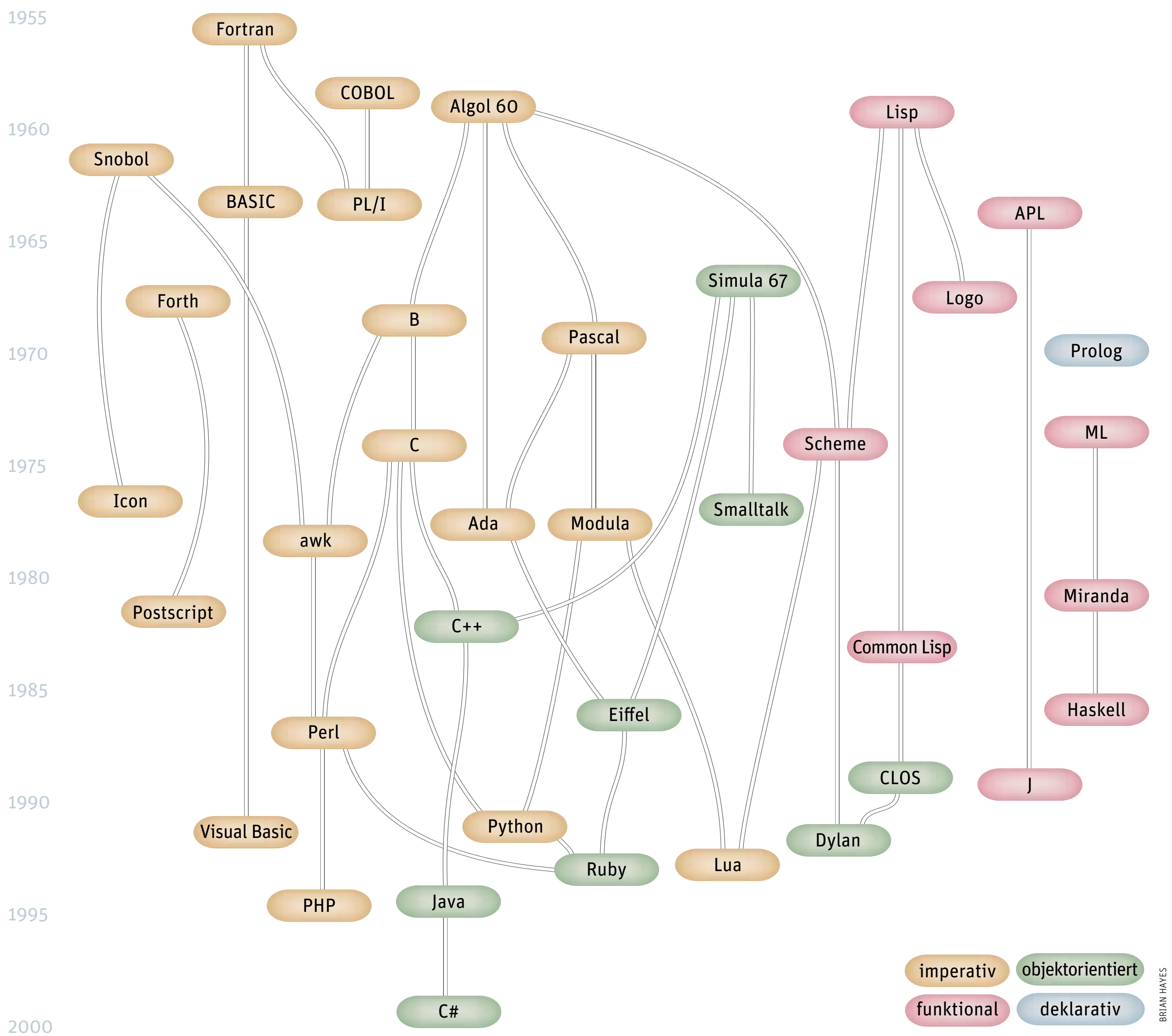
Für Völkerkundler ist die Sprachenvielfalt ein kultureller Schatz, der gepflegt und erhalten werden will, ähnlich der biologischen Artenvielfalt. Alle menschlichen Sprachen sind wertvoll; je mehr es gibt, desto besser. Diese Attitüde ehrfürchtigen Respekts vor der Kulturgeschichte ist in unserem Fall nur mühsam aufrechtzuerhalten. Eine neue Programmiersprache entsteht nicht durch eine unerforschliche Vielzahl winziger evolutionärer Schritte über einen langen Zeitraum, sondern durch einen bewussten Akt mit der Absicht, etwas Vorhandenes zu verbessern. Die Tatsache, dass noch immer neue Sprachen aus dem Boden schießen, beweist: Wir sind mit dem Verbessern noch lange nicht fertig. Wir kennen offenbar noch nicht die beste Schreibweise – oder auch nur eine hinreichend gute –, um einen Algorithmus auszudrücken oder eine Datenstruktur zu definieren.

Etliche Programmierer aus meinem Bekanntenkreis würden an dieser Stelle mit Nachdruck widersprechen. Die richtige Programmiersprache sei längst gefunden, und nur ein böswilliger Ignorant könne das Gegenteil behaupten. Nicht dass sie in allen Einzelheiten perfekt wäre; aber ihre Grundstruktur ist solide, sie löst das Hauptproblem, und nun sollten wir alle uns zusammentun, um sie zu verfeinern und zu verbessern. Leider meint jeder meiner Freunde eine andere Sprache. Es ist Lisp, sagt der eine. Nein, Python. Ruby. Java, C#, Lua, Haskell, Prolog, Curl.

Sprachenvielfalt hat auch eine Schattenseite. Völker unterschiedlicher Sprachen gehen nicht immer friedfertig miteinander um. Unweigerlich kommt einem das Wort »Balkanisierung« in den Sinn. Und ebenso wie die Länder Südosteuropas hatte die Computerbranche unter Abspaltungen und bitteren Konflikten zu leiden. Meines Wissens sind in diesem Streit noch keine Todesfälle zu beklagen, aber der rüden Worte wurden mehr als genug gewechselt – in vielen Sprachen.

Klein-Ender gegen Groß-Ender

Als Gulliver, der Held von Jonathan Swifts 1726 erschienenem fantastischen Roman, schiffbrüchig an die Küste von Liliput angespült wird, erfährt er, dass die Liliputaner mit den Bewohnern der Nachbarinsel Blefuscu einen überaus grausamen Krieg um die Frage führen, ob man ein gekochtes Ei am stumpfen oder am spitzen Ende aufschlagen soll. Der sprichwörtlich gewordene Sturm im Eierbecher fand 250 Jahre später eine Wiederaufführung in der echten (Computer-)Welt. Eine Binärzahl kann entweder »von rechts nach links«, das heißt mit der Einerziffer zuerst, oder »von links nach rechts«, das heißt mit



Diese Chronologie ausgewählter Programmiersprachen ist kein echter Stammbaum; die Linien zeigen nur an, welche Sprachen einander erheblich beeinflusst haben. Auch die Klassifikation der Sprachen (Farbcode rechts unten) ist als Näherung zu verstehen; nur einige »reine« Sprachen gehören ausschließlich in eine dieser Kategorien. Das Material dieser Grafik stammt zum Teil von Éric Lévénez und Pascal Rigaux sowie aus Konferenzen zur Geschichte der Programmiersprachen, welche die Association for Computing Machinery veranstaltete.

der Ziffer mit dem höchsten Stellenwert zuerst, übertragen werden. Da alles, womit ein Computer umgeht, letzten Endes Binärzahlen sind, betrifft die Frage, ob das dicke oder das dünne Ende zuerst kommt, alle Daten überhaupt. Was ist besser? Es macht praktisch keinen Unterschied, aber das Leben wäre um einiges einfacher, wenn alle dieselbe Wahl träfen. Dies ist nicht der Fall, und deshalb müssen beim Übergang zwischen verschiedenen Systemen unter großem Aufwand an Hard- und Software unzählige kleine Binär-Eier umgedreht werden.

Danny Cohen von der Universität von Süd-Kalifornien war der Erste, der den modernen Krieg der Klein-Ender gegen die Groß-Ender mit Swifts Geschichte in Beziehung setzte. Sein glänzend geschriebener Artikel »On holy wars and a plea for peace« (»Über heilige Kriege und ein Friedensgesuch«), der 1980 in der Zeitschrift »Computer« erschien, wurde von vielen gelesen und geschätzt; das Friedensgesuch indes blieb unerhört.

Ein anderer – eingeschlafener, aber nie offiziell beigelegter – Konflikt tobte um das Semikolon. In Algol und Pascal müssen Programmbefehle durch Strichpunkte getrennt werden. In einem Programmstück wie

```
x:=0; y:=x+1; z:=2
```

zeigen sie dem Compiler (dem Programm, welches das Programm des Benutzers verarbeitet), wo ein Befehl endet und der nächste anfängt. Auch C-Programme sind voller Strichpunkte; dort aber zeigen diese nicht die Grenze zwischen zwei Befehlen an, sondern das Ende eines Befehls. Worin besteht der Unterschied? In C muss nach dem letzten Befehl ein Semikolon stehen, in Pascal nicht. Das war eine der Diskrepanzen, über die Brian W. Kernighan von den AT&T Bell-Laboratorien 1981 in dem berühmt gewordenen Pamphlet »Why Pascal is not my favorite programming language« (»Warum Pascal nicht meine Lieblingssprache ist«) herzog. Die ▷

▷ Streitschrift wurde nie veröffentlicht, fand jedoch unter der Hand weite Verbreitung (immerhin ist Kernighan einer der Schöpfer des Betriebssystems Unix und zahlreicher Programmiersprachen) und läutete – wie man im Rückblick erkennen kann – das Ende von Pascal als einer ernst zu nehmenden Programmiersprache ein.

Einen weiteren Grund für fortdauernde Streitigkeiten liefert die Zählweise. Soll man die Elemente einer Liste mit 0, 1, 2 oder mit 1, 2, 3 nummerieren? Jeder im Reich der Computer weiß die richtige Antwort und vertritt sie mit tiefster Überzeugung. Aber welche ist die richtige? Nehmen wir den Java-Ausdruck `Date(2006, 1, 1)`. Für welches Kalenderdatum steht er Ihrer Meinung nach? Die Antwort lautet: für den 1. Februar 3906. In Java werden die Monate von 0 an gezählt, Tage ab 1 und Jahre ab 1900.

Programmierstile

Ein und dasselbe Problem kann sowohl im imperativen als auch im funktionalen Stil gelöst werden. Die hier gezeigten Programme berechnen beide die Fakultät der Zahl *n*: das Produkt aller natürlichen Zahlen zwischen 1 und *n*.

Die imperative Prozedur (oberes Schema) arbeitet mit einer Iterationsschleife (»`for i = 1,n do`«): Dem Computer wird befohlen, immer wieder den Wert einer Variablen namens `accumulator` zu überschreiben.

Das funktionale Programm (unten) dagegen sagt dem Computer nicht direkt, was er der Reihe nach zu tun hat. Vielmehr führt es die Lösung des Problems »Berechne die Fakultät von *n*« auf die des einfacheren Problems »Berechne die Fakultät von *n*–1« zurück. Zusammen mit der Startbedingung »Die Fakultät von 1 ist 1« ist auch dies eine vollständige Anweisung. Der Computer führt dieses Programm aus, indem das Funktionsunterprogramm `factR` sich immer wieder selbst aufruft, bis das innerste Unterprogramm den Wert 1 zurückgibt und daraufhin das Produkt von innen nach außen berechnet wird.

Diese »rekursive« Formulierung ist häufig die geschickteste Weise, die Lösung eines mathematischen Problems auszudrücken; mit dem funktionalen Stil wird das zugehörige Computerprogramm ebenso elegant wie die mathematische Lösung.

Es gibt Programmiersprachen, die ausschließlich imperativ oder ausschließlich funktional sind. Diese Beispiele sind jedoch beide in der Sprache Lua geschrieben, die in den 1990er Jahren an der Bischöflichen Katholischen Universität in Rio de Janeiro entwickelt wurde.

Imperatives Programm

```
function factI (n)
   local accumulator = 1
   for i = 1,n do
      accumulator = accumulator*i
   end
   return accumulator
end
```

Programmablauf

```
factI(4):
         accumulator = 1
i = 1    accumulator = 1 * 1
         accumulator = 1 * 2
         accumulator = 2 * 3
         accumulator = 6 * 4
return 24
```

Funktionales Programm

```
function factR (n)
   if n == 1 then
      return 1
   else
      return n*factR(n-1)
   end
end
```

Programmablauf

```
factR(4):
4 * factR(3) =
    3 * factR(2) =
        2 * factR(1) =
                1
                1
            1
        2
    6
24
```

BRIAN HAYES

Selbst Programmteile, die eigentlich gar keine sind, sorgen für Zwietracht. »Kommentare« sind für menschliche Leser gedacht und müssen in irgendeiner Weise gekennzeichnet werden, damit der Compiler sie ignoriert. Das sollte doch nicht so schwer sein, ein einheitliches Zeichen zu wählen, das in allen Programmen diesem Zweck dient. Doch das von Pascal Rigaux zusammengestellte Kompendium der Programmiersprachen-Syntax – ein wunderbares Dokument, nebenbei gesagt –, zählt 39 inkompatible Kommentar-Kennzeichen auf: `#` in awk; `\` in Forth; `(* ... *)` in Pascal; `/* ... */` in C … . Außerdem tobt eine Diskussion darüber, ob Kommentare »schachtelbar« *(nestable)* sein sollten – ob es also erlaubt sein soll, Kommentare innerhalb von Kommentaren zu schreiben.

Und dann gibt es noch die Kontroverse über den *CamelCase*. Die meisten Programmiersprachen schreiben vor, dass die vom Programmierer zu vergebenden Namen für Variable, Prozeduren und so weiter aus einem einzigen Wort bestehen müssen, das keine Leerzeichen enthalten darf. Aber Allewörterzusammengeschrieben ist schlecht lesbar. So kam man darauf, die entsprechenden Wörter nicht in *upper case* (Großbuchstaben) oder *lower case* (Kleinbuchstaben), sondern in *CamelCase* zu schreiben – mit dem Höcker in der Mitte. Offen gestanden: Es gibt zurzeit keinen ernsthaften Streit darum, ebenso wie sich nur noch die ärgsten WahrerInnen der deutschen Rechtschreibung über das Binnen-I in Wörtern wie »StudentInnen« oder »ErstsemesterInnen« aufregen. Aber der Name allein gab Anlass für gelehrte und spaßige Diskussionen, in denen auch traditionelle Artnamen wie *Camelus dromedarius* und *C. bactrianus* aufgegriffen wurden. Außerdem öffneten sie den Blick für Verfeinerungen wie den *sulkingCamelCase* (»Kamel mit hängendem Kopf«).

Wenn ich mich hier – zu Recht, glaube ich – über einige Mätzchen lustig mache, möchte ich keineswegs den Eindruck erwecken, es gehe nur um kosmetische Belange oder alle Programmiersprachen seien im Grunde dasselbe. Im Gegenteil: Das Faszinierende an Programmiersprachen ist, wie extrem sie sich unterscheiden. Ich halte dafür, dass beispielsweise C und Lisp weiter voneinander entfernt sind als zwei beliebige menschliche Sprachen.

Noam Chomsky behauptet, alle menschlichen Sprachen hätten dieselbe »Tiefenstruktur«, die vielleicht sogar im Gehirn »fest verdrahtet« sei. Auch bei den Computersprachen scheint es universelle Eigenschaften zu geben. Fast allen liegt eine einheitliche Struktur zu Grunde, die so genannte kontextfreie Grammatik. Auch auf der semantischen Ebene sind sie fast alle gleich mächtig: Was man in der einen Sprache programmieren kann, gelingt auch in jeder anderen – mit entsprechendem Aufwand. Doch diese formale Äquivalenz ist irreführend. Den Anwendern geht es nicht um die Grenzen der Ausdrucksmöglichkeiten, sondern darum, wie schnell und elegant man dem Computer sagen kann, was er tun soll.

In den 1930er Jahren behaupteten die Linguisten Edward Sapir und Benjamin Lee Whorf, dass das, was man denken kann, von der Sprache abhängt, in der man denkt. In Bezug auf die natürlichen Sprachen ist die Sapir-Whorf-Hypothese vielfach angefochten worden; bei Computersprachen indes hat sie einiges für sich. Die Konstrukte, die eine Sprache bereitstellt, beeinflussen in hohem Maße die Denkansätze und Problemlösungsstrategien des Programmierers.

Es gibt 39 verschiedene Vorschriften, wie man den Computer anzuweisen hat, gewisse Teile des Programms nicht zur Kenntnis zu nehmen

Kleine Computersprachkunde

Programmiersprachen werden üblicherweise in vier Gruppen unterteilt. Imperative Sprachen basieren auf Befehlen: Mach dies, mach das, mach das nächste (Kasten links). Die Ausführung eines Befehls verwendet abgespeicherte Daten und wirkt auf sie zurück, wobei im Prinzip der Zustand des Systems an jeder Stelle verändert werden kann. Alle frühen Sprachen wie Fortran, Cobol und Algol sind in erster Linie imperativ.

Hinter den funktionalen Programmiersprachen steckt die Idee der mathematischen Funktion, zum Beispiel $f(x) = x^2$. Eine Funktion ist ein Unterprogramm, eine *black box*, die als Input ein Argument (das x) entgegennimmt und als Output den Funktionswert (das $f(x)$) liefert. Wesentlich ist, dass die Berechnung nur von den Argumenten abhängt und nur den Ausgabewert beeinflusst; es gibt keine Nebenwirkungen. Diese Eigenschaft macht das Verhalten funktionaler Programme durchschaubar: Um zu verstehen, was geschieht, wenn das Programm eine Funktion ausführt, kann man den gesamten Rest des Computers getrost außer Acht lassen. Funktionales Programmieren begann mit Lisp (Spektrum der Wissenschaft 4/1983, S. 14, 5/1983, S. 13, 6/1983, S. 10 und 7/1983, S. 6); allerdings erlaubt diese Sprache auch andere Programmierstile. John Backus, der führende Entwickler von Fortran, der auch Beiträge zu Algol leistete, wurde später zum Verfechter der funktionalen Sprachen. Seitdem sind einige »reine« funktionale Sprachen entwickelt worden, darunter Miranda und Haskell, sowie einige nicht ganz so reine wie ML.

Beim objektorientierten Programmieren werden Befehle und die zugehörigen Daten zu einer geschlossenen Struktur verbunden. Man ▷

»Fortran ist eine frühkindliche Entwicklungsstörung, PL/I eine tödliche Krankheit, und das Lehren von Cobol sollte als Misshandlung des Gehirns unter Strafe gestellt werden«

schreibt also nicht mehr eine Prozedur, die gewisse Dinge mit einer Datenstruktur tun soll, sondern definiert eine Datenstruktur (eine »Klasse«, deren Elemente einzelne »Objekte« sind); in der Definition enthalten sind die Prozeduren, die auf diese Datenstruktur wirken können. Bei den meisten objektorientierten Sprachen gibt es auch den Begriff der »Vererbung«: Elemente einer untergeordneten Klasse »erben« alle Eigenschaften, die den Objekten der übergeordneten Klasse zukommen, sodass der Programmierer nur noch das definieren muss, was die Elemente der speziellen Klasse von denen der allgemeinen unterscheidet. Die objektorientierten Sprachen gehen auf SIMULA 67 zurück, erregten jedoch erst mit Smalltalk in den 1980er Jahren Aufmerksamkeit. Kurioserweise wurde das Prinzip der Objektorientierung sehr populär, nicht jedoch die Sprache Smalltalk; stattdessen wurden objektorientierte Merkmale an bestehende Sprachen angeschraubt. C beispielsweise entwickelte sich über C++ und Objective-C bis zu C#; Java zählt ebenfalls zu dieser Familie. Inzwischen hat objektorientiertes Gedankengut fast alle neuen Sprachen infiltriert.

Die vierte Gruppe von Programmiersprachen wird als logische, relationale oder deklarative Sprachen bezeichnet. Sie haben gemeinsam, dass der Programmierer dem Computer nicht Schritt für Schritt vorschreibt, was er tun soll, sondern Fakten (»Waldi ist ein Dackel«) und Relationen (»Alle Dackel sind Hunde«) notiert. Dem Computer bleibt es dann überlassen, wie er die Fragen des Benutzers beantwortet, indem er aus Fakten und Relationen Schlüsse zieht. Die bekannteste dieser Sprachen ist Prolog. Verwandte Konzepte gibt es auch in Sprachen für alltäglichere Anwendungen wie Datenbankabfragen und Tabellenkalkulation.

Diese vier Kategorien bezeichnen nur die Unterschiede im Grundprinzip. Programmiersprachen unterscheiden sich in vielen weiteren Merkmalen, zum Beispiel in der äußeren Erscheinung. C ist sehr wortkarg, Cobol sehr weitschweifig. Lisp ist voller Klammern. Perl, sagte ein Witzbold, sieht aus wie Snoopy beim Fluchen: »@&$^^#@!«

Sprachen werden auch in »niedere« *(low-level)* und »höhere« *(high-level)* eingeteilt. Niedere Sprachen erlauben direkteren Zugriff auf die Hardware der Computer wie Adressen im Arbeitsspeicher oder Ein- und Ausgabegeräte. Bei höheren Sprachen liegt eine schützende »Abstraktionsschicht« dazwischen.

In den 1970er Jahren war »strukturelles Programmieren« sehr in Mode. Man wurde gezwungen, strenge Regeln über Datentypen einzuhalten: Für jeden zu verwendenden Speicherplatz musste man vorab festlegen, welche Sorte Daten dort stehen sollte, und war strikt an diese Festlegung gebunden. Außerdem war der Befehl »go to« praktisch verboten: Man durfte das Programm nicht einfach irgendwie von der niedergeschriebenen Folge der Befehle abweichen lassen, sondern musste dafür spezielle grammatische Konstruktionen verwenden. Der klassische Vertreter dieser Sprachengruppe ist Pascal. Bald wurden die strengen Regeln als »bondage and discipline« verspottet; so nennen die Sadomasochisten ihre Fesselungs- und Züchtigungsspiele. Die meisten Programmierer konnten aus dieser Einschränkung ihrer Bewegungsmöglichkeiten nicht den rechten Lustgewinn ziehen; als Gegenreaktion erblickten zügellosere Sprachen wie C das Licht der Welt.

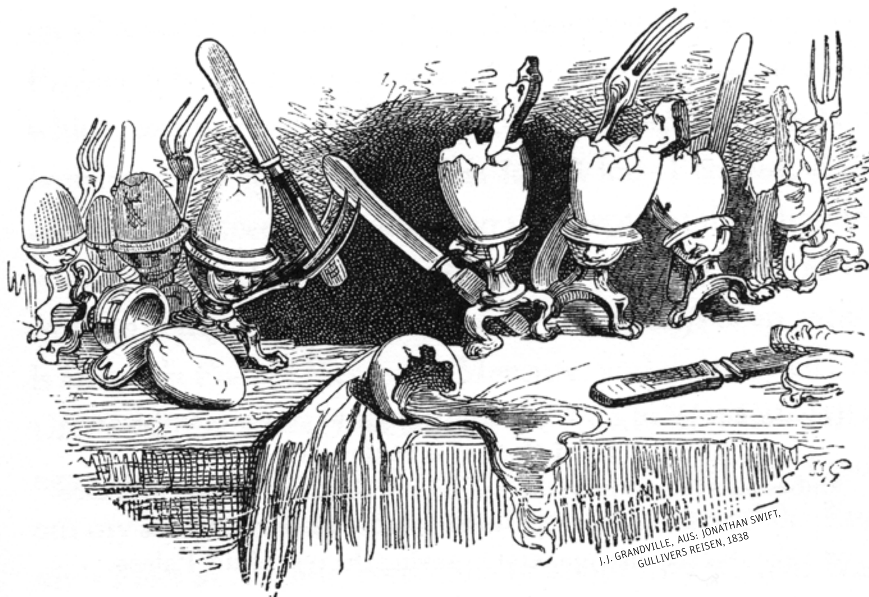
Programmiersprachen unterscheiden sich auch in Zielgruppe und Anwendungsgebiet. Fortran (FORmula TRANslator) begann als Sprache für wissenschaftliche Berechnungen, Cobol (COmmon Business Oriented Language) diente geschäftlichen Anwendungen. Etliche interessante Sprachen waren ursprünglich für Lehrzwecke oder für Kinder gedacht. Basic, Pascal und Smalltalk gehören dazu, ebenso Logo. Alle diese Sprachen mussten zunächst darum kämpfen, von den Erwachsenen ernst genommen zu werden.

Bekehrung aus edelsten Motiven

Nachdem es so viele so verschiedene Programmiersprachen gibt, »lasst tausend Blumen blühen«, freut euch der Vielfalt und lasst jedem seinen Spaß – könnte man sagen. Stattdessen führen wir Kreuzzüge, um die Verblendeten zu konvertieren oder notfalls auszurotten.

Edsger W. Dijkstra, einer der bedeutenden Köpfe aus dem Lager der strukturierten Programmierung, schrieb 1975 ein Thesenpapier mit dem Titel »How do we tell truths that might hurt?« (»Wie sprechen wir schmerzhafte Wahrheiten aus?«). Und bei der Verbreitung seiner »Wahrheiten« nahm er kein Blatt vor den Mund. Fortran sei »eine frühkindliche Entwicklungsstörung«, PL/I »eine tödliche Krankheit«, APL »ein Fehler, der bis zur Perfektion getrieben wurde«. Schüler, die dem Einfluss von Basic ausgesetzt seien, würden »geistig verstümmelt ohne Hoffnung auf Besserung«, und das Lehren von Cobol sollte »als Misshandlung des Gehirns unter Strafe gestellt werden«. Als das Papier einige Jahre später veröffentlicht wurde, zahlten die Verteidiger von Cobol und Basic die Kritik mit gleicher Münze heim, kamen allerdings in der Diktion an Dijkstras ätzenden Sarkasmus nicht heran.

Die meisten Dispute über Programmiersprachen sind nicht so bösartig und humorlos



J. J. GRANDVILLE, AUS: JONATHAN SWIFT, GULLIVERS REISEN, 1838

In diesem Gemetzel am Frühstückstisch endet der Krieg zwischen den »Klein-Endern« und den »Groß-Endern«. Der Stich aus der 1838 erschienenen Ausgabe von »Gullivers Reisen« stammt von J. J. Grandville.

wie Dijkstras »Wahrheiten«-Pamphlet. Heutige Missionare geben sich lockerer; sie wenden mehr Zeit damit auf, ihre eigene Religion zu preisen, als die der anderen schlecht zu machen. Ihre Predigten lauten nicht mehr: »Du wirst in der Hölle schmoren, wenn du in C programmierst«, sondern: »Schau, welch ein Paradies Python dir bietet«. (Irgendwie gefielen mir die alten Sprüche besser.)

Hinter den meisten Bekehrungsversuchen dieser Art stecken die edelsten Motive. Wer ein intelligentes und elegantes Programmiermittel gefunden zu haben glaubt, will die ganze Welt mit seiner guten Tat beglücken. Allerdings ist auch Eigeninteresse mit im Spiel. Damit die Sprache *P* gedeiht, muss sie eine Anwendergemeinde haben: Leute, die Programme in *P* schreiben, Bücher über *P* kaufen, ihre Studenten in *P* unterrichten, *P*-Programmierer einstellen und darauf bestehen, dass *P* auch auf dem neu anzuschaffenden System läuft. Jeder, der zu *P* bekehrt wird, erhöht die Überlebenschancen von *P*. Wenn er von der konkurrierenden Sprache *Q* kommt, umso besser.

Streitereien über die Schreibweise sind nicht auf die Welt der Computer beschränkt. In der Folge des Prioritätsstreits zwischen Gottfried Wilhelm Leibniz und Isaac Newton zankten sich die Mathematiker jahrzehntelang, ob man eine Ableitung als dx/dt (Leibniz) oder als $\dot{x}$ (Newton) schreiben sollte. Chemiker gerieten sich über die Benennung von Molekülen in die Haare. Selbst Schachspieler stritten über die Notation der Züge. Doch in der Computerwissenschaft ist das Problem eine Nummer größer. In der Differenzialrechnung gab es niemals 2500 unterschiedliche Schreibweisen für eine Ableitung.

In all den Jahren wurde das Gezeter um Programmiersprachen wiederholt als Bedrohung für den weiteren Fortschritt der Computerwissenschaft betrachtet. Die übliche Reaktion bestand darin – dreimal dürfen Sie raten –, eine weitere Computersprache zu entwickeln. »Wenn wir nur alle uns zusammentun und uns auf eine einzige, letzte, große Computersprache einigen würden …« Das war in den 1960er Jahren das Motiv für die Entwicklung von PL/I, jener Sprache, die Dijkstra als »tödliche Krankheit« schmähte. Später sollte – auf Betreiben des US-Verteidigungsministeriums – Ada die universelle Programmiersprache werden. Vor zehn Jahren schließlich lag alle Hoffnung auf Java (Spektrum der Wissenschaft 7/1996, S. 17), die vor allem zur Verbreitung übers Internet konzipiert war. »Write once, run anywhere« (»einmal schreiben, überall betreiben«) war der Slogan, der nur beschränkt in Erfüllung gegangen ist.

Einige Computersprachen, vor allem Fortran und Lisp, scheinen nahezu unsterblich zu sein. Die anderen sind wie Wellen, die an den Strand branden und dann im Sand versickern. Auf dem Kamm der jüngsten Welle reiten die so genannten Script-Sprachen wie Python und Ruby. Sie sind von bescheidener Herkunft. Eigentlich ist ein Script eine Folge von Befehlen an ein Betriebssystem wie Unix, mit der man mehrere Programme in Folge ablaufen lassen und die Datenübergabe zwischen ihnen regeln kann. Außerdem gibt es *extension languages*, Erweiterungssprachen, mit denen man ein vorhandenes Programm um gewisse Dinge bereichern kann, ohne in den Programmtext selbst einzugreifen. Doch inzwischen haben sich die Script-Sprachen zu Allzweck-Programmiersprachen gemausert. Heute werden damit oft Internetanwendungen geschrieben. Python wird auch in der Wissenschaft verwandt.

Das Internet brachte der Artenvielfalt einen neuen Aufschwung. Wer heute eine Website verwalten will, muss sich in einem halben Dutzend Programmier- und Datenformatierungs-Sprachen auskennen. Da gibt es HTML *(HyperText Markup Language)* für die Grundstruktur der Webseiten und CSS *(Cascading Style Sheets)* für die Details der Präsentation, dazu Javascript für Dinge wie Pop-up-Fenster, mit denen man den Benutzer nerven kann. Auf dem Serve werden die Inhalte meist in einer Variante von XML *(Extensible Markup Language)* abgelegt und mit einer Datenbankabfragesprache wie SQL abgerufen. All diese Einzelteile werden mit einer Script-Sprache zusammengehalten, etwa PHP, Perl, Python oder Ruby.

Natürlich schreit die Situation geradezu nach einer weiteren Sprache, die alle anderen ersetzt. Zwei Kandidaten stehen schon bereit: Curl und Links.

▷

▷ Mein Friedensgesuch wäre glaubwürdiger, wenn ich mich als unparteiischer Schiedsrichter präsentieren könnte, der kein Interesse am Ausgang des Sprachenstreits hat. Doch es ist Zeit, ein Bekenntnis abzulegen: Auch ich habe eine bevorzugte Programmiersprache, an der ich hänge wie ein Kind an einem abgewetzten Teddybär. Nehmt mir bitte, bitte mein Lisp nicht weg!

Es ist schwer, meine Vorliebe für Lisp zu begründen, ohne selbst in missionarischen Glaubenseifer zu verfallen. Sagen wir's so: Es ist eine sehr einfache Sprache mit einem einzigen Trick; und den beherrscht sie sehr gut. Jeder Lisp-Ausdruck ist eine Liste. Wenn der Computer eine solche Liste bekommt, nimmt er deren erstes Element als den Namen einer Funktion und alle weiteren Elemente als die Argumente dieser Funktion. Ein Beispiel: `(/ (+3 5) 2)` ist ein Programm, das `(+3 5)` durch 2 teilt; dabei ist das erste Argument der Funktion `/` wieder eine Liste, nämlich das Unterprogramm `(+3 5)`, das 3 und 5 addiert. Der Wert dieses ganzen Ausdrucks ist 4. Die Syntax ist extrem einfach, fast schon primitiv, aber darin liegt ihre Stärke. Die Verfechter von Lisp betonen immer, dass Daten und Programme auf dieselbe Weise dargestellt werden. Dies macht es einfach, Programme zu schreiben, die andere Programme beeinflussen.

Das ist wahr, aber was speziell mich begeistert, ist die Einheitlichkeit der Notation. Alles wird auf dieselbe Weise erledigt, man muss sich also nicht viel merken. (Was ich lieber verschweige, ist die Überfülle an Klammern in Lisp (die einige Leute ärgert). (Was die Welt braucht (meiner Meinung nach) ist nicht (eine Lisp-Variante (mit weniger Klammern)), sondern (eine gewöhnliche Schriftsprache (mit mehr)).))

Lisp wurde vor mehr als fünfzig Jahren von John McCarthy entwickelt, der jetzt an der Universität Stanford lehrt. Ich selbst kenne es seit 25 Jahren. Da mag meine persönliche Vorliebe schon etwas antiquiert, stur und hinterwäldlerisch anmuten, so als würde ich darauf bestehen, mich auf Lateinisch auszudrücken. Hat denn die Entwicklung der letzten fünfzig Jahre mit ihren 2500 Sprachen wirklich nichts hervorgebracht, das besser wäre als Lisp?

Nein, das würde ich nicht behaupten. Und natürlich ist das Lisp, das ich heute »spreche«, nicht die gleiche Sprache, die McCarthy vor 50 Jahren einführte. Lisp wurde erweitert, überholt, aktualisiert und aus den zahlreichen Dialekten, in die es zwischendurch zerfallen war, zu einem neuen Standard namens Common Lisp wiedervereinigt. Aber die Teile der Sprache, die mir am besten gefallen, sind diejenigen, die es von Anfang an gab und die am wenigsten verändert wurden.

Vor zwei Jahren fand in Stanford eine internationale Lisp-Konferenz statt. Die Gläubigen waren unter sich, und natürlich sprachen sie auch darüber, wie man das Lisp-Evangelium dem Rest der Welt vermittelt. Zugleich zeigte sich, dass selbst Vertreter derselben Sprache sich bis tief in die Nacht in die Haare geraten können.

Nehmt mir meinen Teddybär nicht weg!

Am Ende der letzten Sitzung ergriff John McCarthy das Wort. Er ließ seinen Blick über die Zuhörer schweifen und sagte: »Wenn jemand in diesem Raum eine Bombe zündete, würde die Hälfte der weltweiten Lisp-Gemeinde ausgelöscht. Für die Sprache Lisp wäre das nicht schlecht, weil man sie dann neu erfinden müsste.« Damit wollte er nicht nur zum Ausdruck bringen, dass der Common-Lisp-Standard weiteren Innovationen im Wege steht – das Gute, einmal fest etabliert, ist immer der Feind des Besseren –, sondern dass er selbst, wenn er noch einmal von vorne anfangen könnte, einiges anders machen würde. Selbst der Schöpfer einer Sprache sieht sie also als weiter verbesserungsfähig an. Ich fand McCarthys Offenheit erfrischend, aber zugleich kroch der Gedanke in mir hoch: Nein, nein, bastel bitte nicht daran herum. Ich mag Lisp genau so, wie es ist.

Ja, ich bin überzeugt, dass Lisp zu den besseren Programmiersprachen zu zählen ist. Aber das ist nicht der eigentliche Grund meiner Liebe. Ich programmiere in Lisp aus demselben Grund, wie ich Prosa auf Englisch schreibe: nicht weil es die beste Sprache wäre, sondern weil es diejenige ist, die ich am besten beherrsche. ◁

AUTOR UND LITERATURHINWEISE

Brian Hayes ist Mathematiker und Redakteur von American Scientist.

© American Scientist
www.americanscientist.org

Selected papers on computer languages. Von Donald E. Knuth. CSLI Publications, Stanford (Kalifornien) 2003

History of programming languages II. Von Thomas J. Bergin und Richard G. Gibson jr. (Hg.). ACM Press, New York 1996

History of programming languages. Von Richard L. Wexelblat (Hg.). Academic Press, New York 1978

Weblinks finden Sie unter www.spektrum.de/artikel/872689.